Arduino PID Research

Table of Contents

PID Theory

AVR221: Discrete PID controller

In Figure 1 a schematic of a system with a PID controller is shown. The PID controller compares the measured process value y with a reference setpoint value, y0. The difference or error, e, is then processed to calculate a new process input, u. This input will try to adjust the measured process value back to the desired setpoint.

The alternative to a closed loop control scheme such as the PID controller is an open loop controller. Open loop control (no feedback) is in many cases not satisfactory, and is often impossible due to the system properties. By adding feedback from the system output, performance can be improved.

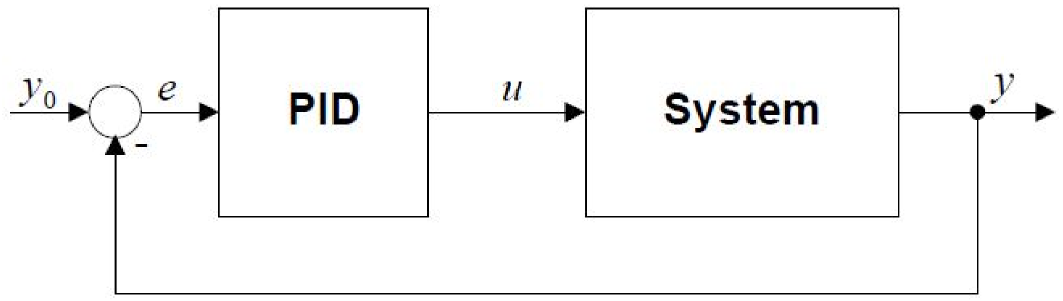

Figure 1 Closed Loop System with PID controller

Unlike a simple *proportional* control algorithm, the PID controller is capable of manipulating the process inputs based on the *history* and rate of change of the signal. This gives a more accurate and stable control method.

The basic idea is that the controller reads the system state by a sensor. Then it subtracts the measurement from a desired reference to generate the error value. The error will be managed in three ways, to...

1. **handle the present**, through the **proportional** term,
2. **recover from the past**, using the **integral** term,
3. **anticipate the future**, through the **derivative** term.

Figure 2 shows the PID controller schematics, where Kp, Ti, and Td denote the time constants of the proportional, integral, and derivative terms respectively.

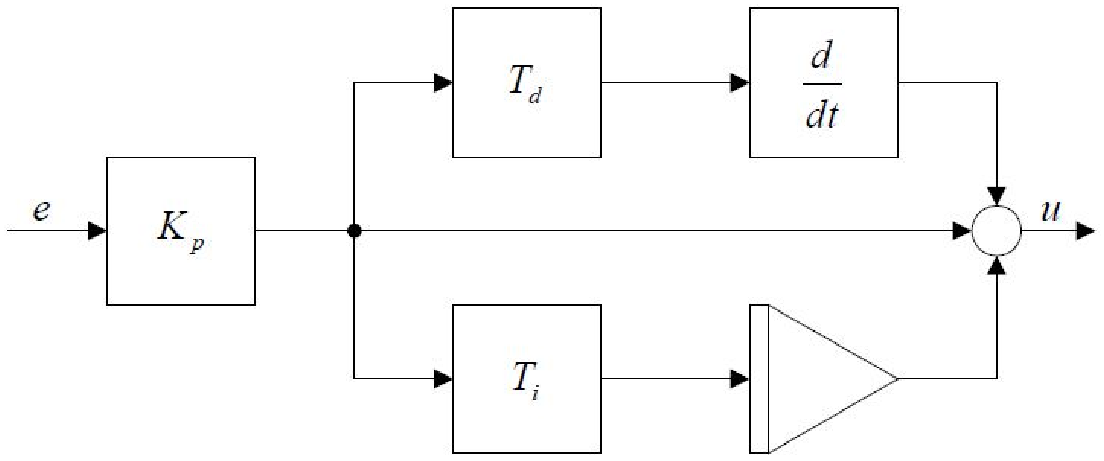

Figure 2 PID controller schematic

Click here to continue the article...

Read this article to see why there is always a **stationary error** $u(t)/K_p$ with a P only controller.

PID Control Code Examples

In this section I am going to look at three control examples, with a balancing robot example hopefully comming soon.

1. Bare Bones (PID) Coffee Controller
2. AeroQuad
3. Arduino PID library with accompanying Tutorial "Improving the Beginner's PID: Direction" by Brett Beauregard
4. RepRap Extruder Nozzle Temperature Controller
5. MIT Lab 4: Motor Control introduces the control of DC motors using the Arduino and Adafruit motor shield. A PID controller is demonstrated using the *Mathworks* SISO Design Tools GUI with accompanying Mathworks PID tutorial "Designing PID Controllers."
6. Balancing Robot

The first three control examples are presented in order of the complexity of the PID controller implementation. The coffee controller is a single PID, which can be documented in a single Arduino PDE file. The AeroQuad PID software is a modified version of the BBCC: Bare Bones (PID) Coffee Controller with the ability to control multiple control loops. The RepRap uses the Arduno PID Library (PIDLibrary). The PIDLibrary is an industrial standard PID controller. RepRap is the feed forward example (PIDSample3) in the PID_Beta6 project folder. Here is a quick comparison of features.

	BBCC	AeroQuad	PIDLibrary
Complexity Factor	low	medium	high
Ability to Change Tunings on the Fly	?	?	yes
Inputs Normalized	no	no	yes
Input and Output Limits	no	no	yes
Multiple Control Loops	no	yes	yes
Reset-Windup Mitigation	somewhat	somewhat	yes
Integration Calculation includes dT	no	yes	no
Tuning	Processing	Labview	Processing

Like the AeroQuad PID, the coffee controller does not normalize the input. Both include non industrial-standard reset-windup mitigation code. Unlike the AeroQuad PID, the coffee controller does not calculate the integral term as a function of delta time. The Arduino PID library (PID_Beta6)

The inclusion of dT in the computation of accError is simply an academic point of interest (and calculations are accelerated without it). Mathematically, dT can be extracted out of the sum and simply included in the taur term.

```
taur *= ((double)NewSampleTime)/((double) tSample);
```

Tuning

Tuning the PID is where most of the “magic wand” action occurs. For some of the software control examples, the term “Configurator” is used for the development environment used for tuning the PID. It is not clear what IDE was used to develop the Configurator used by AeroQuad. The Coffee Controller used a Processing as its IDE for developing a simple configurator. The PIDLibrary also used Processing as illustrated here. It has since been ported to the mbed platform. Here is a very nice step-by-step Tuning Example using mbed.

The Ziegler-Nichols method is outlined in the AVR221: Discrete PID controller article. The first step in this method is setting the I and D gains to zero, increasing the P gain until a sustained and **stable oscillation** (as close as possible) is obtained on the output. Then the critical gain Kc and the oscillation period Pc is recorded and the P, I and D values adjusted accordingly.

Discrete PID Controller - Sample Period

*A discrete PID controller will read the error, calculate and output the control input at a given time interval, at the sample period T . The **sample time** should be less than the shortest **time constant** (36% of normalized output) in the system.*

On the other hand setting the sample period to high will result in an increase in the **integration error** from the gyro (converting from angular velocity to an angle). Note: this is a different integration than the I in PID.

Windup

Source: AVR221: Discrete PID controller

When the process input, *u*, reaches a high enough value, it is limited in some way. Either by the numeric range internally in the PID controller, the output range of the controller or constraints in amplifiers or the process itself. This will happen if there is a large enough difference in the measured process value and the reference setpoint value, typically because the process has a larger disturbance / load than the system is capable of handling, for example a startup and/or reset.

If the controller uses an integral term, this situation can be problematic. The integral term will sum up as long as the situation last, and when the larger disturbance / load disappear, the PID controller will overcompensate the process input until the integral sum is back to normal. This problem can be avoided in several ways. In this implementation the maximum integral sum is limited by not allowing it to become larger than MAX_I_TERM. The correct size of the

MAX_I_TERM will depend on the system and sample time used.

PID Control Software Examples

AeroQuadPID Control Software

Useful Links:

1. AeroQuad Home - http://aeroquad.com/content.php
2. The Manual - http://code.google.com/p/aeroquad/downloads/detail?name=AeroQuad%20Manual%20V9.pdf&can=2&q=
3. The Software *including the Configurator* - http://code.google.com/p/aeroquad/downloads/list
4. Configurator Help - http://aeroquad.com/content.php?116

The following AeroQuad header and pde files are key to understanding the AeroQuad PID software.

AeroQuad.h

This header file defines AeroQuad mnemonics

```
// Basic axis definitions
#define ROLL 0
#define PITCH 1
#define YAW 2
#define THROTTLE 3
#define MODE 4
#define AUX 5
#define XAXIS 0
#define YAXIS 1
#define ZAXIS 2
#define LASTAXIS 3
#define LEVELROLL 3
#define LEVELPITCH 4
#define LASTLEVELAXIS 5
#define HEADING 5
#define LEVELGYROROLL 6
#define LEVELGYROPITCH 7

float G_Dt = 0.02;
```

DataStorage.h

This header file is used to read and write default settings to the ATmega EEPROM.

```
// contains all default values when re-writing EEPROM
void initializeEEPROM(void) {
 PID[ROLL].P = 1.2;
 PID[ROLL].I = 0.0;
 PID[ROLL].D = -7.0;
 PID[PITCH].P = 1.2;
 PID[PITCH].I = 0.0;
 PID[PITCH].D = -7.0;
 PID[YAW].P = 3.0;
 PID[YAW].I = 0.0;
 PID[YAW].D = 0.0;
 PID[LEVELROLL].P = 7.0;
 PID[LEVELROLL].I = 20.0;
 PID[LEVELROLL].D = 0.0;
 PID[LEVELPITCH].P = 7.0;
 PID[LEVELPITCH].I = 20.0;
 PID[LEVELPITCH].D = 0.0;
 PID[HEADING].P = 3.0;
 PID[HEADING].I = 0.0;
 PID[HEADING].D = 0.0;
 PID[LEVELGYROROLL].P = 1.2;
 PID[LEVELGYROROLL].I = 0.0;
 PID[LEVELGYROROLL].D = -14.0;
 PID[LEVELGYROPITCH].P = 1.2;
 PID[LEVELGYROPITCH].I = 0.0;
 PID[LEVELGYROPITCH].D = -14.0;
 windupGuard = 1000.0;
```

FlightControl.pde

This C++ program calls the PID updatePID function and zeroIntegralError subroutine. Here are a few example calls.

```
updatePID(receiver.getData(ROLL), gyro.getFlightData(ROLL) + 1500, &PID[ROLL]
));
updatePID(receiver.getData(PITCH), gyro.getFlightData(PITCH) + 1500,
&PID[PITCH]));
updatePID(receiver.getData(YAW) + headingHold, gyro.getFlightData(YAW) +
1500, &PID[YAW]
```

PID.h

The PID data structure and PID algorithm

```
struct PIDdata {
 float P, I, D;
 float lastPosition;
 float integratedError;
} PID[8];
float windupGuard; // Read in from EEPROM

// Modified from http://www.arduino.cc/playground/Main/
BarebonesPIDForEspresso
float updatePID(float targetPosition, float currentPosition, struct PIDdata
*PIDparameters) {
 float error;
 float dTerm;

 error = targetPosition - currentPosition;

 PIDparameters->integratedError += error * G_Dt;
 PIDparameters->integratedError = constrain(PIDparameters->integratedError, -
windupGuard, windupGuard);
```

```
 dTerm = PIDparameters->D * (currentPosition - PIDparameters->lastPosition);
 PIDparameters->lastPosition = currentPosition;

 return (PIDparameters->P * error) + (PIDparameters->I * (PIDparameters-
>integratedError)) + dTerm;
}

void zeroIntegralError() {
 for (axis = ROLL; axis < LASTLEVELAXIS; axis++)
      PID[axis].integratedError = 0;
}
```

Bare Bones (PID) Coffee Controller

As commented on in the code, the AeroQuad PID software is a modified version of the BBCC: Bare Bones (PID) Coffee Controller The coffee controller is a single PID and so is a little simpler to understand.

Like the AeroQuad PID, the coffee controller does not normalize the input. Both include non industrial-standard reset-windup mitigation code. Unlike the AeroQuad PID, the coffee controller does not calculate the integral term as a function of delta time. The Arduino PID library (PID_Beta6)

```
float updatePID(float targetTemp, float curTemp)
{
  // these local variables can be factored out if memory is an issue,
  // but they make it more readable
  double result;
  float error;
  float windupGaurd;

  // determine how badly we are doing
  // error = setpoint - process value
  error = targetTemp - curTemp;

  // the pTerm is the view from now, the pgain judges
  // how much we care about error we are this instant.
  pTerm = pgain * error;

  // iState keeps changing over time; it's
  // overall "performance" over time, or accumulated error
  iState += error;

  // to prevent the iTerm getting huge despite lots of
  //  error, we use a "windup guard"
  // (this happens when the machine is first turned on and
  // it cant help be cold despite its best efforts)

  // not necessary, but this makes windup guard values
  // relative to the current iGain
  windupGaurd = WINDUP_GUARD_GAIN / igain;

  if (iState > windupGaurd)
```

```
      iState = windupGaurd;
  else if (iState < -windupGaurd)
      iState = -windupGaurd;
  iTerm = igain * iState;

  // the dTerm, the difference between the temperature now
  //  and our last reading, indicated the "speed,"
  // how quickly the temp is changing. (aka. Differential)
  dTerm = (dgain* (curTemp - lastTemp));

  // now that we've use lastTemp, put the current temp in
  // our pocket until for the next round
  lastTemp = curTemp;

  // the magic feedback bit
  return  pTerm + iTerm - dTerm;
}
```

PIDLibrary - PID_Beta6.cpp

You will need AVR Studio to view this file with color coding the C++ code.

```
/* Compute()
 ******************************************************************
 *  This, as they say, is where the magic happens.  this function should
 *  be called every time "void loop()" executes.  the function will decide
 *  for itself whether a new pid Output needs to be computed
 *
 *  Some notes for people familiar with the nuts and bolts of PID control:
 *  - I used the Ideal form of the PID equation.  mainly because I like IMC
 *    tunings.  lock in the I and D, and then just vary P to get more
 *    aggressive or conservative
 *
 *  - While this controller presented to the outside world as being a Reset
 *    Time controller, when the user enters their tunings the I term is
 *    converted to Reset Rate.  I did this merely to avoid the div0 error
 *    when the user wants to turn Integral action off.
 *
 *  - Derivative on Measurement is being used instead of Derivative on Error.
 *    The performance is identical, with one notable exception. DonE causes a
 *    kick in the controller output whenever there's a setpoint change.
 *    DonM does not.
 *
 *  If none of the above made sense to you, and you would like it to, go to:
 *  http://www.controlguru.com .  Dr. Cooper was my controls professor, and

*  is gifted at concisely and clearly explaining PID control
*****************************************************************************
**/
void PID::Compute()
{
    justCalced=false;
    if (!inAuto) return; //if we're in manual just leave;

    unsigned long now = millis();
    //millis() wraps around to 0 at some point. depending on the version of
the
    //Arduino Program you are using, it could be in 9 hours or 50 days.
    //this is not currently addressed by this algorithm.

    //...Perform PID Computations if it's time...
    if (now>=nextCompTime)
    {

       //pull in the input and setpoint, and scale them into percent span
       double scaledInput = (*myInput - inMin) / inSpan;
       if (scaledInput>1.0) scaledInput = 1.0;
       else if (scaledInput<0.0) scaledInput = 0.0;

       double scaledSP = (*mySetpoint - inMin) / inSpan;
       if (scaledSP>1.0) scaledSP = 1;
       else if (scaledSP<0.0) scaledSP = 0;

       //compute the error
       double err = scaledSP - scaledInput;

       // check and see if the output is pegged at a limit and only
       // integrate if it is not. (this is to prevent reset-windup)
       if (!(lastOutput >= 1 && err>0) && !(lastOutput <= 0 && err<0))
       {
             accError = accError + err;
       }

       // compute the current slope of the input signal
       // we'll assume that dTime (the denominator) is 1 second.
       double dMeas = (scaledInput - lastInput);
       // if it isn't, the taud term will have been adjusted
       // in "SetTunings" to compensate

       //if we're using an external bias (i.e. the user used the
       //overloaded constructor,) then pull that in now
       if(UsingFeedForward)
       {
             bias = (*myBias - outMin) / outSpan;
       }

       // perform the PID calculation.
       double output = bias + kc * (err + taur * accError - taud * dMeas);

       //make sure the computed output is within output constraints
       if (output < 0.0) output = 0.0;
       else if (output > 1.0) output = 1.0;

       lastOutput = output;          // remember this output for the windup
                               // check next time
       lastInput = scaledInput; // remember the Input for the derivative
                               // calculation next time

       //scale the output from percent span back out to a real world number
                  *myOutput = ((output * outSpan) + outMin);
```

```
            nextCompTime += tSample;   // determine the next time the computation
                                       // should be performed
            if(nextCompTime < now) nextCompTime = now + tSample;

            justCalced=true;  //set the flag that will tell the outside world
                              // that the output was just computed
        }
    }
```

Appendix A: Source Material

Control

There is no magic control design technique. Any linear control technique yields a filter (the compensator) in the end, and there is only so much that the filter can do. Just because the filter comes from PID, or LQG, or QFT, or H-infinity, or LQR - http://www.control.com/thread/966880428

The robot acts as an inverted pendulum and works better when the weight is high.

PIDS

http://www.controlguru.com/pages/table.html

maybe I can post the links in a reply:
http://www.arduino.cc/playground/Code/PIDLibrary
http://www.arduino.cc/playground/Main/BarebonesPIDForEspresso
http://www.societyofrobots.com/member_tutorials/node/185
Arduino Forum › General › Exhibition › PID Tutorial
Error(current) = Setpoint - Ptime
P = Error(current) * Kp
Error(sum) = Error(current) + Error(last) + Error(secondtolast)
I = Ki * Error(sum)
Error(delta) = Error(current) - Error(last)
D = Kd * Error(delta)
Drive = P + D + I
Error(secondtolast) = Error(last)
Error(last) = Error(current) 'Print "PID value = " ; Drive

Designing a PID

Making your robot get from point A to point B smoothly
http://thecodebender.com/journal/2009/3/30/designing-a-pid-making-your-robot-get-from-point-a-to-point.html

Learning

If you are unfamiliar with PID control, a good place you start might be: http://en.wikipedia.org/wiki/PID_controller
http://www.controlguru.com . Dr. Cooper is gifted at concisely and clearly explaining PID control

Tuning

http://mbed.org/cookbook/PID

BASIC TUNING SUGGESTIONS from PIDLibrary
(these are by no means comprehensive)
Unless you know what it's for, don't use D (D_Param=0)
Make I_Param about the same as your manual response time (in Seconds)/4
Adjust P_Param to get more aggressive or conservative control
If you find that your controller output seems to be moving in the wrong direction , change the sign of the P_Param (+/-)
(these are not mine but far better)
The PIDLibrary has been ported to the mbed platform. The backend calculations are the same, so their Tuning Example can be followed step by step.

Graphical Front-End for the Arduino PID Library

A NEW TUNING OF PID CONTROLLERS BASED ON LQR OPTIMIZATION

Kalman Filter

http://en.wikipedia.org/wiki/Kalman_filter
Kalman filter module works pretty well but is still a Black Box for me
I tried hard to understand and finally gave up.
This code is a modified version from the AeroQuad project from Ted Carancho
http://code.google.com/p/aeroquad/

I spent some hours playing with Q_angle, Q_gyro and R_angle, and finally reverted to the original parameters

Some additional lectures for the brave:
http://academic.csuohio.edu/simond/courses/eec644/kalman.pdf
http://www.cs.unc.edu/~welch/kalman/index.html#Anchor-Rudolph-6296
http://forum.sparkfun.com/viewtopic.php?t=6186

Here is a comparison between the raw accelerator angle (red) and the Kalman filtered angle (blue)
The smoothing effect is impressive (source) http://www.youtube.com/v/H5Drlqv1t3s? This is a simple custom development in LabView. The Arduino sends 2 integers separated by a comma. Data can be sent alternatively to IDE serial monitor or to LabView. Angles are scaled in Quids (360° = 2 PI = 1024 Quids)
This unit is convenient and can be manipulated as integer while retaining a good resolution
Code:
```
void serialOut_labView() {
  Serial.print(Angle + 512);        Serial.print(",");       // in Quids
  Serial.print(ACC_angle + 512);      Serial.print("\n");
}
```

If your robot rolls about the x axis, you only need GYR_X, ACC_Y and ACC_Z

I notice that signal quality is much better when the sensor is upward, apparently, the IMU does not appreciate to be upside down This was posted here on 9/11/2010

```
// Kalman filter module

 float Q_angle  =  0.001;
 float Q_gyro   =  0.003;
 float R_angle  =  0.03;

 float x_angle = 0;
 float x_bias = 0;
 float P_00 = 0, P_01 = 0, P_10 = 0, P_11 = 0;
 float dt, y, S;
 float K_0, K_1;


  float kalmanCalculate(float newAngle, float newRate,int looptime) {
    dt = float(looptime)/1000;
    x_angle += dt * (newRate - x_bias);
    P_00 +=  - dt * (P_10 + P_01) + Q_angle * dt;
    P_01 +=  - dt * P_11;
    P_10 +=  - dt * P_11;
    P_11 +=  + Q_gyro * dt;

    y = newAngle - x_angle;
    S = P_00 + R_angle;
    K_0 = P_00 / S;
    K_1 = P_10 / S;

    x_angle +=  K_0 * y;
    x_bias  +=  K_1 * y;
    P_00 -= K_0 * P_00;
    P_01 -= K_0 * P_01;
    P_10 -= K_1 * P_00;
    P_11 -= K_1 * P_01;

    return x_angle;
  }
```

A low-cost and low-weight attitude estimation system for an autonomous Helicopter uses complementary filters - referenced from this nBot site.
http://www.geology.smu.edu/~dpa-www/robo/balance/inertial.pdf

a kalman filter must be used with accelerometers and many other sensors..

Wheel Encoder

http://www.mindspring.com/~tom2000/Delphi/Codewheel.html

Parts

Motor

The servos do not have a fast enough response (RPM = 60). Use of DC motors with an RPM over 200 is needed. - Build a (simple) balancing robot

Video

Balancing Arduino Robot

Balancing Arduino Robot
http://www.youtube.com/watch?v=RaN0itBbVR4
Self balacing robot using Arduino
http://www.youtube.com/watch?v=QskYp5lM1BE&NR=1
Makershed.com
Parallax Memsic 2125 2-axis accelerometer
How-to Tuesday: Arduino 101 Accelerometers (Talks about balancing robots)
http://www.youtube.com/watch?v=HYUYbN2gRuQ&NR=1&feature=fvwp
Sun Tracking Solar Panel w/ Arduino
http://www.youtube.com/watch?v=ATnnMFO60y8&feature=related
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1275324410
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1280253883

Segway = Arduino + Lego
http://www.youtube.com/watch?v=7BsN4fL8GkI&feature=related

Balancing Robot Forum Threads

Balancing robot for dummies

Arduino Forum › General › Exhibition › Balancing robot for dummies
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1284738418
Part I
Arduino Forum › General › Exhibition › DC motor control with PID
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1282384853/0

```
int updatePid(int command, int targetValue, int currentValue)   {
// compute PWM value
float pidTerm = 0;
// PID correction
int error=0;
static int last_error=0;
  error = abs(targetValue) - abs(currentValue);
  pidTerm = (Kp * error) + (Kd * (error - last_error));
  last_error = error;
  return constrain(command + int(pidTerm), 0, 255);
}
```

Self Balancing Robot

Arduino Forum › General › Exhibition › Self Balancing Robot
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1225283209/all
Sure, i'll post the code this evening as i get home. Be warned that the code is a real mess
Anyway, i adapted the code for the kalman filter from these two sites:
http://diydrones.com/profiles/blog/show?id=705844%3ABlogPost%3A23188
http://tom.pycke.be/

The rest is the pid controller (not that complex), something to use the LCD and other glue.
I don't know if you already know it. In few words, it's a well know way to take into account three different parameters into the calculation of the final speed: the Proportional (that is the error, in our case, the current angle of the robot), the Derivative (that is, the angular velocity, or how fast it is falling/raising), and the Integrative, which is a sum of the errors in all previous iterations.
Ah ok, than that's the reason. You cannot just sum up the readings from the gyro to get the absolute angle, it will drift quite a lot with time like any kind of dead reckoning.
The kalman filter is what helps in this situation: using data from accelerometers (that sense the gravitational acceleration) you can get an absolute value for the current angle. The problem is that while on the long time it averages to the right value, it will change badly as the robot swing, rotate or stop. The kalman filter is what makes an adeguate "combination" of the gyro and accelerometers to get a value that is correct both in the short time and in the long time.

Here's a serie of article that explain this far better than i did:

http://tom.pycke.be/mav/69/accelerometer-to-attitude
http://tom.pycke.be/mav/70/gyroscope-to-roll-pitch-and-yaw
http://tom.pycke.be/mav/71/kalman-filtering-of-imu-data
The "looseness" you describe is a real problem with balancers. I used motors similar to yours at first, but was able to achieve much tighter and more responsive balancing by switching to zero-backlash servos (Dynamixel AX-12).

PID Library

Neurasonic asked for a story on PID control. Here's my take.
PID stands for Proportional, Integral, differential. What do THEY stand for? Start with comparing what you have with what you want with what you have. The difference is called the error signal. If you use only this error signal to drive the output, you have a proportional controller (if the error signal is linear or proportional). This can be OK, but there HAS to be an error to keep driving the output, so it's never exactly right.
OK, how about toting up the small difference, time after time, when you are as close as you can get with the proportional circuit, and driving the output with THAT too? That can work and hit the desired output on the nose.
Now what about if we next want a step change in output?
If we grow restless with a fast start, and a slower slower approach to the desired output, we can get bold. By monitoring the rate that the error signal changes we reduce the time to reach the target value, and (hopefully) minimize the over shoot and hunting chances.
Having said all that PID controllers can be a pig to tune. Some people find fuzzy logic controllers are much easier to understand and to set up.

Brian Whatcott
Altus OK
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1226431507/30
...I find that the blessing and the curse of the PID is it's ease of configurability. you can get a control loop to "good 'nuf" pretty quickly. the problem is that sometimes that's a ways away from "good." I'd say, as far as arduino is concerned, simple fuzzy logic is the right place to start. that can get real complicated real quickly though, and at that point it's time for this library (IMO)

I don't understand why you have problems making your PID faster....maybe because I've never used it . Sometimes the measurement is not at the same frequency than the Computing frequency, maybe you can't achieve the maximum speed because you are not using timer interrupts, and almost every PID ready-to-use software made by big companies uses timer interrupts. When you use timer interrupts the only limit is the clock speed and the timer's pre-escalator, ahh one last thing, the common acronyms are:
Set Point, (SP)
input: PV (Present Value)
output: MV(manipulated variable)
error
dt : (delta t) that's what you call calc period
Kp
Ki,Ti (I prefer Ki than Ti, where Ki=Kp/Ti)
Kd,Td (same with Kd, Kd=Kp*Td)

EDIT: PV: process variable
the issue isn't with when the computation occurs, it's with how long the computation takes when it's excecuted. the computation fires at whatever interval the user desires (set by the SetSampleTime function) the interrupt idea is interesting. I'll have to look into that.

what I was talking about when I said "faster" was the computation itself. the fewer processor cycles it takes to perform its calculation, the more time the arduino has to do other things. and, just as a point of pride, I want the pid to be as efficient as possible.

as far as abbreviations, some are certainly more common than others, but they do vary from manufacturer to manufacturer. off the top of my head:
Process Variable: PV, ME
SetPoint: SP, SV
Controller Output: CO, CV, OUT, OT, OP.

dt is commonly used. I settled on SampleTime (which is also commonly used) because I felt it was easier to understand.

at any rate, give the library a shot! I'd love to hear any more suggestions you might have.
...I just looked at the source, and it looks very well, I can see that you have done something to prevent **windup**, I also prefer the Ideal form of the PID algorithm, everything is done efficiently (computing time-speaking), maybe millis() is taking too much, I don't see where else the code could be made more efficient, I will try to do it with interrupts and I will let you know if it works better, also I'm going to implement a **kalman** filter, so maybe that will be my first contribution to the arduino project, for those who don't know what I'm talking about, a **kalman filter must be used with accelerometers and many other sensors**..
By the way PV is often referred as "Process Value" in engineering terms
There is no problem to run it in master slave configuration. Just run it twice with different inputs. We have used this successfully

Last year I have trained a student during 3 month in order to make a complex regulation, for this purpose we have developed and implemented the Broïda method that allows to identity PID coefficient by measuring the response of the open loop system to square by measuring an amplitude and two time (at 0.28*deltaA* and 0.40*deltaA) it gives parameters depending of the kind of PID you want (with or without overshoot). This method seems to work up to the 6th

order.
What do you think about coding this into the arduino platform ?
I think it can works on a lot of cases !
this may be a good place to start: http://www.arduino.cc/playground/Main/InterfacingWithHardware (scroll down to ADC/DAC)

I am building a simple temperature controller and I'm attempting to use your PID library. The arduino reads the temperature from an LM35 and then outputs a slow PWM pulse to a solid state relay which will switch a standard electrical output. I figure I'll plug in either a water bath heater or an electric smoker to the outlet, it seems like it should work just as well for either application. Anyhow, I don't have much of the hardware yet (it's all on order), but I put together the code and I wanted to make sure I was utilizing the PID library correctly. As you can see, I'm using the Timer1 library to run a 1 second PWM pulse. I wouldn't think a heating element would need anything faster than that. The PID output adjusts the duty cycle of the pulse:
first off, there's a timer library?! why did I not know this?

your code looks good. be advised that you will need to adjust the tuning parameters (3,4,1) depending on what you're connected to. it's unlikely that the default values will give you the response you want, or that the same values will work for both the water bath heater and smoker.
Yeah, I understand about the tuning parameters, I guess I'll just have to eyeball things until I get them set up right. Like I said, nothing is built just yet, it's still diagrams and connections floating around in my head. I've heard it'll probably take a long time (hours? days?) to tune due to the slow response of my systems (I'm thinking the smoker will probably be faster than the water bath though).

The Timer1 library is something I stumbled upon while trying to figure out how to slow down the arduino PWM. I think I understood the software "window" method that you wrote into the second example on the PID library page, but it seemed like there must be a better way. I don't claim to understand it completely, but it seems like just the ticket for what I'm trying to do:

http://www.arduino.cc/playground/Code/Timer1

Thanks again!
sorry for the confusion. there's various forms of the pid equation. (http://en.wikipedia.org/wiki/PID_controller scroll down to "Alternative nomenclature and PID forms")
they're mathematically equivalent, but the one listed as the "standard form" is the one i chose. it's a little more work on the backend as you mentioned, but I find it easier to tune. if you know the time constant of the process, you just make that the I term, then adjust P to get more aggressive or conservative)
I started a new thread specific to PID Tuning:http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1280251311 From Brett (the creator)
.....
http://www.rcgroups.com/forums/showthread.php?t=1066119&page=51

I think I'll drop in my two cents on this one. The simplest explanation that I've heard from anyone was given by Prof. George Ellis who talked about an observer in an IMU system that acted like a low-pass filter on some of the feedback and a high-pass filter on other parts of the feedback.

The accelerometer provides really good and reliable data for low frequency feedback (static up to some cutoff frequency), but any noise causes the data to drift when you integrate it. On the other hand, the gyro provides really good data at higher frequencies, but the DC offset drifts (as you guys were talking about). So if you low-pass filter the accelerometer data and high-pass filter the gyro data, you actually get a much fuller picture of the state of the helicopter than you would with either sensor individually.