Building an Arduino-based self-balancing robot – Part 1

https://roboticdreams.wordpress.com/2015/04/24/building-an-arduino-based-self-balancing-robot-part-1/

Posted on April 24, 2015 by Mike Jacobs

There are many examples on the internet that show how people have built their own self-balancing robots. Many blindingly accept the approaches of others with little explanation. Most include source code and very few include explanations of how the many magic numbers were determined or how to apply their approach to other projects. This series is intended to not only explain the workings of my approach but the lessons I have learned that hopefully you can apply to your self-balancing robot project. I plan to create a first generation prototype and then use what I learn to build ever more improved versions. The current prototype is a bit of Frankenstein. It combines components from Lego Mindstorms v1.0, an Arduino Uno, motor control shield, IMU breakout board, NiCd battery from an old HandyBoard and chassis boards from a Tamiya Universal Plate Set from Japan. Do you like the pipe insulation bumpers?

This posting covers the very heart of the system – the inertial measurement unit or IMU.

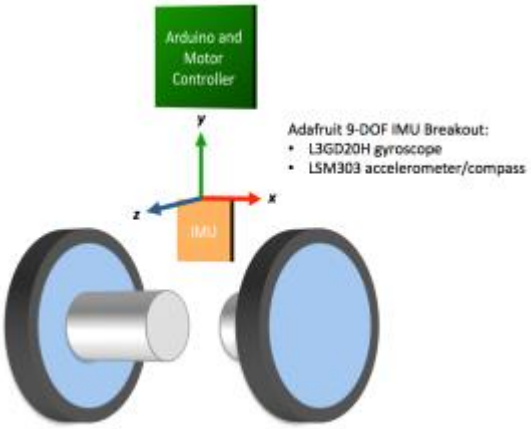
Not a Funny Looking Bird


Were talking about IMU here, not EMU!

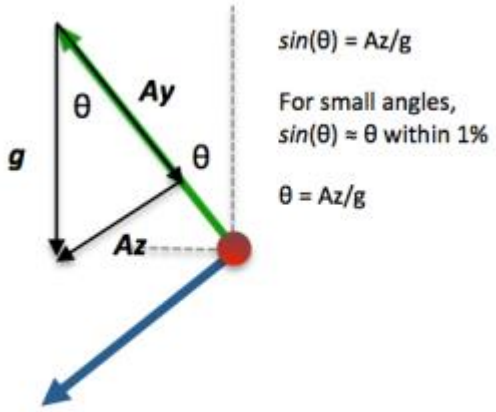
The job of the IMU is to quickly and accurately report the angle of your robot. The angle is used to decide to move your robot forward or backward in an attempt to keep it upright. I

elected to use the Adafruit 9 degree-of-freedom IMU Breakout. This tiny board combines a L3GD20H gyroscope with a LSM303 accelerometer/compass with a convenient Arduino library. It's called 9 degrees-of-freedom (or DOF) because it measures the x, y and z values for all three measurements (angular velocity, linear acceleration and direction).

Much of the internal workings of an IMU implementation depends on the physical installation and orientation of the hardware. The following diagram shows how mine is installed.



The IMU is installed so that the "front" of the robot points in the positive z direction. When the robot tilts forward, it corresponds to a positive rotation about the x-axis and a downward (negative) acceleration of the z-axis. The tilt angle can be determined based on a combination of the accelerometer and gyroscope readings. The accelerometer readings return the linear acceleration for each of the three axis. What which one to use? Have a look at the following diagram.



Tilting forward by an angle of θ from the side

As the robot tilts forward by an angle of θ, gravity exerts a force on the y and z aspects of the accelerometer. After projecting the force of gravity along those axis, we can express the z-axis acceleration Az as:

$\sin(\theta) = Az/g$

For angles less than 30 degrees (our intended control range for the robot), the angle in radians can be approximated within 1% or less by using the the sine of the angle as a close approximation of the actual angle:

$\theta(t) = Az/g$ (for a point-in-time from the accelerometer)

The tilt angle from the accelerometer is a point-in-time measurement and is not constant over any period of time.

As the robot tilts forward by an angle of θ over a period of time, it acquires an angular velocity. The angular velocity is measured by a gyroscope as the number of radians per second and is often designated ω. In this implementation I decided to make leaning forward be designated as a positive tilt angle. Using the right-hand rule, the above example will have a positive angular velocity. In order to make an angle determination, a time element must be considered in conjunction with the angular velocity – the value must be integrated. The gyroscope based tilt angle change can be estimated to be:

$\Delta\theta(t) = \omega * \Delta t$ (for a short time period Δt from the gyroscope)

The tilt angle from the gyroscope is an average over a short period of time, typically 10 milliseconds or so for a balancing robot.

Given the volatility of the accelerometer and gyroscope measurements, how can we determine the current tilt angle? We can start by using time discrete integration of the angle like this:

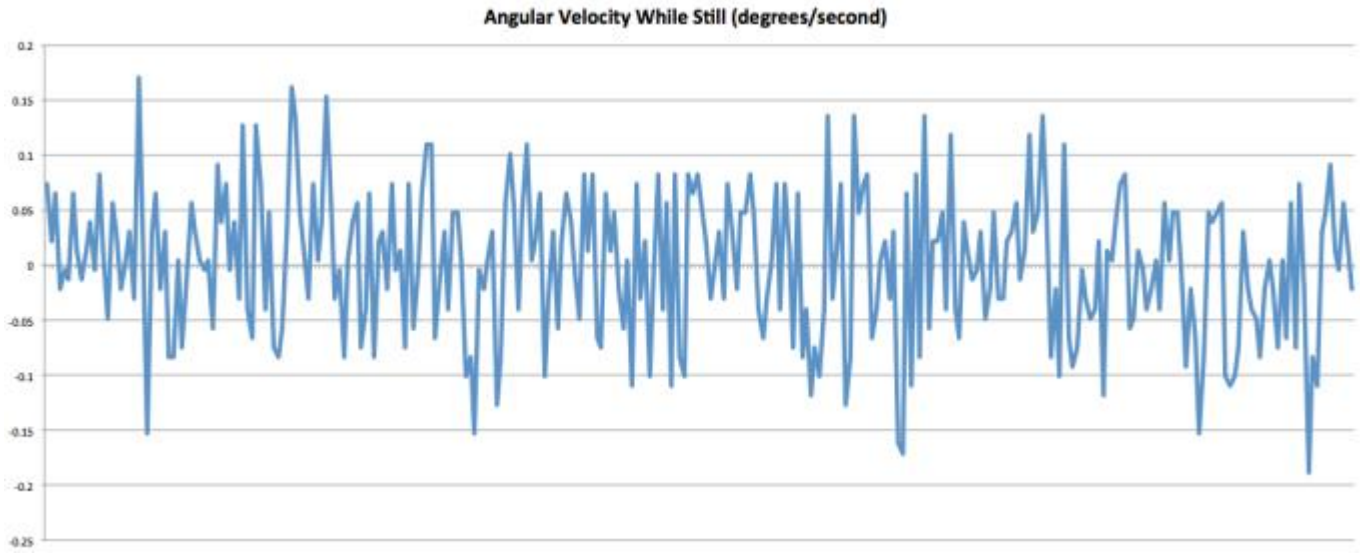
$\theta(t) = \theta(t - 1) + \omega * \Delta t$

In other words, accumulate the angle value by continuously taking the previous measurement and adding the change based on the angular velocity and the time difference between the last measurement and the current measurement. Theoretically, this can be done with the gyroscope measurement alone. However, gyroscopes have several sources of error: constant bias, bias stability and angle random walk.

Constant bias is the average output of the gyroscope (ω) when it is still. Since ω is integrated through the multiplication with Δt, this error can accumulate quite substantially. Before we begin using the robot, we need to measure the average of ω over a period of time and subtract this bias from every use of ω. This makes the determination of the tilt angle:

$\theta(t) = \theta(t - 1) + (\omega - bias) * \Delta t$

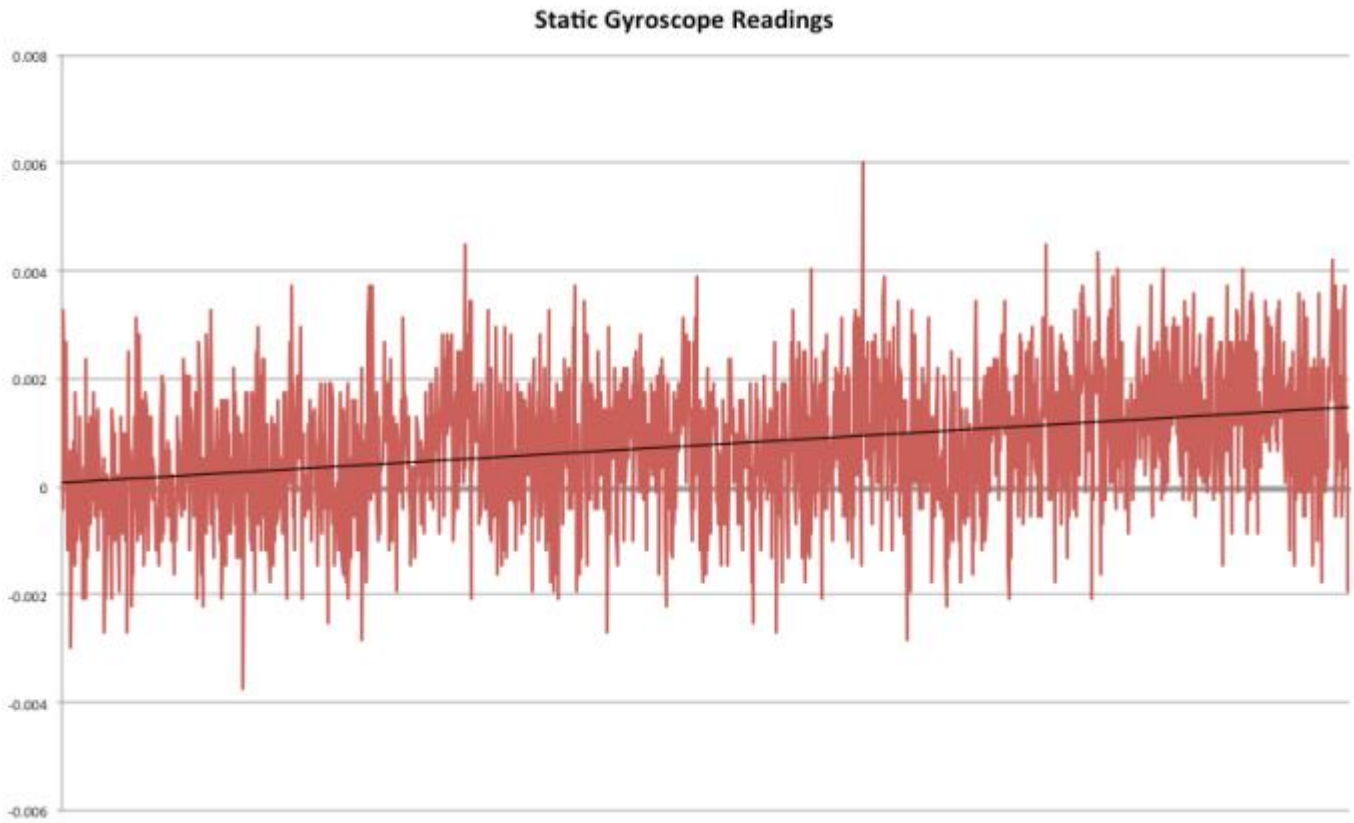
However, the constant bias is not really constant. The bias varies slowly over time and is typically modeled as white noise or a "random walk". This is measured by something called the "bias stability". While this value is typically on the gyroscope data sheet, I was unable to find a value for the ST Microelectronics LSM303DLHC. MEMS gyroscopes like this exhibit very high frequency noise that is caused by thermo-mechanical events internal to the chip. This cumulative white noise is typically called drift. I was curious about this noise level and recorded the following data during a 30 *second* period while the robot was perfectly still.



This is a 30 second sampling of the angular velocity reported by the gyroscope when the robot was perfectly still. Note that the constant bias has been incorporated and the signal is fairly symmetrical about zero. As we'll see later 30 seconds is too short of time period to draw any conclusions about bias stability or drift.

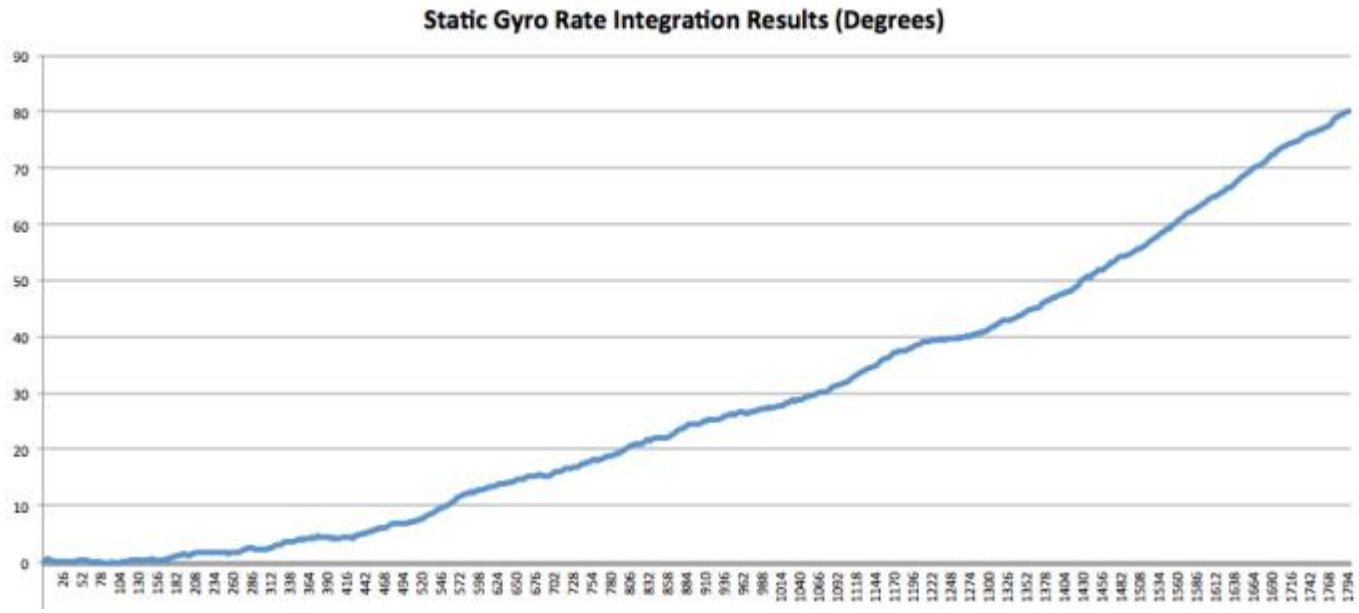
What is the effect of the gyroscope noise?

The effect of this noise can be experimentally determined. The *Angle Random Walk (ARW)* value can be thought of as the variation (or standard deviation) of the result of integrating the output of a stationary gyro over time due to the noise. I did some experimenting and found that the ARW for my ST Microelectronics LSM303DLHC is about 1.377 degrees per square root hour (with 100 millisecond intervals measured over a 30 *second* period). In other words, if the robot ran for an hour, the inherent gyroscope noise would contribute to a variation of ±1.377 degrees from the ideal vertical position. This sounded far too good to be true. In order to measure bias stability, I used a much longer measurement period of 30 *minutes*, measuring once per second. The results are below:



This graph shows the gyroscope readings (radians/second) of a perfectly still robot. The bias was adjusted prior to collecting the readings. The bias is clearly trending upwards with a drift of 0.0014 radians/second over the 30 minute measurement period.

After looking at the graph above, I thought that 0.0014 radians/second drift over 30 minutes didn't sound too bad. Then I realized that this is in fact very bad because each reading would be integrated and accumulated on top of the previous error. After integrating with the Δt and converting to degrees, the bias drift error became very obvious:



This graph depicts the accumulated angle of a perfectly still robot using only the gyroscope. The bias drift can become quite substantial. The y-axis is degrees and the x-axis is seconds.

So while the gyroscope readings have relatively small short-time errors, left unchecked the errors can become quite large over longer periods of time. While this experiment sampled the gyroscope once per second, a self-balancing robot is likely to check far more frequently (e.g. once every 10 to 15 milliseconds). At what point does this drift cause significant error? My next experiment was to measure a perfectly still robot at 15 millisecond intervals to see where the accumulated error reached 1 degree. It took 98.5 seconds.

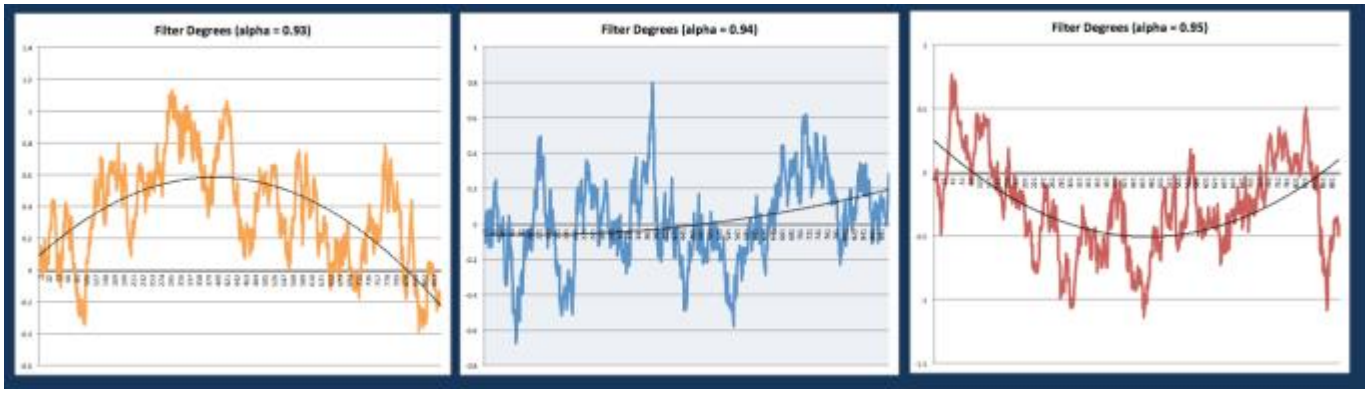
There is nothing that can be done to eliminate the drift effects on the gyroscope output. However, the effective tilt angle can be corrected against these effects through sensor fusion.

Sensor Fusion Confusion

We now know that the accelerometer can provide a point-in-time tilt angle and the gyroscope can provide an angular velocity that can be used to calculate a tilt angle change. The gyroscope will drift as demonstrated above and need a constant correction. Since the accelerometer can provide an instantaneous tilt angle, it could be used to correct the drift. But why not just use the accelerometer by itself? The accelerometer is subject to vibration from the motors and sudden stops and starts as the robot stays balanced – in other words it is too noisy by itself. When I first started my research on how to approach sensor fusion, I found that there were two schools of thought: Kalman filter and complementary filter. The clearest explanation of why a Kalman filter is overkill for a balancing robot is in Kerry D. Wong's write up. The complementary filter essentially combines the gyroscope and accelerometer readings to take advantage of their respective strengths and filters out their weaknesses. The values are combined through a weighting that heavily favors the gyroscope over the accelerometer through a single variable α. This approach makes the tilt angle calculation:

$$\theta(t) = \alpha * (\theta(t - 1) + (\omega - bias)* \Delta t) + (1 - \alpha)*Az/g$$

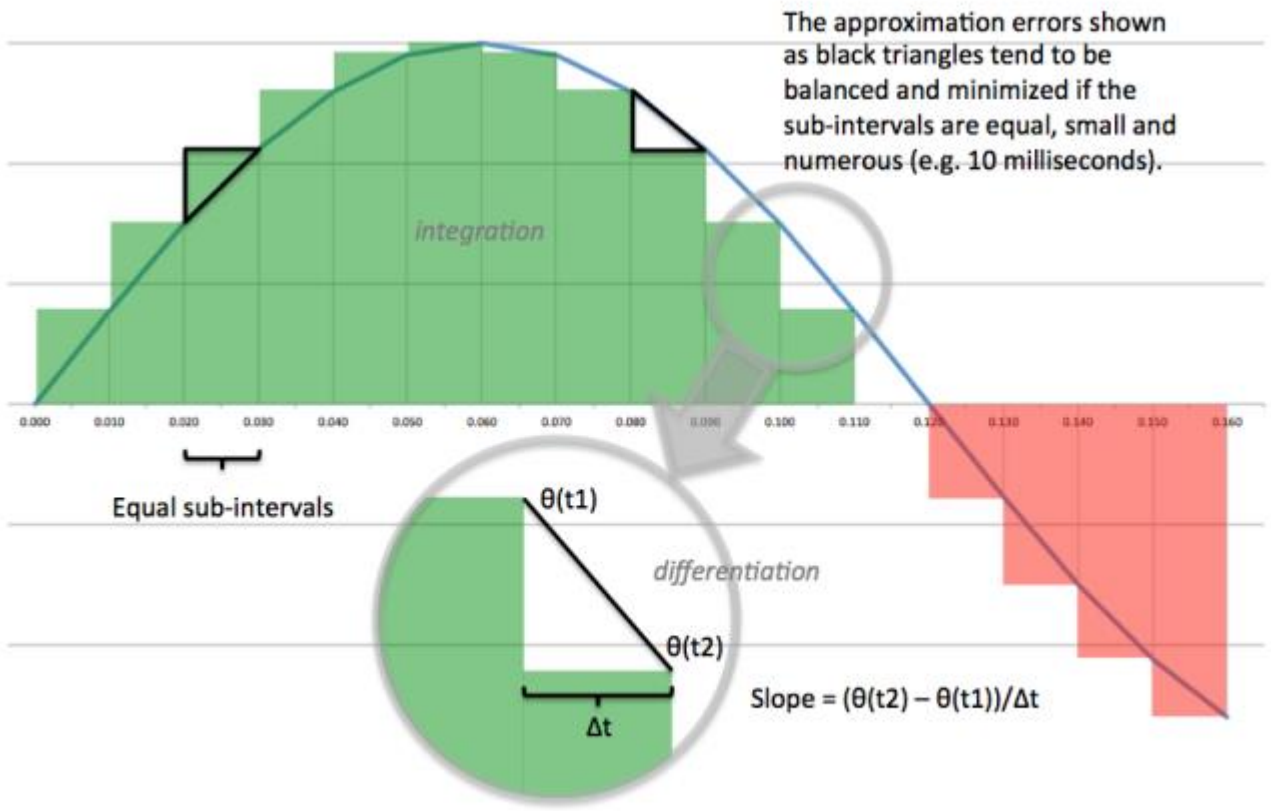
However, I have not found an explanation of how to tune the complementary filter variable α other than it should be greater than 0.9. In the absence of a mathematical foundation for tuning, I turned to experimentation. What value of α enables the accelerometer reading to correct the gyroscope drift? I iterated on several values and plotted a polynomial trend line on the filter output when the robot was perfectly vertical and still.



Here are three of the iterations to determine how best to correct the gyroscope drift by adjusting the alpha variable. All of these graphics show a 15 minute duration.

At least for my IMU, a value of $\alpha = 0.94$ nearly eliminated the gyroscope drift over a 15 minute measurement period. I deemed this value good enough.

Now that we have an effective means to measure a tilt angle, how do we apply that to the control of the robot? That is covered next in this posting.



Building an Arduino-based self-balancing robot – Part 2

Posted on May 10, 2015 by Mike Jacobs

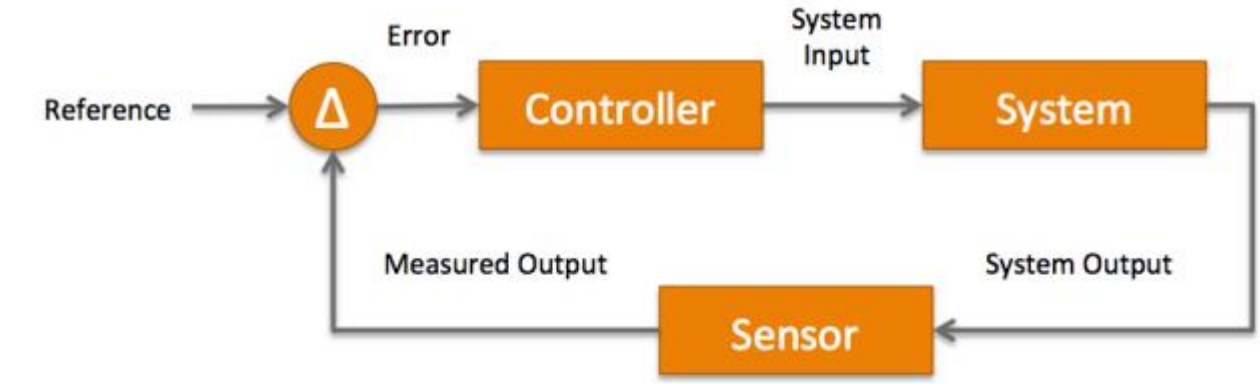
In my previous post, I explained how to design an inertial measurement unit or IMU for use with a self-balancing robot. The IMU is used to measure the current tilt angle of the robot. The idea is to use the tilt angle as a way to determine the direction the robot should

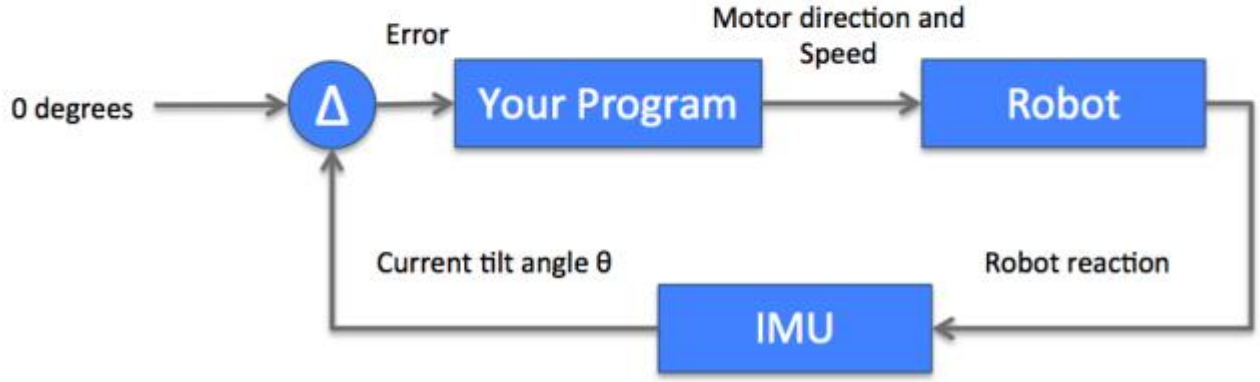
move to bring it back to a near vertical position. This problem is similar to the situation most of us have intuitively solved when balancing something like a broom on the palm of our hand. We move our hand so that it is under the center of mass – if the broom tilts left, we move the bottom to the left to catch up with the rest of the broom before it falls. We make many small adjustments like this to keep the broom balanced. These adjustments are based on our observation that the broom is beginning to tilt too much in one direction and we need to move our hand fast enough so that the broom doesn't fall. In this case, we may need to move our hand forward and backward as well as left and right.

The PID and the Pendulum

[Apologizes to Edgar Allen Poe]. The self-balancing robot balancing act is similar and somewhat simpler than balancing the broom in that we only need to move the robot forward and backward – but how to do this? This problem happens to be a classic control theory problem called an inverted pendulum. Most descriptions of this problem go to the trouble of mathematically describing the motion equations of the inverted pendulum before describing how to control it. This is mostly for the benefit of those with system simulation software and ideal parameters to create the perfect solution. The math is frighteningly complex involving signal processing techniques like transforming the time domain problem into the frequency domain through Laplace transforms and bringing stability to the system by moving the roots into the negative s-plane. Do you need to understand that? I don't think so – but if you do, that could help later. I haven't touched Laplace transforms in a very very long time and I didn't need to learn them again. So what do you need to know to control the robot? A little bit of control theory.

You have likely encountered control theory without even realizing it while filling a bath tub. You may start with both hot and cold water running while you monitor the water temperature in the tub. If the temperature is too cold, you might turn the cold water down or turn the hot water up. You monitor the temperature and make adjustments until the right water temperature and level is reached. What you've done is used the difference between the desired and actual water temperature as a guide to repeatedly adjust the incoming water temperature mix. Your hand is the *sensor* that provides feedback of the current temperature. The difference between the desired (or *reference*) and actual temperatures is called the *error*. You use the error signal to control adjustments to bring the temperature closer to the desired output – you are the *controller*.





The orange boxes depict a generic sensor feedback loop back to a controller that adjusts the system input based on the difference between the reference and the measured output. The blue boxes depict the control approach for the self-balancing robot.

You can see from the diagram above that "Your Program" is the controller and it uses the IMU output to decide how to move the robot. This type of feedback control has been extensively used for over one hundred years to control all sorts of machines. A typical approach to a controller is to use the Proportional, Integral, Derivative or PID design. This design combines the use of two or three of:

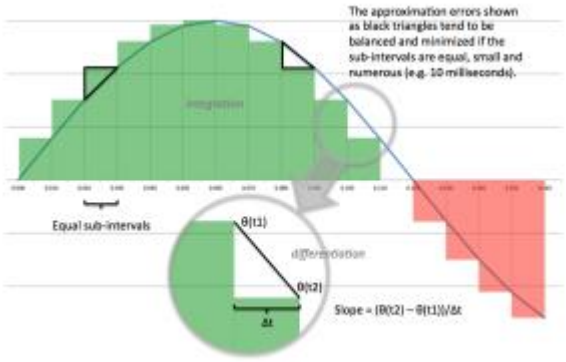
- A proportional adjustment (e.g. multiplier) of the current tilt angle. This adjustment by itself cannot result in a stable system and must be combined with one or both of the other adjustments.
- An adjustment based on the total accumulated error. Multiple adjustments to the robot should make the tilt angle approach zero degrees (e.g. vertical). The actual measurements will show that the adjustments may have fallen short or over shot the desired angle resulting in an error. This error is accumulated and should be accounted for by the controller. Because this is a sum over time, it is mathematically an integration and thus the name integral.
- An adjustment that considers the rate at which the error is changing so that the system can become stable more quickly. Because it is related to the rate of change of the error over time, it is mathematically a derivative of the error.
- Controllers that use only the Proportional and Integral adjustments are referred to as PI controllers. Those that only use the Proportional and Derivative adjustments are called PD controllers. A very informative and comically dated MIT lecture on YouTube provides a very good detailed mathematical foundation for control feedback

of an inverted pendulum. The lecture only uses a PD design, while I understand most systems rely on a PI design. This series uses all three adjustments just for fun. The PID controller output can be expressed mathematically as:

$$controller(t) = K_p\theta(t) + K_i\int_0^t \theta(\tau)d\tau + K_d\frac{d}{dt}\theta(t$$

- The PID controller output equation with the tilt angle (theta) substituted for the error.

- Where θ is the tilt angle. The tilt angle is considered the "error" since the reference is zero degrees so anything other than zero is considered an error. Each adjustment has a multiplier called a gain. The Kp, Ki, and Kd gains are for the proportional, integral and derivative adjustments respectively. All of these gains must be positive to achieve stability. There are other equivalent representations of this equation that we'll discuss when it comes to determining the values of each of the gains. As you can see there is a need to do integration and differentiation. We need to find ways to approximate these functions.

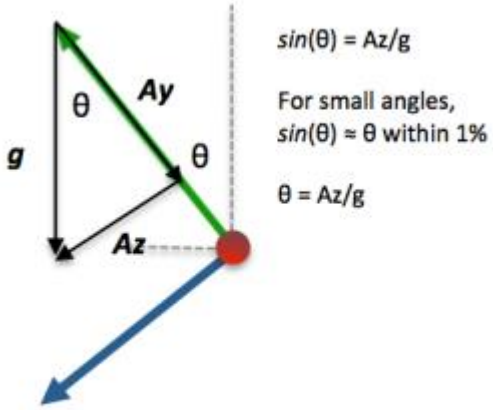


- Numeric integration can be accomplished through the rectangle method of summing the areas of many narrow rectangles having equal widths. Numeric differentiation can be done through a slope calculation of adjacent tilt angle measurements.

- Integration is the calculation of the area under a function. Fans of calculus also know that integration is the sum of infinitesimally wide rectangles as the width approaches zero. Your program can accomplish a close approximation of integration through numerical integration. This example (and all that I have seen elsewhere) specifically uses the rectangle method. Imagine a graph of the accumulated tilt angle on the y-axis versus time on the x-axis like the one to the right. The rectangle method sums the rectangles formed by a series of *equal sub-intervals* to calculate the area providing a nice approximation to integration. There is some error in the calculation as shown with the black triangles but is minimized through the use of equal sub-intervals. This approach led me to use a timer interrupt to ensure the time intervals were as equal as possible. Many other examples you might find tend to use the main Arduino loop for this evaluation and I suspect additional error may be introduced in the numeric integration as a result.
- A derivative is the slope of a function and is accomplished through differentiation. Your program can implement a close approximation of a derivative through the use of numeric differentiation. This example uses the finite difference approach where the slope of a secant line between two adjacent tilt angle

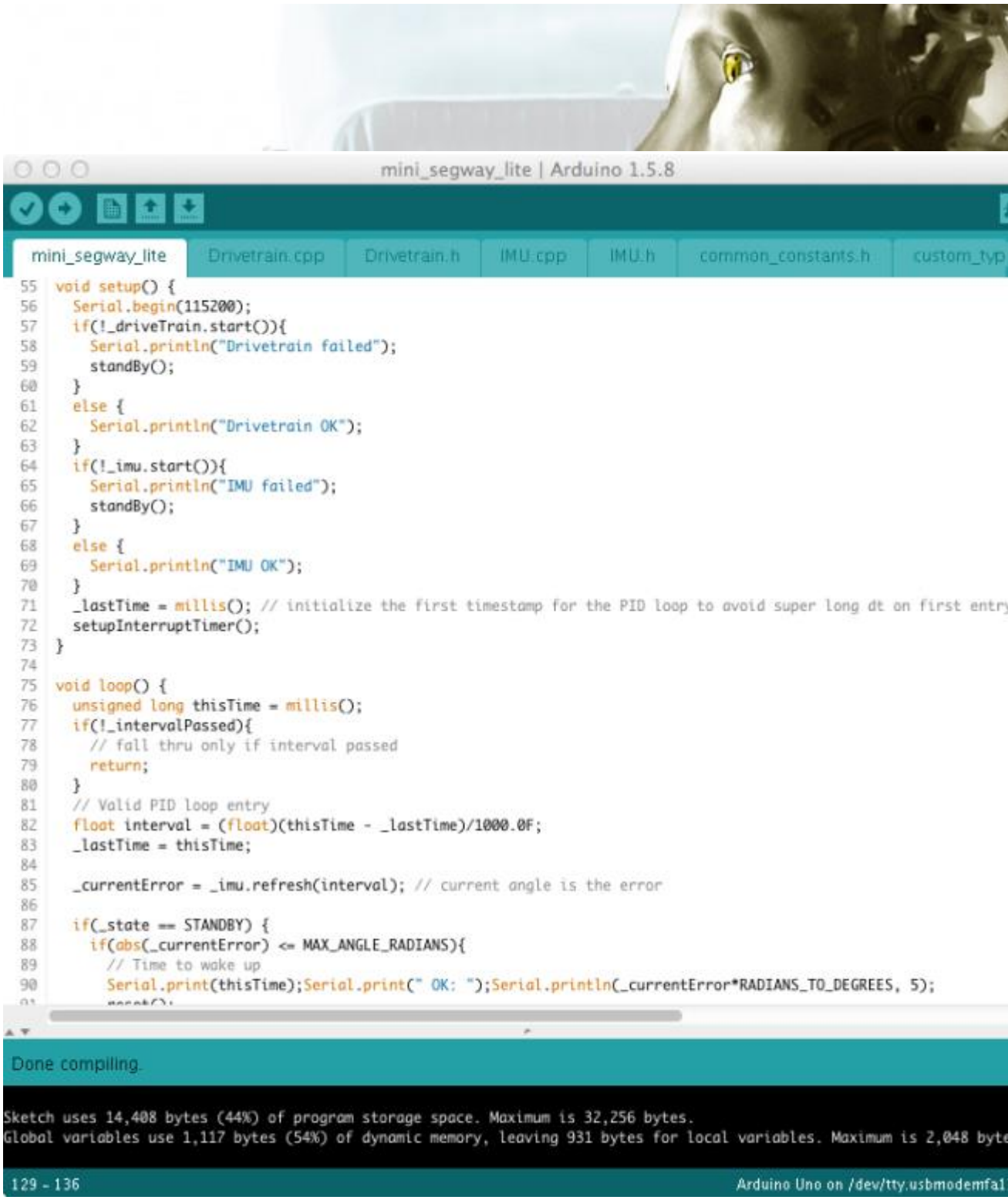
measurements is used. This is sufficiently accurate for our use provided the sub-interval time is very short.

- So now that we have the controller output foundations established, how should that be used? Don't worry about the gain values yet as we'll cover them in detail when we finally get to some code.
- **May The Torque Be With You**
- Ultimately we need to drive the motors of our robot and the details are quite implementation specific. Your program needs to determine a direction and speed to keep the robot balanced. The way to do that depends on the weight of the robot, the torque of the motors you've selected and how you've attached the motors to your Arduino. I use the Adafruit Motor/Stepper/Servo Shield because it's convenient, powerful and relatively inexpensive at US$20. You might be able to put together a cheaper alternative with some extra work. A few things I like about this shield is that is has a dedicated PWM chip, connects over just two I2C pins and has a nice library to access it. The direction is set with a dedicated function call using FORWARD or BACKWARD constants. The speed is set by using another function call passing a value ranging from 0 to 255 – the realized speed depends on the motor, the current power supply level and the load.



Tilting forward by an angle of θ from the side

- As you may recall from my previous post, the IMU arrangement makes a positive rotation about the x-axis correspond to tilting forward. Your design may use a different convention for the axis or direction. As the tilt angle becomes positive, the robot needs to move forward to stay balanced. Looking at the control equation, we can see that a positive and increasing tilt angle θ produces a positive control output. The sign of the control value can be used to determine the required motor direction.
- The absolute value of the control should not be directly used as the speed or throttle of the motor because that value must be within the range of 0 to 255. With some care of data types and ranges, the control value can be mapped into the proper throttle value through the use of the Arduino 'constrain' and 'map' functions.
- This completes the background and foundational concepts to design a self-balancing robot. Next we will *finally* get to some code to make these concepts real and then determine the PID gain values to make it work.

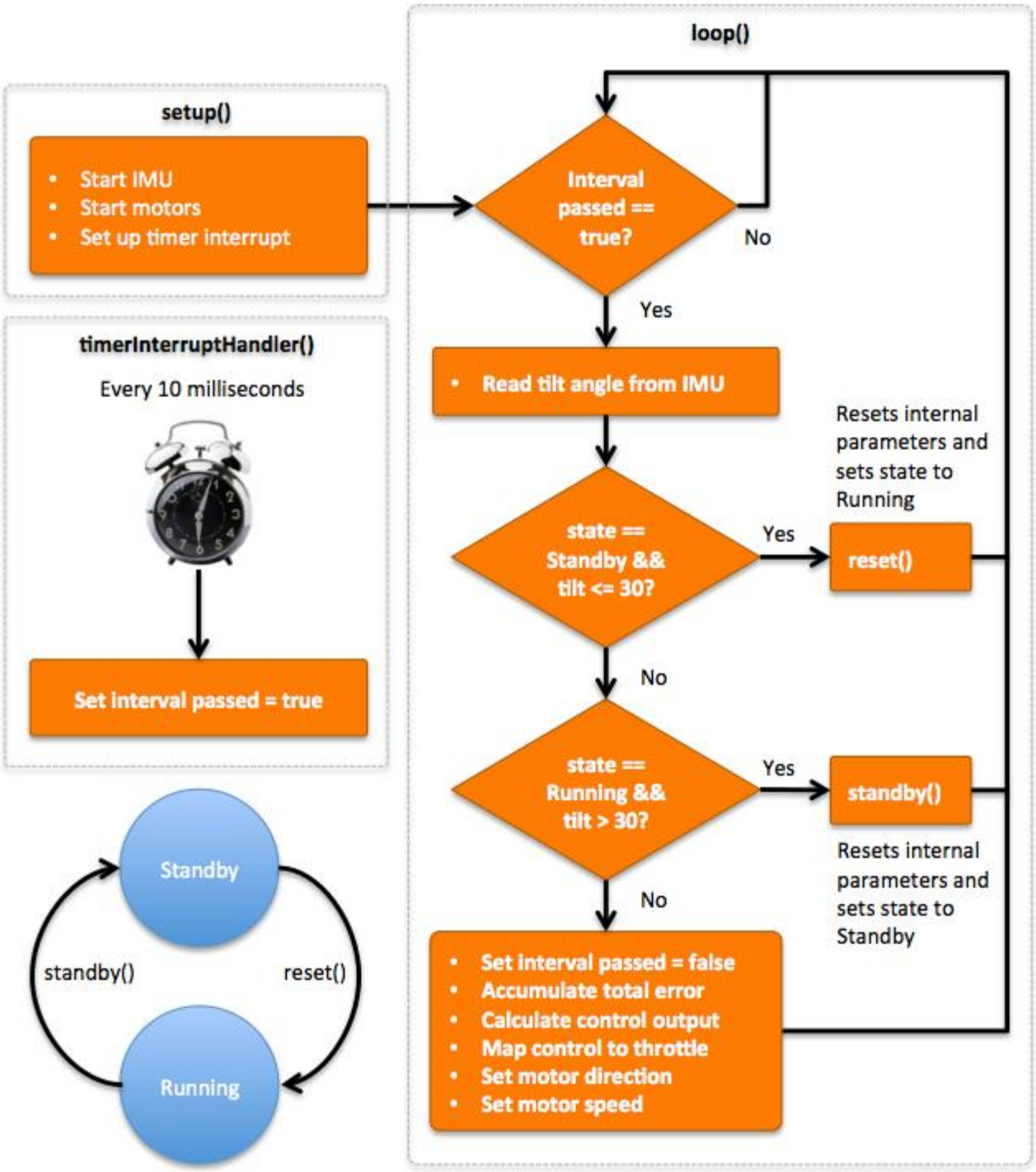
- 

Building an Arduino-based self-balancing robot – Part 3

- Posted on May 25, 2015 by Mike Jacobs
- This is the third installment of my series describing the details of building your own self-balancing robot. In the first post, we covered the IMU to measure the tilt angle of

the robot. The second post explained the foundations for controlling the robot to keep it balanced. In this post, we will finally get to the code to put it all together.

- [***Note**: This series is intended to help you design and code your own self-balancing robot. I include lots of code in the articles, but I won't have a download of all the code. Why? There are too many people who expect to download, compile and run code and have it work unchanged for their situation and then expect help from the author when it doesn't work for them. This series will include everything you need and more. All of the code snippets can be copied and pasted and adjusted to your robot design – isn't that what you'd do with a download anyway? The tips and explanations in these articles should hopefully help you get yours working.*]
- **Program Design**
- In case you haven't programmed an Arduino, here is a very brief overview. An Arduino program is called a Sketch. Your sketch is written in a programming language called Wiring which is based on Processing and looks like C++. In fact, you can and we will mix C++ into our robot code. You can develop your sketch in any IDE, but there is an Arduino-specific IDE that includes most of what you need. Additional libraries can be added to your project to extend the capabilities of the core framework. Arduino boards like the Arduino Uno we are using can be expanded through add-on boards called shields. The motor controller we will be using is an example. Your program is compiled on your desktop and transferred to your Arduino board through a serial link (typically a USB cable).
- Your sketch has two functions: setup() and loop(). As the name implies, the setup() function is called once with the intention of getting things initialized and setup. The loop() function then runs continuously.



- This is the overall design of the code for the self-balancing robot. It consists of the setup(), timerInterruptHandler() and loop() functions. The real work is done in the loop() function.

- *It's About Time*
- [*After writing that heading I was reminded of an old TV show that had an ear worm of a theme song. You're welcome.*]
- One important aspect of an accurate IMU discussed in my first post, is the use of a consistent time interval. There are a number of hardware timers available on the Arduino boards that would provide just the thing we need for a consistent time interval. These timers allow your code to be called on a regular basis based on a hardware timer and an interrupt that calls your function. Since we are using an Uno

board, there are three timers (timer0, timer1 and timer2) available two of which are 8 bits and one that is 16 bits. The number of bits determines the resolution of the timer and along with a scaling factor, how often your code is called. The typical Arduino board runs at 16MHz (there are some exceptions!) and the timers are incremented on every cycle or every 63 nanoseconds. When the value of the timer reaches a limit you specify, it wraps around to zero – this is called an overflow. The overflow creates an interrupt and calls your interrupt handler.

- I elected to have the control loop run every 10 milliseconds or 100 times per second (a frequency of 100Hz). This was an arbitrary choice on my part and you could use something slower. If you think you need something faster, be sure to monitor how long your code takes to run within your loop.
- Which timer should we use? Some libraries and functions use a specific timer, so be aware of the timers used by the libraries you use. For example, delay() and millis() use timer0 so I avoided using it. If you're using the Servo library (which this project does not use) then you'll want to avoid using timer1. How should we choose between the remaining two timers and how should they be configured? I found this detailed explanation very informative and I will summarize the concepts here.
- You can specify how often the timer gets updated by specifying a divisor called the prescaler. The prescaler value can be one of 1, 8, 64, 256 or 1024. The Arduino will essentially divide the 16 MHz frequency by the prescaler to create a slower frequency. Instead of updating every 63 nanoseconds running full tilt, a value of 1024 would provide a counter frequency of 15.625 kHz or an update every 64 microseconds. Since we want a 10 millisecond interval, we would want the counter to count to 156.25 (10 milliseconds divided by 64 microseconds) or 156 ticks for an update cycle of 9.984 milliseconds. An 8 bit timer can hold a value of up to 255 so either an 8 bit or 16 bit timer would work. See the table below for the other options for a 100 Hz interval.

	Prescale = 1		Prescale = 8		Prescale = 64		Prescale = 256		Prescale = 1024	
# Bits	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
8	62.5 kHz	16.0 MHz	7.81 kHz	2.0 MHz	976.56 Hz	250.0 kHz	244.14 Hz	62.5 kHz	61.04 Hz	15.63 k
16	244.14 Hz	16.0 MHz	30.52 Hz	2.0 MHz	3.81 Hz	250.0 kHz	0.95 Hz	62.5 kHz	0.24 Hz	15.63 k

- This table shows the minimum and maximum timer update frequencies for 8 bit and 16 bit timers based on the prescaler value used. For example, a 16 bit timer with a prescaler divisor of 64 can support updates between 3.81 Hz through 250 kHz.

- Now you could get into the low level register settings (and I may do that as an optimization later) but I decided to use the Timer1 Library for convenience. This library lets you specify the period of the update (e.g. 10 milliseconds) and the function to call when that time has expired.
- *Let's Get The Party Started!*
- So let's look at the code for setup().
- `Drivetrain _driveTrain;`
- `IMU _imu;`
- `State _state = STANDBY;`
- `volatile bool _intervalPassed = false;  // whether or not the PID feedback time interval has passed`
- `unsigned long _lastTime = 0; // The last time the PID loop was evaluated`
-
- `void setup() {`

```
    Serial.begin(115200);
    if(!_driveTrain.start()){
       Serial.println("Drivetrain failed");
       standBy();
    }
    else {
       Serial.println("Drivetrain OK");
    }
    if(!_imu.start()){
       Serial.println("IMU failed");
       standBy();
    }
    else {
       Serial.println("IMU OK");
    }
    _lastTime = millis();
    setupInterruptTimer();
}
```

- The first you might notice is the use of a few global variables (starting with an underscore) for the drive train, IMU, state, interval passed and last time. The drive train and IMU are C++ abstractions for the motor controller shield and 9 DOF IMU breakout board. You might have noticed that there are is no call to new or malloc for the drive train or the IMU. This is one difference between Wiring and C++: Wiring allocates your C++ objects for you. The state variable is for the very simple state machine that toggles between RUNNING and STANDBY (an enum not shown here). When the robot tips over, we want the motors to shut down and wait until it is stood up again (STANDBY). When the robot is upright, it will start the motors again (RUNNING). The intervalPassed variable is a flag that indicates that our 10 millisecond interval has passed. Notice the volatile declaration? That's a directive to the compiler not to use internal registers to cache the value for optimization because the loop() and the interruptHandler() both change the value and we want the most up to date value in memory when the loop() procedure reads it. Have a look at the relevant interrupt code below.

```
void setupInterruptTimer(){
   Timer1.initialize(TIMER_INTERVAL_MICROS);
   Timer1.attachInterrupt(timerInterruptHandler);
}

void timerInterruptHandler(){
   _intervalPassed = true;
}
```

- You can see that during setup(), the setupInterruptTimer() procedure is called. The Timer1 library is used to set up the interval (a static variable not shown here) and register the function to call when the interval is reached. The timerInterruptHandler() is called when the timer expires and all it does is set the intervalPassed global to true. The loop() sets this to false later. You always want to keep the code within a interrupt handler short and sweet.

Making it Go

- This section covers a specific motor controller, namely the Adafruit Motor/Stepper/Servo Shield for Ardunio V2 kit V2.3. The class I used to encapsulate

this choice can be used as a starting point for the motor controller that you are using. To start with, let's have a look at the C++ header file snippet:

```
#include
#include
#define LEFT_MOTOR 4
#define RIGHT_MOTOR 1

class Drivetrain {
  public:
    Drivetrain(void);
    bool start();
    bool selfTest(uint8_t);
    void forward(uint8_t);
    void reverse(uint8_t);
    void release();
    void setSpeed(uint8_t);
  private:
    Adafruit_MotorShield _motorShield;
    Adafruit_DCMotor *_leftMotor;
    Adafruit_DCMotor *_rightMotor;
};
```

In retrospect, I probably should have named this Powertrain because it includes the motors but oh well. The interface consists of functions to initialize and control the direction and speed of the motors. The implementation of this C++ class looks like this:

```
#include "Arduino.h"
#include "Drivetrain.h"

Drivetrain::Drivetrain(){
  _motorShield = Adafruit_MotorShield();
  _leftMotor = _motorShield.getMotor(LEFT_MOTOR);
  _rightMotor = _motorShield.getMotor(RIGHT_MOTOR);
}

bool Drivetrain::start(){
  _motorShield.begin(); // using default PWM frequency
  delay(1000); // wait for motor shield to start up
  return true;
}

bool Drivetrain::selfTest(uint8_t aSpeed){
  delay(500);
  forward(aSpeed);
  delay(500);
  release();
  reverse(aSpeed);
  delay(500);
  release();
  return true;
}

void Drivetrain::forward(uint8_t aSpeed){
  setSpeed(aSpeed);
```

```
  _leftMotor->run(FORWARD);
  _rightMotor->run(FORWARD);
}

void Drivetrain::reverse(uint8_t aSpeed){
  setSpeed(aSpeed);
  _leftMotor->run(BACKWARD);
  _rightMotor->run(BACKWARD);
}

void Drivetrain::release(){
  _leftMotor->run(RELEASE);
  _rightMotor->run(RELEASE);
}

void Drivetrain::setSpeed(uint8_t aSpeed){
  _leftMotor->setSpeed(aSpeed);
  _rightMotor->setSpeed(aSpeed);
}
```

If you're interested in the library used to interface with this shield have a look at this technical documentation.

Measuring the Tilt

This section covers a specific IMU breakout board, namely the Adafruit 9-DOF IMU Breakout – L3GD20H + LSM303. The class I used to encapsulate this choice can be used as a starting point for your IMU. To start with, let's have a look at the C++ header file snippet:

```
#include
#include
#include

typedef enum {
  APPROXIMATE = 0,
  EXACT = 1
} IMU_Mode_t;

class IMU {
  public:
    IMU(void);
    bool start();
    bool selfTest(int iterations = 1000, int delayValue = 8);
    float refresh(float);
    bool reset();
    void calibrate();
    void measureARW();
    void testFilter();
    void findFilterTau();
    void logData(bool);
  private:
    Adafruit_LSM303_Accel_Unified _accelerometer;
    Adafruit_L3GD20_Unified _gyroscope;
    float _angle = 0.0F;
    IMU_Mode_t _calculationMode = EXACT;
    float resetAccelerometer();
```

```
    void calibrateGyroscope();
    void calibrateAccelerometer();
    bool _loggingData = false;
};
```

This article will cover only the essential functions leaving a few to be discussed when it comes time to tune our robot. One of the first things you'll want to do is calibrate the accelerometer and gyroscope based on your implementation. I set my robot on a known level surface without the wheels and ran the calibration routines to establish the bias offsets:

```
float IMU_CALIBRATION_ANGLE = -0.09799358F; // DO NOT USE THESE VALUES FOR
YOUR ROBOT
float IMU_CALIBRATION_OMEGA = -0.02030516F;

const int CALIBRATION_ITERATIONS = 100;
const int RESET_ITERATIONS = 50;
void IMU::calibrate(){
  calibrateAccelerometer();
  calibrateGyroscope();
}

void IMU::calibrateAccelerometer(){
  Serial.print("Accel");
  IMU_CALIBRATION_ANGLE = 0.0F;
  float readings = 0.0F;
  for(int i = 0; i < CALIBRATION_ITERATIONS; i++){
    readings = readings + resetAccelerometer();
    if(i % 5 == 0) {
      Serial.print(".");
    }
    delay(100);
  }
  IMU_CALIBRATION_ANGLE = readings/((float)CALIBRATION_ITERATIONS);
  Serial.println(" ");Serial.print("IMU_CALIBRATION_ANGLE =
");Serial.print(IMU_CALIBRATION_ANGLE, 8);Serial.println("F;");
}

float IMU::resetAccelerometer(){
  sensors_event_t accelerometerEventData;
  float reading = 0.0F;
  for(int i = 0; i < RESET_ITERATIONS; i++){
    _accelerometer.getEventGs(&accelerometerEventData);
    float thisReading = constrain(accelerometerEventData.acceleration.z -
IMU_CALIBRATION_ANGLE,-1.0F, 1.0F);
    reading = reading + thisReading;
    delay(10);
  }
  reading = reading/((float)RESET_ITERATIONS);
  _angle = (-1.0F)*asin(reading);
  return _angle;
}

void IMU::calibrateGyroscope(){
  Serial.print("Gyro");
  IMU_CALIBRATION_OMEGA = 0.0F;
  sensors_event_t gyroEventData;
  float readings = 0.0F;
  for(int i = 0; i < CALIBRATION_ITERATIONS; i++){
```

```
    _gyroscope.getEvent(&gyroEventData); // internally converts degrees/sec
to radians/sec
    readings = readings + gyroEventData.gyro.x;
    if(i % 5 == 0) {
      Serial.print(".");
    }
    delay(100);
  }
  IMU_CALIBRATION_OMEGA = readings/((float)CALIBRATION_ITERATIONS);
  Serial.println(" ");Serial.print("IMU_CALIBRATION_OMEGA =
");Serial.print(IMU_CALIBRATION_OMEGA, 8);Serial.println("F;");
}
```

The calibration routines simply make the readings 100 times and averages them. Since the robot is not moving and is presumably level, any reported values are the offsets that need to be accounted for when the robot is moving. You may recall from my first article that the tilt angle equation is:

$$\theta(t) = \alpha * (\theta(t-1) + (\omega - bias) * \Delta t) + (1 - \alpha) * Az/g$$

where α is the high pass filter (HPF) coefficient for the gyroscope integration and (1-α) is the low pass filter (LPF) coefficient for the accelerometer-based small angle estimate. Here's how this equation translates into code:

```
const float HPF = 0.94f;  // Determined experimentally to reduce gyroscope
drift.  See Part 1
const float LPF = 1.0f - HPF;

float IMU::refresh(float dt){ // dt is in seconds
  sensors_event_t accelerometerEventData;
  sensors_event_t gyroEventData;
  /*
    Update the gyro and accel data from the device - the library gets all
three dimensions
    Accelerometer: Leaning forward reports a negative value - switch it
around
    Gyroscope: Leaning forward reports a positive value
  */
  /*
    The call below uses an updated version of the library to use native 'g'
units
    because sin(theta) = Az/g.  The native result is in Gs, however
    the Adafruit library multiplies this by g.  We would ordinarily
    end up dividing the result by g, so eliminate those steps by
    keeping the units in Gs.
  */
  _accelerometer.getEventGs(&accelerometerEventData);
  _gyroscope.getEvent(&gyroEventData); // internally converts degrees/sec
to radians/sec
  float gyroRate = gyroEventData.gyro.x - IMU_CALIBRATION_OMEGA;
  float gyroRotation = gyroRate * dt;

  float accAngle = 0.0F;
  float accReading = constrain(accelerometerEventData.acceleration.z +
IMU_CALIBRATION_ANGLE,-1.0F, 1.0F); // contrained # of Gs
  /*
    With this hardware configuration, falling forward results in a negative
acceleration due to gravity
```

```
    and a positive angular velocity about the x-axis.  Align the readings
by multiplying the acceleration
    by -1.0.
  */
  if(_calculationMode == EXACT){
    accAngle = (-1.0F)*asin(accReading);
  }
  else {
    /* Approximate Mode
      Because this value is a factor of g's, the value can be used
      to approximate Sin(theta).  At small angles (+/- 30 degrees),
      this value can be used to closely estimate the angle of rotation
(radians).
    */
    accAngle = (-1.0F)*accReading;
  }
  /*
      Use a Complementary filter to determine the angle of rotation by
combining
      the measurements of the accelerometer and gyroscope.
  */
  _angle = HPF * (_angle + gyroRotation) + LPF * accAngle;

  // Determine if a mode change is needed
  if(_angle <= MAX_SMALL_ANGLE){
    _calculationMode = APPROXIMATE;
  }
  else {
    _calculationMode = EXACT;
  }
  return _angle;
}
```

Here is where you need to really understand what your accelerometer library returns. The Adafruit breakout board uses the ST Microelectronics LSM303DLHC. When polled, the accelerometer chip returns the number of G's for each axis. The Adafruit library uses the accelerometer sensitivity settings (essentially how many thousandths of a G per bit in the A/D conversion) to arrive at the number of G's. The library then multiplies this by 9.8 meters/sec² and returns the values. Returning values with units of meters/sec² makes sense for the library, but not for our use. The equation above (Az/g) would have us take that value and divide by 9.8 meters/sec² essentially undoing part of what the library just did. To avoid any rounding issues, I extended the Adafruit library to return the number G's instead. You'll also notice the IMU will switch between EXACT mode using the inverse sine (asin) function and APPROXIMATE mode when the angle is ±30 degrees.

Feeling Loopy

Now that we know how to measure the the tilt angle and make the robot move back and forth, it's time to implement the loop() function. The Arduino framework automatically calls this function repeatably within a loop that runs forever. If you have a look at the design diagram, you'll see that the function first checks to see if our interrupt driven interval has passed. If it has passed, then we read the angle from the IMU. The next set of decisions depends on the current angle reported by the IMU. If the robot is in STANDBY mode and the angle is now ±30 degrees, it switches over to RUNNING state. If it's in RUNNING mode and the angle is outside this angle range, it shuts down the motors and switches to STANDBY. Once in

RUNNING mode with a reported current angle of ±30 degrees, PID logic is invoked. Here is how the function looks so far:

```
const long MIN_THROTTLE = 50;  // high enough to move the robot when tilted
a few degrees
const long MAX_THROTTLE = 255;
// These values are determined experimentally - to be covered in the next
article
const float Kp = 0.000160F;
const float Ki = 0.0060F;
const float Kd = 0.00000100F;
const float SCALER = 1000000.0f;
long MAX_CONTROL = (long)(Kp*SCALER*MAX_ANGLE_RADIANS_CONTROL +
Ki*SCALER*TIMER_INTERVAL_SECONDS*MAX_ERROR_RADIANS +
(Kd*SCALER*MAX_DELTA_ERROR_RADIANS/TIMER_INTERVAL_SECONDS));

void loop() {
  unsigned long thisTime = millis();
  if(!_intervalPassed){
    // fall thru only if interval passed
    return;
  }
  // Valid PID loop entry
  float interval = (float)(thisTime - _lastTime)/1000.0F;
  _lastTime = thisTime;

  _currentError = _imu.refresh(interval); // current angle is the error

  if(_state == STANDBY) {
    if(abs(_currentError) <= MAX_ANGLE_RADIANS){       // Time to wake
up       Serial.print(thisTime);Serial.print(" OK:
");Serial.println(_currentError*RADIANS_TO_DEGREES,
5);       reset();     }     return;   }    else if(_state == RUNNING)
{     if(abs(_currentError) > MAX_ANGLE_RADIANS_CONTROL){
      // There is no hope of balancing. Cut the motors
      _driveTrain.release();
      standBy();
      return;
    }
  }
```

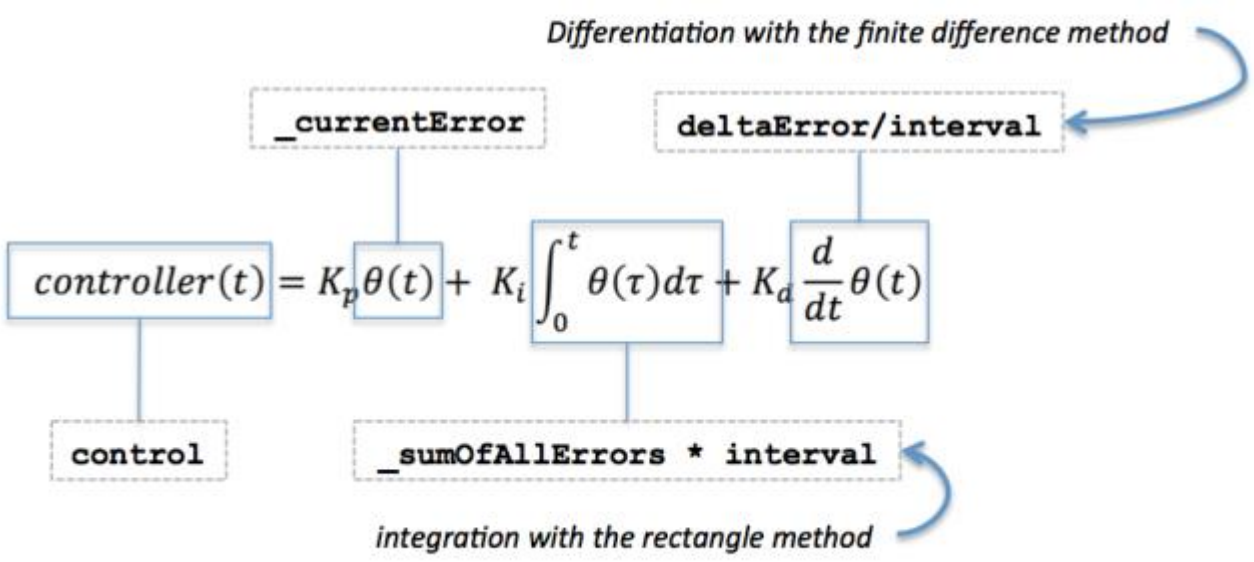
You may recall from [my second article describing the PID controller](), the equation for the controller feedback is:

$$controller(t) = K_p\theta(t) + K_i \int_0^t \theta(\tau)d\tau + K_d \frac{d}{dt}\theta(t)$$

The values of Kp, Ki and Kd appear to be “magic numbers” in the code above. There were determined experimentally and depend on many aspects of your robot so these values will not work for yours. The determination of these values will be covered in the next article as it’s one of the hardest parts of getting your robot to work. For now, we will go over how these values are used.

Before we go over the controller code, here is a map of the original equation to the associated code.

```
float interval = (float)(thisTime - _lastTime)/1000.0F;
_currentError = _imu.refresh(interval);
_sumOfAllErrors = _sumOfAllErrors + _currentError;
float deltaError = _currentError - _previousError;
```



This is how the controller equation is mapped to the variables and code in the PID logic section of the loop() function.

The actual code for the controller and throttle determination is shown below (a continuation of the loop() function).

```
  if((_state == RUNNING) && (abs(_currentError) <= MAX_ANGLE_RADIANS))
{     _intervalPassed = false; // reset for next time        // PID control
feedback     _sumOfAllErrors = constrain(_sumOfAllErrors + _currentError, -
MAX_ERROR_RADIANS, MAX_ERROR_RADIANS);     float deltaError = _currentError
- _previousError;       long control = (long)((SCALER*Kp*_currentError) +
(Ki*SCALER*interval*_sumOfAllErrors) +
(Kd*SCALER*deltaError/interval));         long throttle =
constrain(map(abs(control), 0L, MAX_CONTROL, MIN_THROTTLE, MAX_THROTTLE),
MIN_THROTTLE, MAX_THROTTLE);          if(control > 0){
      _driveTrain.forward((uint8_t)throttle);
    }
    else{
      if(control < 0) {
        _driveTrain.reverse((uint8_t)throttle);
      }
      else {
        _driveTrain.release();
      }
    }
    _previousError = _currentError;
  }
```

```
} // end of loop

void reset(){
  if(_state == STANDBY){
    _currentError = 0.0f;
    _previousError = 0.0f;
    _sumOfAllErrors = 0.0f;
    delay(2500);
    _lastTime = millis();
    _imu.reset();
    _state = RUNNING;
  }
}
void standBy() {
  if(_state == RUNNING){
    delay(1000);  // Let the IMU catch up to avoid false positives on the
control angle envelope
    Serial.println("** Waiting for near vertical **");
    _currentError = 0.0f;
    _previousError = 0.0f;
    _sumOfAllErrors = 0.0f;
    _state = STANDBY;
  }
}
```

The calculation of the control variable includes a SCALER that is multiplied with each of the PID contributions. This is done to ensure that each contribution is large enough to make a difference across the spectrum of possible throttle values. This was found through experimentation and frustration during the tuning phase. The SCALER could be incorporated into the Kp, Ki and Kp values to save the math time. The sign of the control value determines the direction the robot needs to take. The absolute value of the control variable is mapped to a throttle range by using the map() function. This function does not guarantee that the result is within the supplied bounds so the constrain() function is used to ensure that the throttle is within the right range. The sign of the control variable and the resulting throttle is used to move the robot in the right direction with the right speed. In case the robot has achieved perfect balance, there is code to release the motors – probably only for a very short time.

Next we will tune the PID coefficients to make the robot balance itself.