

# PID Motion Control for the Lego NXT using leJOS Java

*Andy Shaw*

## Introduction

This document describes an investigation into the use of a PID based motion control system for the Lego NXT programmable brick. The system is implemented using the leJOS Java environment and it is assumed that the reader is familiar with this and the NXT.

leJOS has a set of motor control classes as part of the standard class library. Details of this are presented and it is used as the basis of comparison with the PID based alternative. A number of tests are described which demonstrate the characteristics of the two implementations.

## leJOS motor control

The standard motor control system as implemented via the *Motor* class provides both regulated and unregulated control of one or more Lego motors, only the regulated mode is discussed here. The regulator is provided via a thread per motor, this thread performs the following actions:

- Smooth acceleration through velocity ramp up.
- Speed regulation during the move operation.
- Monitoring of the number of degrees rotated and optional termination when a previously set number of degrees are reached.

An additional timer thread provides monitoring of battery voltage (used to adjust initial power levels), and to calculate the current motor speed every 100ms.

Smooth acceleration is provided by increasing the current speed of the motor from zero until either the end point is reached or the motor velocity reaches the desired value. The acceleration is provided as part of the standard regulation process.

The regulation process runs continuously sampling the current motor position. This position is compared with the expected position (created by multiplying the required velocity by the time spent so far on the move operation) to form an error term.

$$\text{Error} = (\text{Elapsed} * \text{Speed}) - (\text{CurrentTachoCount} - \text{BaseTachoCount})$$

This error term is used to correct the current motor power setting

$$\text{Power} = \text{BasePower} + K_p * \text{Error}$$

The delta of the power setting is used to smoothly adjust the base power

$$\text{BasePower} = \text{BasePower} + K_i * (\text{Power} - \text{BasePower})$$

The constants  $K_p$  and  $K_i$  are selected by observation to provide smooth speed regulation. The current version of the regulator makes use of the motor in float mode (which according to the Lego documentation provides smoother running and consumes less power). This mode has a large deadband (power settings which until exceeded provide no motor rotation). This deadband is overcome by the initial setting of BasePower, this setting is also used to provide compensation for any drop in battery voltage. The current deadband is approximately 35%.

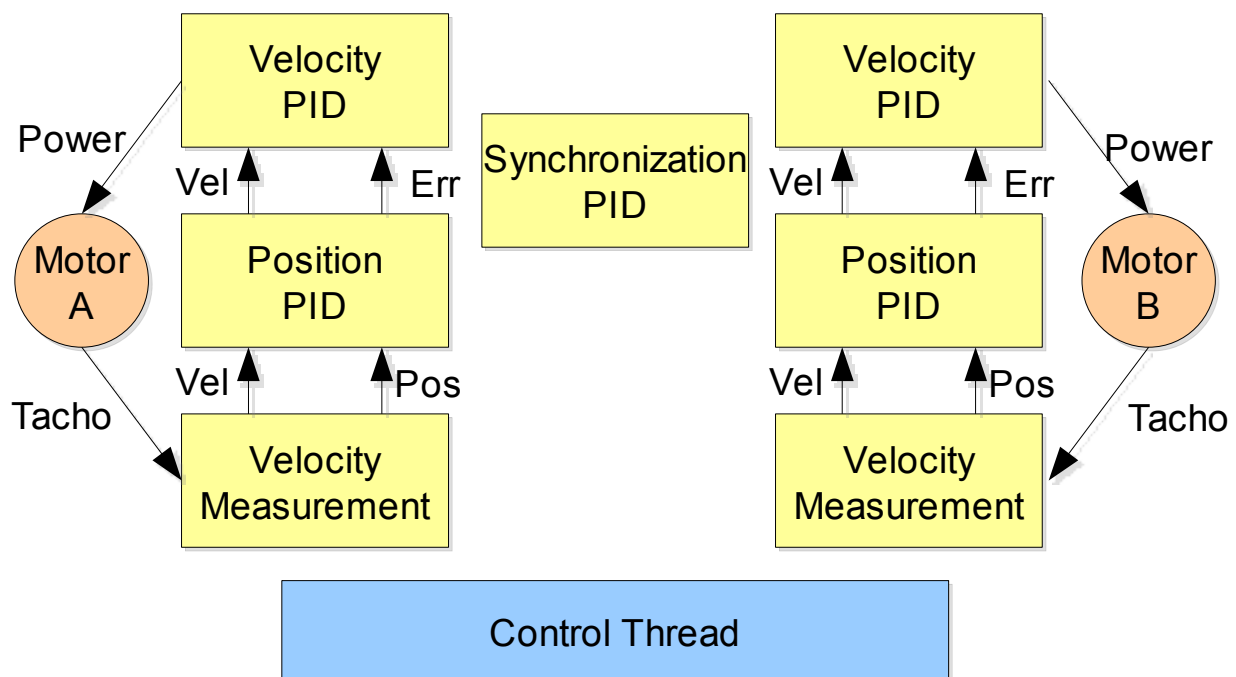
This regulator is similar to a PI (proportional and integral) control system, with the error being based on overall velocity error. The use of overall velocity is interesting in that it provides an additional smoothing factor, rather than the interval based sampling often used with PID controls.

The positional control is provided by constant monitoring of the tachometer output. When the position reaches a pre-calculated stop point then the motor is braked to a stop. The stop point is calculated from the desired end point along with the requested velocity such that the motor will stop

just short of the request end point. Finally one or more small moves (a nudge), are used to move the motor to within 3 degrees of the end point.

## Experimental PID motor control

The experimental PID control system describe here was created by deriving a new class *PIDMotor* from the existing *Motor* class. This class overrides many of the methods of the base class and in particular disables the threads used by that class. One of the principle aims of the experiment was to investigate the use of active synchronization between motors, as part of this system all speed and movement control is driven by a single thread shared between all motors. The diagram below shows the major components



As can be seen from the above each motor is controlled by two PID stages. The single control thread consists of three inner loops each loop operates over all three motors. The first loop calls the velocity measurement function (for each motor in turn), the second loop calls the position PID and the final loop calls the velocity PID. Finally the thread sleeps for what is remaining of the overall loop time before starting the process again. The aim here is capture velocity and position readings for each motor and to issue new power settings as closely together as possible, thus improving the synchronization between motors. The overall loop runs 50 times per second, with the position PID running every other time through the main loop (so 25 times per second). These rates seem to provide good overall control. All of the stages in this controller operate using fix point or integer math.

The first part of this process is the measurement of the current motor velocity and the generation of the PID velocity error terms. The normal way to do this is to read the current position from the tachometer and then the previous reading to give the velocity over the current sample period. This is then compared with the current requested velocity to generate the error term. However with a sample period of only 20ms and with the encoder placed on the output shaft (encoders are often placed on the faster spinning motor shaft), then there is a resolution problem. At slow speeds (say 50 degrees per second), then in a 20ms sample we would expect only a single count, which is not really sufficient. To work around this issue the measurements make use of a sliding window that spans five samples, giving a period of 100ms and a much more healthy count of 5 per sample. To

ensure that the error terms are correct (and not skewed by sudden changes in the requested velocity), the last five velocity requests are also kept and combined to form the overall velocity term. Using this mechanism it has been possible to control motors operating at speeds as low as 10 degrees per second.

The Position PID takes as input the current velocity and position (plus the requested velocity and stopping position for this request) and generates a new output velocity. This first stage controls the position of the motor and is responsible for generating the smooth ramp up in velocity and stopping the motor when the requested position is reached. It uses the current position and the required end position to form the PID error term. The PID controller then uses this to generate a target value (between -100..+100) which is combined with the current required velocity to generate a target velocity for the velocity PID. The current velocity normally starts at zero and is then ramped until the final requested velocity is reached (providing smooth start up). Under normal operation the position PID will return a value of 100, however as the motor approaches the end point then the PID control comes into operation and generates a term dependent upon the distance from the end point, plus the velocity at which the end point is being approached. Normally this will result in the speed being smoothly ramped down and the motor coming to rest within a few degrees (specified within the definition of the position PID and currently 2) of the end point. If the motor overshoots or stops short then the PID will continue to generate velocity requests (based primarily on the integral term), driving the motor until the end point is reached. This action will then continue after the motion has halted with the PID working to keep the motor at the specified location until a new movement request is made or the motor is turned off.

The second PID stage is used to control the actual velocity of the motor. The error term for this PID is the difference between the target velocity (from the position PID), and the velocity measured from the tachometers (see above for more details). This PID stage makes use of an additional term known as velocity feed forward to drive the final power output. This term is simply a proportional term based on the request velocity, it is in this term that the deadband associated with the Lego motor is included. A major difference here is that this controller uses the motor in brake mode. In this mode power is supplied through the entire PWM cycle. This mode has a much smaller deadband (the current implementation uses 6%). Following experiments with both modes it was found that brake mode also seems to offer better control when decelerating which is particularly important when used with a PID controller.

## **Active synchronization**

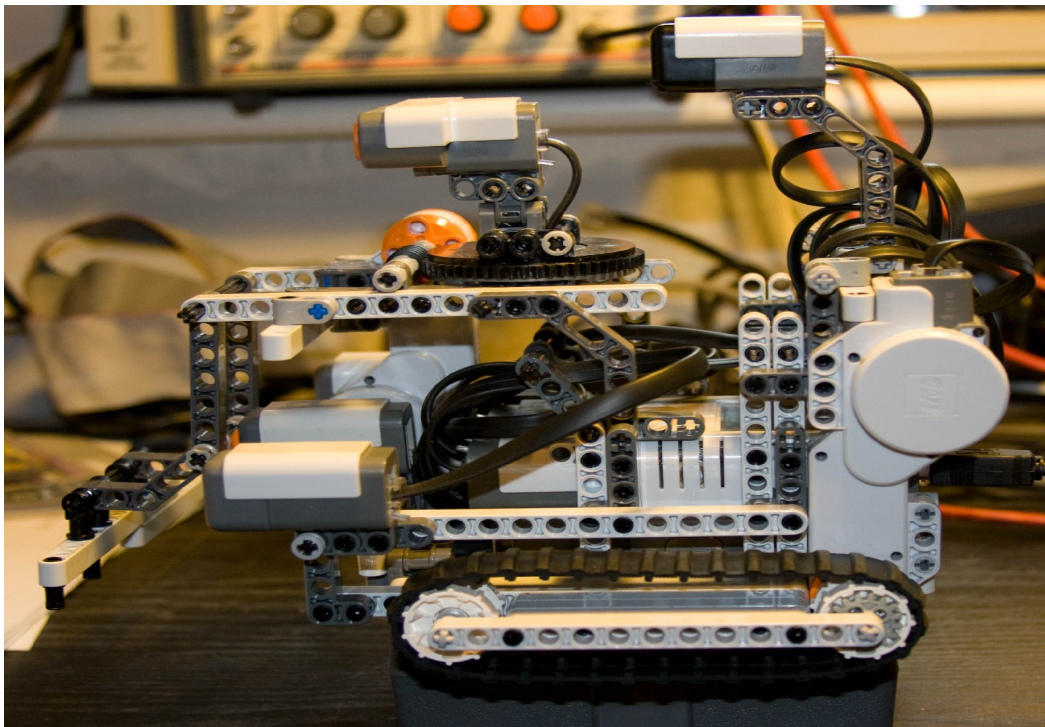
The experimental controller also features the capability to provide active synchronization between a pair of motors. When this mode is enabled two additional controls come into play.

The first is to synchronize the start of any movement command. To do this the system will not begin to move either of the synchronized motors until move requests (rotateTo etc), have been issued to both motors. This has the effect of delaying the start of the movement of the first motor until the movement request for the second motor is issued. This mechanism ensures that both motors will begin to move very closely together.

The second control adds a further PID stage into the system. This PID takes the current velocity of the two motors and combines them to generate an error term. The PID output is then used to adjust the target velocities of the two motors (as fed into the velocity PID) to ensure that they operate at the same speed (and via the integral term to ensure that any past differences are corrected). The effect of this is to link the two motors together, so that if for instance one motor is stalled then the other will also come to a halt. In theory this should ensure that such things as hitting small objects (which slows only one motor), will be handled correctly.

## Testing the controller

In order to test how well the system operates a number of tests have been created. Each test and the results are described below. All of tests were run using the same test vehicle, this vehicle uses tracks and is pictured (on the test stand used for some of the tests) below.



**Test vehicle**

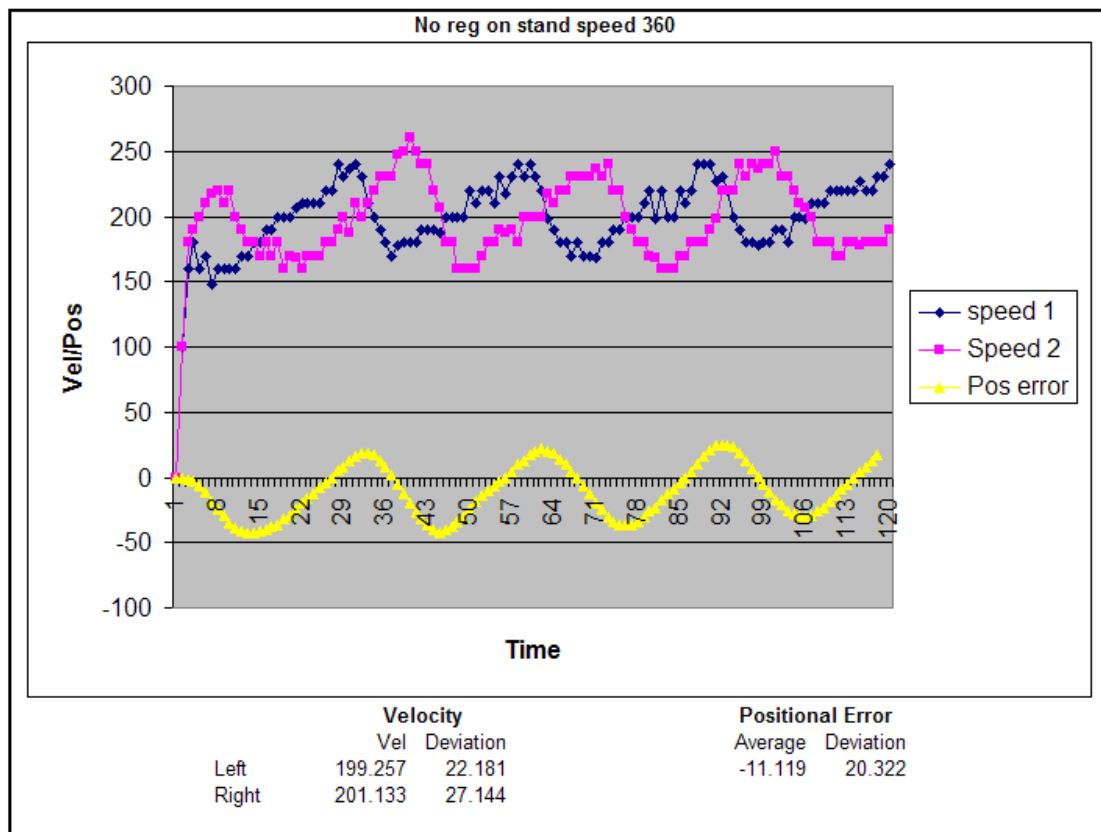
All of tests make use of the same set of programs. These consist of a program downloaded to the NXT that performs the requested move operation and a capture program running on a PC. During the move operation the program loops sampling the current tachometer readings from the two motors every 100ms these readings along with the calculated velocities are stored in an array. Once the move operation is completed then the captured data is uploaded via bluetooth to a program running on a PC. The data is then exported to Excel for further processing.

The Excel program is used to produce a graph of the velocities for the left and right motors along with a plot of the difference between the tachometer readings for each sample. In an ideal world this difference would be zero, here it is a good indication of any errors in the synchronization of the motor drive. In addition to the graph, the overall velocity during the centre part (the section that should be at constant speed) of the move and the standard deviation of this velocity is displayed. Finally the average error (again during the centre part of the move), plus the standard deviation of the error is included. This last figure is particularly useful as an indication of the variation in synchronization over the length of the run.

### Test 1. Open loop motor

To get a feel for exactly why we need all of this complex motor control (and to show how good a job these controllers do), the first test is to run the motors without any motor control. The vehicle is placed on a stand (so no floor friction). The motors are driven via the standard leJOS interface with `regulateSpeed` called to turn off the motor regulator. As with all of the other tests smooth acceleration is enabled. Both motors are set to rotate at a speed of 360 degrees per second (the default speed for the standard software, and a good all round speed to use). The actual movement is made by calling `rotateTo` with an argument of 4000 (requesting a rotation of 4000 degrees before stopping). This combination provides around 110 data points which gives a nice graph.

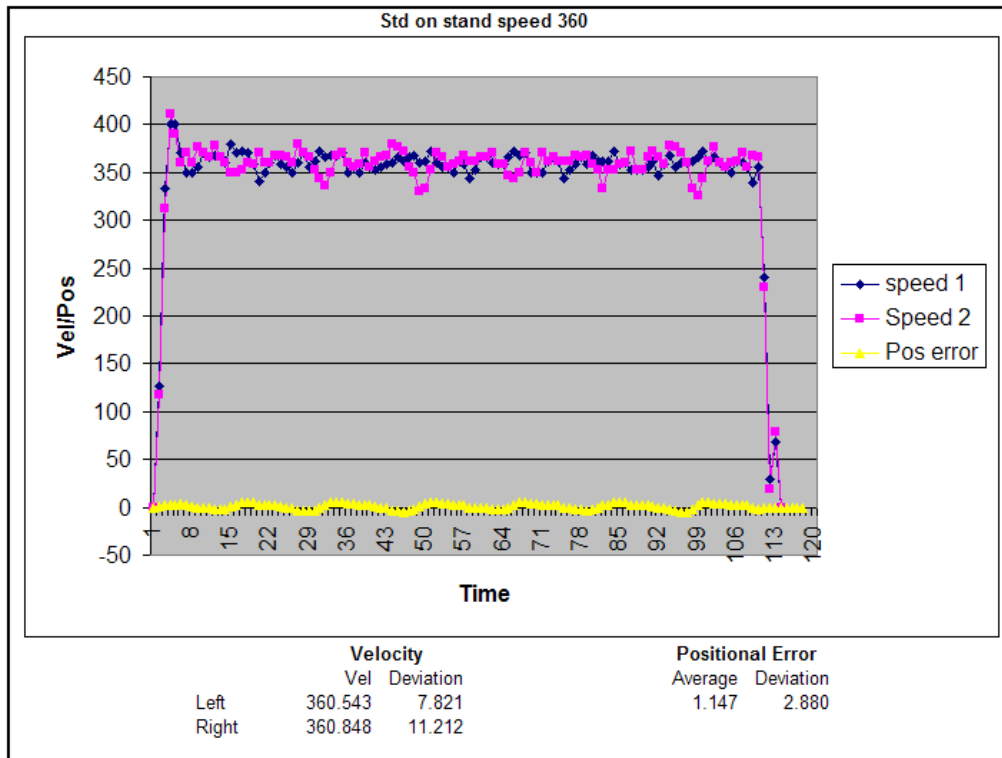
The results of this initial test are shown below, the velocity/position axis is in degrees (per second for velocity), the time axis is in 100 millisecond units.



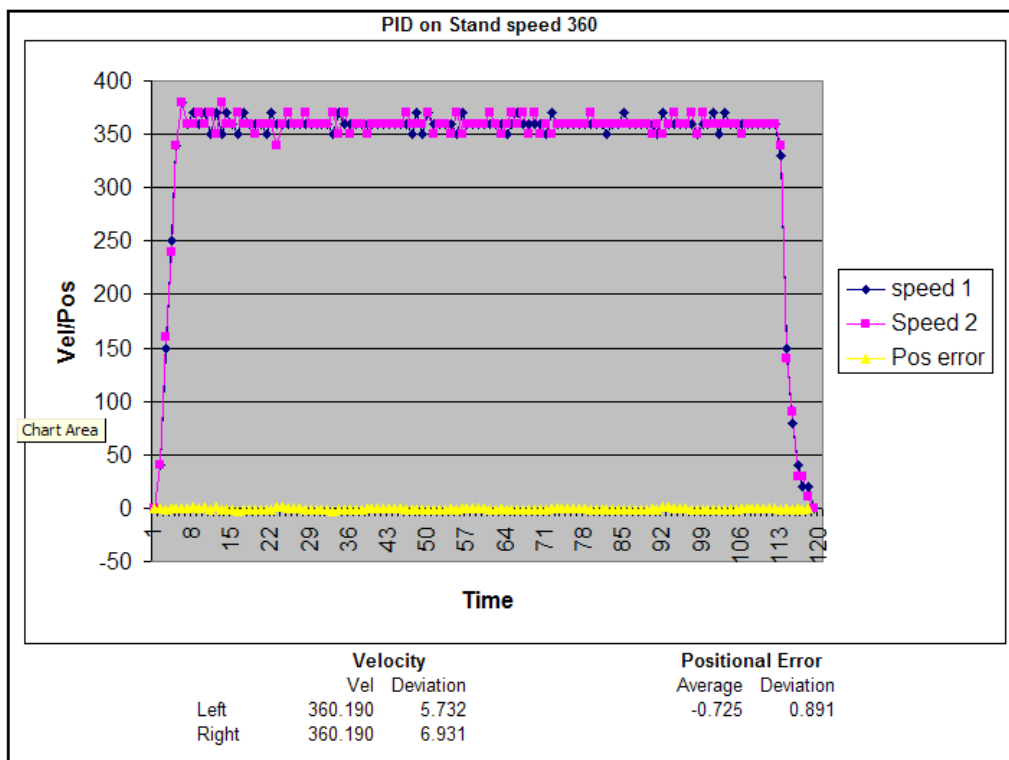
The results are pretty bad. The velocity of the two motors is much less than required (averaging around 200 rather than 360). This is to be expected since without regulation the controller is simply choosing a power level based on the requested speed. This choice is not able to take in to account things like friction, variation between motors etc. and is one of the faults of so called open loop control. Perhaps what is surprising is the way the speed seems to oscillate. What seems to cause this is friction caused by the tracks used on the test vehicle, basically they are harder to drive at some points than others. Again this is a good example of why we need a more sophisticated control system. The positional error can be clearly seen in the error plot, and the deviation gives an indication of how this varies. Finally note that because the velocity was so low we failed to capture the end of the movement (we ran out of space to record the results).

## Test 2: On stand tests

The first set of tests with regulation enabled are based on the above test. The test vehicle is on a stand (and is running off mains supplied power). For the first test we will again use a target velocity of 360 degrees per second. The results for the standard leJOS class is shown below:

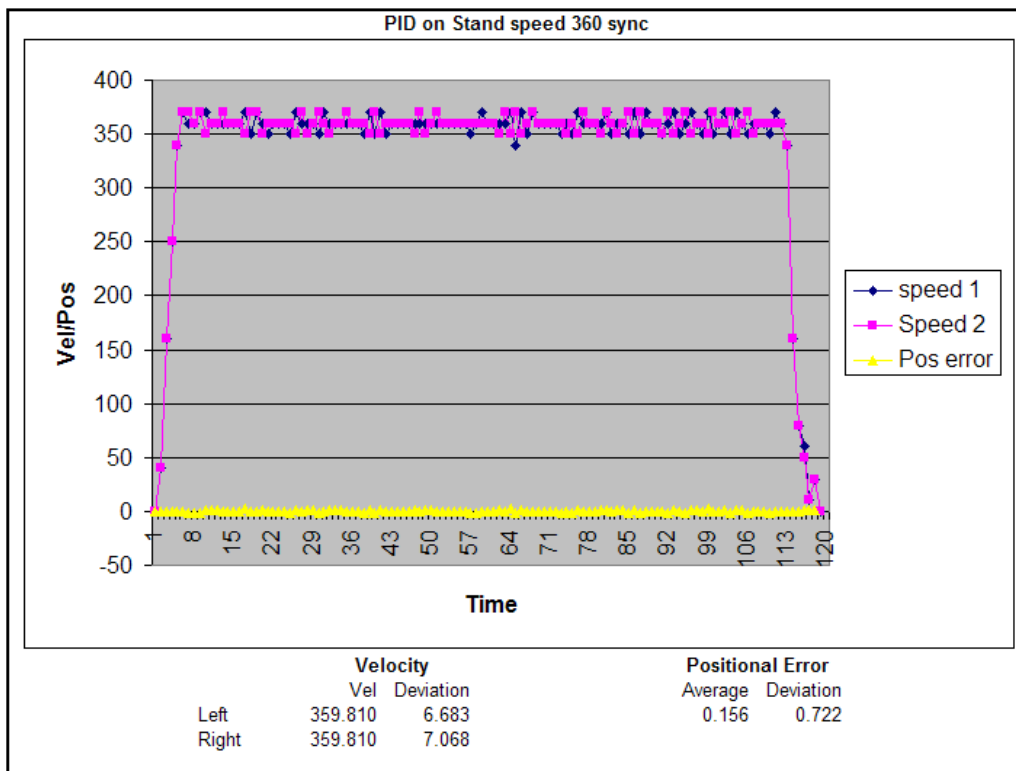


The improvement is huge. With the exception of a slight overshoot, the velocity of the two motors is centered nicely on 360. Almost all of the ripple seen above has been controlled (although a small residue can be seen in the error plot). Finally the “stop and nudge” action can clearly be seen at the end point.



The second graph shows the results using the PID implementation. Again there is a slight overshoot

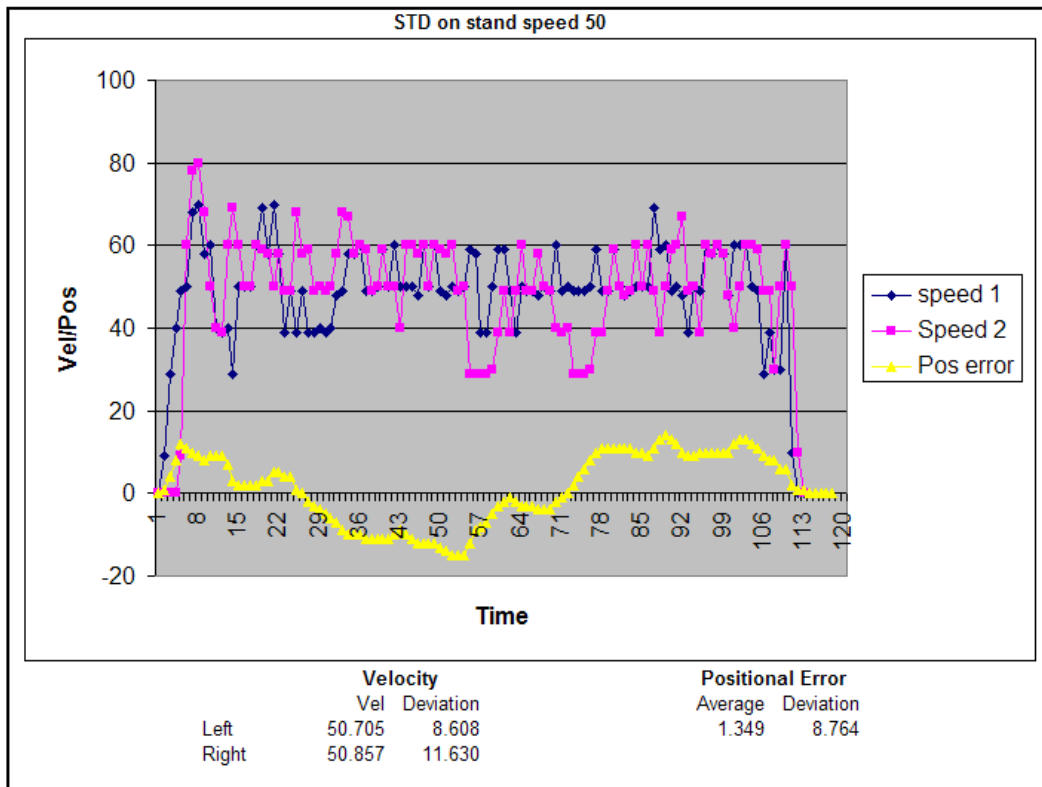
of the initial acceleration. Virtually all of the ripple has gone (again a small amount shows in the error plot and can be seen in the standard deviation). It can be seen that the initial acceleration is slightly slower with this controller, and the way that the position PID slows the velocity as the end point is reached can clearly be seen. Both of these characteristics lead to the movement taking longer overall.



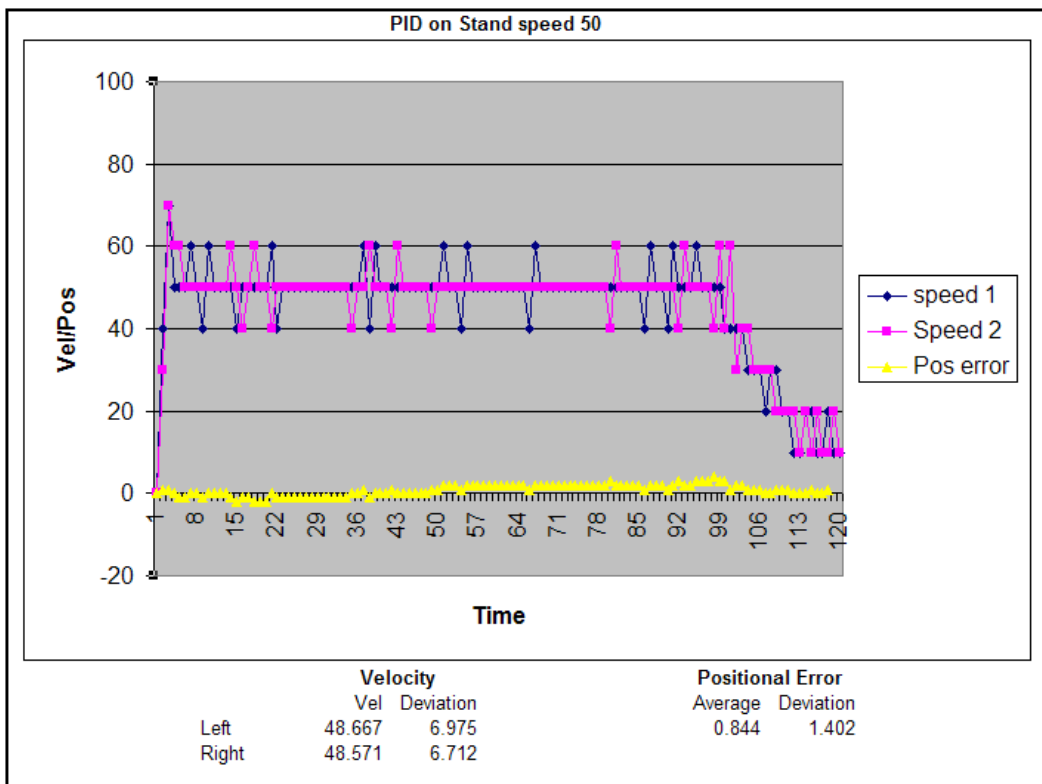
The final graph from this set shows the PID controller with synchronization enabled. The (already low) positioning error has been reduced yet further. However this test does show a couple of interesting side effects of the synchronization. There is a slight kick as the motors slow down which seems to be caused by the synchronization PID interacting with the position PID as it attempts to slow to the end point. Also the standard deviation of the velocities is slightly higher, this is to be expected as the synchronization tends to force the two curves to follow each other adding a little to the variation.

### Test 3: On stand low speed

The above tests provide a good indication of how the two controllers operate, but without any friction etc. and operating at an optimal speed they do not really stress things very much. The next test attempts to push things a little more by reducing the speed of the test from 360 to 50 degrees per second. With this test we are operating at the low end of what the motors are actually capable of, so the control systems need to work much harder to keep things moving smoothly. All of the other conditions are the same as for test one. Note however that the scale of the following graphs has changed (due to the reduced range of the velocity). Again the first graph shows the results from the standard leJOS class.

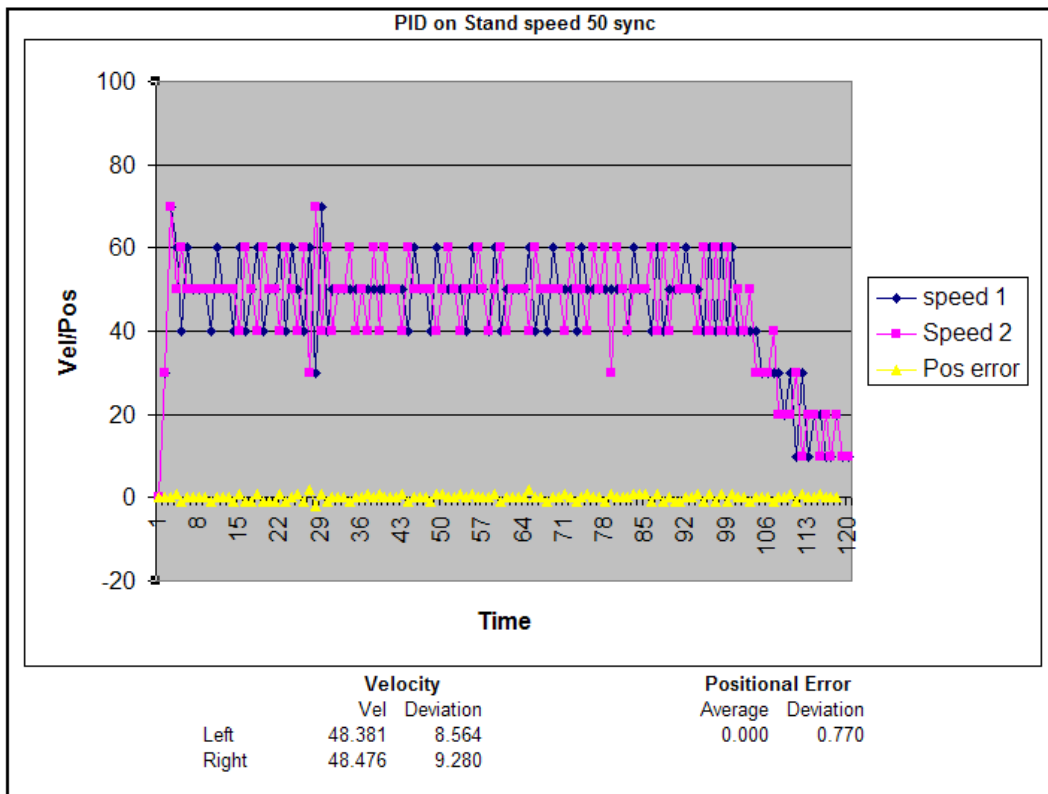


It can be seen that at this speed the regulation is having a harder time removing all of the ripple (the standard deviation of the error plot is over 3 times higher). However overall the velocity is well controlled and centred nicely on the target speed. At this speed the braking at the end of the move works well. Overshoot is well controlled (actually being less than before, but looking more because of the change in scale).



With the PID version the ripple is well controlled (though almost twice as high as before). Overall speed regulation is good. However when slowing to the end point, the position PID probably starts the deceleration too soon, this then requires a number of moves at a very slow speed (10/20 dps) to

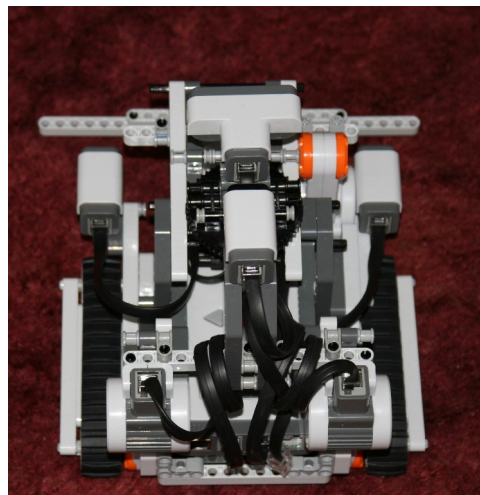
bring the motor to the correct position. The actual movement though while doing this is nice and smooth (if a little slow!).



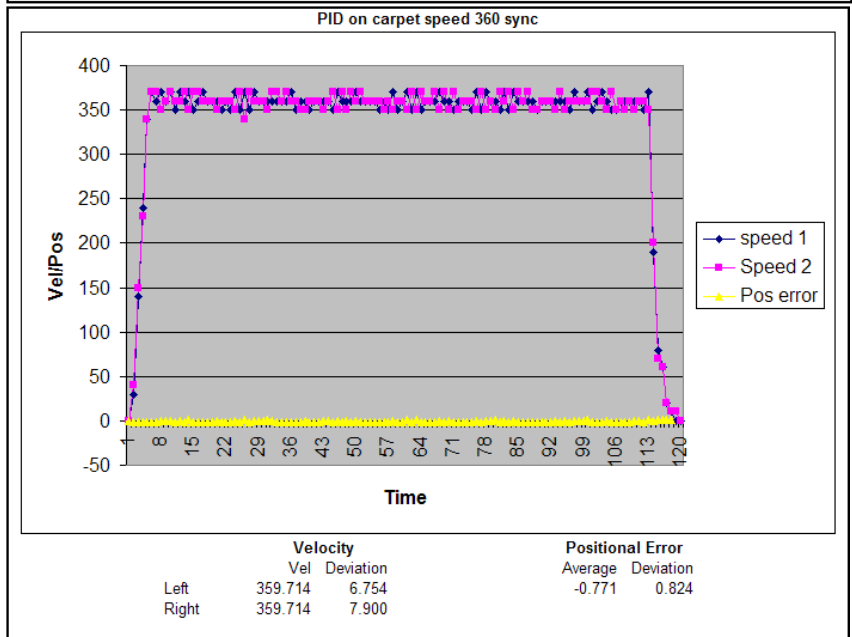
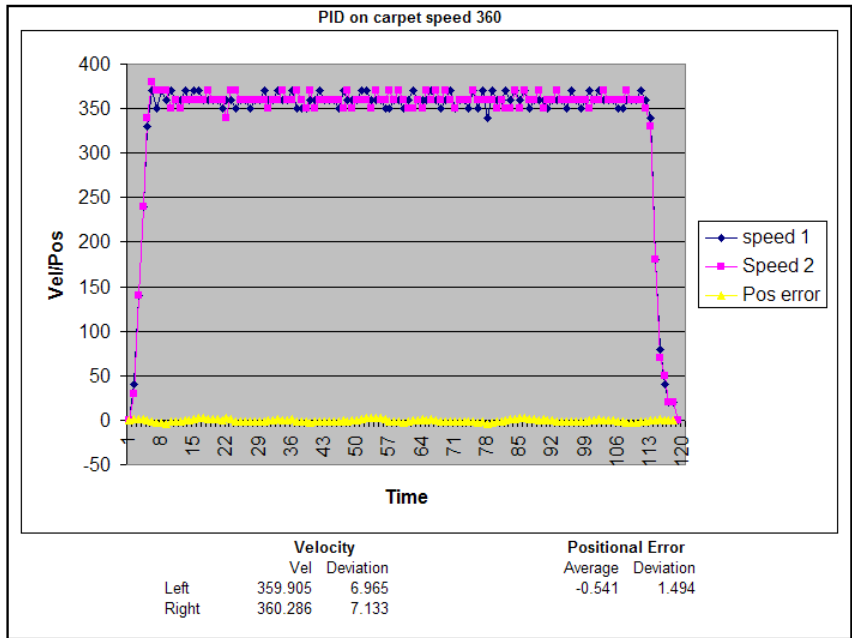
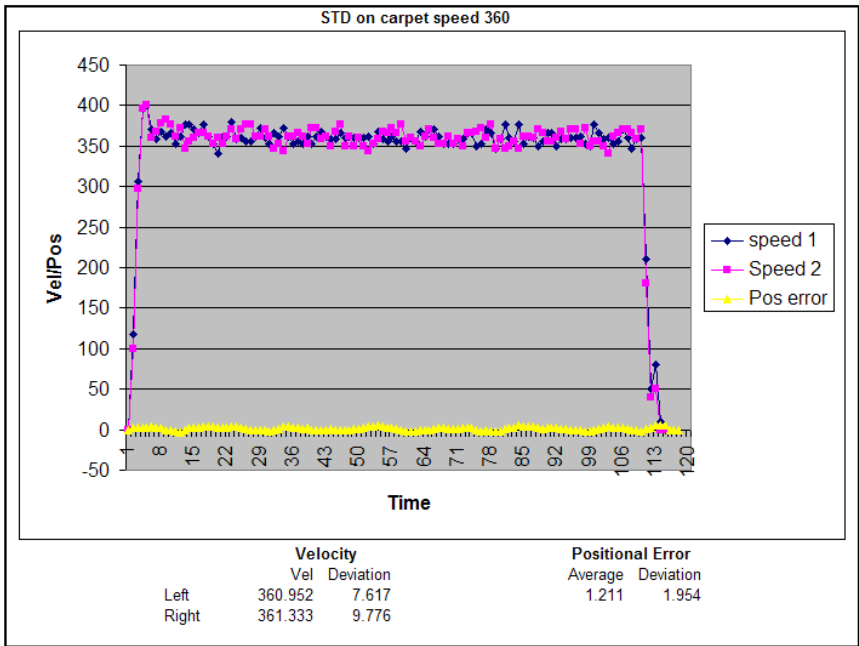
The final results for this set show synchronization added to the PID controller. The impact on the error plot can be seen with the deviation almost halved and the average offset reduced. The way that the synchronization PID forces the two velocities to track each other is also very evident in this plot. Once again the final speed reduction is perhaps a little too slow.

#### Test 4: Test vehicle moving over carpet

For this set of tests we move to a more realistic environment for a Lego robot, movement over a carpeted surface. The actual surface is rather Lego unfriendly having long strands and being somewhat uneven, thus providing a good test environment. All of the tests were run at the default speed of 360 dps. The following page shows the results from this test, following the same format as before. There are no real surprises here with all of the controllers performing well. Again the addition of synchronization can be seen to tighten things up a little for the PID controller.

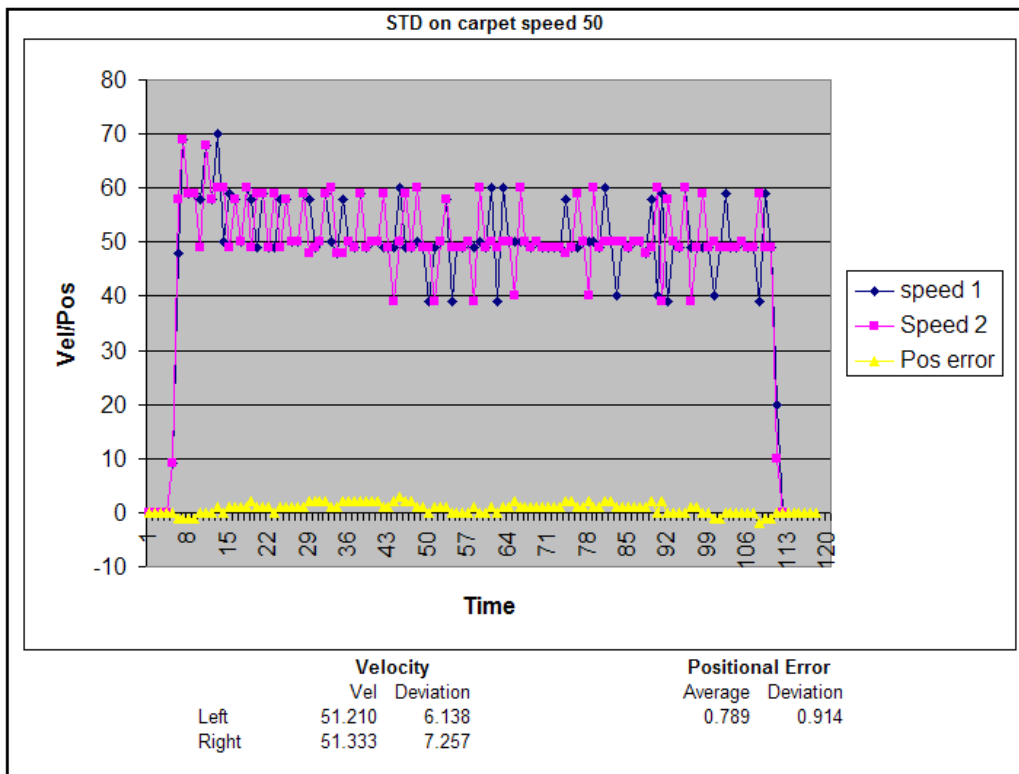


Test vehicle about to move on carpet

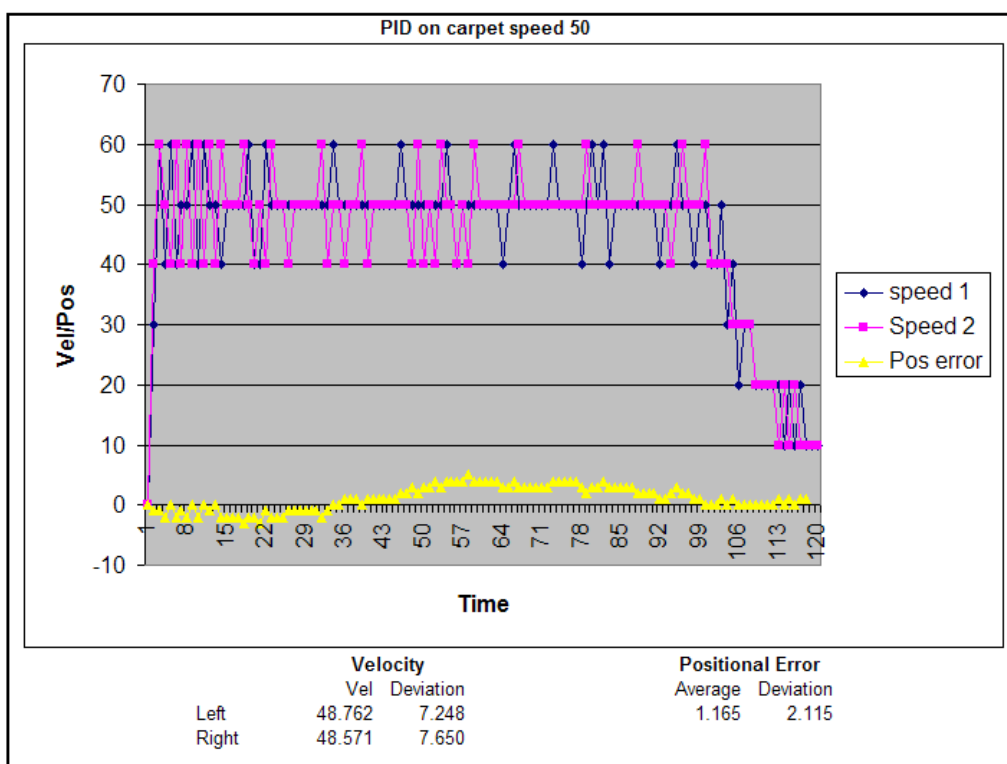


## Test 5: On carpet slow speed

More interesting than the previous results are those shown below from running the test vehicle at a slow speed over the carpeted surface. As before a speed of only 50dps was used.

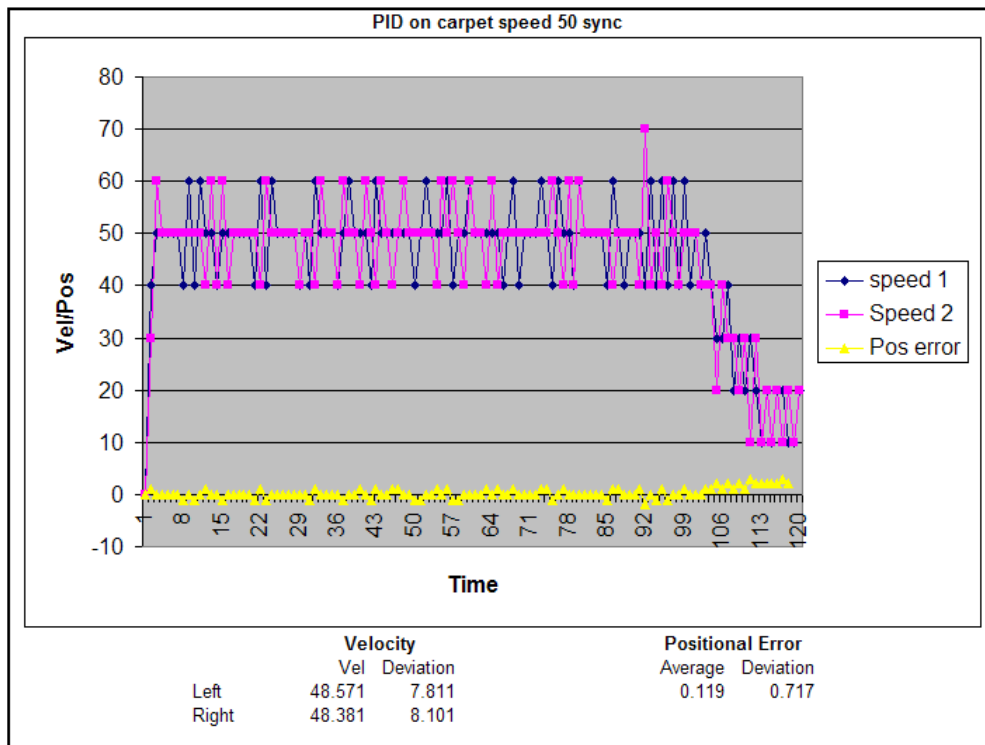


The first surprise is when using the standard leJOS controller. Note how the error term is much lower than when running at the same speed on the stand. The velocity is also much more controlled. It is likely that what we are seeing here is the effect of the additional load on the motors adding a damping factor to the movement. Also note the period of zero velocity at the start of the move, probably caused by insufficient initial power, until the error term builds.



A further surprise is the shown above with the PID based controller showing a relatively high error

term (though note the slightly different scales used for the graphs). Although the velocity is well controlled the positional error seems to accumulate in the mid part of the run.



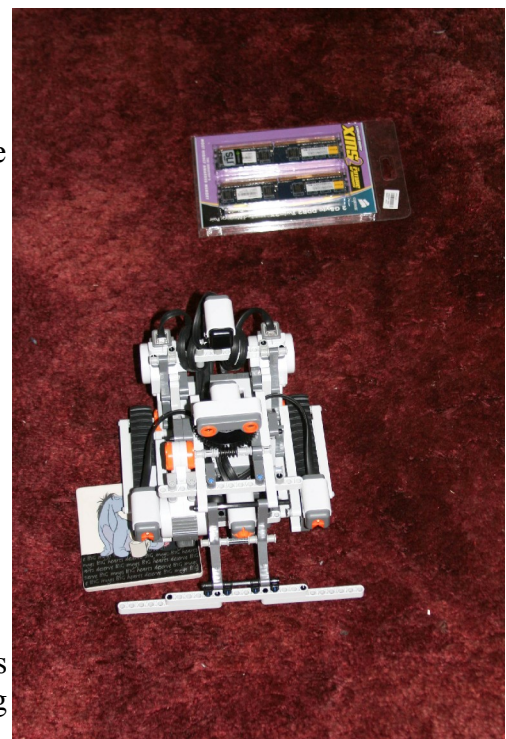
With the final set of results for this test we can see how the addition of synchronization brings the positional error back under control. Though again it can be seen to rise slightly as the vehicle slows indicating that synchronization does not work well during this phase of the move.

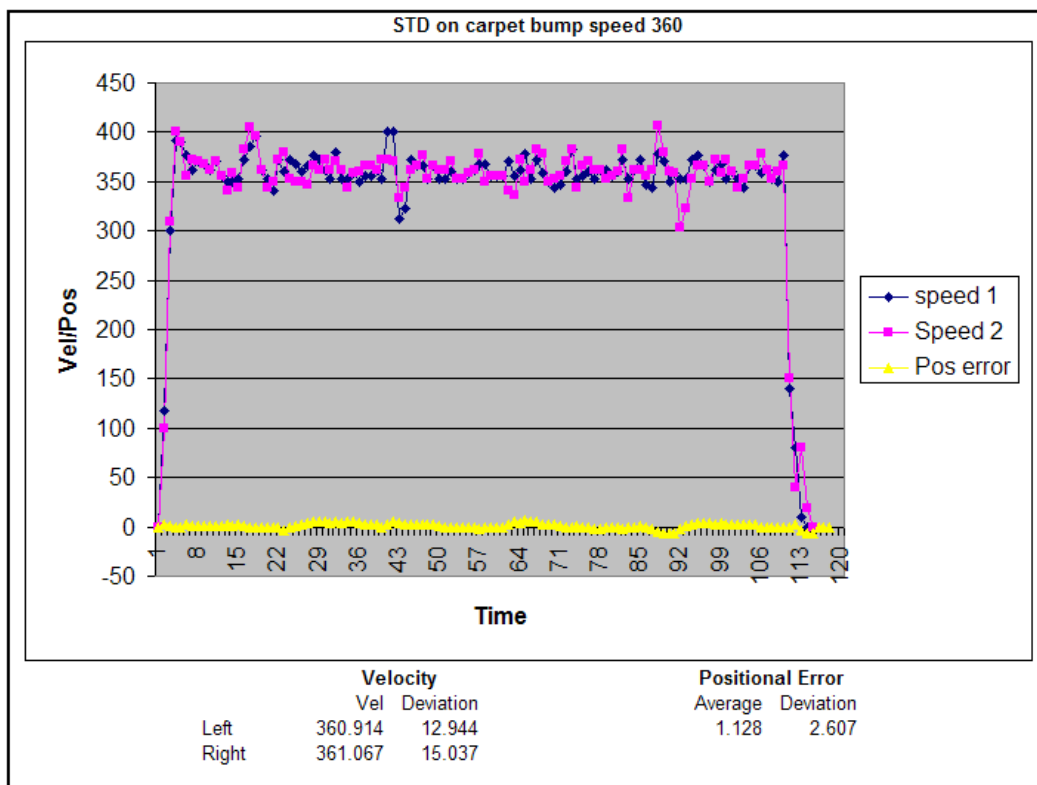
## Test 6: Impulse tests

The final movement test introduces small bumps into the test run, to see how well the systems cope with sudden changes in workload. Two bumps are added one in line with each set of tracks. The first bump is provided by a drinks coaster, and provides a simple flat square section bump. The second provided by some packaging (actually a package for computer memory), provides a series of small ridges. The inset picture shows the vehicle just clearing the first bump and heading towards the second. Again this test was operated at the default velocity of 360dps. Several runs were made to ensure that variations in how the vehicle hit the objects did not change the results. There was very little difference between these runs.

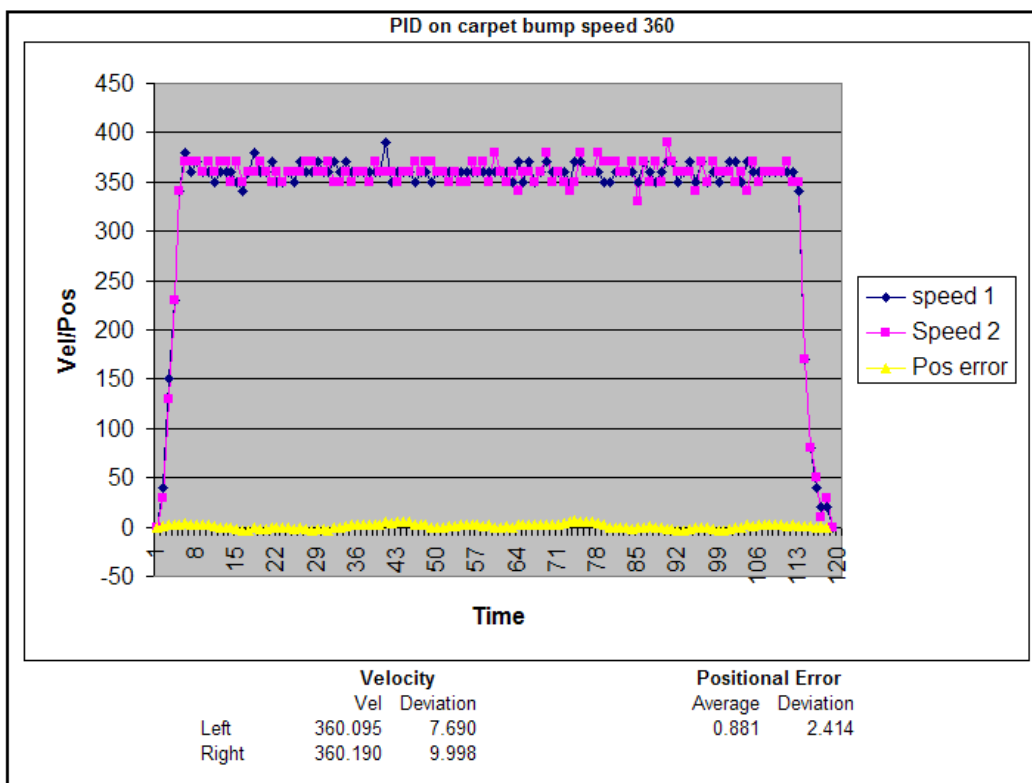
Various other types of tests were tried including ramps and other surfaces but they did not produce any results of note, with both controllers producing good results.

Again the first set of results are for the standard leJOS controller. The first disturbance which starts at around point 15 on the time axis is when the left hand tread hits the front edge of the first object. A further disturbance can be seen as the vehicle drops off from the object around point 43. Further disturbances begin at around 64 as the vehicle encounters the second object and then continue as it bounces over the various edges on the packaging until finally moving



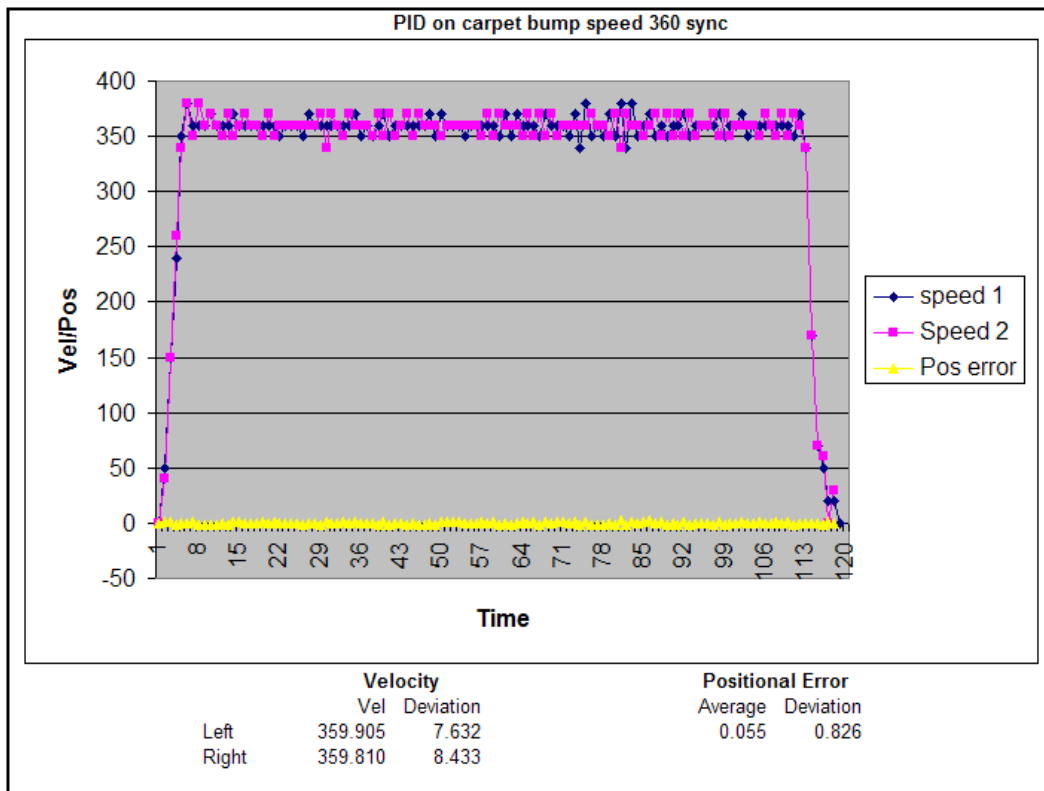


off at around 92. The standard controller copes well with this test, it can be seen how the position error increases at the various bumps, but then is corrected as the vehicle moves on. It seems likely that this correction is provided by using the overall position during the run as the basis for error calculations allowing the system to correct over time.



The results for the PID class show how it handles the actual impulse events with greater control. This is almost certainly due to the use of the extra differential term in the PID control system which allows for rapid reaction to velocity changes. However note that the error deviation is not so good with this test. This controller does not seem able to correct the positional error quite as quickly as the standard controller. This is probably caused by a combination of the error term being based on a

relatively short time sample (which is good for detecting the velocity impulse), and a relatively small integral term being used by the velocity PID.



The final results above show how again the synchronization PID can be used to provide better control of the positional error. The standard deviation in this case being reduced by a factor of 3.

## Test 7: CPU usage

The final test was an investigation of the cpu usage of the two controllers. To test this the standard test program was amended slightly. The main recording loop was modified to run as fast as possible (that is the sleeps and logging were removed) incrementing a loop counter each time around the loop. The number of iterations were recorded every 100ms. The code for this is shown below:

```
int now = (int)System.currentTimeMillis();
int startTime = now;
int loopCnt = 0;
while(m.isMoving() || m2.isMoving())
{
    loopCnt++;
    now = (int)System.currentTimeMillis();
    if (now >= startTime + 100)
    {
        startTime = now;
        log(loopCnt);
        loopCnt = 0;
    }
}
```

To obtain a base value a modified version of the PID version was used. As already noted when operating this version terminates all of the threads associated with the standard motor control class, in addition the PID thread was disabled as were the timer threads used by the motor base class. The PID version of isMoving was modified to return false after 500,000 calls, to allow data to be

collected. All other threads used by the rest of the system were left as is. With all of the motor threads disabled the test loop averaged 469 iterations per 100ms period. This was used to represent 100% of the cpu being available to the main thread. The test was then repeated performing test 2 (on stand 360dps) using each of the controllers and measuring the number of loops available to the main thread. In the case of the PID controller the synchronized version of the test was used as this is likely to consume more cpu (due to the additional PID operations). Using the standard controller the main thread averaged 262 loops indicating that the controller was consuming approximately 44% of the available cpu. When using the PID controller the test loop averaged 364 loops indicating that this controller consumes approximately 22% of the available cpu.

These results are a little surprising (and the method used to determine cpu usage may not be the best), so we should look in a little more detail:

- The standard regulator uses a thread for each motor (and all three threads will be in the system even if only two motors are in use, this was confirmed by terminating the thread associated with the unused motor and re-running the test, the cpu usage dropped).
- In addition each motor has an additional timer thread associated with it (though this is mainly in a sleep state waking every 100ms).
- The regulator loop runs continuously (using yield to allow other threads to run), when operating the thread makes a number of calls to the firmware (to obtain tachometer counts and system time), it also performs a number of floating point operations.
- Other benchmarks (notably the speedtest sample), also run faster when the motor regulation tasks are disabled indicating that these results may be correct.
- The PID regulator only uses a single thread.
- The thread only executes code every 20ms (sleeping for the remaining part of the time).
- The calculations performed by the PID regulator are all integer operations.

## Conclusions and further work

The major conclusion to draw from these tests is that both controllers work really well. Both are capable of providing a high degree of velocity control, while minimizing positional errors, both are capable of bringing a rotating motor to a halt at a specified point. From the tests presented here it would seem that the PID based controller can provide some improvements in positional error control over the standard controller (particularly when using the additional synchronization PID). However the gains seem to be rather small and it is unclear if they are large enough to warrant replacing the current system. In particular the current system has been in use with many different models and in many different systems and seems to operate well, the PID system has only been used in much more limited situations. So my overall recommendation would be to keep the existing system.

Having said that there are a number of areas that seem should be investigated for both implementations. They are listed below in no particular order starting with the current leJOS controller:

- Consider the use of brake mode for the motor PWM control. It seems to offer a smaller deadband, has better characteristics when slowing a motor and (because of the smaller deadband) offers more “useful” power steps.
- Investigate the cpu usage of the system in more detail. It would appear that the current motor control system can use a large percentage of available cpu. It would be good to find a way to validate (or refute) the results described here.
- Consider if it is practical to include some of the PID elements in the existing system. Adding

a differential type term may improve the response to impulse events. It may be possible to use a PID style position control system rather than the existing stop and nudge mechanism.

- Consider if any form of cross motor synchronization would be beneficial.

For the PID based controller

- Complete the implementation to allow the PID controller to be a drop in for the standard system to allow testing with a wider range of models and environments.
- Investigate potential improvements to the current positional control when operating at slow speeds to avoid the current slow move to final position.
- Investigate the interaction between the synchronization PID and the Position PID while slowing the motors.
- Provide a more generic access mechanism to the synchronization PID to allow error terms from other sensors (compass etc.) to be used to control directional corrections.

There are probably many more but that would seem plenty to be getting on with!

## **Acknowledgments**

Many thanks to the leJOS team for providing the NXT Java environment, which has made this work possible (and fun!). Particular thanks to Roger Glassey the main author of the current leJOS motor control system, which inspired this work.

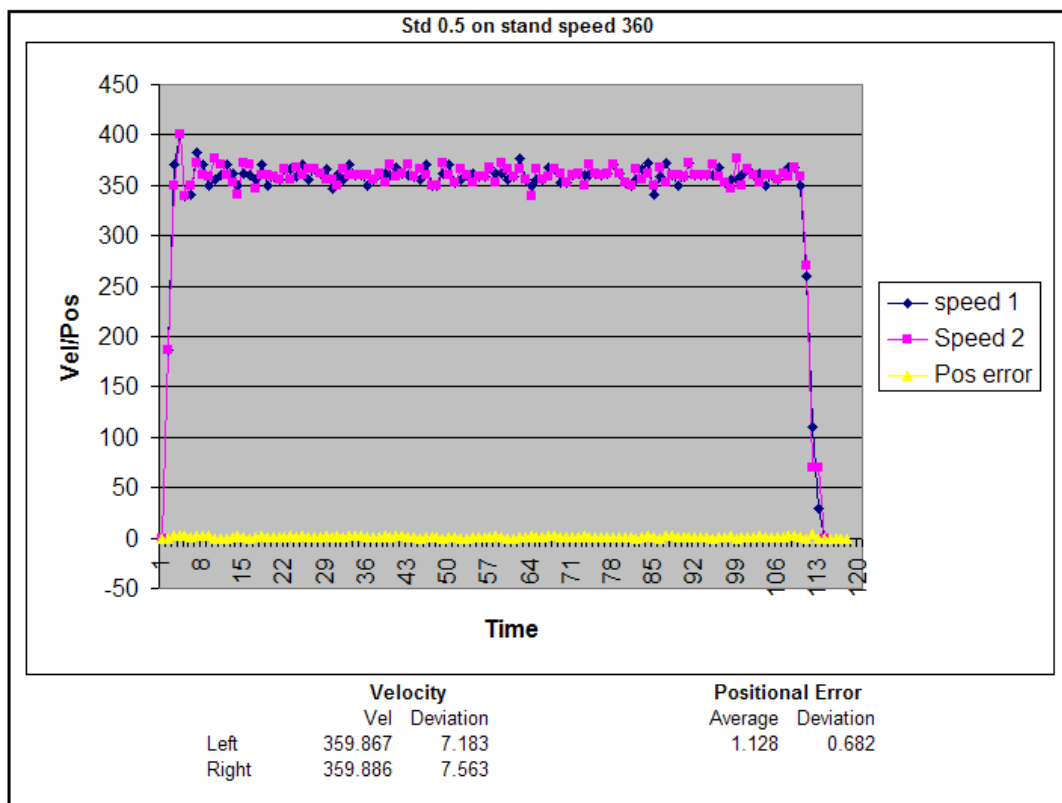
## Update: leJOS 0.5

The motor control system has been updated for the latest build of leJOS (currently likely to be released as leJOS version 0.5). This version has two major changes:

- The constants  $K_p$  and  $K_i$  have been tuned, providing a more to provide a faster reaction to velocity errors.
- New code has been added to “kick start” the movement process this takes the form of a loop which applies full power and then waits until the motor has begun to turn before exiting to the normal control process.

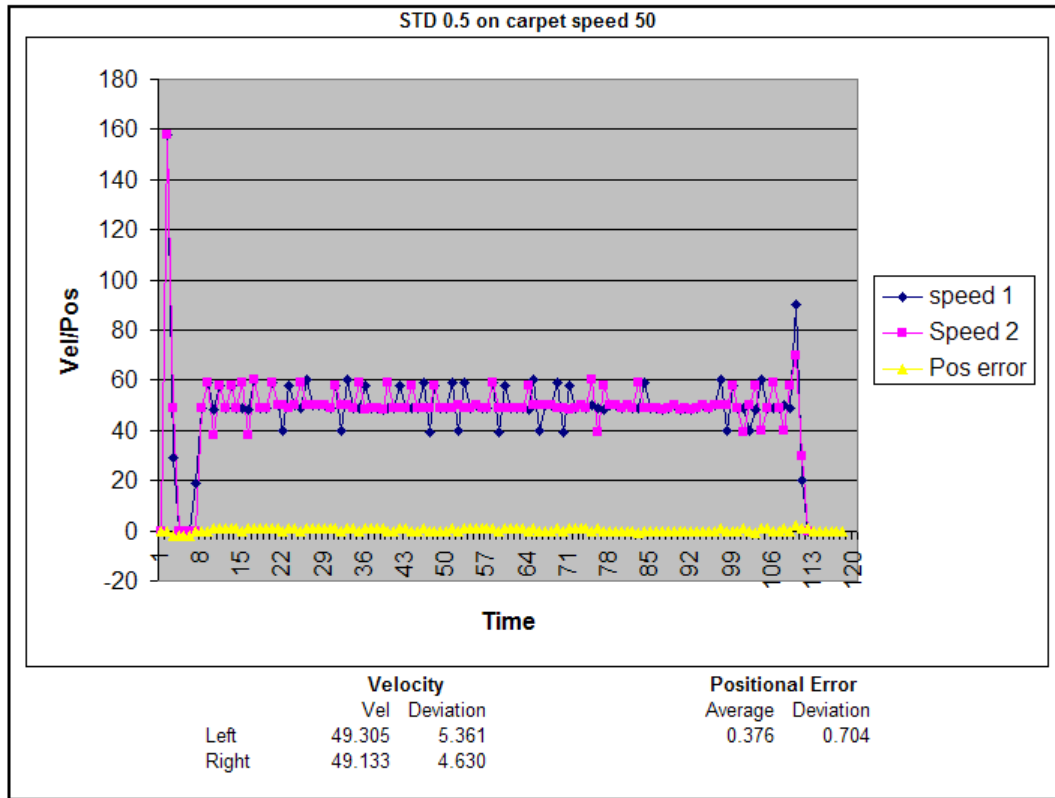
To evaluate the impact of these changes all of the tests have been re-run. The more interesting results are shown below. In addition a number of “high speed” tests have also been conducted using a velocity of 720dps. The results of these tests are consistent with the other results presented in this document and are not therefore presented here.

In general the new changes have resulted in a large improvement in the overall speed regulation of the system. As can be seen from the results for the “on stand velocity 360dps” test shown below, the ripple removal has been much improved.



The only small downside in the above result is the initial overshoot and slight ringing at motor start up. However this seems to be well controlled and is quickly damped.

Again it is the low speed runs which show the more interesting results. The results for a re-run of the “on carpet 50dps” test are shown below.



The overall error control is much better. However a problem can be seen at the start of the move. The new “kick start” code can be seen making the motors move, this seems to generate a large overshoot. But the bigger problem is that after the initial “kick start” the motors do not continue to move. Instead they stop and remain stopped until the error term builds sufficiently to make the motors move again.