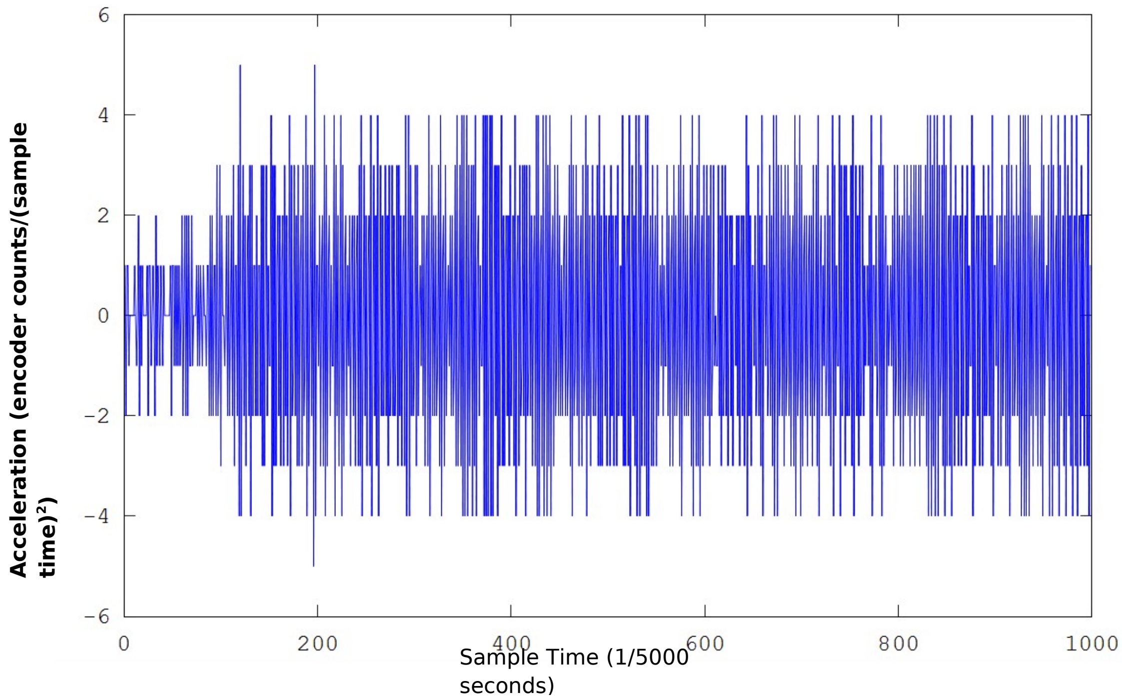
Estimating Angular Acceleration from Magnetic Encoder Measurements

Introduction and Motivation

In many control applications, accessibility to all three states of motion – position, velocity, and acceleration – can allow for greatly improved system responses. By placing gains on the feedback path of each state, a controls engineer can create an optimized control law by selecting the best location for the system's dominant poles. In many cases, such a controller, which often happens to be a linear-quadratic regulator, can provide the engineer with a simple yet elegant means of controlling an extraordinarily complex plant. In addition to full state feedback, the availability of velocity and acceleration values can also aid in numerous other controls endeavors, such the identification of various inertias through acceleration measurement. For these reasons and many others, it is important to provide a motor's user with reliable acceleration and velocity output.

The Challenge of Acceleration Estimation

By far, obtaining clean acceleration output is the most difficult aspect of achieving full state feedback. Because the position values from the magnetic encoder must be differentiated twice, a great deal of noise is injected into the acceleration signal. In most cases, a simple filter could achieve reasonable velocity output, since the noise contribution from a single differentiation is somewhat minimal. Therefore, the focus of this research endeavor will be to obtain reliable acceleration values.



An example of twice differentiated position data obtained via magnetic encoder

Weighted Linear Regression

One of the first techniques attempted in the effort to achieve the estimation of the motor shaft's velocity and acceleration was Weighted Linear Regression.

The method to formulate running linear regression is as follows:

Define the instantaneous position to be a parabola of the form,

$$p = p_0 + vt + \frac{1}{2}at^2$$

where the acceleration is assumed to be constant over a given window of points. In the case of this experiment, the window size was twenty samples long, the exact number of encoder values recorded at a sample rate of 5kHz between every other cycle of the outer 1 kHz control loop. In this equation p indicates position, v indicates velocity, a indicates acceleration and t represents time.

The goal of linear regression is to approximate the coefficients p_0, v, and a. This process is started by forming the matrix Φ, which consists *i* rows of the values 1, t_i, t_i^2. In this specific case, *i* is equivalent to 20.

The next matrix essential to the linear regression algorithm is the matrix θ, which is merely equivalent to:

$$\begin{bmatrix} p_0 \\ v \\ a \end{bmatrix}$$

Now, finally, can the measured values *p* be related to the coefficient matrix θ, as shown below:

$$(\Phi^T\Phi)\theta = \Phi^T p$$

Thus, to solve for θ, two additional sub-matrixes *A* and *b* may be formed:

$$A = \lambda(\Phi^T\Phi),$$

where λ is a one-dimensional matrix of length *i,* consisting of tuned weighting factors, and

$$b = \lambda(\Phi^T p).$$

The matrix equations now simplify to $A\theta = b$, which can be simply solved as $\theta = A^{-1}b$.

To compute the *A* and *b* matrixes more easily, their formation can be simplified to two summations, which are described below:

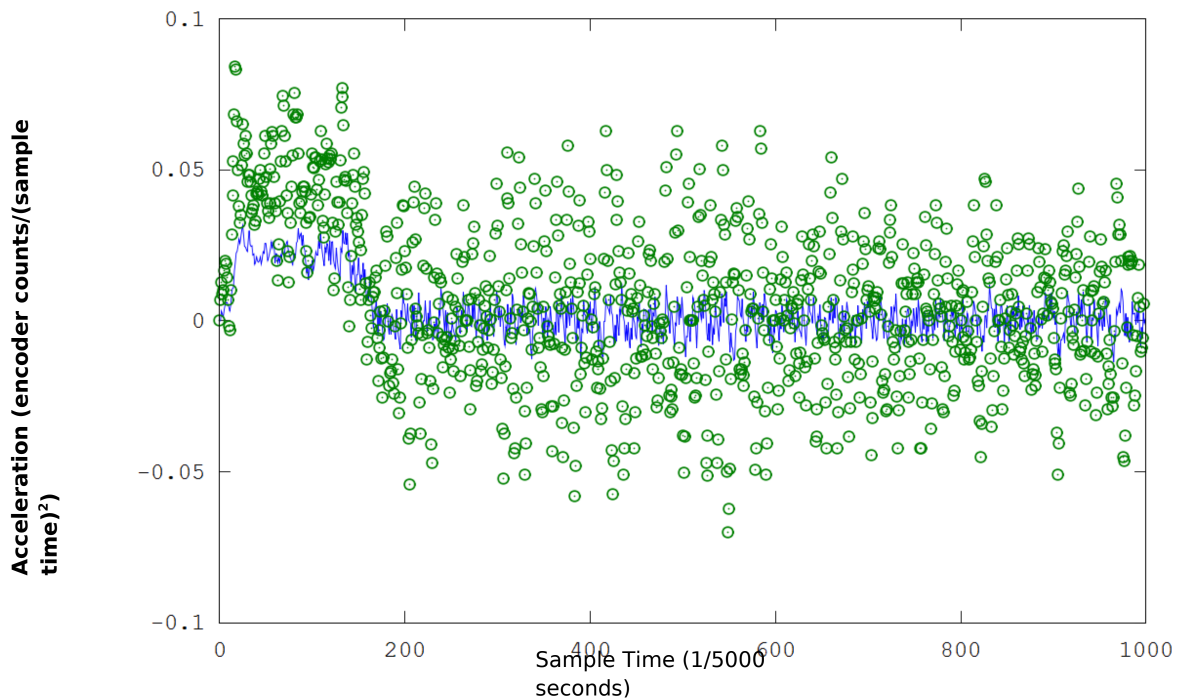
$$A_{ij} = \sum_{k=1}^{k=20} \lambda_k \left(\Phi_{ki}\Phi_{kj}\right)$$

$$b_i = \sum_{k=1}^{k=20} \lambda_k \left(\Phi_{ki} p_k\right)$$

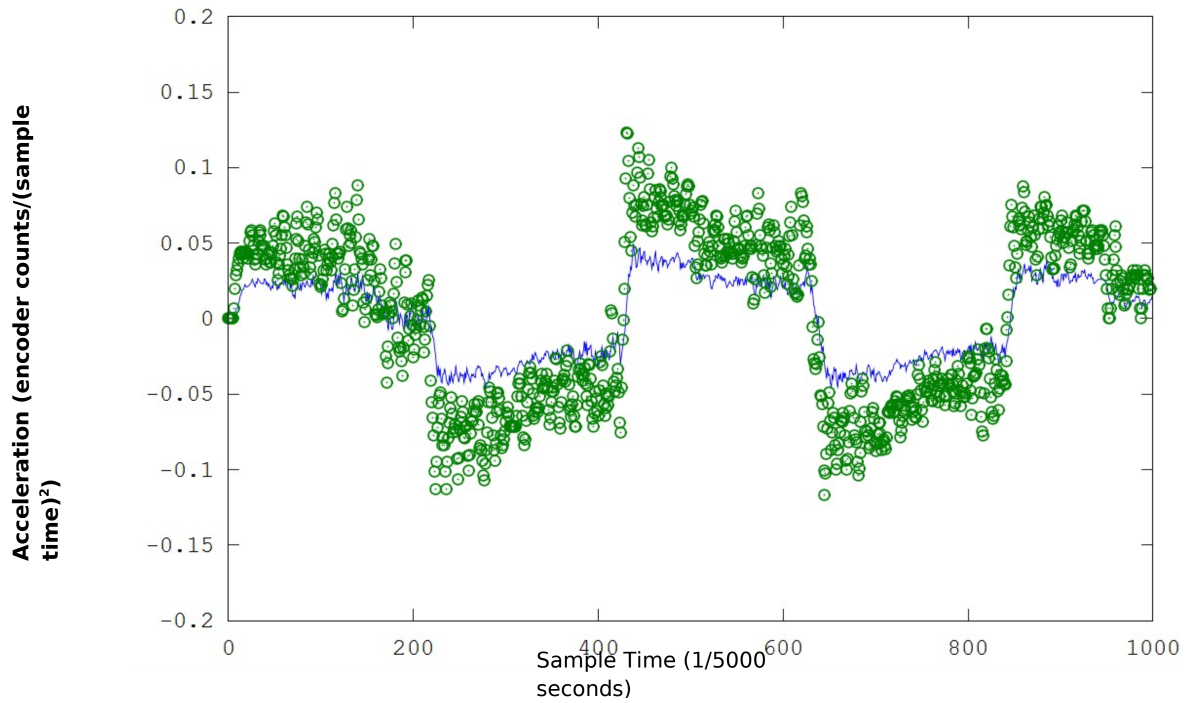
NOTE: In the above equations, k represents the numbers from 1 to the length of the moving window, and both *i* and *j* represent the dimensions of the computed matrixes (In this case, 3 and 3 respectively).

This algorithm was implemented first using Octave and then in C. However, it was found that this algorithm was too computationally intensive and yielded too poor of results to be the best way to obtain accurately estimated acceleration data. See "linear_regression.m" for Octave implementation and test trials. The c-code has also been included (linear_regression.c) but may not be suitable for compilation. Furthermore, one may also try adjusting the weights in the linear regression file. It is important to notice that the weights provide little difference in output acceleration values and tuning all twenty weights by hand is an extremely tedious process.

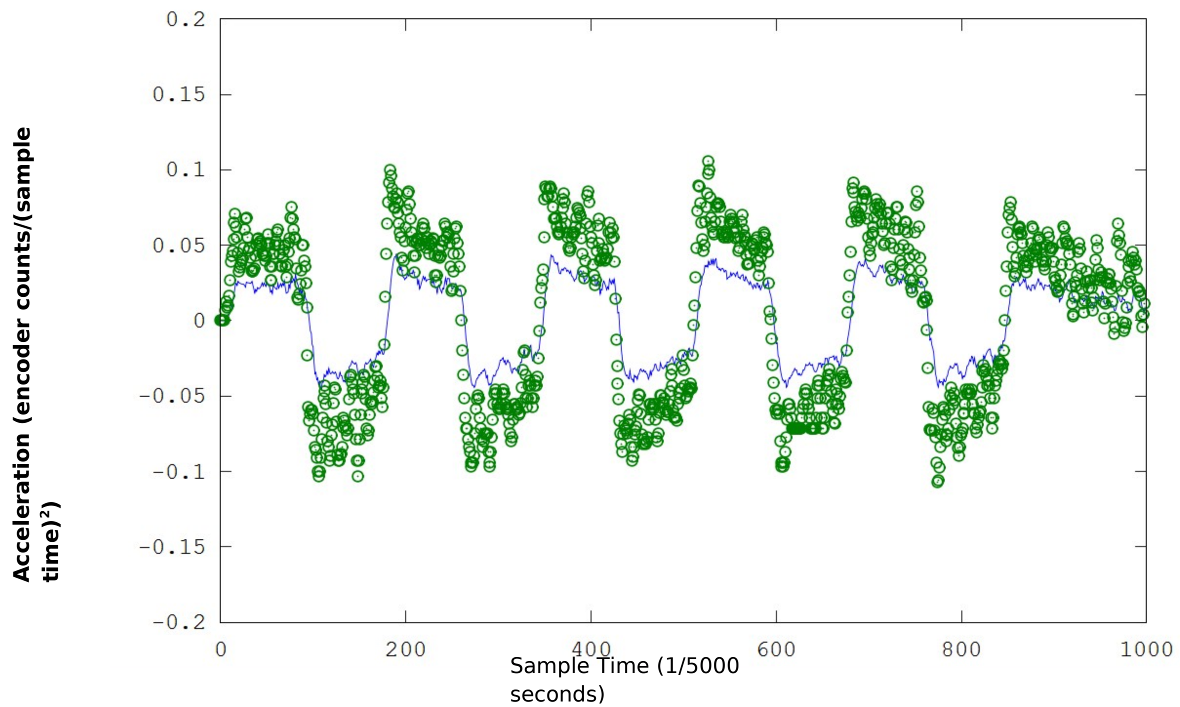
Note: For all figures depicting acceleration, green circles represent the simply filtered (FIR) accelerations as shown in the Octave testfiles, and the blue plots represent the results of the given acceleration estimation algorithm.



Linear Regression Algorithm: Step Input



Linear Regression Algorithm: Medium Frequency Square Wave Input



Linear Regression Algorithm: High Frequency Square Wave Input

FIR and IIR Filtering

After the weighted linear regression proved to be unsatisfactory, a second method involving digital filtering was investigated.

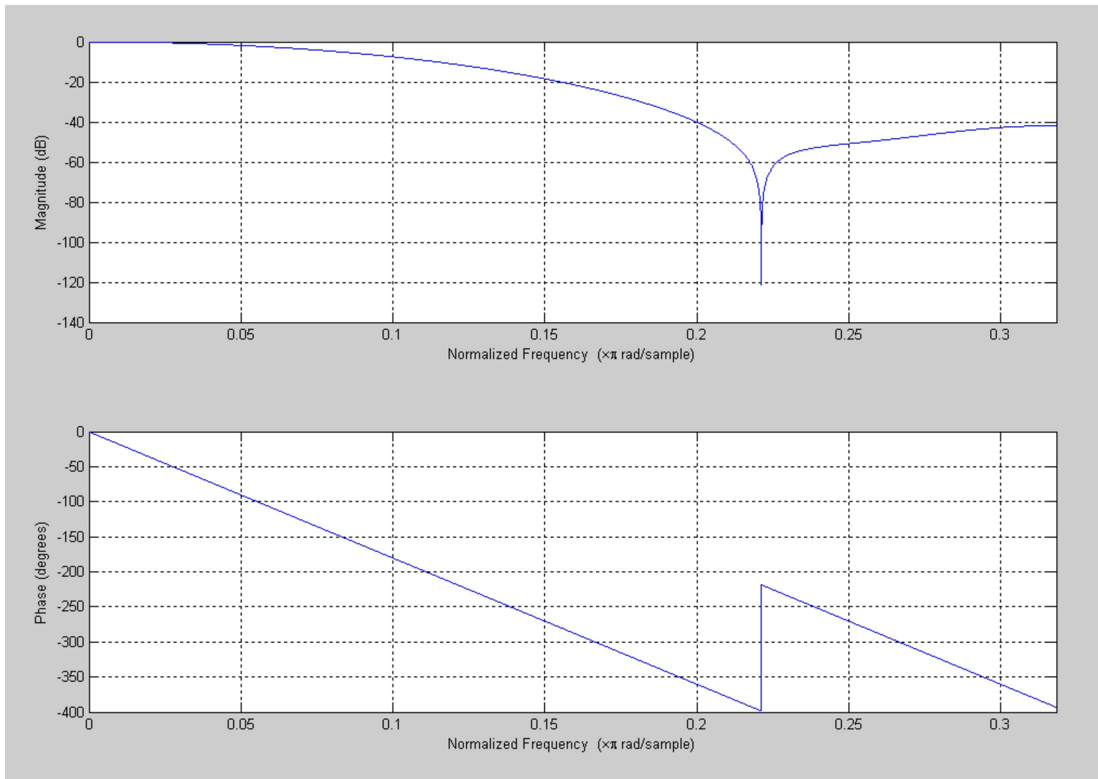
The simplest method of digital filtering involves the use of a moving average filter. A moving average filter is a FIR (finite impulse response) filter whose coefficients are all equal to one. The only difference between an averaging filter and an FIR filter is that the coefficients in the general FIR filter are allowed to vary to achieve better performance.

In explaining the FIR filter, it is easiest to describe the algorithm itself, which consists of a single, moving summation. In the discrete domain, the filter can be described as,

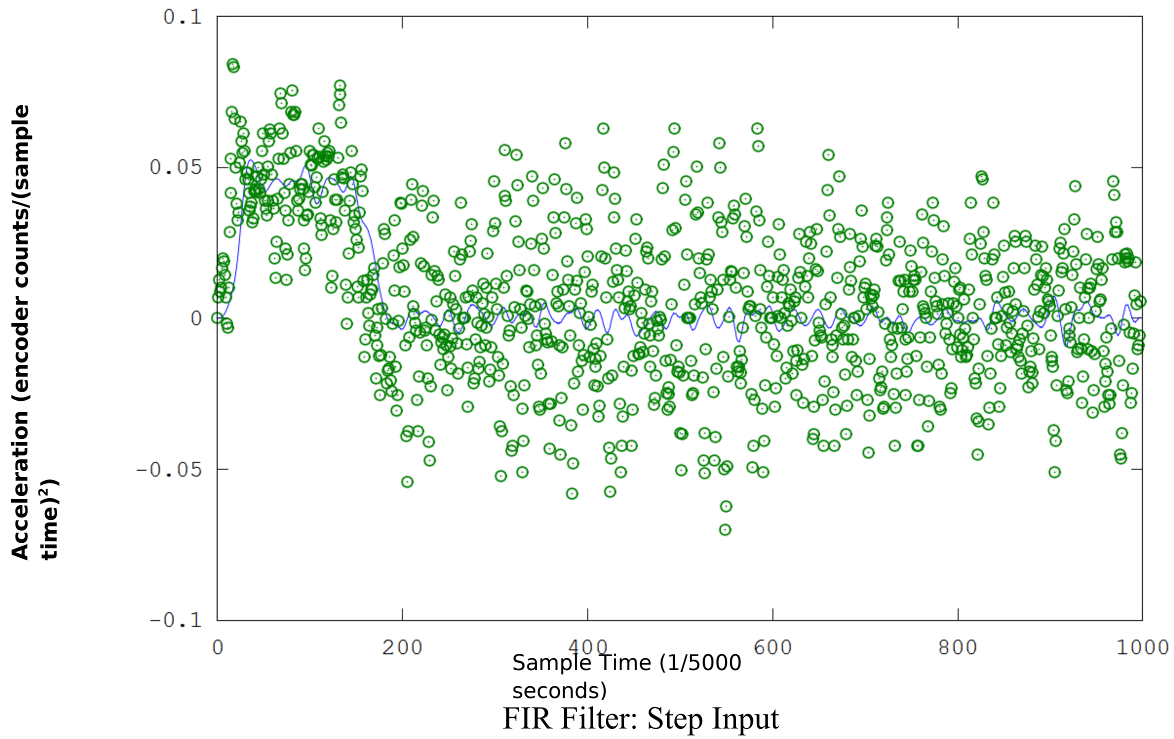
$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots b_N x[n-N]$$

where $b_0 \ldots b_N$ are the filter coefficients, y is the output, x is the input and N is the filter order. There are many benefits to the FIR filter. The first benefit is that the filter can be implemented very easily in software with a simple code. Another benefit is that the phase response varies linearly with increasing order, which leads to decreased delay and improved delay prediction. In this investigation, two FIR filters were placed in cascade to improve the performance of the FIR filtering technique. Although this action increases the delay of the acceleration signal slightly, it does not seem to impede on overall performance.

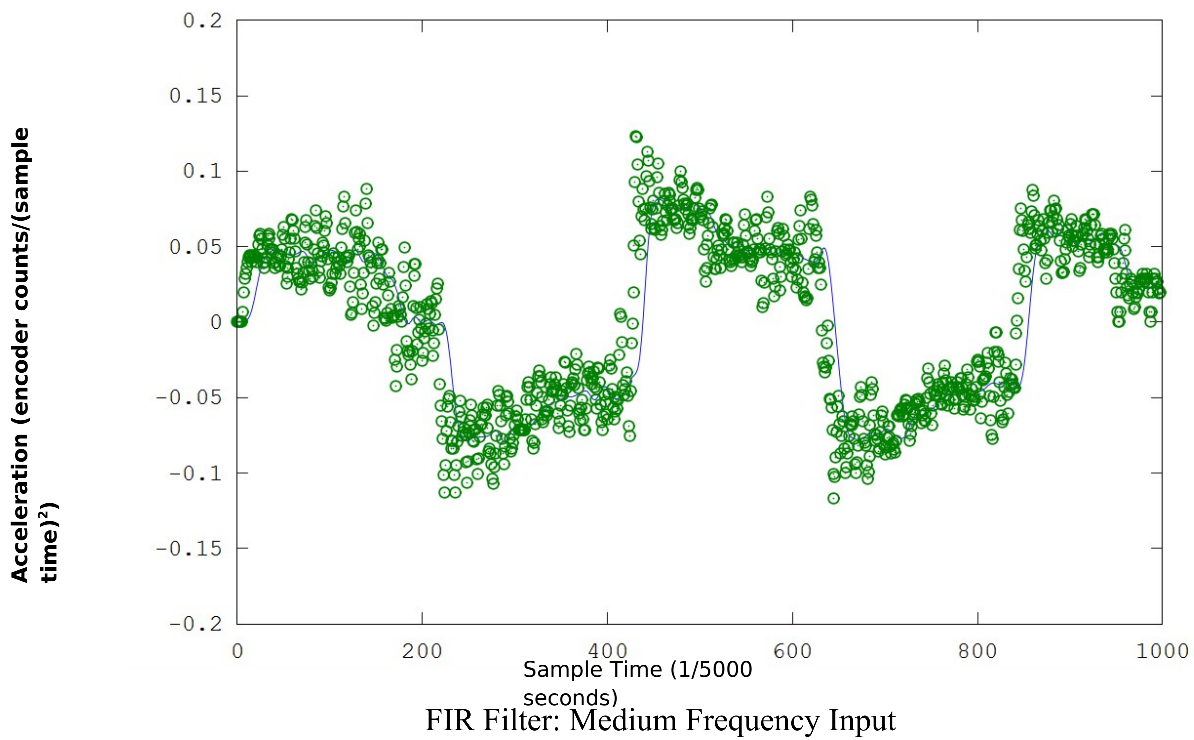
To carry out filter design, MATLAB has a built in command "fir1", which allows one to receive a set of FIR filter coefficients from the parameters cutoff frequency and filter order. The specific code used for filter design in MATLAB is contained in the MATLAB only file, "filters_matlab.m". In this investigation, a filter order of 20 was selected along with a cutoff frequency ratio of 0.02 (cutoff frequency divided by twice the sample time, i.e. (50Hz/(2*5kHz))). The plot below depicts the Magnitude and Phase response of the designed 20th order FIR filter. Notice how the phase decreases linearly with increasing frequency.



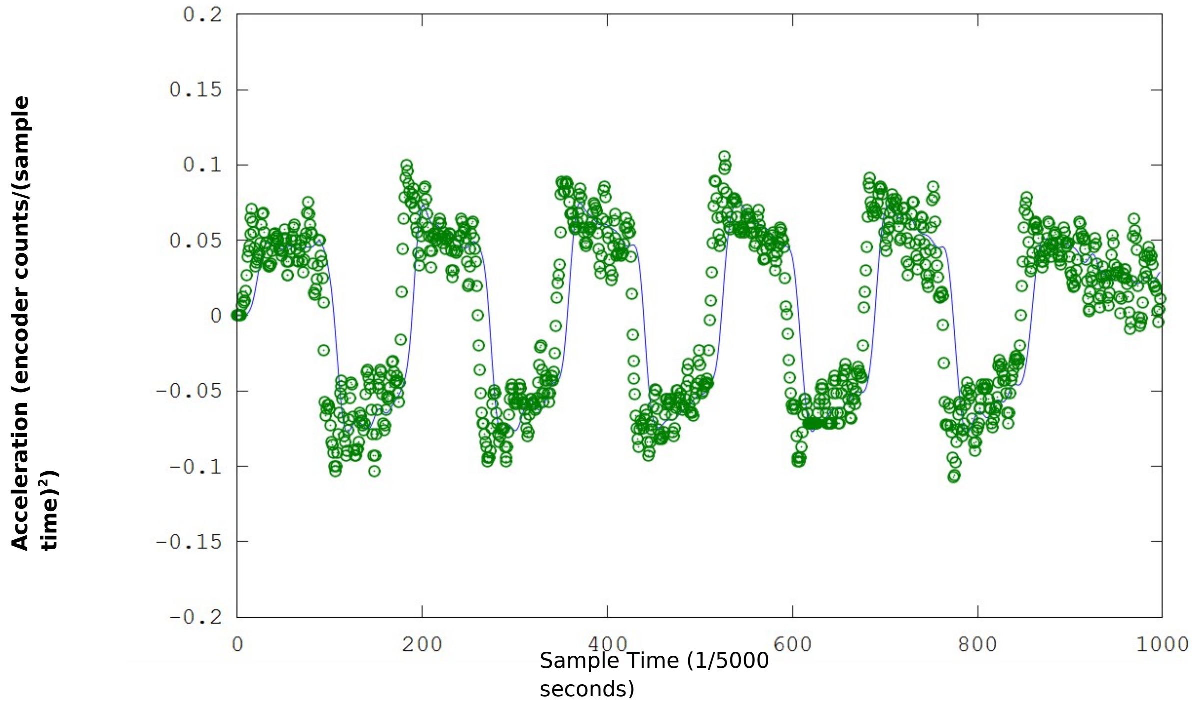
Once designed, the FIR filter was implemented using real data in Octave. The results of this implementation are shown in the plots below. It is important to note that the double differentiation takes place before the filtering in an effort to eliminate the possibility for overflow (acceleration values are typically much smaller than position values). Preliminary tests reveal that an extremely high order filter is needed to output reasonable acceleration values. In addition to increasing delay, a high order filter requires a significant amount of storage, especially when the integers being used are 32 bits in length. One can also observe the increased delay with increasing frequency. For these reasons, the FIR filter was implemented in C-code on a DSP, but only to provide velocity output. See “fir.m” to examine the Octave implementation of the FIR filter. “FIR.C” in the DSP subfolder implements the FIR filter in C.



FIR Filter: Step Input



FIR Filter: Medium Frequency Input

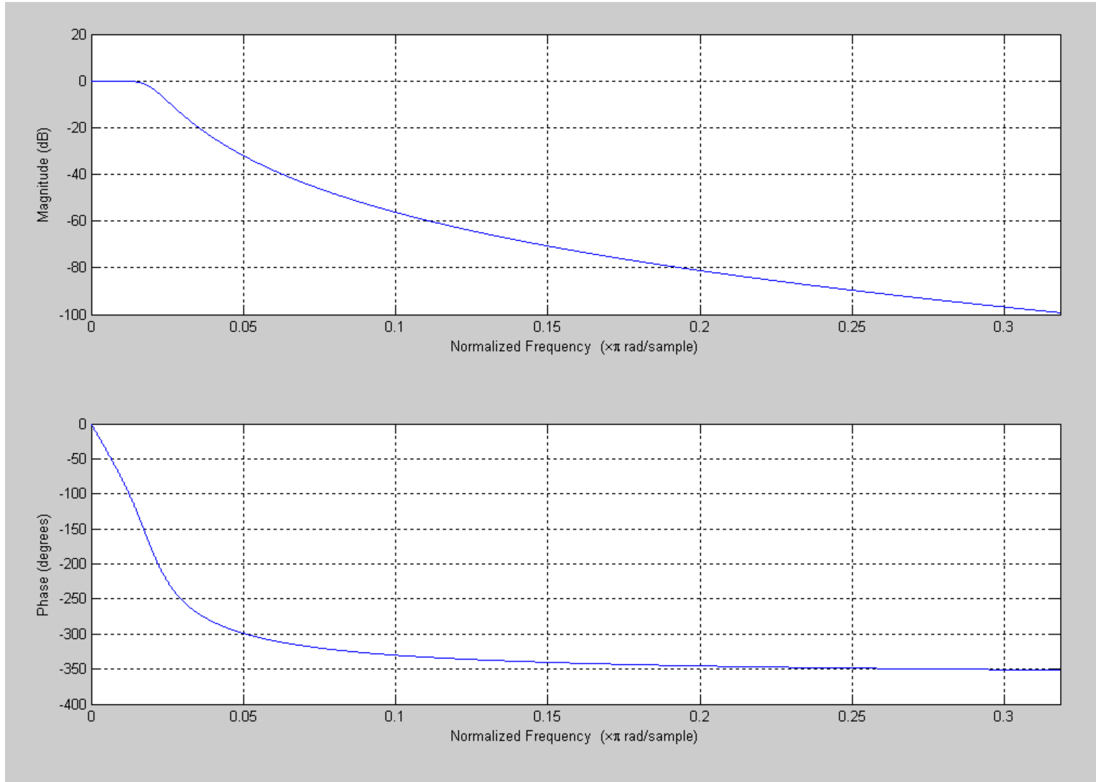


FIR Filter: High Frequency Input

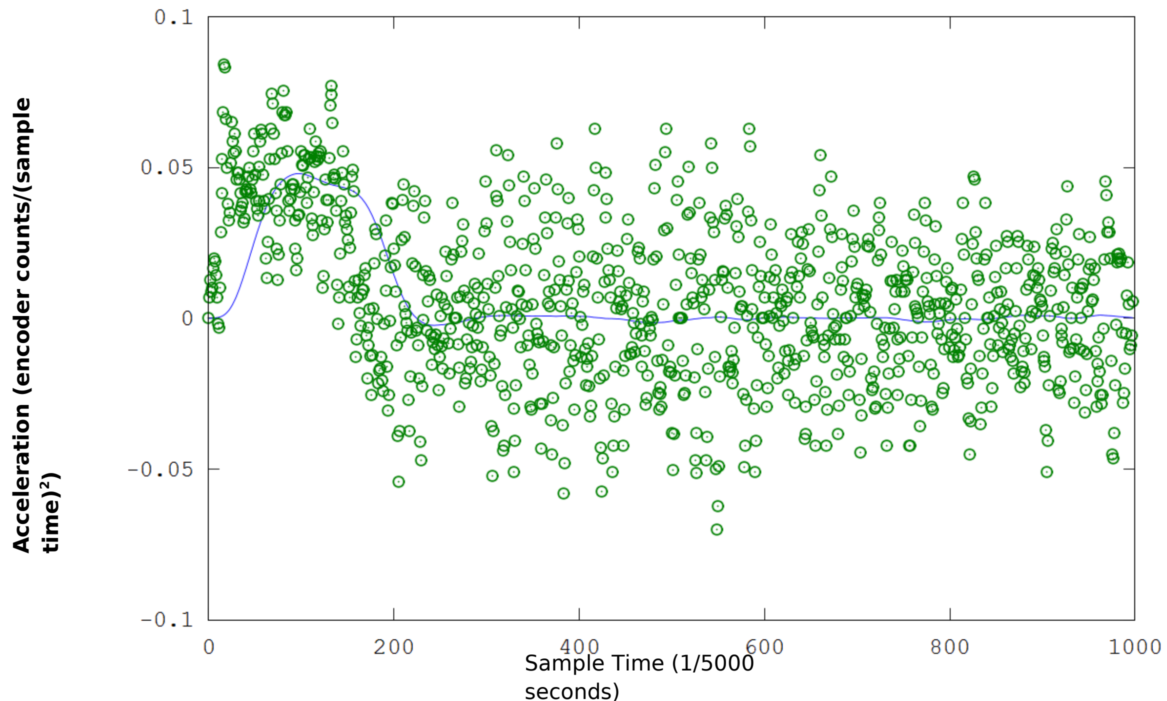
The second digital filter investigated for the purpose of extracting acceleration data was the IIR (infinite impulse response) filter. The IIR filter differs from the FIR filter in that it not only takes into consideration the input to the system, but it also considers the past filter outputs. The difference equation for the IIR filter is as follows,

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots b_N x[n-N] - a_0 y[n] - a_1 y[n-1] - \cdots a_N y[n-N]$$

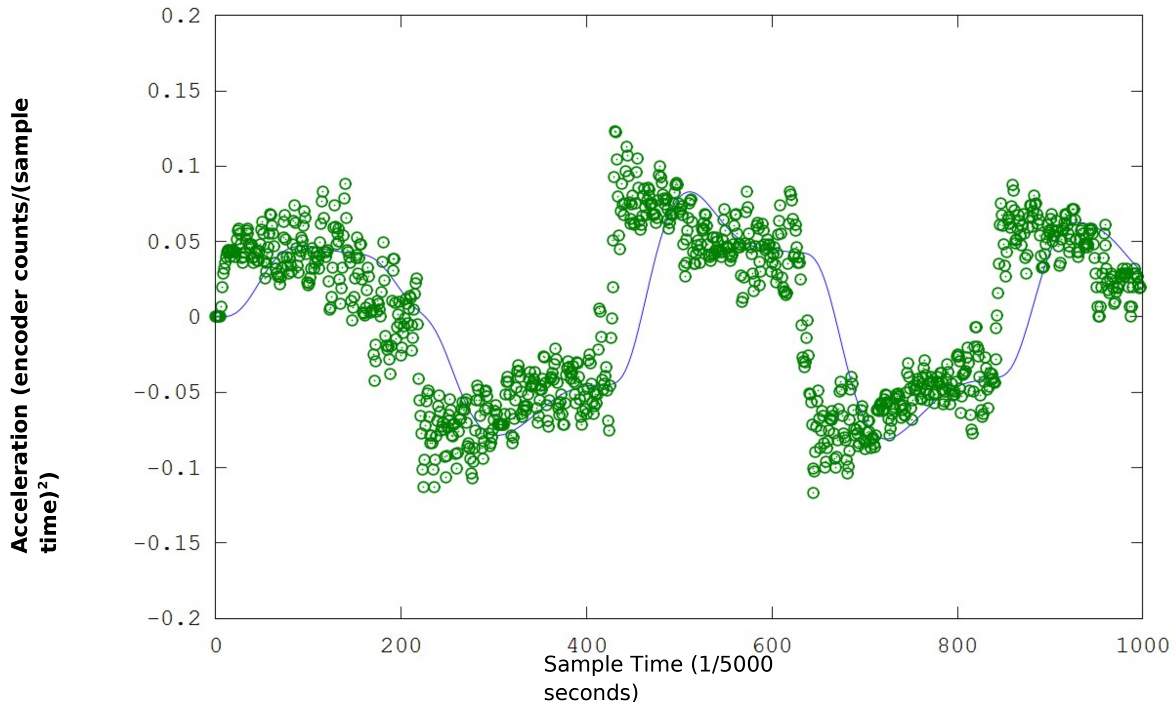
where y is the output, x is the input, $b_0 \ldots b_N$ are the input coefficients $a_0 \ldots a_N$ are the output coefficients and N is the filter order. Fortunately, as in the case of the FIR filter, MATLAB provides software to design IIR filters. Because of its extremely smooth frequency response, a 4th order Butterworth (MATLAB Command "butter") filter was designed to filter the acceleration data at 50 Hz, which corresponds to 0.02 in the digital domain (50Hz/(2*5kHz)). Once the 4th order response was seen to have desirable characteristics, the filter was implemented in C-code as two cascaded 2nd order filters. By cascading the filters, one helps reduce the occurrence of massive integer overflow. Although the Butterworth filters seemed to be an excellent solution for obtaining acceleration data from position measurements, it was soon found that they created a significant amount of signal delay at higher frequencies (phase increases exponentially with increasing frequency) and created a much more rounded response, getting rid of not only the high frequency noise but also the high frequency changes in acceleration which could be important in more advanced controls situations. The magnitude and phase plot of the Butterworth filter can be seen in the plots below. Notice how the phase increases exponentially with increasing frequency.



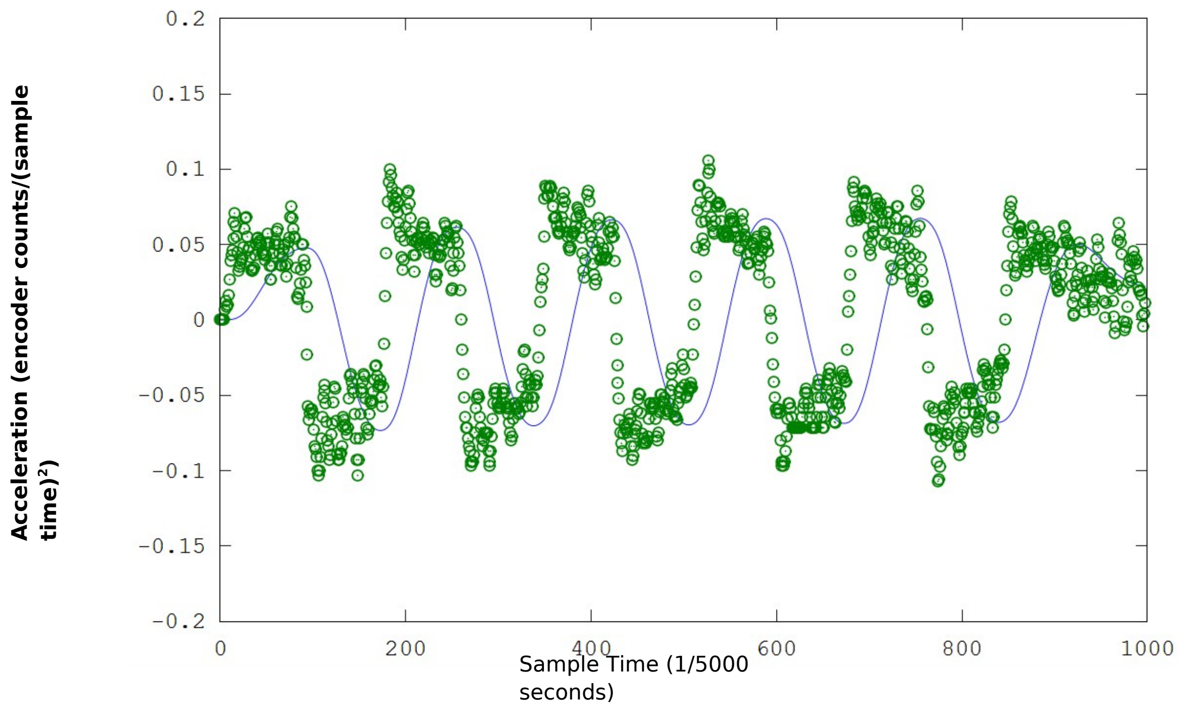
Once the Butterworth filter coefficients were obtained, the filter was implemented in Octave. The results of this analysis of can be found in following plots, which were generated by running the file "bw.m" in Octave. The filters themselves have been implemented in the file BW.C.



Butterworth Filter: Step Input



Butterworth Filter: Medium Frequency Input



Butterworth Filter: High Frequency Input

Kalman and Alpha-Beta-Gamma Filtering

The final attempt to extract acceleration data from the magnetic encoder measurements involved the implementation of a Kalman/Alpha-Beta-Gamma Filter. The Kalman filter is a recursive filter which uses a model and collected data to estimate the actual values of the states occurring in a given system. The Kalman filter hinges on the fact that the values of the states lie somewhere between the model values and the measured values, and tries to pinpoint the location of these state values through probabilistic and optimal methods.

Before beginning the explanation of the Kalman filter, a review of discrete state space will be attempted. Note: All of the following work assumes linear systems.

All linear systems can be represented in the form,

$$x[k] = A_k x[k-1] + B_k u[k]$$

$$y[k] = C_k x[k] + D_k$$

Where x[k] represents a marix of current state values, x[k-1] represents the previous state values, u[k] represents the current input, Ak and Bk are matrixes which represent the model of the system, Ck determines which of the states, and Dk is zero. As the system moves through time, k increments during each sample, yielding a new current set of states that can be derived from the previous states and the current input.

To describe the Kalman filter itself, it is best to begin with an explanation of the various matrices.

First one must define A_k (mxn) and B_k (mx1) which define the state transition for the system.

Then, one must define the P matrix, which is the error/confidence matrix. This matrix will help determine whether more weight goes to the model or more weight goes to the measurement value. It has the same dimensions as the matrix A_k (mxn), and oftentimes, the P matrix is initialized with some constant value along the diagonal. Occasionally, this constant value corresponds to the covariance of the model, but many times this matrix can be left full of zeros, since this matrix will change during the Kalman filter cycle.

The next matrix to be defined is Q. The Q matrix, which also has the same dimensions as A_k (mxn), represents the process noise covariance. The Q matrix represents errors present in the model, and will give either a higher or lower "weight" to the measured data. Usually the values of the diagonals can be tuned to provide the desired response.

The matrix C_k (1xm) determines which of the states is compared to the measured value, and corresponds directly to the C_k matrix in the state space representation.

The value R (typically 1x1) represents the measurement noise covariance. This value will often vary from one sample to another, and can be computed by squaring the value of a running standard deviation. In the case of this report, R was assumed to be a fixed value, and because the magnetic encoder is fairly accurate, this assumption seemed to work well. When readings are less accurate it may be advantageous to compute a running covariance of the incoming measurement data.

Now that all the primary matrixes have been defined, the Kalman filter equations can be explored.

The first four equations are used solely to find the Kalman gains,

$$P1 = (A_k)(P)(A_k)^T + Q$$

$$S = (C_k)(P1)(C_k)^T + R$$

$$K = (P1)(C_k)(S)^{-1}$$

$$P = P1 - K(C_k)(P1)$$

Where K (mx1) is a matrix a matrix of Kalman gains, S (1x1) is an intermediate matrix, and P1 (mxn) is an intermediate matrix. The actual measurement values have an impact on the formation of the Kalman gains only if the measurement noise covariance change is taken into consideration. The final equation in the above set only seeks to iterate the values of P.

The next two equations of the Kalman filter deal with the model of the system and the Kalman gains determined in the previous steps.

$$ERROR = Z - \{(C_k)(A_k)x[k-1] + B_k u[k]\}$$

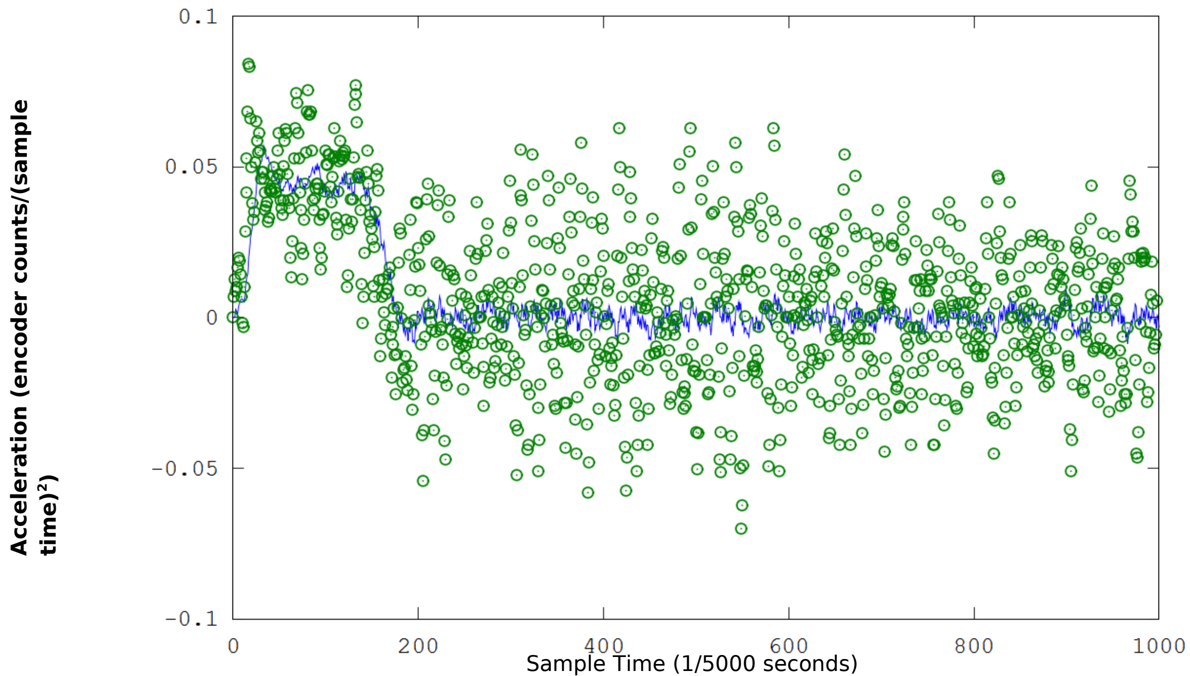
$$x[k] = \{(C_k)(A_k)x[k-1] + B_k u[k]\} + K(ERROR)$$

Note: In the above equations, Z represents the measured values, and ERROR represents the difference between the predicted value and the measured value.

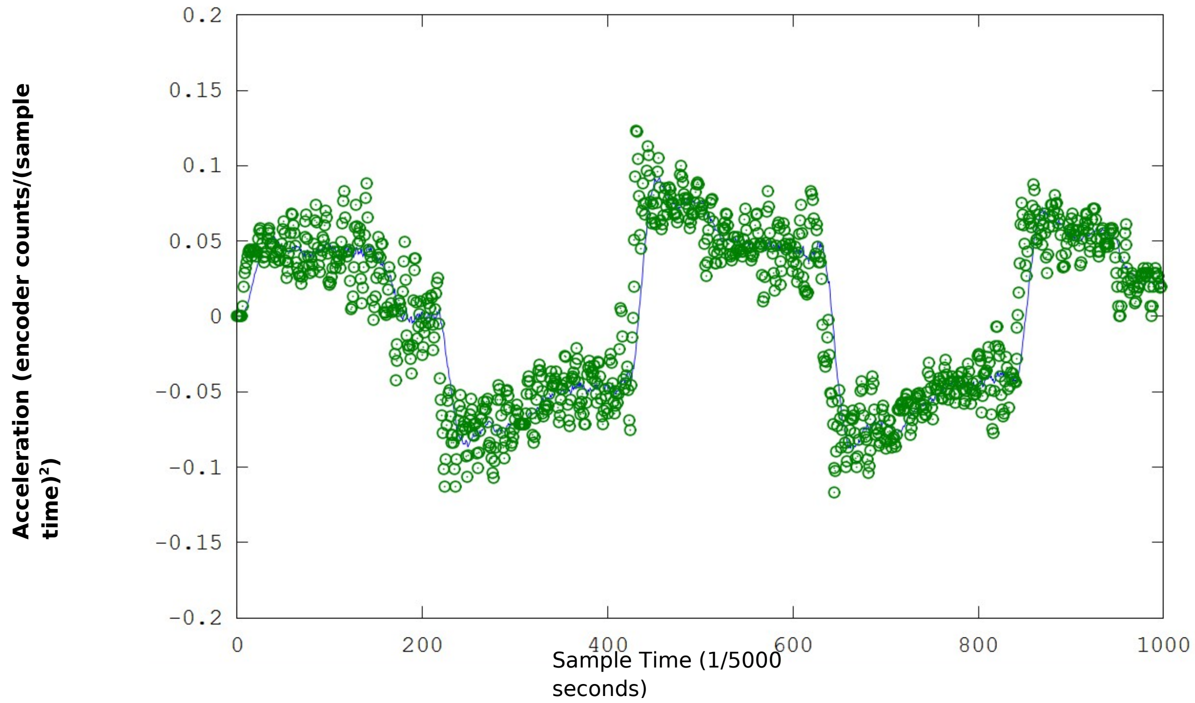
Originally a three state Kalman filter was implemented in Octave and then in C-code. However, due to overflow issues, the filter was eventually reduced to a two state system. This was accomplished by a simple differentiation and filtering of the incoming position data to yield velocity input. A system was then developed to relate velocity and acceleration in the DC motor, and then used in the Kalman filter. Many Kalman filters and Alpha-Beta-Gamma filters are demonstrated in the Octave m-files. The files “kalman_without_model.m” and “alpha_beta_gamma_without_model.m” demonstrate the implementation of both filter types where only the constant acceleration equations of classic physics are used to guide the filter. The files “kalman_with_model.m” and “alpha_beta_gamma_with_model.m” demonstrate the use of a

third order model (three states) which relates the position of the motor to the torque input. The files “kalman_with_model2x2.m” and “alpha_beta_gamma_with_model2x2.m” implement only the second order model (two states) which relates velocity to the torque input to the motor. Although a three state Kalman filter was implemented in c-code using 64-bit computation, this was not the final version implemented on the DSP since it proved to be too computationally intensive. Instead a simpler, two state Kalman filter was implemented in c-code and executed on the DSP. These two c-files can be found in “kalman.c”, “kalman_reborn.c” respectively. In the DSP, the file “KALMAN.C” only implements an alpha-beta filter, whereas the file “KALMAN2.C” actually implements the full Kalman filter algorithm.

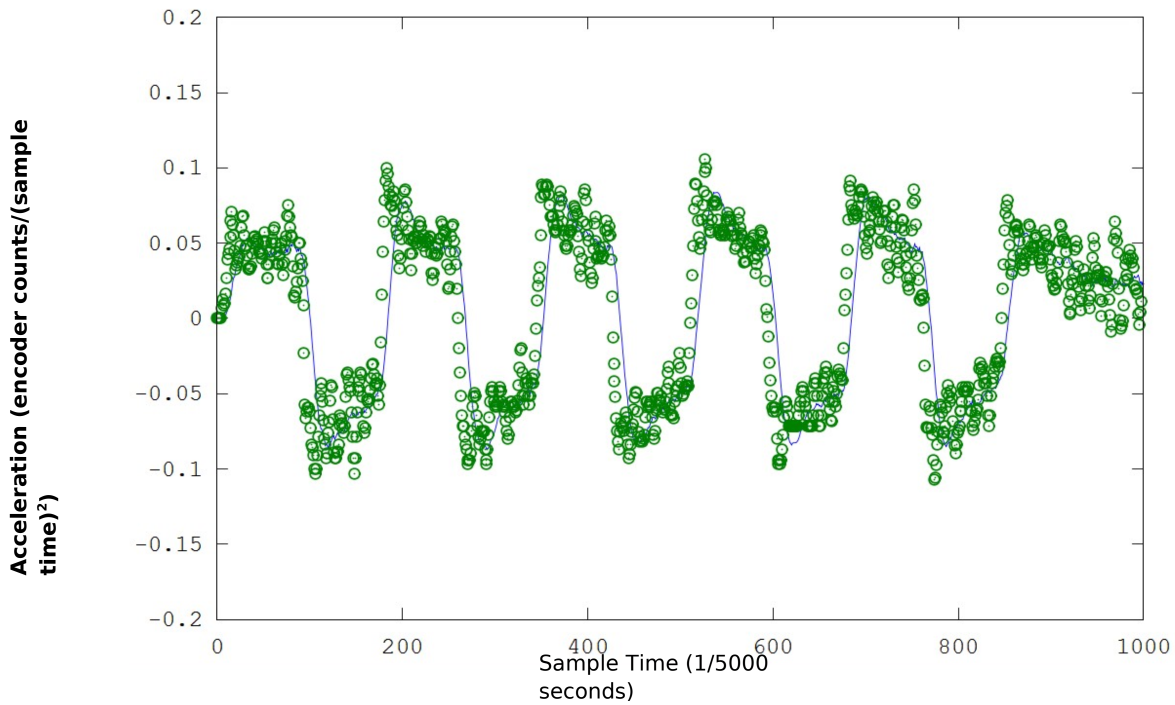
As seen below, the results of Kalman filtering proved to be very successful give a reasonable model.



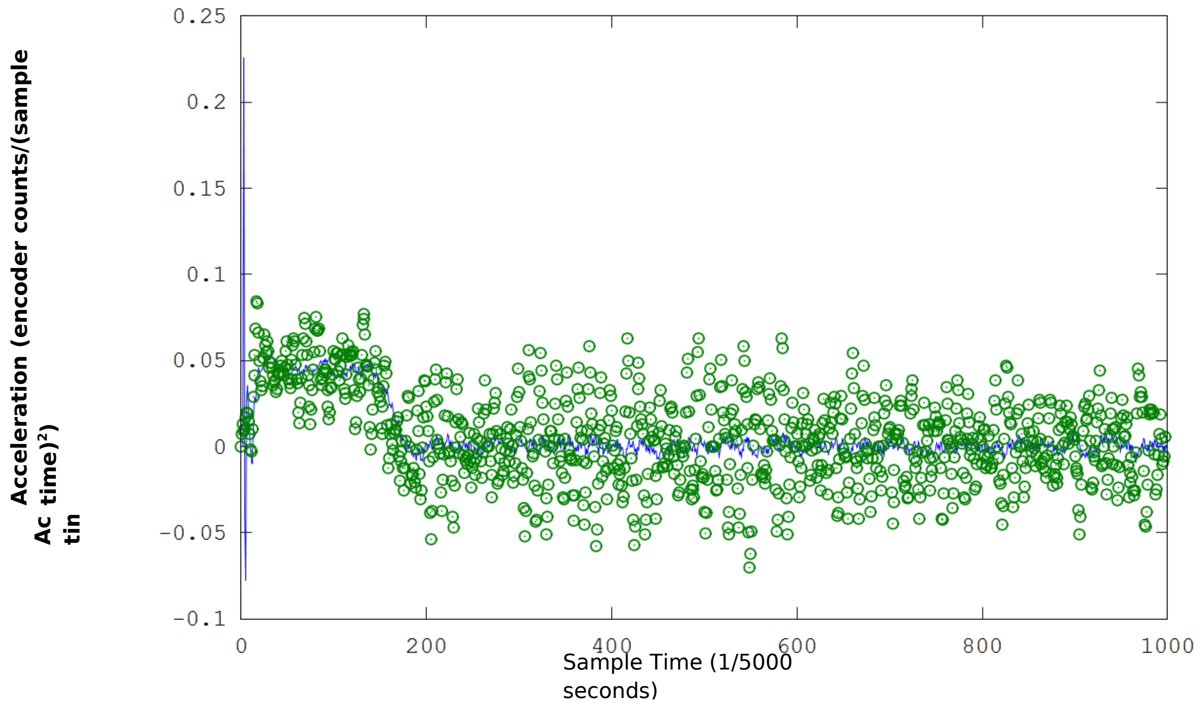
Alpha-Beta-Gamma Filter without Model: Step Input



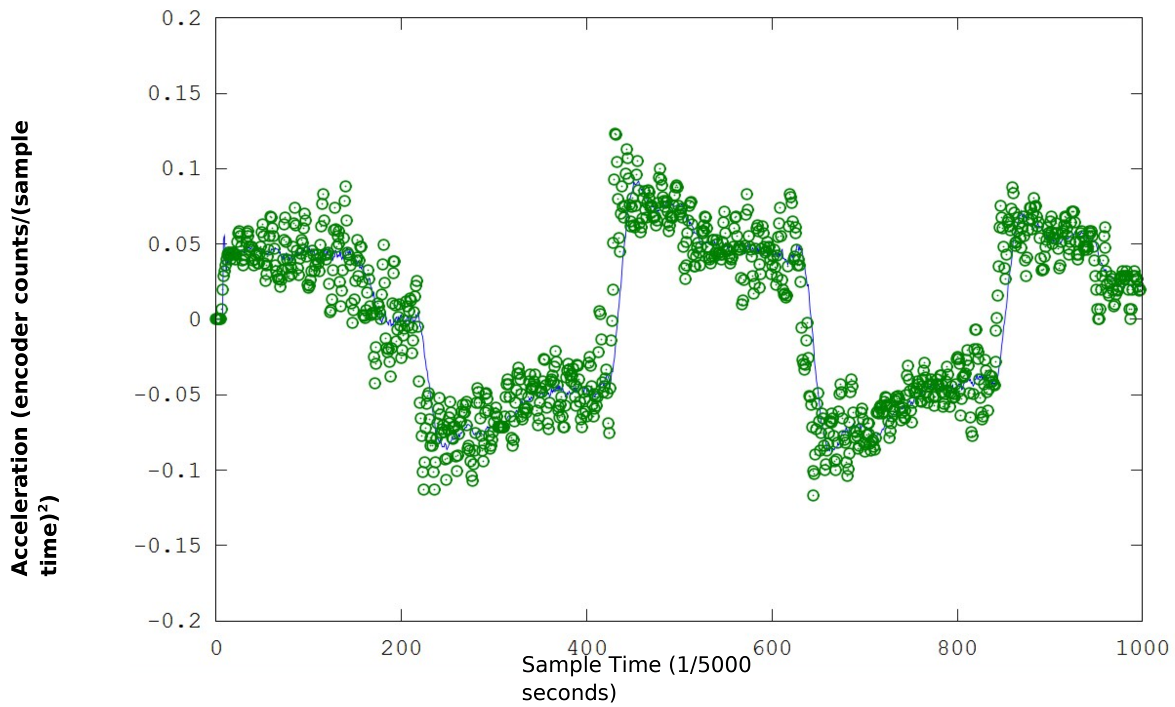
Alpha-Beta-Gamma Filter without Model: Middle Frequency Torque Input Square Wave



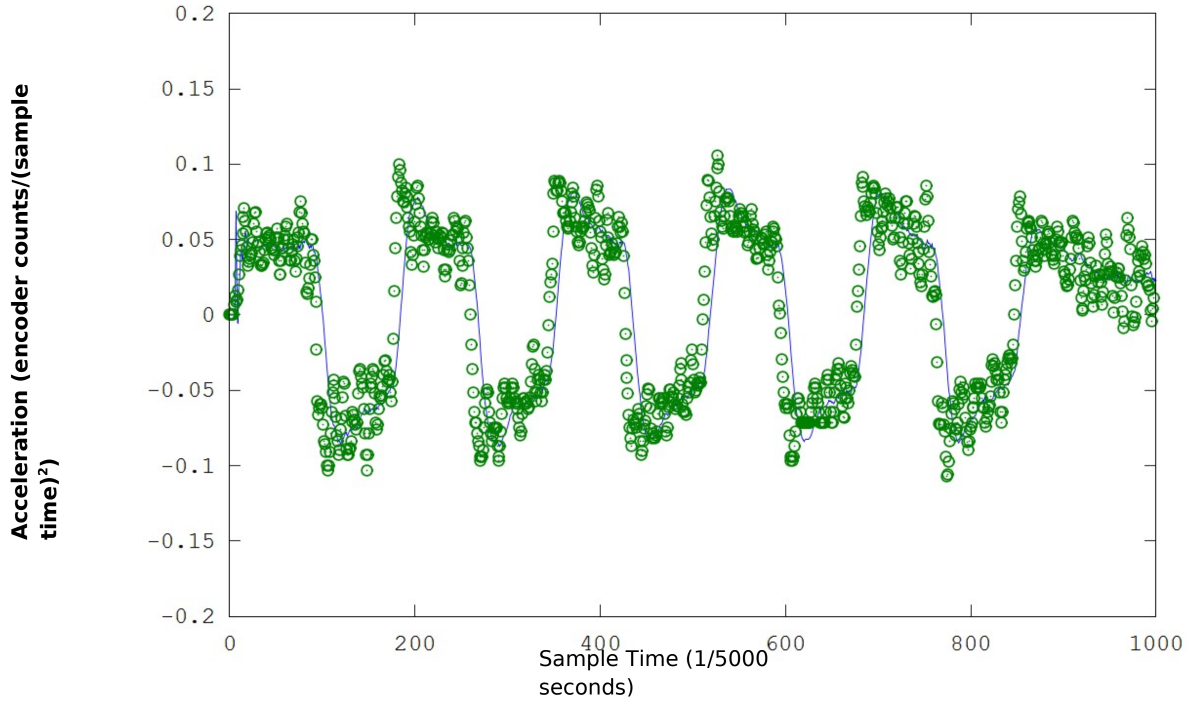
Alpha-Beta-Gamma Filter without Model: High Frequency Torque Input Square Wave



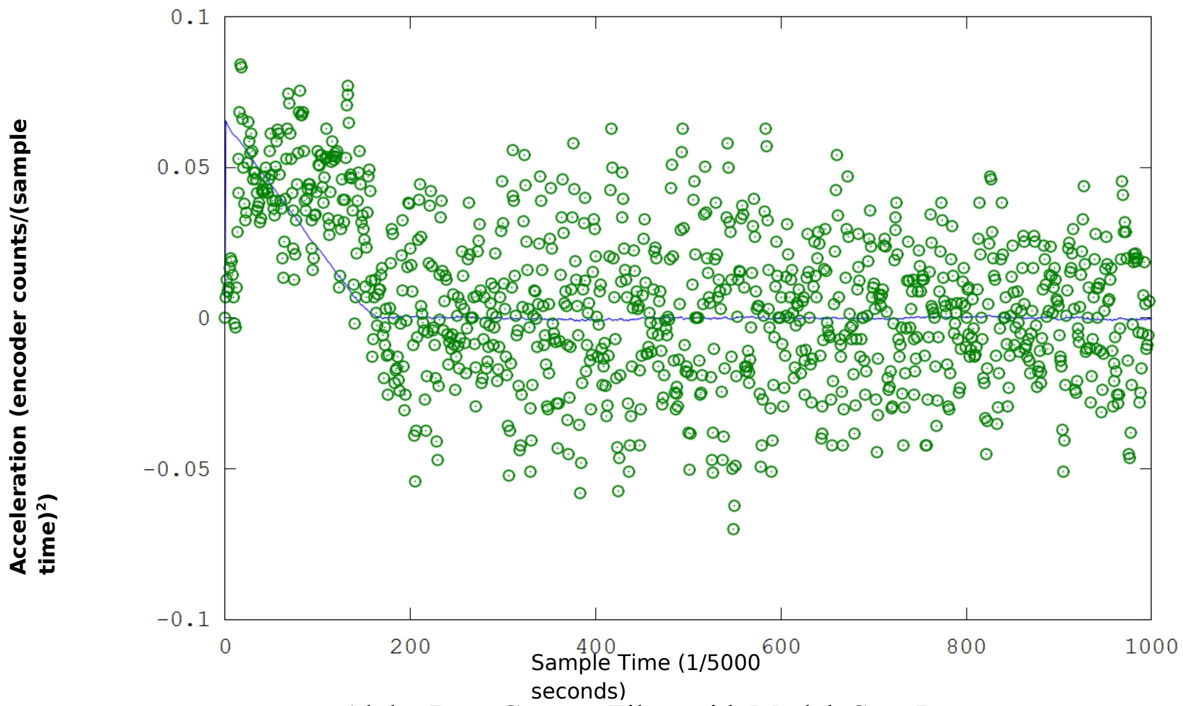
Kalman Filter without Model: Step Input



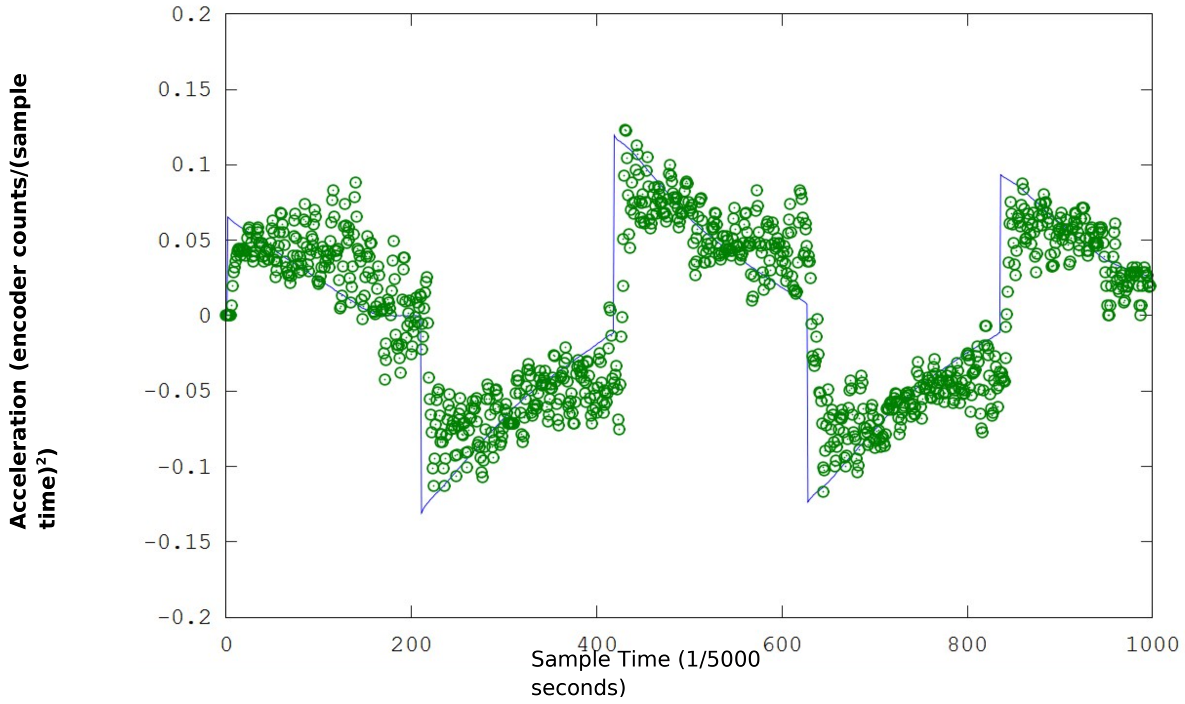
Kalman Filter without Model: Middle Frequency Square Wave



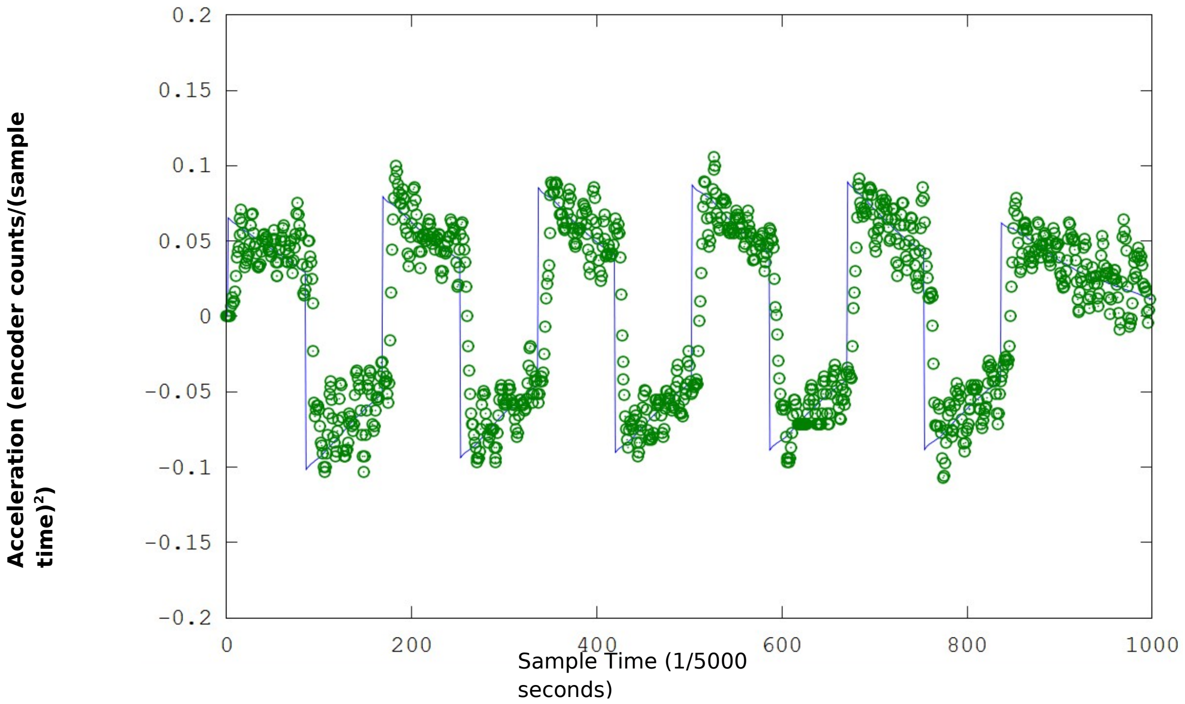
Kalman Filter without Model: High Frequency Square Wave



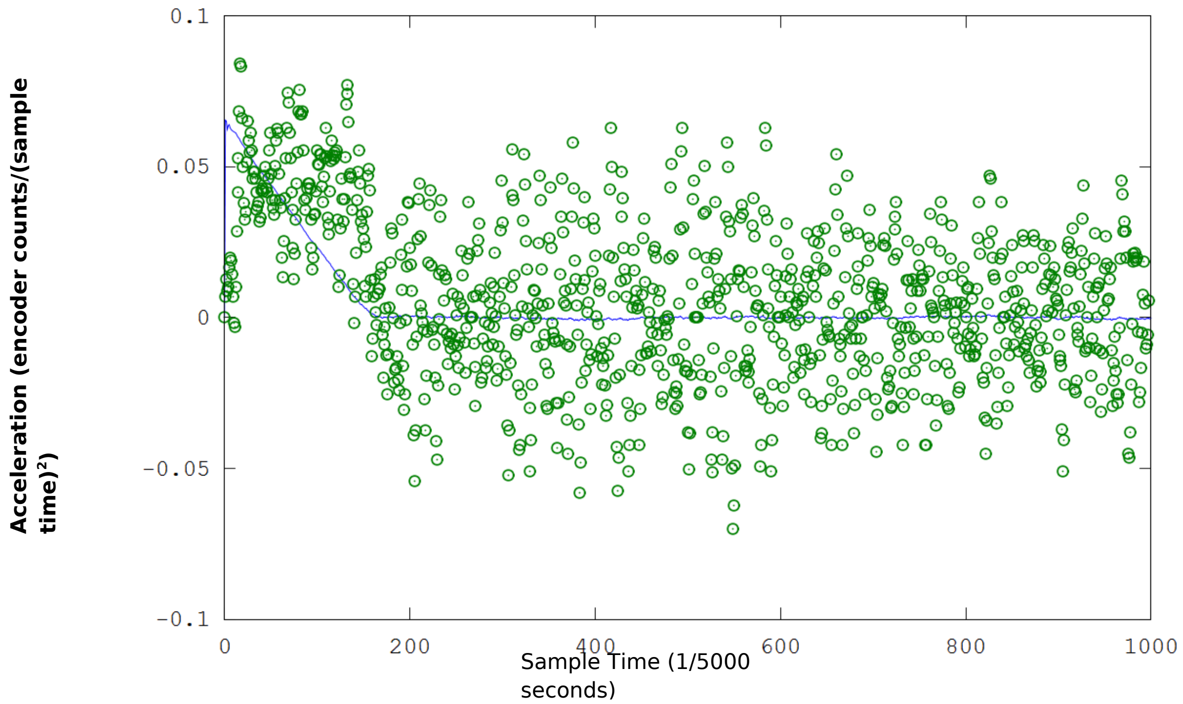
Alpha-Beta-Gamma Filter with Model: Step Input



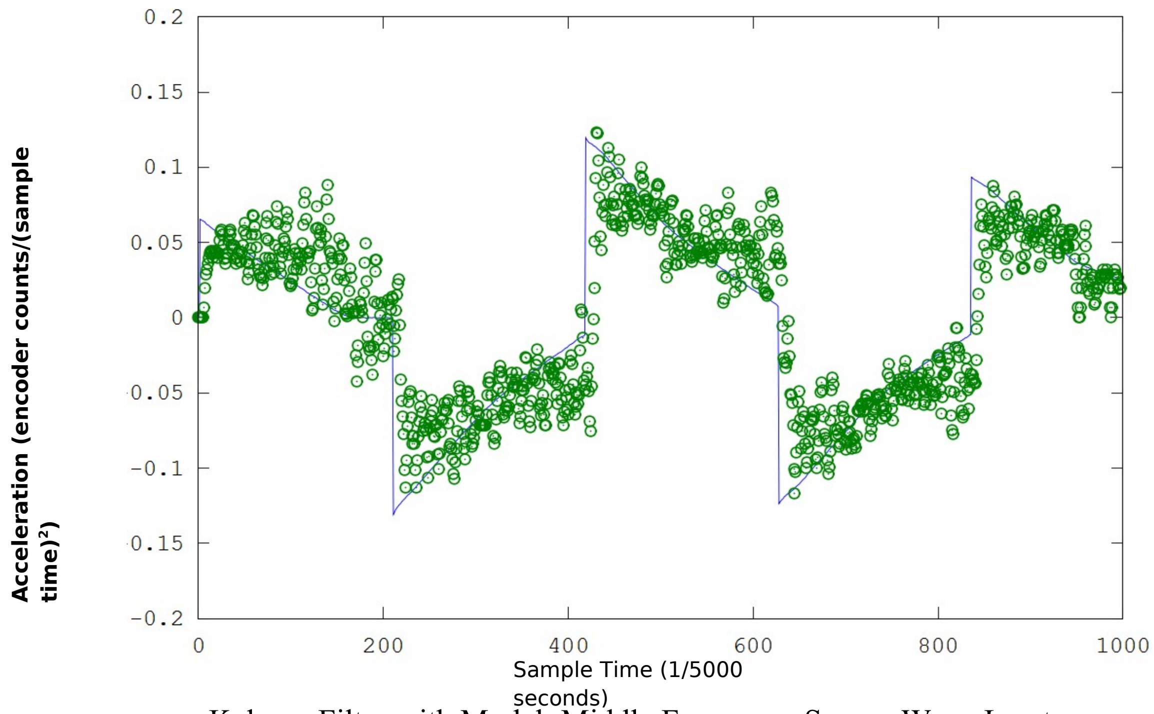
Alpha-Beta-Gamma Filter with Model: Middle Frequency Square Wave Input



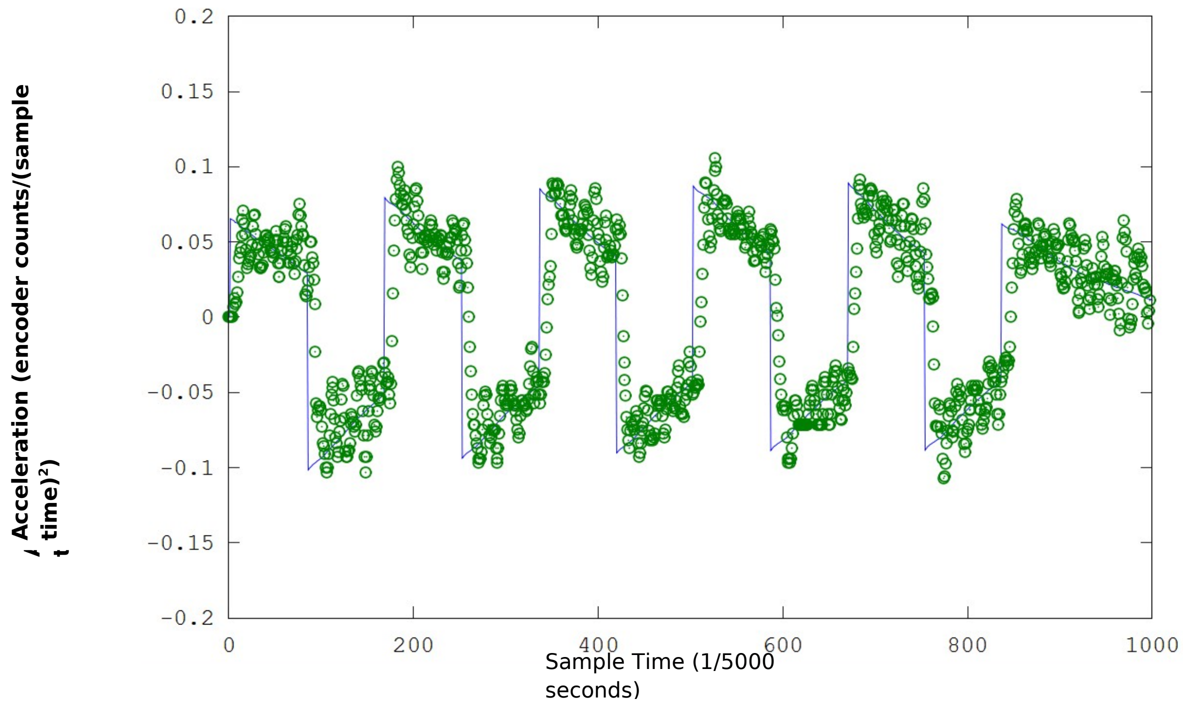
Alpha-Beta-Gamma Filter with Model: High Frequency Square Wave Input



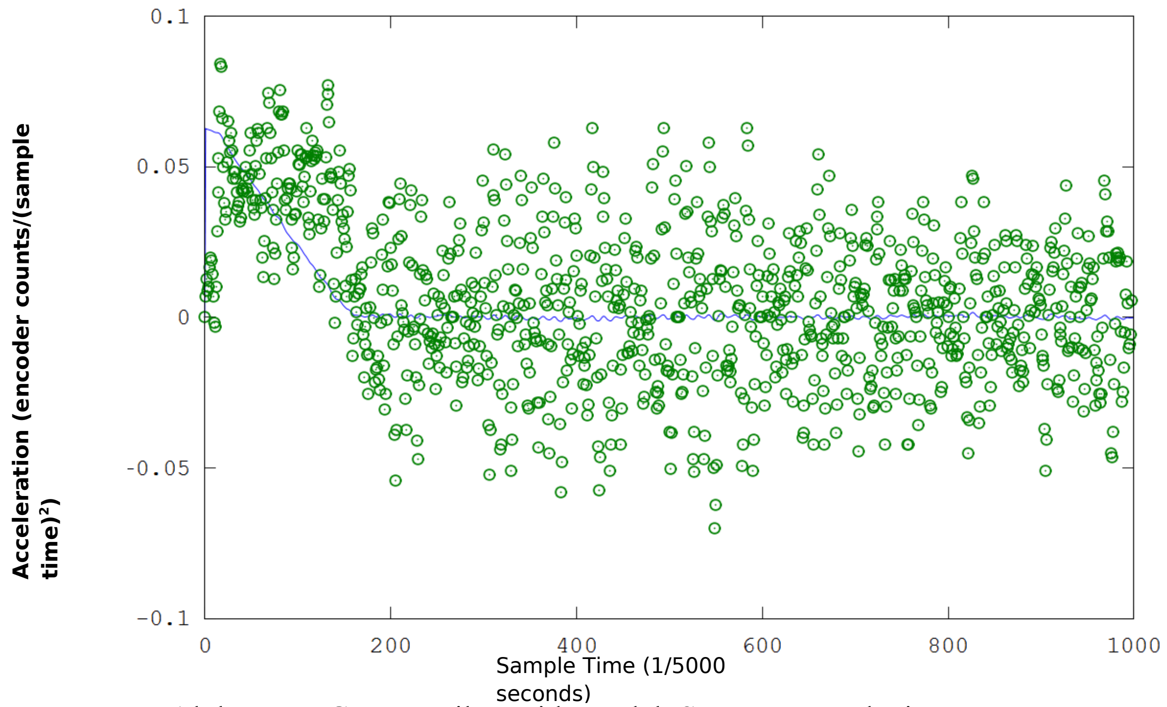
Kalman Filter with Model: Step Input



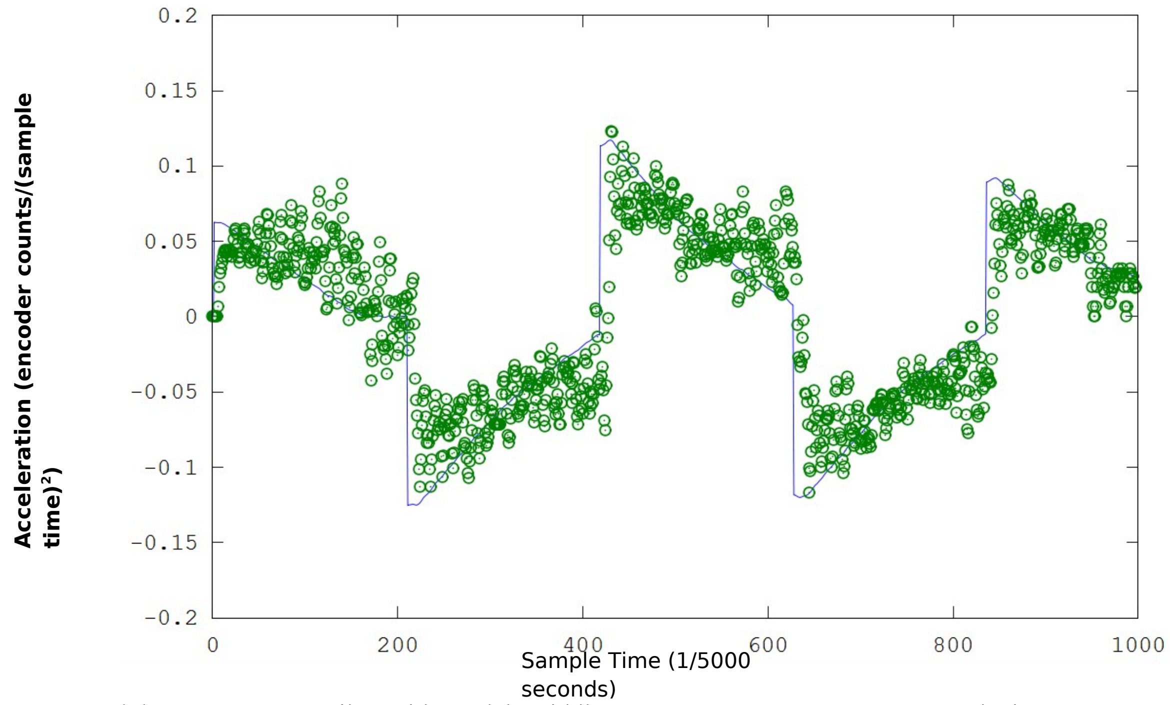
Kalman Filter with Model: Middle Frequency Square Wave Input



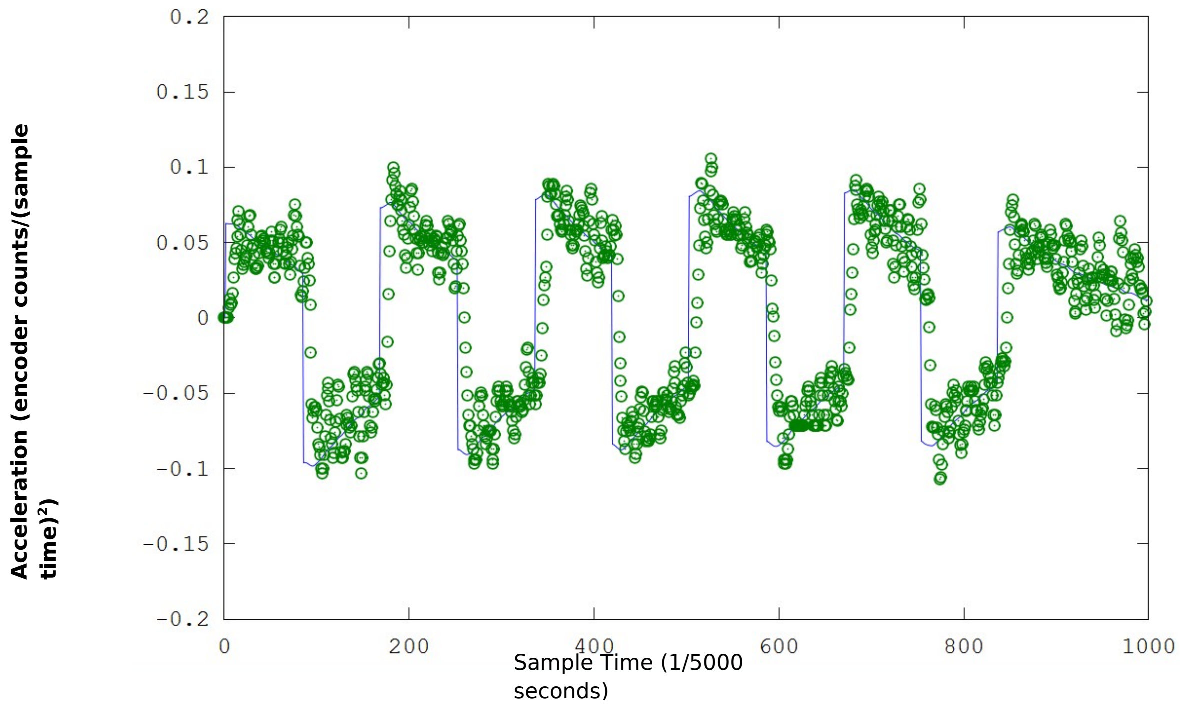
Kalman Filter with Model: High Frequency Square Wave Input



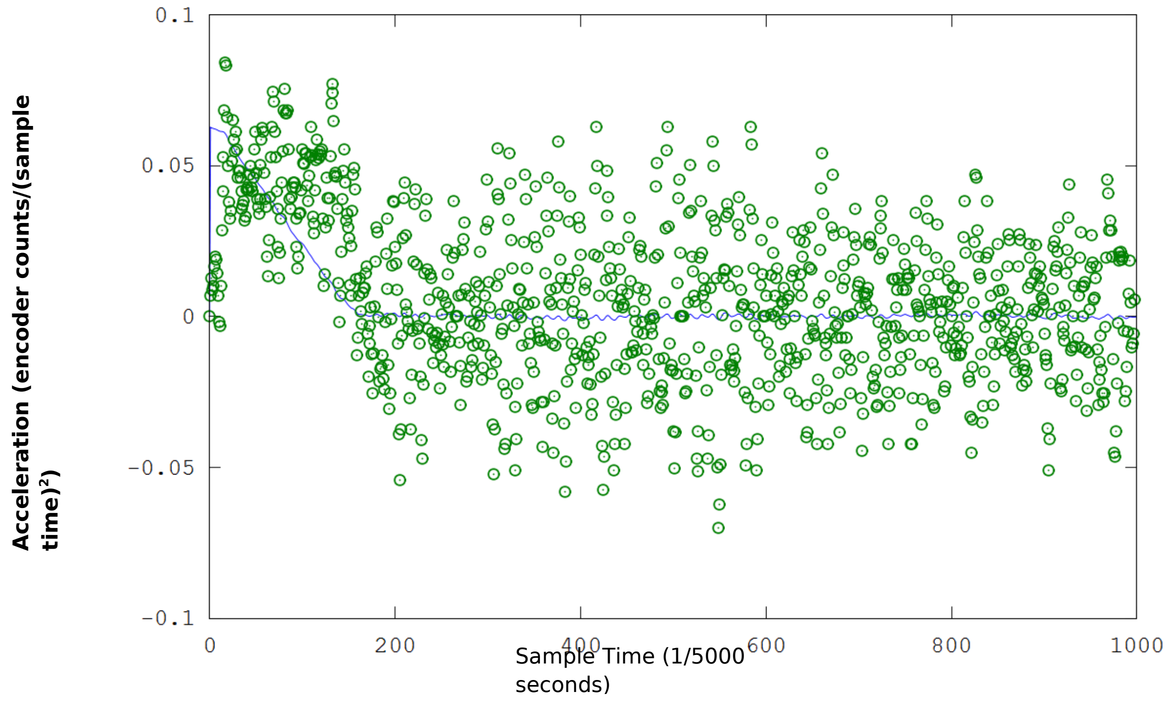
Alpha-Beta-Gamma Filter with Model: Step Input, Velocity Measurement



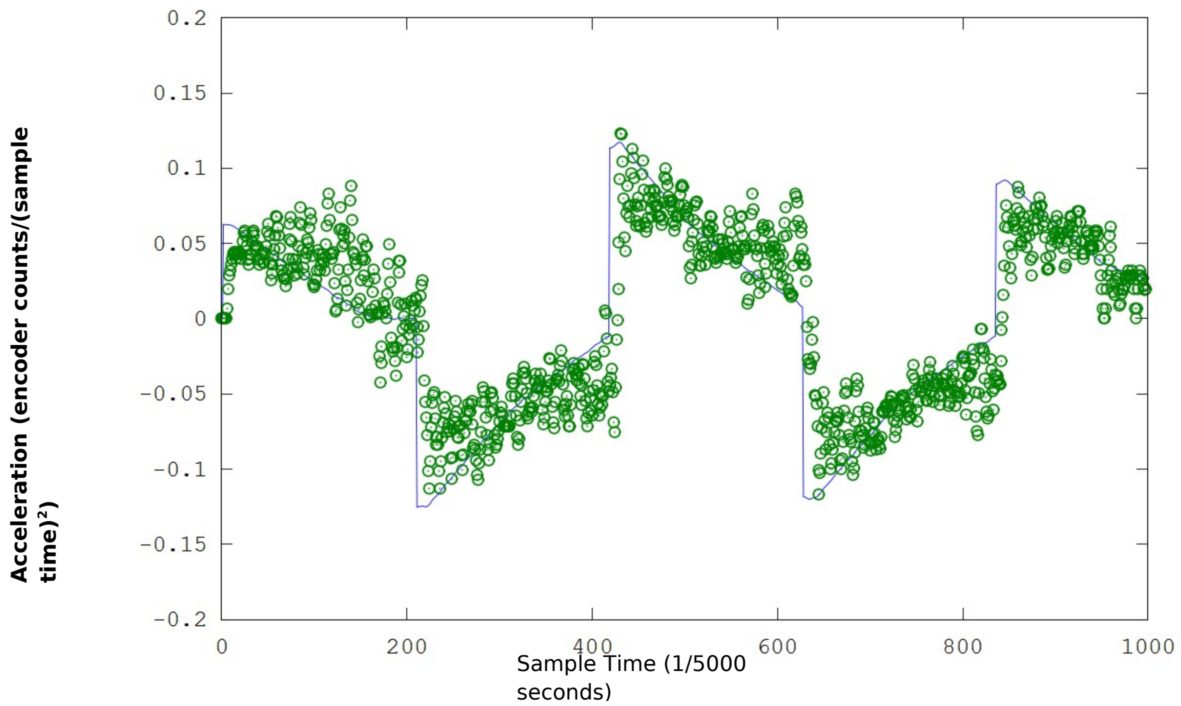
Alpha-Beta-Gamma Filter with Model: Middle Frequency Square Wave Input, Velocity Measurement



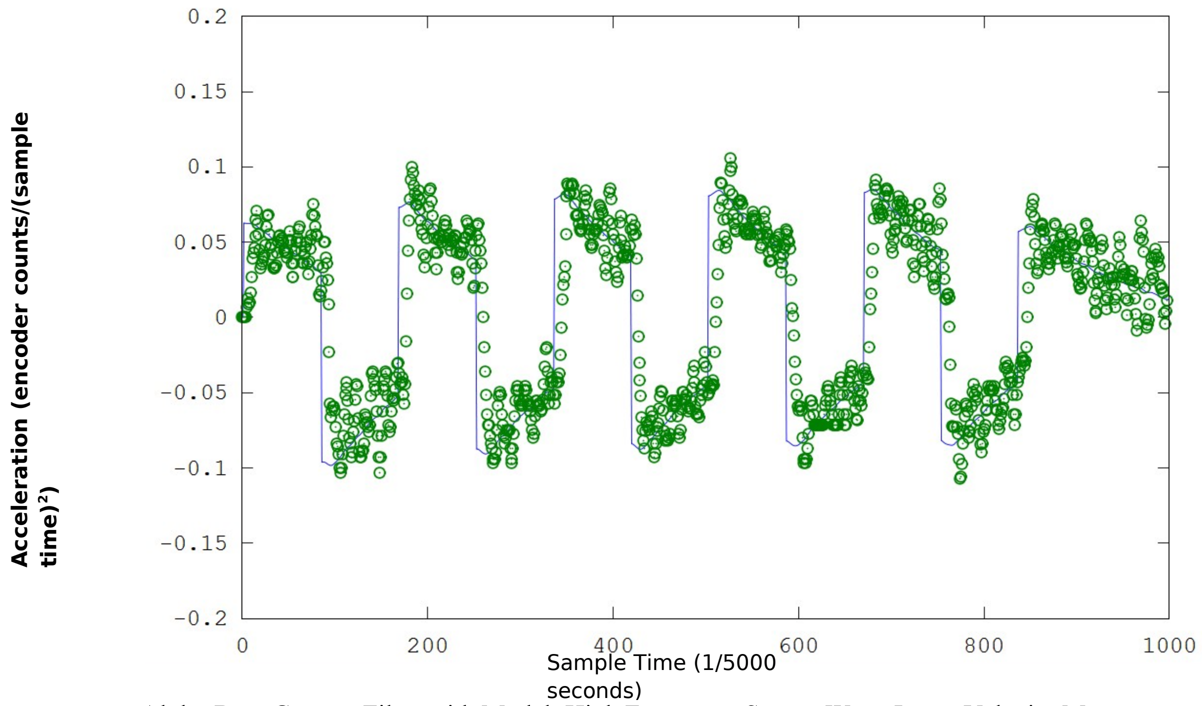
Alpha-Beta-Gamma Filter with Model: High Frequency Square Wave Input, Velocity Measurement



Kalman Filter with Model: Step Input, Velocity Measurement

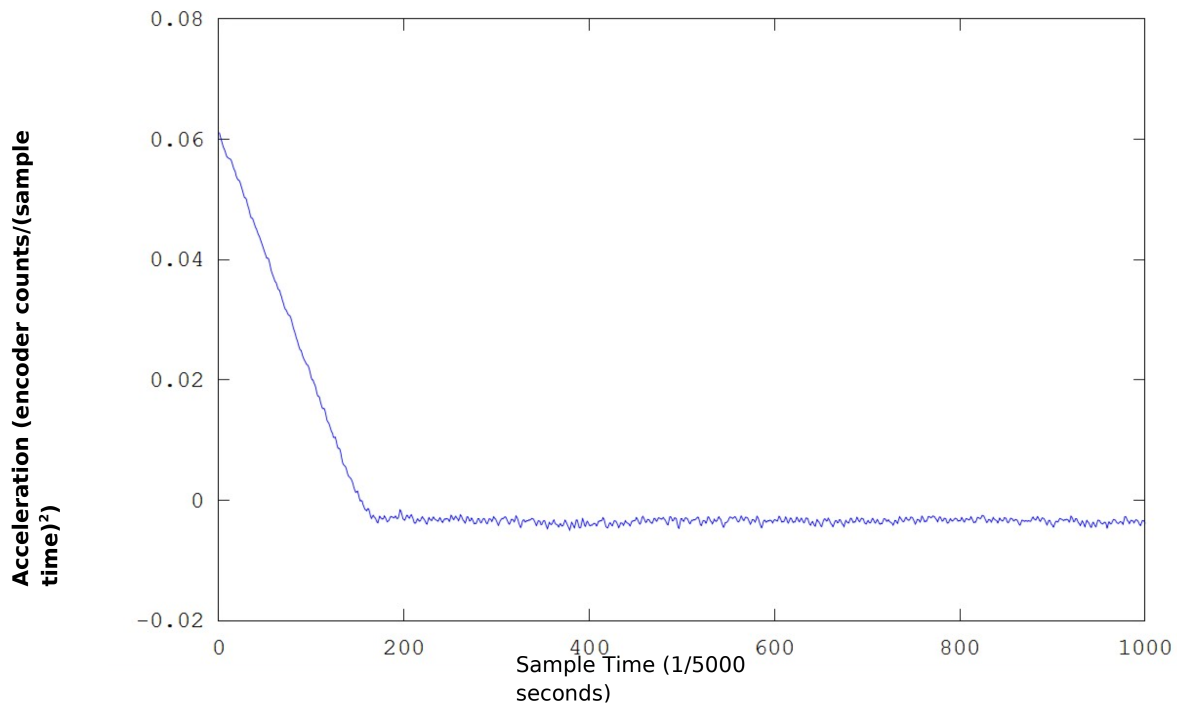


Kalman Filter with Model: Middle Frequency Square Wave Input, Velocity Measurement

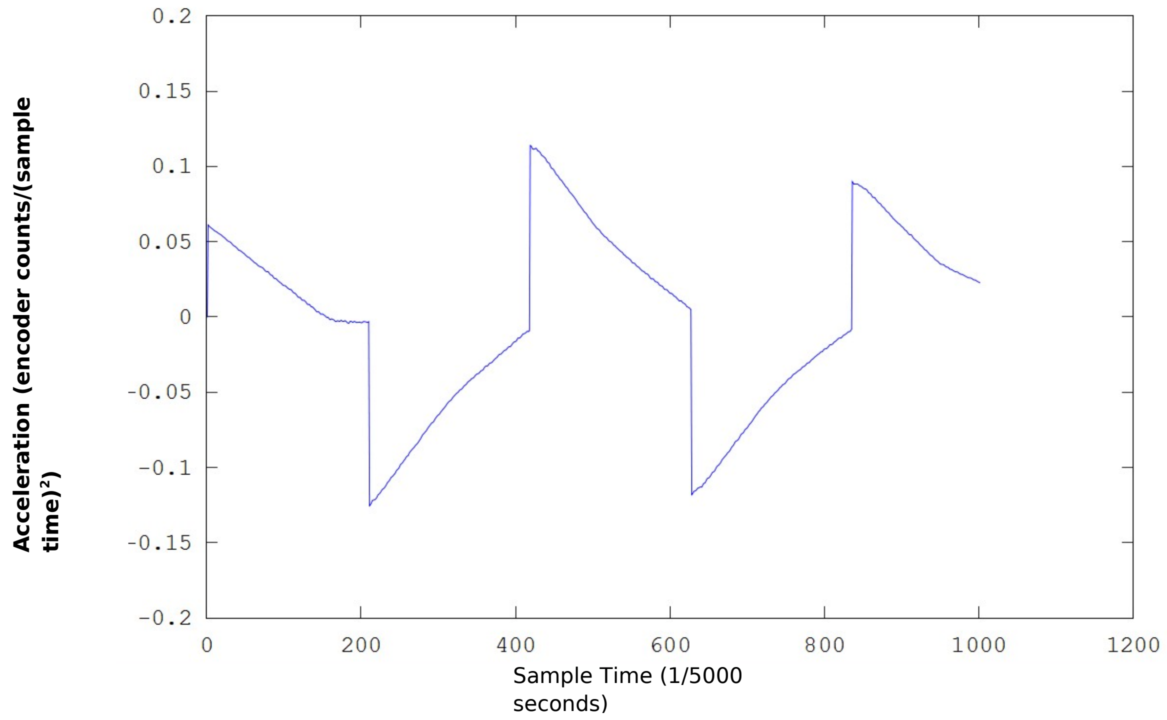


Alpha-Beta-Gamma Filter with Model: High Frequency Square Wave Input, Velocity Measurement

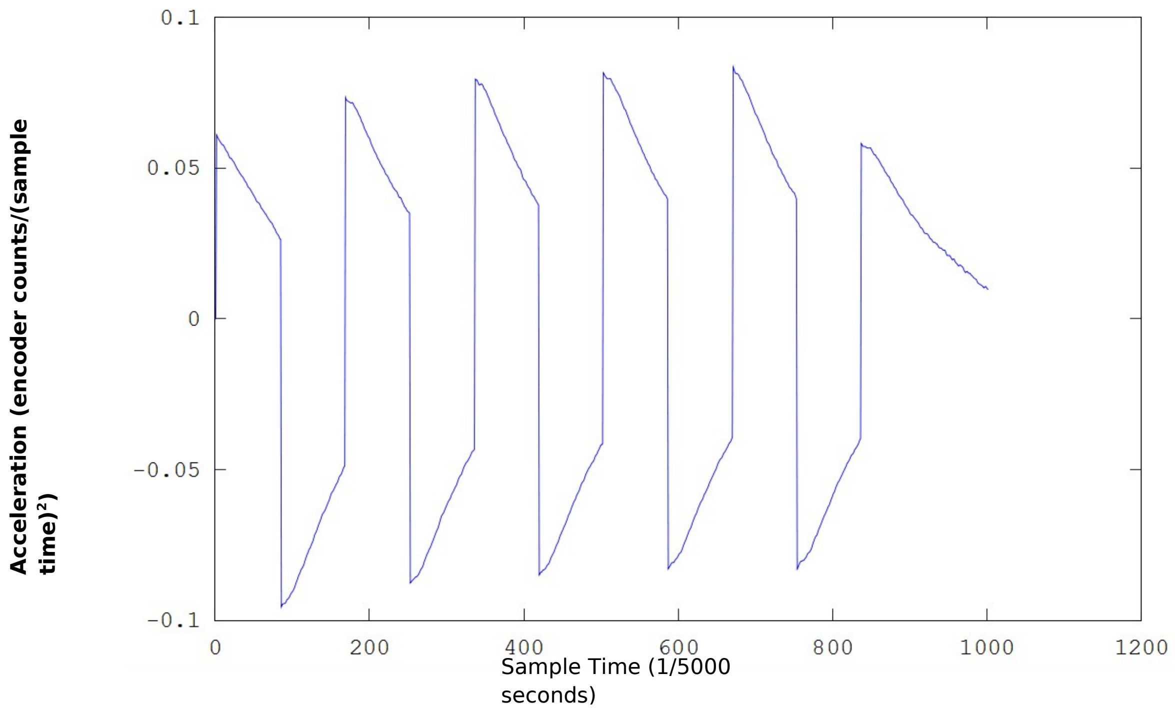
FINAL IMPLEMENTATION (C-CODE, FIXED POINT)



Kalman Filter with Model: Step Input, Velocity Measurement, Fixed Point C-code



Kalman Filter with Model: Middle Frequency Square Wave Input, Velocity Measurement, Fixed Point C-code



Kalman Filter with Model: High Frequency Square Wave Input, Velocity Measurement, Fixed Point C-code

System Identification

Fortunately, in most situations, the model of a DC motor can be approximated as a first order linear system. Whether torque controlled or position controlled, most motors will reach a steady state velocity under some step input. If the transient velocity rise time can be captured in data, then one could easily use the properties of a first order system to obtain an approximate model.

Since we know that a first order system has the form,

$$\frac{Y(s)}{X(s)} = \frac{1}{Js + b}$$

where Y(s) is the output, X(s) is the input, J is the overall inertia of the system and b overall is the damping of the system. Note: J and b do not refer merely to physical inertia and damping, but also encapsulate the motor's electrical properties as well.

To determine b, one must merely examine the steady state speed and the value of the input. The equation for finding b is,

$$b = \frac{X}{\omega_{ss}}$$

where X is the input (mA in this specific case) and ω_{ss} is the steady state speed.

To find J, all one needs is a value for the first order time constant, which is the time value at 63% of the steady state speed.

Once the time constant is obtained, J is found through the equation:

$$J = \tau b$$

Now, since we measure position, not acceleration, an integration term must be added to force the output of the system to become position. A revised first order model, where V(s) represents velocity and T(s) represents applied torque, is shown below:

$$\frac{V(s)}{T(s)} = \frac{1}{Js + b}$$

Integrating velocity to obtain position is carried out as follows:

$$\left(\frac{1}{s}\right)\frac{V(s)}{T(s)} = \frac{1}{Js + b}\left(\frac{1}{s}\right)$$

$$\frac{X(s)}{T(s)} = \frac{1}{Js^2 + bs}$$

In the previous equation, X(s) represents position.

To achieve a state space representation, the transfer function above must be converted from the frequency domain to the time domain.

$$\frac{X(s)}{T(s)} = \frac{1}{Js^2 + bs}$$

$$s^2X(s)J + sX(s)b = T(s)$$

$$J\ddot{x}(t) + b\dot{x}(t) = T(t)$$

Before proceeding to create a state space system, it is important to realize that the system above is only second order. To have access to the state of velocity, a third order differential equation is needed. This can be obtained by perturbing the third derivative of position, $\dddot{x}(t)$.

$$0 = T(t) - J\ddot{x}(t) - b\dot{x}(t)$$

$$k\dddot{x}(t) = T(t) - J\ddot{x}(t) - b\dot{x}(t)$$

Now, that a third order differential equation has been created, state space formation can take place. The variable k can be selected as an arbitrary small value so that it only slightly changes the system's response.

$$\dddot{x}(t) = \left(\frac{1}{k}\right)T(t) - \left(\frac{J}{k}\right)\ddot{x}(t) - \left(\frac{b}{k}\right)\dot{x}(t)$$

$$\frac{d}{dt}\begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -\frac{b}{k} & -\frac{J}{k} \end{bmatrix}\begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{k} \end{bmatrix}T(t)$$

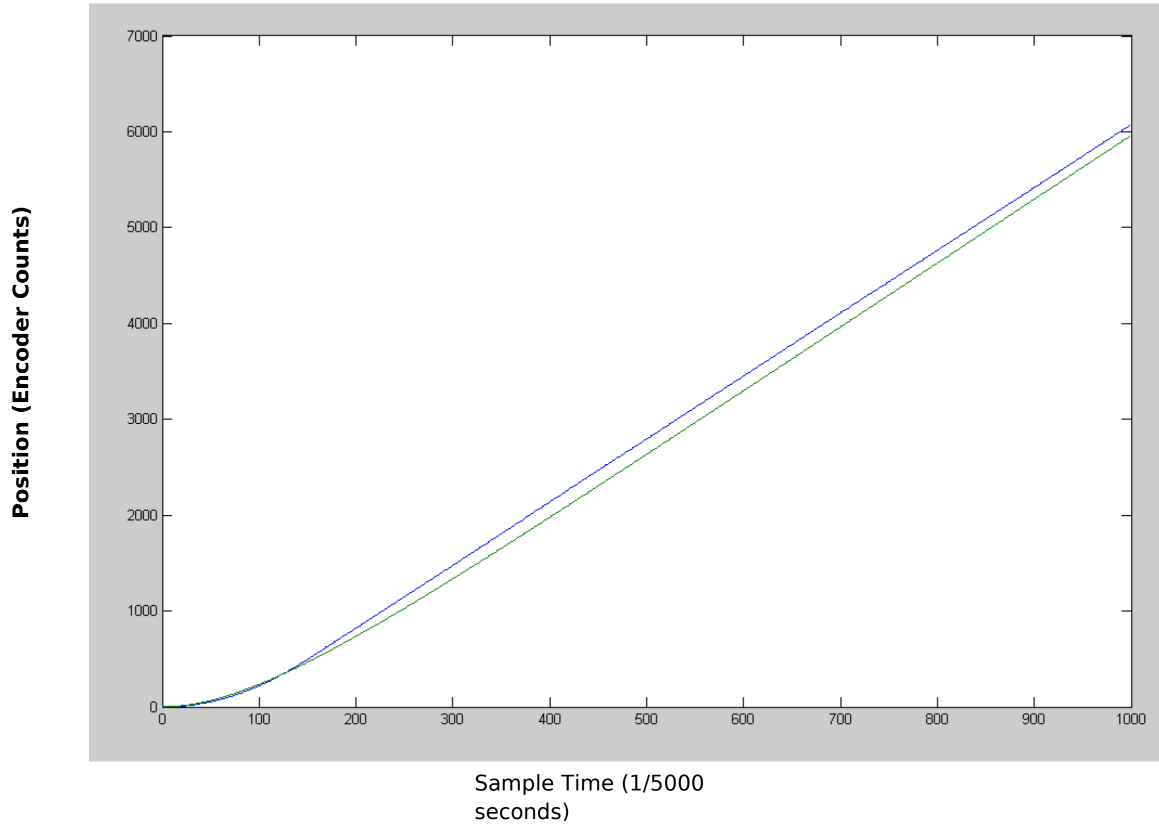
If a full state feedback system were to be designed, the above model would be sufficient. If the design method will only involve root locus or frequency domain techniques, the transfer function,

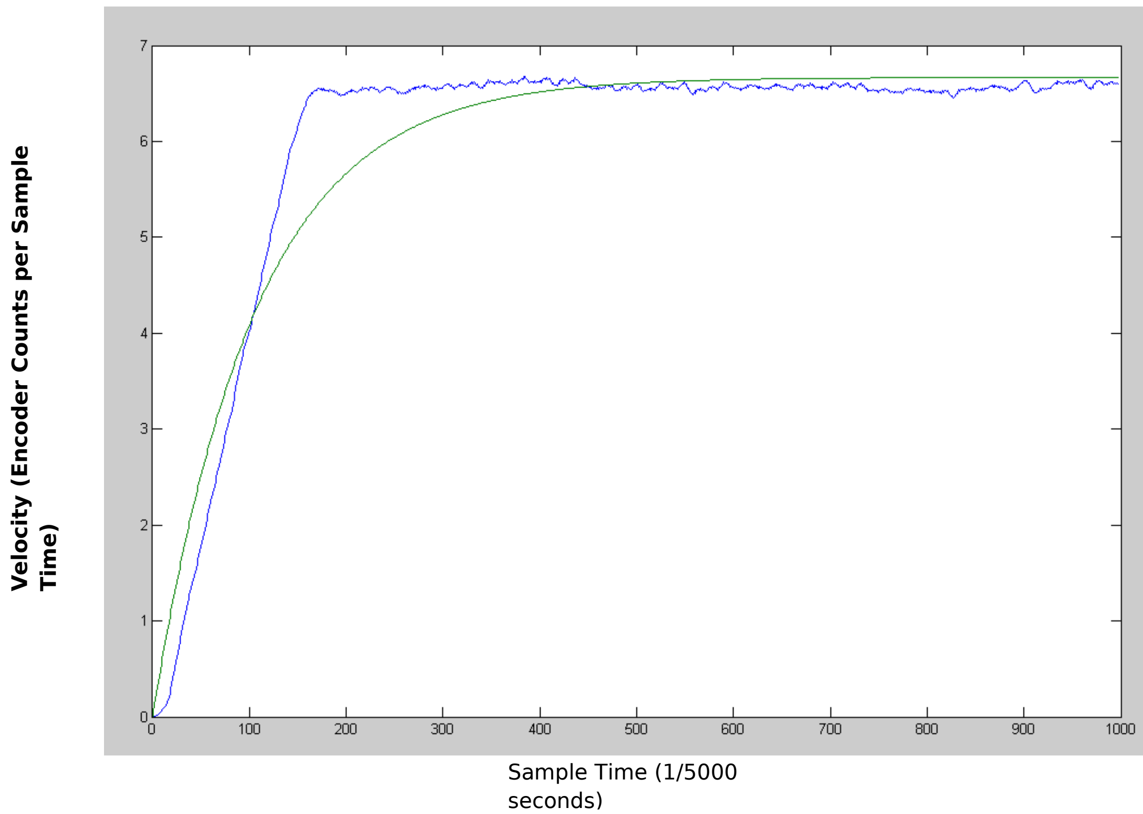
$$\frac{X(s)}{T(s)} = \frac{1}{Js^2 + bs}$$

would be sufficient.

If, instead of for control design, a Kalman filter observer was desired, the state space model above would need to be discretized. This is easily accomplished in a program like MATLAB or Octave and can be carried out in "make_model.m". Plots of actual collected data and the

corresponding model are shown below. The green curves represent the model, whereas the blue curves represent actual data (or differentiated and filtered data).





Adjust the values for J and b to correspond to the values output by the DSP and record the discrete transfer function values. Note: It is important to use a zero order hold to discretize the system. Both MATLAB and Octave accomplish this by numerically determining the matrix exponential of the A and B state space matrixes. Methods such as Tustin are not typically exact enough for this filtering application.

To relate velocity to the torque input command, as is done in the C-code, one must merely change the state space representation as follows:

$$\frac{d}{dt}\begin{bmatrix}\dot{x}\\ \ddot{x}\end{bmatrix} = \begin{bmatrix}0 & 1\\ -\frac{b}{k} & -\frac{J}{k}\end{bmatrix}\begin{bmatrix}\dot{x}\\ \ddot{x}\end{bmatrix} + \begin{bmatrix}0\\ \frac{1}{k}\end{bmatrix}T(t)$$

Furthermore, when rounding the values in the state space matrix for implementation in the C-code, it is important to remember that in the discrete 2x2 A-matrix, the first element must not be rounded to one. It is important that this value is set to be close to the actual floating point value of the first element in the discretized matrix (i.e. 0.99=> 127/128). Values which are close to but not equal to one must be maintained as such to prevent instability.

Operation in Firmware Code

In the firmware, the filters and model generation code are accessed using several key firmware parameters.

To select a filter, use property X2. Setting X2 to 1, filters velocity using a 2nd order Butterworth filter. Setting X2 to 2, filters acceleration using a 4th order Butterworth filter. Setting X2 to 3 filters acceleration using a 4th order Butterworth filter and velocity using a 2nd order Butterworth filter. This feature, although desirable, still has a bug which must be worked out. Setting X2 to 4, filters velocity with a 20th order FIR filter. Setting X2 to 5, filters both velocity and acceleration using a Kalman filter.

Note: The Butterworth filter coefficients could easily be created as parameters able to be accessed from outside the DSP. This could also be done for the FIR coefficients, but at least 10 additional properties would be needed.

Both the FIR filter and the Butterworth filter use coefficients in Q22.10 format. Velocity is output in the same format, whereas acceleration values are output in Q14.18 format.

To determine a given motor's model, set property PLOG (101) to 2. This will apply a step input of 500 mA to the motor, and fit a first order model to the velocity curve. The coefficients J and b as discussed in this document will be output to properties X0 and X1 respectively. These

coefficients can then be used as previously mentioned to design a controller or to design an observer as used in the Kalman filter.

The Kalman filter, because of the need to eliminate overflow in the intermediate calculations while maintaining a reasonable accuracy, is slightly complicated to modify. After a 2x2 *discrete* A matrix and a 2x1 *discrete* B matrix are determined using the provided Octave code, these parameters must be added into the Kalman filter code. In the Kalman filter without automatic gain tuning (i.e. alpha_beta filter), the A and B matrix values are represented with a numerator and a denominator. The denominator is required to be a power of 2 and the numerator is some integer value, which when divided by the denominator, approximates the fractional values present in the A and B matrixes. Since the top left value in the A matrix will have the largest value, this value will be what determines the maximum velocity input value before overflow. If the velocity is scaled up by 2^18 to provide adequate acceleration resolution, then it is important to regulate the numerator of the first value in the A matrix allows for ample velocity input. For example, the maximum value for the a11 numerator is 2^7 which leaves 2^6 encoder counts per commutation loop (2^(31-7-18)) available for maximum velocity input. This velocity value equates to 78 rev/s. If the Kalman filter with automatic gain tuning is used, then additional values for the A matrix numerator must be provided for the automatic gain tuning operations. These special A matrix numerator values all must be in Q16.16 format.

In the case of the Kalman filter with the manually tuned gains, the 4 values in the A matrix and the 2 values in the B matrix should have their numerator and denominator values as configurable properties. The two gains should also have their numerators and denominator values as configurable properties. In the case of the Kalman filter with the automatically tuned gains, the A and B matrixes should have their numerator and denominator values as configurable properties as well as the special Q16.16 values for the Q16.16 A matrix. Instead of configurable gains, there should be a configurable Q-matrix value which is also in the format Q16.16.

The data acquisition array in the Puck is also able to be exported to the PC via the property command. One way to accomplish this is by setting property 101 to 1, after torque mode (property 8, value 2) has been initiated. Executing this command will cause a step input to the motor, which will be recorded for a 1000 samples at the commutation rate of the motor. Then, if one calls property 102 a thousand times, and saves each value received, the values will recorded into the data array will be read out of the Puck and may be stored in an array on the user's PC. In addition, instead of setting property 101 to 1, one could also set property 40 to 1. This will begin the data acquisition process in the puck. However, if property 40 is used instead of property 101, one must execute the torque or position commands through the set property command in a C-file. This works best if code is executed in a real time operating system, such as XENOMAI.

REFERENCES:

http://www.embedded.com/story/OEG20010529S0118

http://www.innovatia.com/software/papers/kalman.htm

http://www.techsystemsembedded.com/Kalman.html

http://www.mathworks.com/matlabcentral/fileexchange/loadCategory.do : An Intuitive Introduction to the Kalman Filter by Alex Blekhman.