Barometric Apogee Detection Using the Kalman Filter

A NAR Research and Development Report
"C" Division
NARAM 44, 2002

By David Schultz
NAR # 63255

Table of Contents

Abstract

Altimeters currently used for apogee deployment do only a fair job of detecting apogee. Barometric based systems sense when pressure begins increasing and therefore are always late. How late depends on the details of the hardware and software. Accelerometer based systems integrate acceleration to get velocity and are subject to many errors which corrupt the estimate. Accelerometer systems cannot “detect” apogee and at best provide a rough estimate.

It is important that the altimeter deploy the recovery system as close to apogee as possible. At apogee the rocket is traveling at its minimum velocity and deployment loads will be minimized. At apogee a one second difference in deployment time translates into a 32 ft/sec change in rocket velocity. While it is possible to build the rocket to withstand these loads, it is always best to minimize the loads.

Two methods of improving apogee detection using barometric sensors are developed. The first is a simple low pass filter that is suitable to altimeters with extremely limited computational resources. This also serves as an introduction to some of the concepts used in the Kalman filter. The second is a Kalman filter that provides even better performance but requires more computation. Both methods are tested using previously recorded flight data.

The Kalman filter is shown to provide a robust, accurate, and repeatable method of determining the time of apogee.

Introduction

I recently acquired a new altimeter, the RDAS by AED. The RDAS can record acceleration and pressure data with 10 bit resolution and at 200 samples per second (SPS) and was my primary reason for purchasing it. After looking at the data from its first flight and the data from the NAR level 3 certification flight of Dave Schaefer (using Bryan Nelson’s RDAS) I was concerned over the late deployments and I was certain that it was possible to improve on the apogee detection.

The RDAS uses an accelerometer to measure acceleration on the rockets longitudinal axis during flight. By integrating this reading during flight, an estimate of the vehicles velocity is obtained. When this estimate decreases to zero, the apogee deployment charge is fired. There are several possible sources of errors that affect the accuracy of this technique.

One source is that the effect of the earth’s gravity is assumed to act along the longitudinal axis of the rocket. If the rocket flies perfectly straight up, this is fine. But any deviation from vertical will introduce error. Another source of error is the flight dynamics of the rocket. If its flight angle oscillates, the assumption that the motor thrust is acting in the direction of flight is invalidated. If we do not have accurate values for the sensitivity and offset of the accelerometer chosen, it will lead to errors in apogee detection. This is largely mitigated by measuring the pre-launch acceleration and calling that value “1G”.

If the accelerometer is so fraught with errors, why use it? Because barometric sensors are sometimes fooled by flight dynamics. If the rocket gets close to the speed of sound, a pressure wave will travel along the rocket body and this pressure increase would be seen as a decrease in altitude and therefore, apogee. This is sometimes handled by using a “Mach inhibit” timer that prevents the altimeter from taking any actions until after a timer expires. This works will so long as the rocketeer correctly sets the delay.

The primary source for error in determining the time of apogee with barometric sensors is that they have to wait for the measured pressure to begin increasing. This threshold must be set high enough so that random noise in the sensor will not cause false triggering. The result is that barometric altimeters are always late. The barometric sensor does have one big advantage over the accelerometer: it is actually measuring something that varies with altitude.

Examples of two problematic flights

Flight 1: Dave Schaefer's Level III certification flight.

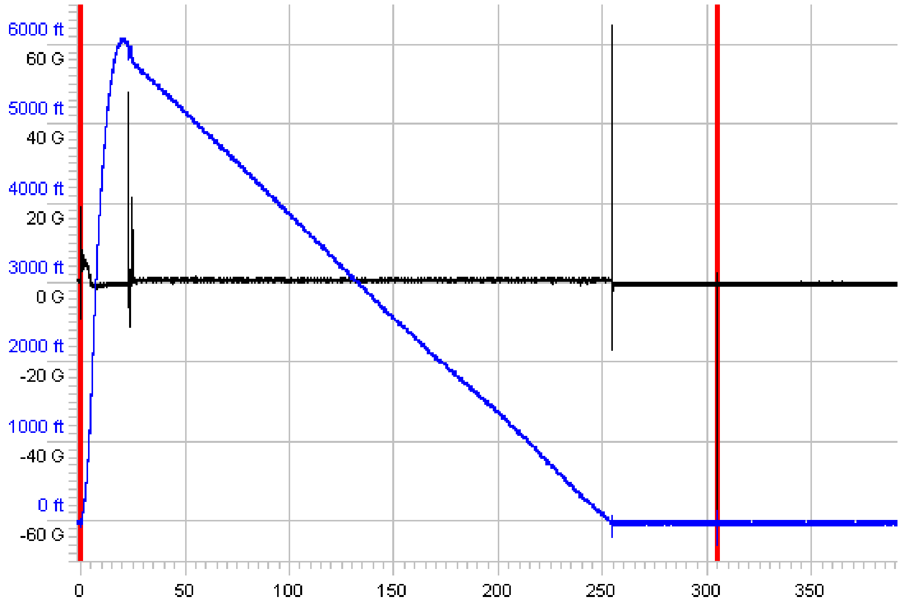

Figure 1, Complete plot of Dave's level III data

Figure 1. is a plot of the RDAS data recorded during Dave's flight. (acceleration is black, altitude is blue, and events are vertical red lines)What surprised (and alarmed) Dave was that the RDAS fired its apogee and main charges after the rocket was on the ground!

The RDAS determines the time of apogee by integrating acceleration values to get velocity. When velocity reaches zero it has detected apogee and fires the apogee charge. Only then does it begin checking to see if the altitude has decreased below the level set for the main parachute.

The required redundant altimeter used for this flight was a Missile Works RRC2, which uses barometric apogee detection. Lets zoom in and look at the events at apogee. (See Figure 2.)

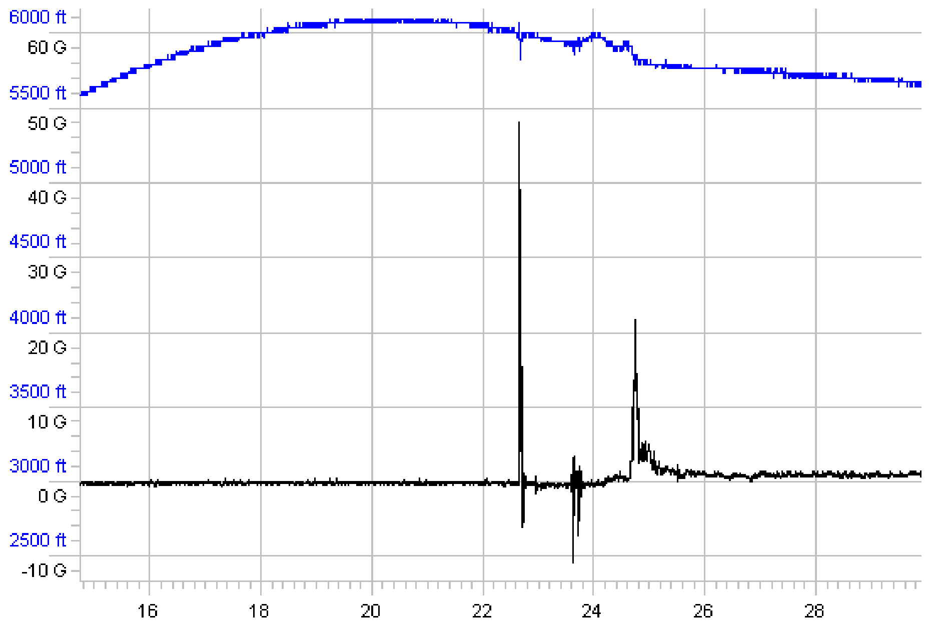

Figure 2, Detail view of apogee data

The acceleration spike at ~22.5 seconds is from the RRC2 firing its apogee charge. Notice that it is approximately 2 seconds past apogee. This charge separated the motor and nose sections of Dave's Nike Smoke and the motor section then recovered on its own parachute. Another acceleration spike is seen at ~23.5 seconds and this is the second event on the RRC2, which was set for one second after the first. This fired the parachute for the nose section out of the tip of the nose cone. The nose section then came down with "the pointy end up" and this is where the RDAS was located.

The next graph shows what the RDAS velocity estimate was doing. (See Figure 3.)

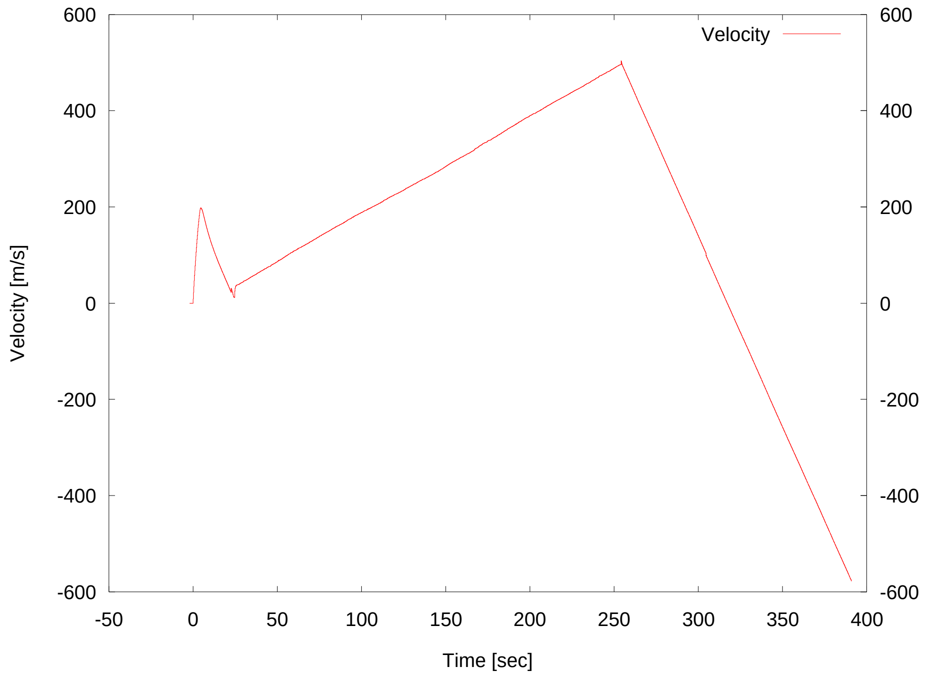

Figure 3, RDASPlot output showing velocity

Notice that the estimated velocity never reached zero at apogee. And once the nose section was under parachute it started "increasing" in velocity again. If the correct offset had been used in the RDAS integral, this period should have shown an acceleration of zero. Instead it was accelerating at ~2.2 m/s. Only after the nose was on the ground (and on its side) did the velocity estimate start decreasing.

Flight 2. The unusual flight of my PAC-3 model.

The first flight of my RDAS was on board a 4" diameter model of a PAC-3 using a K560 motor. This flight exhibited poor dynamic stability with an oscillating flight angle that got progressively worse as the K560 burned. Figure 4 shows the flight data.

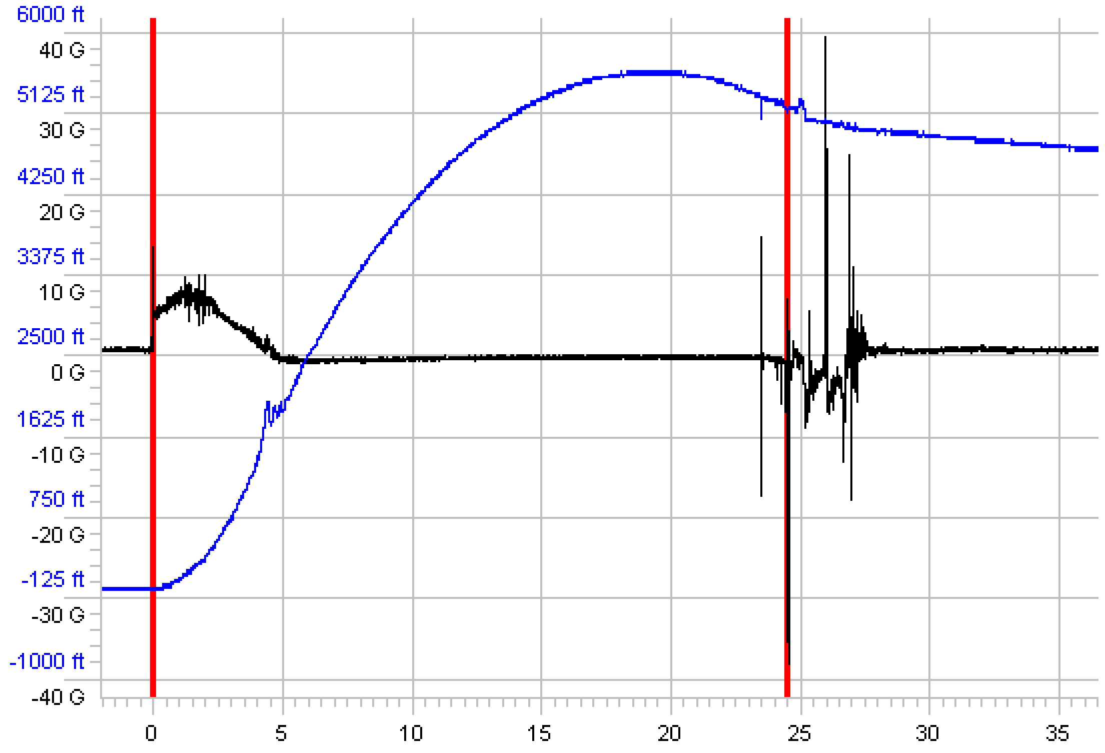

Figure 4, My PAC-3 flight data

Figure 4 clearly shows that the RDAS detected apogee around five seconds late. But the other altimeter on board (a Black Sky AltAcc which is also accelerometer based) was late as well, just not quite so late. The spike in acceleration at 23.5 seconds is from the AltAcc ejection charge) Figure 4 also shows an oscillation in the pressure data at 4 seconds. This was caused by the oscillating flight angle.

While looking at the data from these two flights I decided that there had to a better method of detecting apogee than was used by any of these altimeters.

Improving barometric detection

Can a barometric sensor detect apogee with greater accuracy than an accelerometer and can its problems be overcome? The answer to both of these questions is yes.

The primary limitations of barometric apogee detection are due to quantization error and noise. Quantization error is caused by the analog to digital conversion process having a finite resolution. Typical converters used in altimeters have between 8 and 12 bits of resolution. With an 8 bit converter, the smallest pressure difference that can be measured is equivalent to approximately 100 feet of altitude. This varies with altitude because of the non-linear relationship between pressure and altitude. The resolution also depends on the pressure range being measured. If you are only interested in altitudes up to 5,000 feet you will have much better resolution than if you want to be able to measure up to 100,000 feet.

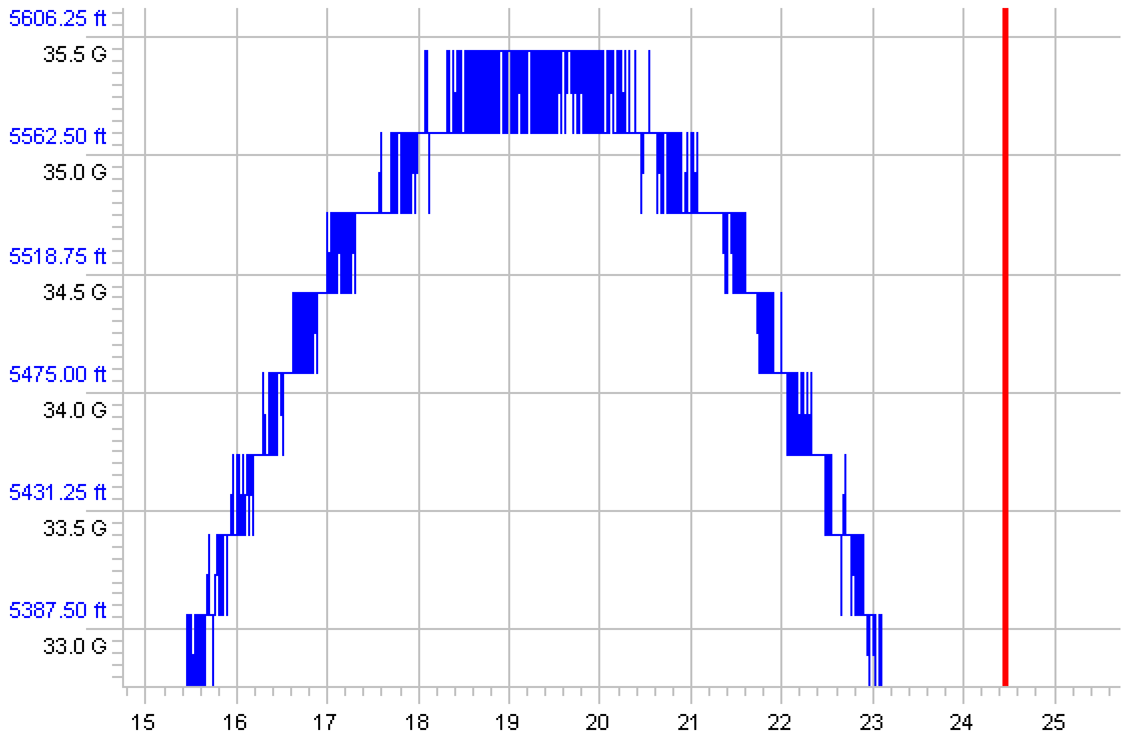

Figure 5, PAC-3 flight apogee

Figure 5 shows data from my PAC-3 flight zoomed in to show the data around the time of apogee. This plot illustrates both quantization error and noise. When was apogee? I don't know but it was somewhere between 18.5 and 20.5 seconds. The pressure is jumping back and forth between two different values because of noise. The stair-stepped nature of the altitude data is the result of quantization error (sampling).

To get an idea of the statistics of the noise, I collected some static data from the RDAS and then computed the mean, standard deviation, and variance. They were:

Mean (average) =	930.8154168 counts
Standard deviation =	0.444125467
Variance =	0.197247431

For those not familiar with statistics, by definition 99% of all data points will be within +/- 3 sigma (standard deviation) of the mean, or +/- 1.3 counts in this case.

I also performed a spectral analysis of the pressure sensor noise using an FFT. This verified that the sensor noise was white. White noise has equal power at all frequencies. This is a good indication that the noise is truly random.

Isn’t noise a bad thing?

Consider what the data would be like as we approached and passed apogee if there were absolutely no noise. The pressure will be decreasing (and the pressure sensor output voltage) and it will pass the threshold between two values in the ADC (go from 713 to 712 counts for example). It will then hold steady at that value for a while and then drop back (to 713 for example) when the pressure starts increasing again.

We really can't nail down the time of apogee very well because there is an uncertainty of 1 count in the final pressure. When the pressure changed from 713 to 712 counts that might be as far as it got. Or it might have almost made it to the point where it would switch to 711 counts. But not quite. With the 10 bit ADC used in the RDAS that would be pretty good since 1 count is about 30 feet (depending on altitude).

But the RDAS pressure reading does have noise in it. The pressure sensor outputs a voltage that on average is correct but has random noise added to it. If we could eliminate the noise by averaging (or filtering) several readings we could get the true value and smooth out the noise as well. Because the noise is random, the voltage we see will be higher (or lower) than the true value half of the time. So if the true value were 713 we would expect to see 713 part of the time. The readings that were not 713 would be equally split between being higher or lower than 713. This would also be true if the true value were 713.5 counts. Even though the ADC cannot output fractional values, we can still get them by averaging the data.

Filtering the Data

First of all lets look at using a simple filter to average several pressure sensor readings and smooth out the data. The filter I have chosen is an recursive filter since it is very simple and doesn't require a lot of data storage or processing so it can be easily implemented on even the simplest micro-controller. This filter is roughly equivalent to a moving average filter except that it does not require the storage of prior samples. The basic idea is to estimate the current altitude by taking a weighted average of our last estimate and the current ADC reading. The formula for this is:

$$P_{n+1} = x * ADC + (1-x) * P_n$$

where:
P_n is our last estimate
P_{n+1} is our new estimate
ADC is the latest reading from the ADC
x is the weight ($0 <= x <= 1$)

Note that when x is 0.5 we get:

$$P_{n+1} = (ADC + P_n)/2$$

or just the average of the two values.

So how does it work?

The weight x can be considered the value of our confidence in the measurement from the ADC. If we are sure that it is absolutely accurate, the weight (x) would be 1. So we would always believe the ADC and ignore the last estimate.

At the other extreme would be a weight of 0. This would mean that we have absolutely no confidence in our measurement so we stick with our last estimate. Needless to say the estimate is then a constant value. Reality lies somewhere in between these two extremes.

Also, as the value of x is decreased, the response of the filter slows down. Thus there has to be a balancing act. If we use a very small value of x the data will be very smooth but it will be delayed so much that we will always detect apogee late.

Playing around with the value for x I decided that x = 1/32 is about right. (at 200SPS) This removes a lot of the noise but still responds to changes fast enough so apogee detection isn't delayed too much. The statistics for the sample data set after being filtered are:

Mean (average)	930.8154824
standard deviation	0.110939363
variance	0.012307542

Notice that now 99% of our measurements will be within +/- 0.33 counts (3 sigma) of the true value and the mean is the same.

Why 1/32? That might seem like a very strange number to use but because it is a power of 2 it allows for a very easy implementation on the micro-controllers typical of most altimeters.

If we rearrange the equation a bit we end up with:

$$P_{n+1} = P_n + x(ADC-P_n)$$

That can be done with a couple of additions and one division. And because the division is by a power of two, it can be done simply by bit shifting. We don't even need floating-point numbers to do it. We can use fixed-point numbers instead.

The chosen number representation will have the binary point (just like a decimal point except this is base two) shifted 8 digits to the left. That way we have 24 bits for the integer part and 8 bits for the fractional part. Addition and subtraction work just the same as for regular binary numbers. The only tricky part is multiplication and division. But for this special case we will sidestep that.

So now for an example:

We start out with our old estimate for the altitude and the new ADC reading. We will assume that they are stored in locations called ESTALT and ADCCNT respectively. So here is the pseudo assembly code to do the job.

```
move.l  ESTALT, D0     ; Get a copy of the old altitude estimate
shr.l   #5, D0         ; divide it by 32
move.l  ADCCNT, D1     ; get a scratch copy of the new ADC count
shl.l   #3, D1         ;  shift it left 3 bits to line up the binary points
sub.l   D1, D0         ; subtract ADCCNT from ESTALT
add.l   D0, ESTALT     ; update altitude estimate
```

(Any resemblance between my pseudocode and MC68000 assembly language is purely intentional.)

Because of my choice of both x and the number representation I get to cheat a little on the math. In order to shift the binary point of the ADC value to the correct location I would need to shift it 8 bits to the left. But I also need to divide it by 32. Which is the same as shifting it 5 bits to the right. Eight to the left and five to the right is the same as 3 to the left.

Notice that after filtering we no longer have the 10 bit integer value we started with. We now have a 32 bit fixed point number. This will enable us to detect changes that are less than 1 ADC count (1 Least Significant Bit (LSB)). A 32 bit number is not really needed. What is required is the 8 fractional bits we added plus the bits coming from the ADC, ten in this case. So we need an 18 bit number. Most microprocessors provide support for 8 bit (and sometimes 16) numbers and can fairly easily be extended to 16, 24, 32, etc. 24 bits is enough and I used 32 bits for the C code I used to test the algorithm on a PC.

Figure 6 shows the results of filtering the data. Now it is much easier to determine the time of apogee. (Note that I am using the data processed to provide altitude rather than the raw ADC counts.)

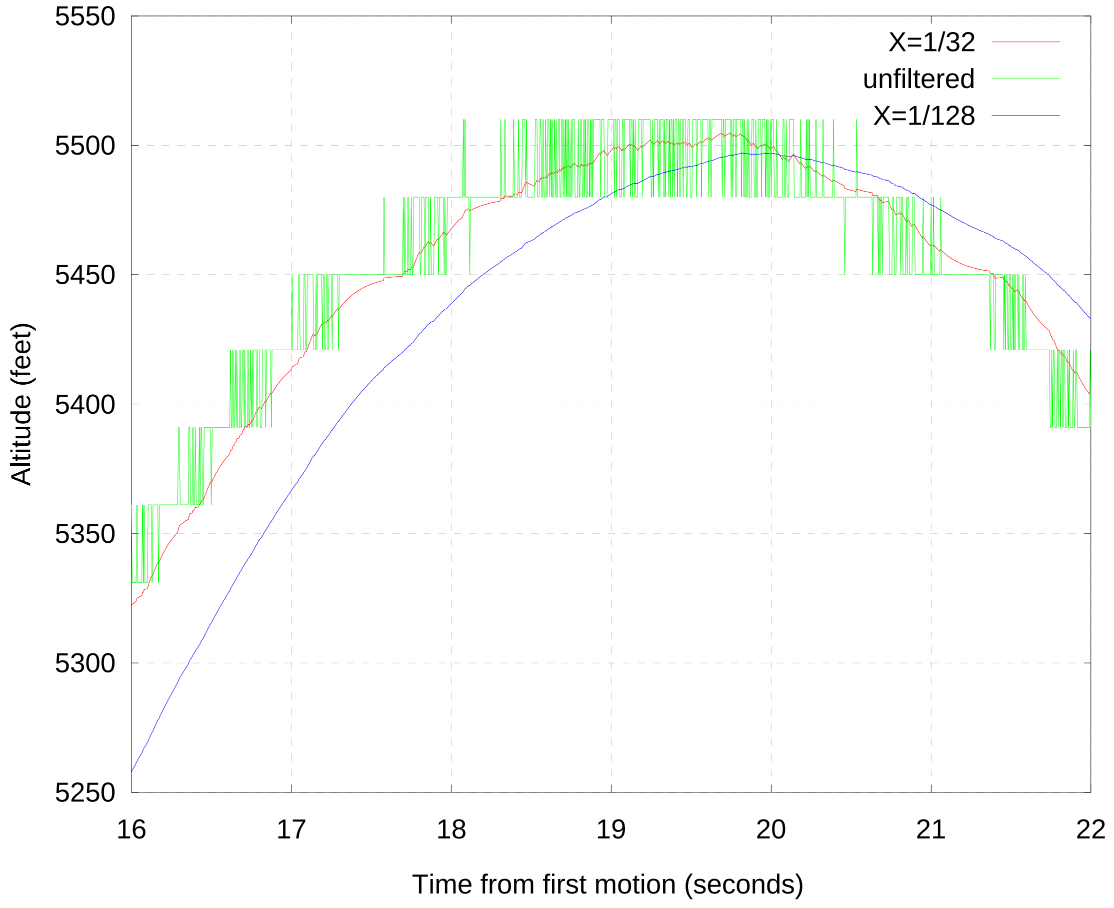

Figure 6, Filtered Pressure data

The tradeoffs in this filter are in how aggressive to make the filter (the value of X) and what threshold to use for apogee detection. Making X smaller provides more smoothing of the data but at the cost of delaying the filter's response to changes. The threshold (level below max altitude) needs to be selected so that false triggering doesn't occur but higher thresholds delay apogee detection.

Avoiding Mach effects

Even if filtering the data allows us to more accurately detect apogee, how do we avoid the problems from speeds near or over Mach 1? Either use a lock out timer or use an accelerometer to estimate velocity.

Although the accelerometer is not very good for detecting apogee, it can be used to get a reasonable estimate of the vehicles velocity and is very good (with most motors) at detecting liftoff. We can then use that estimate of velocity to disable apogee detection until it is below a safe level. We only need a rough estimate of velocity so that the barometric apogee detection can be locked out when we are near Mach 1. Since Mach 1 is about 1100 ft/sec we can set the level to disable apogee detection at a wide range of levels and be safe. If we use 500 ft/sec then the errors in velocity estimation will not be a problem. If the estimate is off by even 300 ft/sec we would be OK.

Algorithm Testing

So I have an idea for an improved apogee detection algorithm. How about testing it? I had data from a few flights available but I really needed a larger data set. So I posted a call for data on the RDAS mailing list. I received several responses and ended up with a nice sample of data. Some of the data was from flights that exceeded Mach 1 and there was a lot of variation in the flight profiles. I want to express my thanks to everyone who provided data.

I wrote a quick and dirty program (see Appendix A) to process the raw data files exported from the RDAS data analysis program. This program was not intended to represent the final algorithm to be used in flight software but was intended to test the concept using raw RDAS data files. After processing the data file, the test program would then print out the following:

The time of apogee as detected by an unbiased velocity estimate (integrated acceleration).
The time that apogee detection was enabled. (Velocity Gate Off)
The barometric apogee time as detected with three different thresholds: ½, ¼, and 1/8 LSB.

The threshold for enabling apogee detection was set to approximately 200 feet/sec.

Table 1 shows the results along with the RDAS apogee time and a "true" apogee time determined by looking at the data with a Mark I eyeball.

Table 1, Recursive Filter Test Results

Flight	Acceleration Apogee	Velocity Gate Off	Pressure Apogee (1/8LSB)	Pressure Apogee (1/4LSB)	Pressure Apogee (1/2LSB)	RDAS drogue time	"True" Apogee (approximate)
My PAC-3 K560 Flight	22.83	16.66	19.87	20.05	20.29	24.45	19.5
Nike Smoke M1315	20.825	15.235	20.765	20.86	21.205	304	20.6
Project P	13.710	8.335	13.87	13.87	13.870	13.85	14.2
Odlid (J330)	12.445	6.615	12.28	12.43	12.755	14.65	12.3
IO_g64	8.450	3.010	9.54	9.54	9.540	NA	8.45
Orange2_I161	12.925	7.845	14.125	14.125	14.125	14.6	13.5
Rosson 1	14.385	10.190	14.265	14.305	14.405	NA	Unknown, early deployment
Rosson 2	15.25	9.82	15.54	15.74	16.11	NA	15.0
Rosson 3	19.435	13.77	18.44	19.115	19.495	19.65	18.5
Rosson 4	23.845	17.975	22.90	23.690	24.445	33.7	23.0
Rosson 5	20.310	14.93	21.09	21.34	21.74	NA	21.0
Rosson 6	26.05	20.65	27.56	27.83	28.31	31	27.2
Rosson 7	16.0	10.40	15.27	16.08	16.73	15.6	16.2? (RDAS was early)
Rosson 8	11.05	9.38	24.3	24.73	24.95	25.1	23.5
Rosson 9	21.06	15.36	21.47	22.11	22.62	25.83	21.5
Rosson 10	20.6	14.6	20.86	20.98	21.17	22.0	20.5
Rosson 11	21.56	15.97	21.22	21.95	22.345	NA	21.1
Rosson 12	21.79	15.85	21.76	22.35	22.74	27.15	21.4
Rosson 13	18.53	13.27	19.03	19.18	19.475	NA	18.8

The results were much better than I had hoped. The 1/8 LSB threshold value (~5 ft.) was early on only one flight! The ¼ LSB threshold was typically within ½ second of apogee and never more than 1 second after. This is a big improvement over the accelerometer algorithm used in the RDAS.

Applications

This technique can be used on any altimeter using a barometric pressure sensor that meets the following conditions:

1) The sensor data has noise with a standard deviation of about 1/2LSB.
2) The sensor is sampled at a reasonable rate. (At least 100 samples/second)
3) Some method is used to disable apogee detection early in flight (velocity inhibit or timer).

Because the delay introduced by the filter is in terms of samples, increasing the sample rate will decrease the delay in terms of time. Increasing the sample rate to 1000 SPS would decrease the delay from an X=32 filter to ~30ms.

While filtering the pressure data certainly improved performance, I still felt that something was missing.

Kalman Filter

One key piece of information was completely ignored by the simple filter: knowledge of the vehicle dynamics. At apogee the rocket is essentially in free fall and aerodynamic drag has a minimal effect on vertical velocity. This allows for a very simple model of the rockets motion.

$$\dot{p} = v$$
$$\dot{v} = a$$
$$\dot{a} = 1$$

Which is a simple way of saying that the time rate of change of pressure altitude is velocity, the time rate of change of velocity is acceleration, and acceleration does not change with time. This representation has two drawbacks. The first is that it requires differential calculus to understand. The other is that it is a continuous time representation.

Because an altimeter will be sampling the pressure reading at discrete intervals, the model must also be discrete. The basic parameter is then Δt, which is the time between samples. The primary assumption used to simplify the model is that acceleration is constant. The acceleration is of course never constant because of motor thrust and constantly changing drag forces. However, at apogee as the velocity is decreasing and the motor has long since burned out, the acceleration is very nearly constant. The equations of motion are then:

$$a_{t+\Delta t} = a_t$$

$$v_{t+\Delta t} = v_t + a_t * \Delta t$$

$$p_{t+\Delta t} = p_t + v_t * \Delta t + a_t * \Delta t^2/2$$

Where a, v, and p are acceleration, velocity, and pressure altitude respectively and represent the state of the system. This representation should be recognizable to anyone who has had high school physics.

The model could be expanded to include the acceleration due to drag but the mass of the vehicle and coefficient of drag are not known. The killer is that drag is proportional to the square of velocity, which would make the model non-linear. There are extensions to the Kalman filter that allow non-linear system models but that is a bit more than I wanted to learn about the Kalman filter and it turns out that this simple model works very well.

The Kalman filter is useful anytime you have a system you want to know the state of and you have a model of the system and one or more measurements of the system, usually corrupted by noise. It is also possible in certain circumstances to see all of the states of the system even if you do not have measurements of them.

The Kalman filter provides an "optimal" estimate of the state of a system given a model of the system, measurements of the system, initial conditions, and knowledge of the statistics of the model noise and measurement noise. The filter is optimal in the sense that it minimizes the variance between the estimated state and the true state. Kalman filters have been around since the dawn of the space age and have seen wide application in guidance and control systems.

The basic process of the Kalman filter is to predict the state of the system using a model of the system. If the model is accurate then the prediction will be very close. This is then corrected using a measurement of the system. The measurement is assumed to be of the true state of the system but is corrupted by noise. The most complicated part of this is in determining the "Kalman gain" used to determine the correction.

The Kalman gain is very similar to the gain used in the recursive filter shown earlier. The primary difference in the filters is that the earlier filter corrected the prior estimate and the Kalman filter corrects the prior estimate after it has been propagated forward in time using the system model.

While I will present the equations for the Kalman filter, the details of its operation are beyond the scope of this paper and you should consult a good text on the subject if you would like to learn more. (Gelb, A. Editor, "Applied Optimal Estimation", The M.I.T. Press, Cambridge MA, 1974 for example)

Kalman Filter Equations

The state update equation is:

$$\hat{x}_k^{(-)} = \Phi \hat{x}_{k-1}^{(+)}$$

Where:
$\hat{x}_k^{(-)}$ is the propagated system state estimate.
Φ is the state transition matrix.
$\hat{x}_{k-1}^{(+)}$ is the final state estimate from the prior cycle of the Kalman filter.

In expanded form:

$$\begin{matrix} \hat{x}1_k^{(-)} \\ \hat{x}2_k^{(-)} \\ \hat{x}3_k^{(-)} \end{matrix} = \begin{bmatrix} 1 & T & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x}1_{k-1}^{(+)} \\ \hat{x}2_{k-1}^{(+)} \\ \hat{x}3_{k-1}^{(+)} \end{bmatrix}$$

Where:
T is the time between estimates.
$\hat{x}1$ is the position estimate
$\hat{x}2$ is the velocity estimate
$\hat{x}3$ is the acceleration estimate

The notation (-) is used to indicate the value of the estimate after the update but prior to correction. (+) indicates the value of the estimate after correction.

This is the equation that applies the system model to update the estimate. Note that the time between samples is the key value here. When performing the calculations on a PC with full floating point numbers, this value is not very important and can be whatever we want it to be. But when trying to stuff this computation into a micro-controller it is very important. For an arbitrary value of T, a micro-controller would be hard pressed to perform the calculation. But if we make the sample rate be a power of two, then T is also a power of two. (T = 1/sample rate) The result is that we can multiply by T simply by bit shifting, which micro-controllers can do easily.

The state correction equation is:

$$\hat{x}_k^{(+)} = \hat{x}_k^{(-)} + K_k \left[z_k - H_k \hat{x}_k^{(-)} \right]$$

Where:
K_k is the Kalman gain matrix
z_k is the measurement vector
H_k is the matrix that maps the state space onto the measurement space

Notice that this equation looks very much like the earlier recursive filter. The matrix notation makes this look fairly complicated and in general it is. But because we only have one measurement, z_k and H_k have only one non-zero element each (and the non-zero element in H_k is equal to 1) . Thus multiplying by the Kalman gain matrix (K_k) requires only three multiplies. In fact, the result is three equations just like the simple recursive filter- one each for altitude, velocity, and acceleration. Only now we cannot select a gain that is computationally convenient but must use the Kalman gain.

The determination of the Kalman gain is very computationally intensive and beyond the capabilities of most micro-controllers. However, we can take advantage of the fact that the Kalman gain depends only on the system model and statistics of the model and measurement. In general these are time varying but in our case they are constant and therefore can be pre-computed by a more capable computer and then the Kalman gain can be hard coded into the micro-controller. The equations required to compute the Kalman gain are:

Error Covariance Update

$$P_k^{(-)} = \Phi_{k-1} P_{k-1}^{(+)} \Phi_{k-1}^T + Q_{k-1}$$

Error Covariance Correction

$$P_k^{(+)} = \left[I - K_k H_k \right] P_k^{(-)}$$

Kalman Gain

$$K_k = P_k^{(-)} H_k^T \left[H_k P_k^{(-)} H_k^T + R_k \right]^{-1}$$

Where:
K_k is the Kalman gain matrix
P_k is the error covariance matrix
R_k is the measurement noise covariance matrix

Q_k is the model noise covariance matrix
superscript "T" indicates matrix transpose
superscript "-1" indicates matrix inversion

Because of the matrix inversion, the Kalman gain computation is complicated. Fortunately, with small matrices, closed form solutions to the inversion exist. Still this is far too much to ask of a micro-controller especially since the Kalman gain matrix rapidly converges to a constant value.

For the case of constant Kalman gain the required computation simplifies to:

State Update:

$$\hat{x}1_k^{(-)} = \hat{x}1_{k-1}^{(+)} + T * \hat{x}2_{k-1}^{(+)} + \frac{T^2}{2} * \hat{x}3_{k-1}^{(+)}$$
$$\hat{x}2_k^{(-)} = \hat{x}2_{k-1}^{(+)} + T * \hat{x}3_{k-1}^{(+)}$$
$$\hat{x}3_k^{(-)} = \hat{x}3_{k-1}^{(+)}$$

State correction:

$$\hat{x}1_k^{(+)} = \hat{x}1_k^{(-)} + K1\left(z - \hat{x}1_k^{(-)}\right)$$
$$\hat{x}2_k^{(+)} = \hat{x}2_k^{(-)} + K2\left(z - \hat{x}1_k^{(-)}\right)$$
$$\hat{x}3_k^{(+)} = \hat{x}3_k^{(-)} + K3\left(z - \hat{x}1_k^{(-)}\right)$$

Where:

$K1, K2, K3$ are the Kalman gains
z is the pressure measurement

Because the model of the rockets dynamics includes velocity and acceleration, we also get estimates of these values. Because the model assumes constant acceleration, we will see the biggest errors in the estimates at the times when the acceleration is changing.

I wrote a program to process RDAS data files using a Kalman filter and then used it on several sets of data. Since this was my first try at implementing a Kalman filter, I started first by working on programs for filters using constant position and constant velocity models. I learned a lot about the process but neither of these models provided good results because the models did not match the system.

Example of Kalman filtered data

For processing the data I used two slightly different versions of the program. One output the data in a text file so that it could then be plotted. The other only printed liftoff and apogee detected times. (This program is listed in Appendix B). It turns out that the data from my PAC-3 flight provides an excellent example of the algorithm because of the pressure data oscillation.

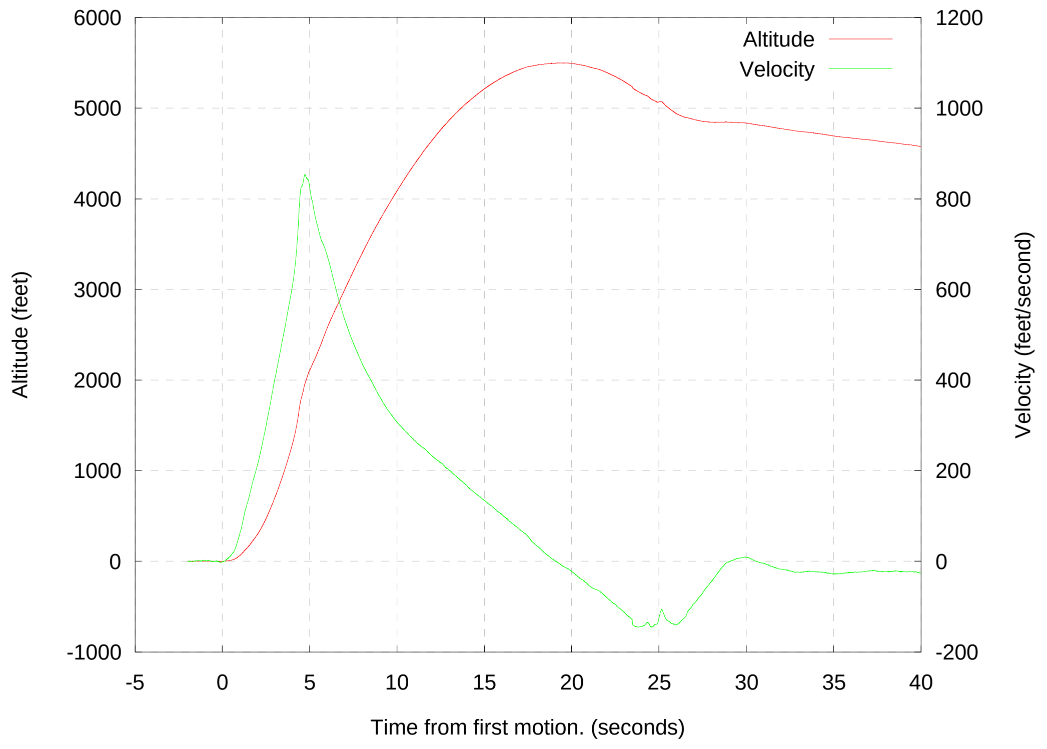

Figure 7, Kalman filtered pressure data

Notice in Figure 7 that the oscillation in the altitude at 5 seconds is gone. There is a slight disturbance in the data but at no time does it show a decrease in altitude. But the velocity estimate is the real gem. It stays positive from first motion until apogee. Therefore the apogee detection criteria is very simple: when does velocity change from positive to negative? To see just how well the Kalman filtered data behaves at apogee, see Figure 8.

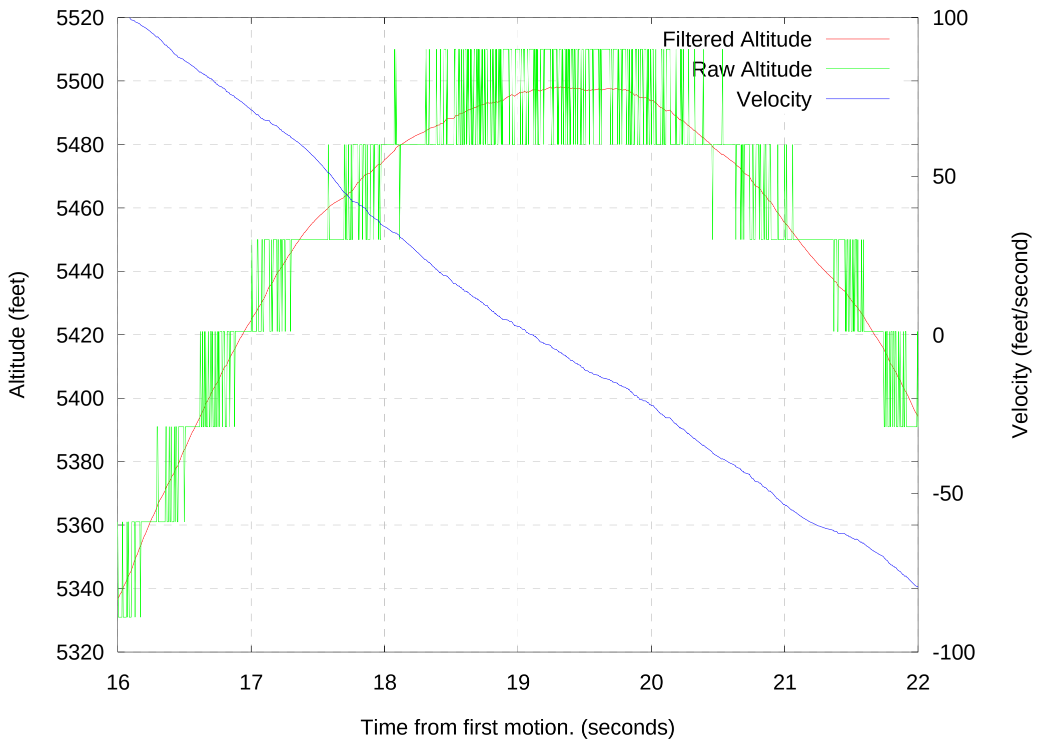

Figure 8, Kalman filtered pressure data at apogee

Notice in Figure 8 that the filtered altitude is not delayed at all with respect to the measured data. This is a vast improvement over the simple recursive filter. With that filter, the smoother the data was, the more it was delayed.

Recall that velocity is the time rate of change of position. So to get velocity from the altitude data the natural thing to do is to divide the altitude difference by the time difference between the two measurements. If you were to do this directly, you would have a very poor result. Because of the measurement noise, the altitude is bouncing around 30 feet between each sample. Therefore the velocity computed this way could and usually would change from a positive 30ft * 200SPS or 6000ft/sec to negative 6000ft/sec from one sample to the next. This could be improved by low pass filtering the data first but the low pass filter also delays the data in time. The magic of the Kalman filter provides a good low noise estimate for us that is not delayed.

Kalman filter compared to recursive low pass filter

Figure 9 is a plot of the unfiltered altitude data along with the recursively low pass filtered data and the Kalman filtered data.

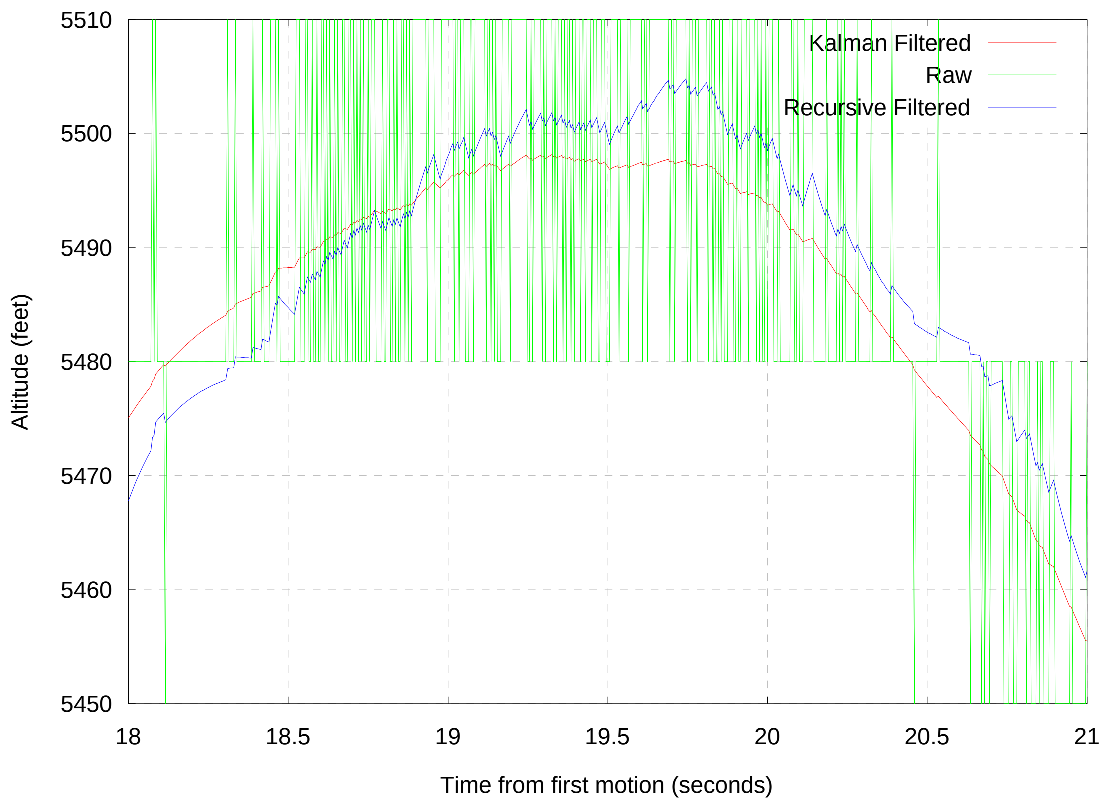

Figure 9, Comparison of Raw and Filtered Data

Figure 9 clearly shows that the Kalman filter has smoothed the data much better than the recursive filter. Plus it does not have the delay that is also clearly visible in the recursively filtered data. While the Kalman filter smooths the data sufficiently to be able to resolve the altitude at apogee to less than one foot, this is not really very useful since the accuracy of the pressure data is far worse than this. What the Kalman filter provides that is extremely valuable in determining the time of apogee is its velocity estimate.

The Kalman velocity estimate provides information that is easily used to determine the time of apogee. Simply watch for the velocity to change from positive to negative. It can also be used to detect liftoff if an accelerometer is not available. Simply wait for the velocity to exceed some reasonable value (100 ft/sec for example). What is even better is that even though the Kalman filter is affected by the pressure effects of passing through Mach 1 (or wild changes in measured pressure), the velocity estimate does not go negative. This is because the filter is tuned to place much more weight on the dynamic model of the vehicles motion than to the measurement.

Kalman Filter Tuning

There are two parameters that are used to tune this Kalman filter. They are the statistical parameters of the measurement noise and the model noise. The concept of measurement noise is fairly obvious and easy to measure. The concept of model noise is not obvious at all. The model noise is due to the model not accurately capturing the system dynamics. For example, my model ignores drag and this is part of the model noise.

Since it is very difficult to quantify the model noise, I used this as the primary parameter to tune the filter response. I used the measured value for the pressure sensor noise and then set the model noise relative to that. Because I am primarily interested in the time at apogee the error due to ignoring drag is very low at that time. Therefore I used a value for the model noise that was very small with respect to the measurement noise. This causes the filter gains to favor the model over the measurement. The result is that the filter tracks very well near apogee but does poorly at other times. To see how the filter is affected by tuning, I determined the apogee time for a wide range of ratios between measurement and model variance for two data sets with anomalies in the pressure data. One data set is from my PAC-3 flight with the pressure oscillation and the other is from "Rosson 4" which was a supersonic flight. The results are shown in Figure 10.

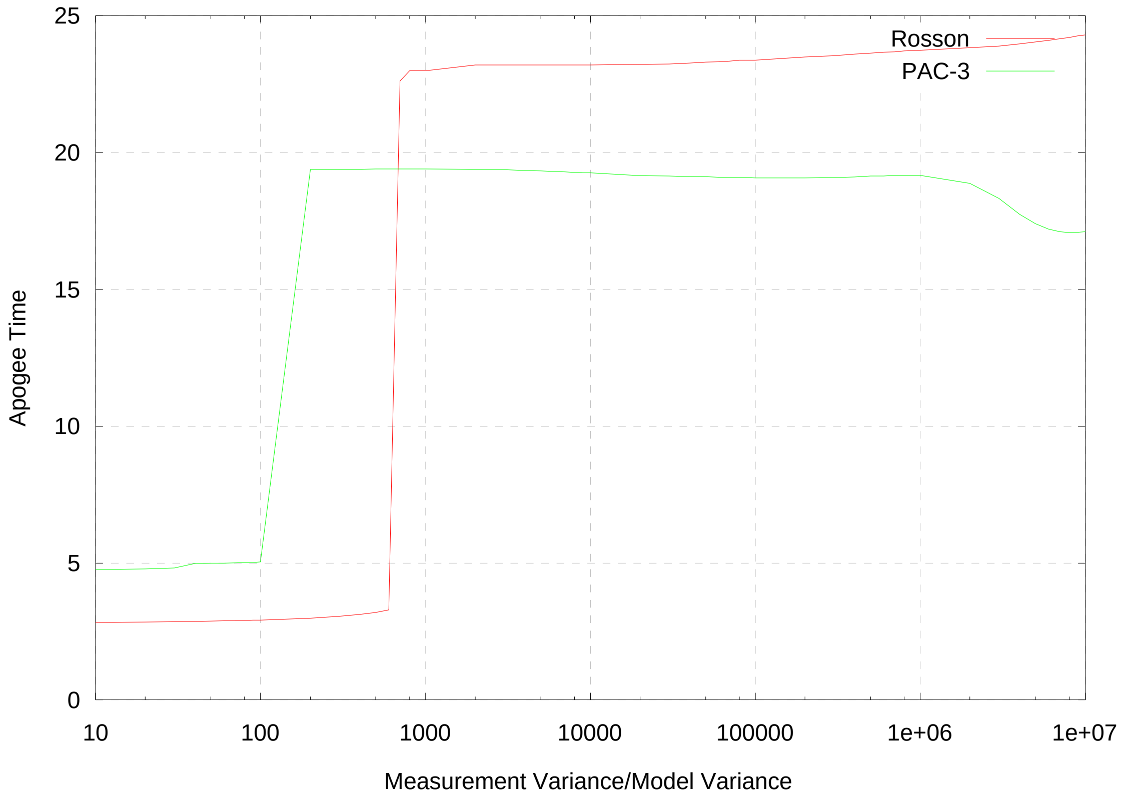

Figure 10, Kalman Filter tuning sensitivity

The filter shows good performance over three orders of magnitude in the ratio of measurement to model variance. The breakdown at low values is due to tracking the oscillation in pressure that occurred there too closely. At high values the filter is very nearly ignoring the pressure measurement. The best results occur in the range of 1000 to 1,000,000. Higher values than the minimum are needed to increase the margin of error during pressure transients caused by Mach transitions or other causes.

Kalman filter for the AltAcc

As an exercise, I modified my code so that it could read the BlackSky AltAcc altimeter data files from my PAC-3 flight. Figures 11 and 12 show the results.

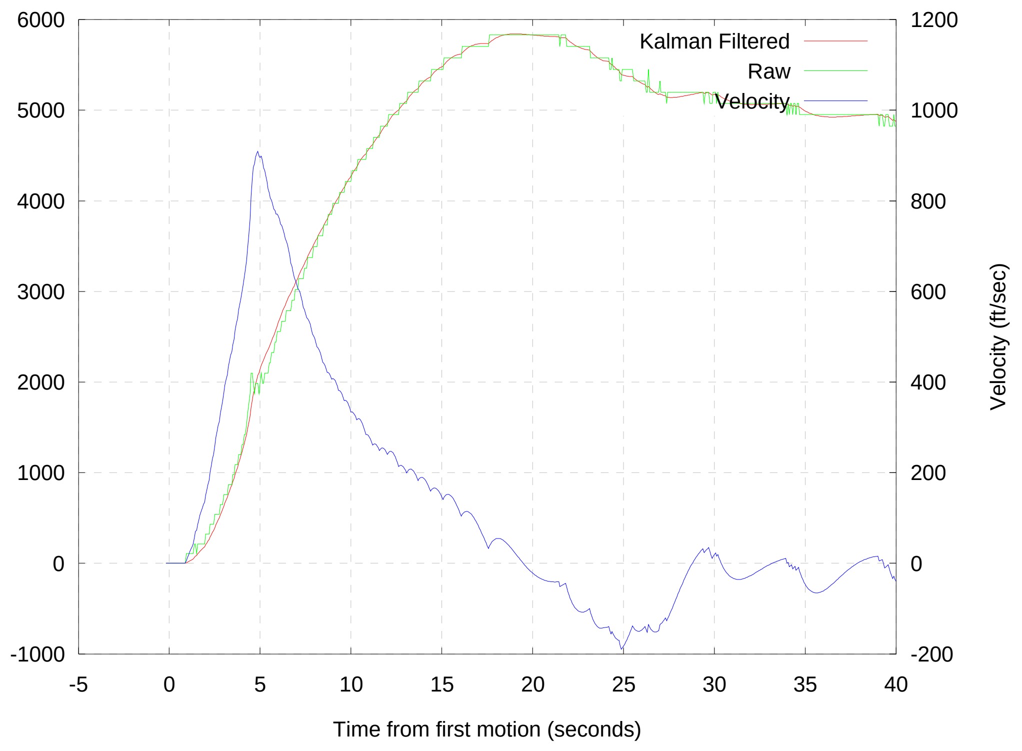

Figure 11, Filtered AltAcc data

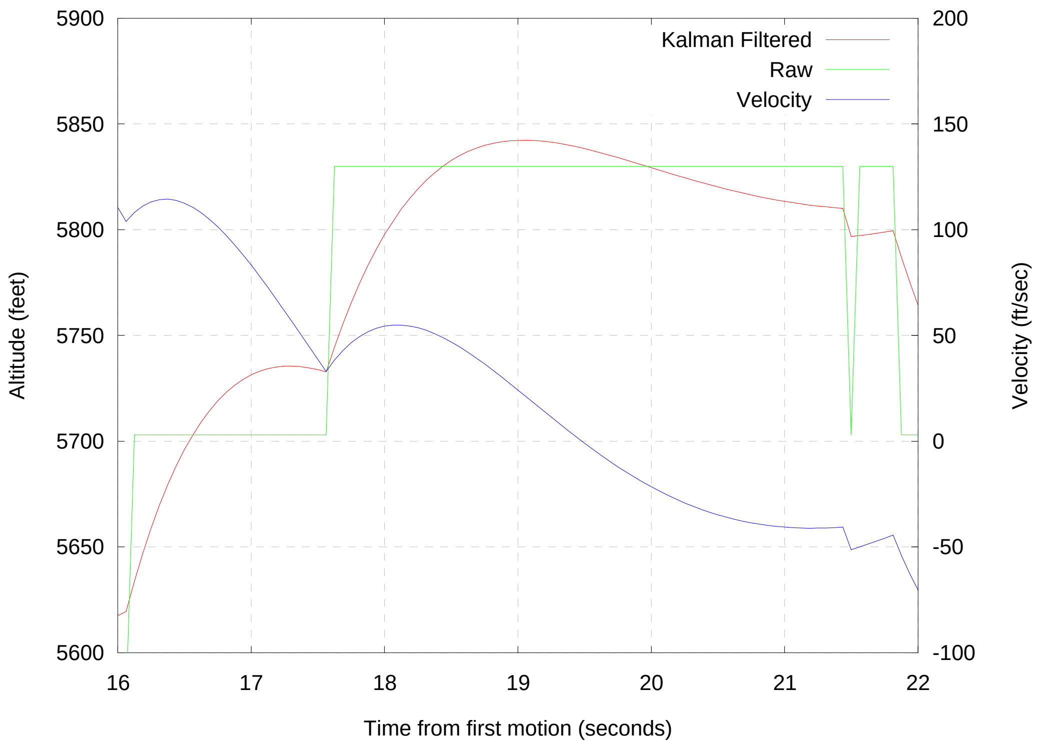

Figure 12, Filtered AltAcc data at apogee

After processing the AltAcc data, I was amazed to discover how well the Kalman filter performed. The AltAcc has a much lower sample rate (18 samples per second vs. 200), lower resolution (8 bit ADC vs. 10 bit), and almost no sensor noise. In spite of this, the Kalman filter shows apogee at 19.5 seconds. Adding some sensor noise would improve the performance.

Kalman Filter Testing

To see how well the Kalman filter works, I ran the same data used to test the simple recursive filter. The results using a measurement to model variance ratio of ~50,000 are shown in Table 2.

Table 2, Kalman Filter Test Results

Flight	Acceleration Apogee	Recursive Filter Apogee (1/4LSB)	Kalman Apogee Time	RDAS drogue time	"True" Apogee (approximate)
PAC-3 K560	22.83	20.05	19.12	24.45	19.5
Nike Smoke M1315	20.825	20.86	20.25	304	20.6
Project P	13.710	13.87	13.69	13.85	14.2
Odlid (J330)	12.445	12.43	12.005	14.65	12.3
IO_g64	8.450	9.54	8.405	NA	8.45
Orange2_I161	12.925	14.125	13.12	14.6	13.5
Rosson 1	14.385	14.305	14.88	NA	Unknown, early deployment
Rosson 2	15.25	15.74	14.985	NA	15.0
Rosson 3	19.435	19.115	18.44	19.65	18.5
Rosson 4	23.845	23.690	22.94	33.7	23.0
Rosson 5	20.310	21.34	20.66	NA	21.0
Rosson 6	26.05	27.83	26.62	31	27.2
Rosson 7	16.0	16.08	15.38	15.6	Unknown, early deployment
Rosson 8	11.05	24.73	23.235	25.1	23.5
Rosson 9	21.06	22.11	21.33	25.83	21.5
Rosson 10	20.6	20.98	20.065	22.0	20.5
Rosson 11	21.56	21.95	21.185	NA	21.1
Rosson 12	21.79	22.35	21.37	27.15	21.4
Rosson 13	18.53	19.18	18.5	NA	18.8

The Kalman filter results are a vast improvement over the performance of the RDAS but it shows less improvement over the simple recursive filter. One of the more dramatic differences is in the flight labeled "Rosson 4". This flight was very emphatically supersonic with very noticeable pressure effects. The Kalman filter was not fooled by the Mach transitions and detected apogee at the correct time.

Although the Kalman filter shows only slight performance improvement relative to the recursive filter it has several advantages over the recursive filter.

First is that the measurement requirements on the recursive filter are more stringent than for the Kalman filter. As demonstrated by the AltAcc version of the filter, the Kalman filter maintains its performance with low sensor noise, low sample rates, and lower ADC resolution. All of these have serious impacts on the performance of the recursive filter. In addition, the recursive filter must be carefully tuned for best performance. There is a fine line between setting the threshold to get earliest possible apogee detection and getting false triggering. Lastly, the recursive filter must use some other technique to lock it out during Mach transitions. The Kalman filter is immune to the Mach transition effect if a suitably high measurement to noise variance ratio is used.

Conclusions

The Kalman filter velocity estimate consistently goes through zero (time of apogee) slightly before apparent apogee (determined by looking at the data: a very subjective measurement). It is far more accurate at determining apogee than the RDAS over a wide range of cases and better than the AltAcc and RRC2 in two specific cases. Its one drawback is that it is consistently early by a few tenths of a second. However, I consider this to be far better than deploying sometime (usually several seconds) after apogee. With a slightly early ejection, by the time the parachute begins to inflate, the rocket will be right at apogee. This will decrease the loads on the recovery system. Not to mention the rocketeer.

The Kalman filter provides a robust, accurate, and repeatable method of determining the time of apogee. It will work with noisy or noise free measurements although some noise is preferred. It works over a wide range of sample rates. (at least over the range between 18 and 200 SPS). It will also work well with either 8 bit or 10 bit data.

Because the Kalman filter will work at low sample rates, most micro-controllers should be capable of performing the computations required by the filter in real time. The availability of code and data space will limit which micro-controllers will be suitable. If the computation required by the Kalman filter is greater than the altimeter can perform, then the simple recursive filter can be used to improve performance.

Future Work

1) Implementation in an altimeter and flight testing.
2) Add acceleration measurement to filter. This might allow the tuning of the filter to be loosened and should improve performance but it could be too complex for an altimeter.

NAR Rule 63.5 Required Information

Objective	Develop a more accurate method of determining the time of apogee.
Approach	Analyze recorded flight data.
Previous R&D reports by author	None
Previous work on subject:	None
Equipment Used	Personal computer
Facilities Used	None
Budget	Zero
Data Collected	See report body
Results	See report body
Conclusions	See report body
Future work	See report body

Appendix A

Source code for recursive filter.

```
#include <stdio.h>

/*
        RDASfilter.c

        This program filters data files output by the RDAS program
        (either raw or interpreted) and outputs a new file. The filter
        is a simple recursive filter and the weight can be specified on
        the command line.

        Usage:

        rfilter [weight] <infile >outfile

        The weight is an integer greater than or equal to one. Larger
        numbers provide more filtering. Default value of weight is 32.
*/

int weight = 32;


main(int argc, char **argv)
{
        int     i;
        char    buf[512];
        char    Rtime[128];
        double  Raccel, Rpressure;
        int     Rdig;
        double  FilteredAcceleration, FilteredPressure;
        double  weight = 32;

        /* check for weight as command line argument */

        if(argc == 2)
                weight = atoi(argv[1]);

        printf("%s %f\n", argv[1], weight);

        /* Copy text at start of file to output. */

        while(1)
        {
                if(gets(buf) == NULL)
                {
                        fprintf(stderr, "No data in file\n");
                        exit(1);
                }
                else
                {
                        puts(buf);
                        if( strstr(buf, "2.00"))
                                break;
                }
        }

        /* Get first line of data to initialize filters */


        /* Get a line of data and chop it up.*/
        if( sscanf(buf, "%s %lf %lf %d ", &Rtime, &FilteredAcceleration, &FilteredPressure, &Rdig) ==
EOF)
                return(1);

        while(1)
        {
          if( gets(buf) == NULL)
            exit(0);
          sscanf(buf, "%s %lf %lf %d ", &Rtime, &Raccel, &Rpressure, &Rdig);

                FilteredAcceleration += (Raccel - FilteredAcceleration)/weight;
                FilteredPressure += (Rpressure - FilteredPressure)/weight;


                printf("%s %lf %lf %d\n", Rtime, FilteredAcceleration, FilteredPressure, Rdig);

        }

}
```

Appendix B

Source Code for kapogee

```
/*


 NAME
        kapogee.c -  A third order Kalman filter for
       RDAS raw data files.

 SYNOPSIS
        kapogee <infile

 DESCRIPTION:
        Performs Kalman filtering on standard input
        data using a third order, or constant acceleration,
        propagation model.

        The standard input data is of the form:

        Column 1: Time of the measurement (seconds)
        Column 2: Acceleration Measurement (ignored)
        Column 3: Pressure Measurement (ADC counts)

       All arithmetic is performed using 32 bit floats.

        The standard output data is of the form:

       Liftoff detected at time: <time>
       Apogee detected at time:  <time>


 AUTHOR
       David Schultz




*/


#include <stdio.h>
#include <string.h>
#include <math.h>


#define  MEASUREMENTSIGMA     0.44
#define  MODELSIGMA           0.002
#define  MEASUREMENTVARIANCE  MEASUREMENTSIGMA*MEASUREMENTSIGMA
#define MODELVARIANCE         MODELSIGMA*MODELSIGMA

main()
{
       int     liftoff = 0;
       char    buf[512];

       float time, accel, pressure;
       float last_time, last_pressure;
       float   est[3] = { 0, 0, 0 };
       float  estp[3] = {0, 0, 0 };
       float   pest[3][3]    = { 0.002, 0, 0,
                                0, 0.004, 0,
                                0, 0, 0.002 };
       float  pestp[3][3]   = { 0, 0, 0,
                                 0, 0, 0,
                                 0, 0, 0 };
       float   phi[3][3]     = { 1, 0, 0,
                               0, 1, 0,
                               0, 0, 1.0 };
       float  phit[3][3]    = { 1, 0, 0,
                                0, 1, 0,
                                0, 0, 1.0 };
       float  gain[3] = { 0.010317, 0.010666, 0.004522 };
       float  dt;
       float   term[3][3];

/* Initialize */
```

```
        /* Skip text at start of file. */

        while(1)
        {
                if(gets(buf) == NULL)
                {
                        fprintf(stderr, "No data in file\n");
                        exit(1);
                }
                else
                {
                        if( strstr(buf, "-2.000"))
                                break;
                }
        }
        sscanf(buf, "%f %f %f", &time, &accel, &pressure);
        est[0]  = pressure;
        last_time = time;

#ifdef DEBUG
        printf("%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf\n",
        time, pressure, est[0], est[1], est[2], sqrt(pest[0][0]),
        sqrt(pest[1][1]), sqrt(pest[2][2]), gain[0], gain[1], gain[2]);
#endif


        if(gets(buf) == NULL)
        {
                fprintf(stderr, "No data\n");
                exit(1);
        }
        sscanf(buf, "%f %f %f", &time, &accel, &pressure);

        est[0] = pressure;
      dt = time - last_time;

#ifdef DEBUG
        printf("%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf\n",
        time, pressure, est[0], est[1], est[2], sqrt(pest[0][0]),
        sqrt(pest[1][1]), sqrt(pest[2][2]), gain[0], gain[1], gain[2]);
#endif

        last_time = time;
/*
 Fill in state transition matrix and its transpose
*/
      phi[0][1]    = dt;
      phi[1][2]    = dt;
      phi[0][2]    = dt*dt/2.0;
      phit[1][0]    = dt;
      phit[2][1]    = dt;
      phit[2][0]    = dt*dt/2.0;


        while( gets(buf) != NULL)
        {

                sscanf(buf, "%f %f %f", &time, &accel, &pressure);
                if( last_time >= time )
                {
                        fprintf(stderr, "Time does not increase.\n");
                        exit(1);
                }


                /* Propagate state */

                estp[0] = phi[0][0] * est[0] + phi[0][1] * est[1] + phi[0][2] * est[2];
                estp[1] = phi[1][0] * est[0] + phi[1][1] * est[1] + phi[1][2] * est[2];
                estp[2] = phi[2][0] * est[0] + phi[2][1] * est[1] + phi[2][2] * est[2];

                /* Simplified version (phi is constant)


                estp[0] = est[0] + est[1]*dt + est[2]*dt*dt/2.0;
                estp[1] =          est[1]    + est[2]*dt;
                estp[2] =                      est[2]; */

                /* Propagate state covariance */
```

```
         term[0][0] = phi[0][0] * pest[0][0] + phi[0][1] * pest[1][0] + phi[0][2] *
pest[2][0];
         term[0][1] = phi[0][0] * pest[0][1] + phi[0][1] * pest[1][1] + phi[0][2] *
pest[2][1];
         term[0][2] = phi[0][0] * pest[0][2] + phi[0][1] * pest[1][2] + phi[0][2] *
pest[2][2];
         term[1][0] = phi[1][0] * pest[0][0] + phi[1][1] * pest[1][0] + phi[1][2] *
pest[2][0];
         term[1][1] = phi[1][0] * pest[0][1] + phi[1][1] * pest[1][1] + phi[1][2] *
pest[2][1];
         term[1][2] = phi[1][0] * pest[0][2] + phi[1][1] * pest[1][2] + phi[1][2] *
pest[2][2];
         term[2][0] = phi[2][0] * pest[0][0] + phi[2][1] * pest[1][0] + phi[2][2] *
pest[2][0];
         term[2][1] = phi[2][0] * pest[0][1] + phi[2][1] * pest[1][1] + phi[2][2] *
pest[2][1];
         term[2][2] = phi[2][0] * pest[0][2] + phi[2][1] * pest[1][2] + phi[2][2] *
pest[2][2];

         pestp[0][0] = term[0][0] * phit[0][0] + term[0][1] * phit[1][0] + term[0][2] *
phit[2][0];
         pestp[0][1] = term[0][0] * phit[0][1] + term[0][1] * phit[1][1] + term[0][2] *
phit[2][1];
         pestp[0][2] = term[0][0] * phit[0][2] + term[0][1] * phit[1][2] + term[0][2] *
phit[2][2];
         pestp[1][0] = term[1][0] * phit[0][0] + term[1][1] * phit[1][0] + term[1][2] *
phit[2][0];
         pestp[1][1] = term[1][0] * phit[0][1] + term[1][1] * phit[1][1] + term[1][2] *
phit[2][1];
         pestp[1][2] = term[1][0] * phit[0][2] + term[1][1] * phit[1][2] + term[1][2] *
phit[2][2];
         pestp[2][0] = term[2][0] * phit[0][0] + term[2][1] * phit[1][0] + term[2][2] *
phit[2][0];
         pestp[2][1] = term[2][0] * phit[0][1] + term[2][1] * phit[1][1] + term[2][2] *
phit[2][1];
         pestp[2][2] = term[2][0] * phit[0][2] + term[2][1] * phit[1][2] + term[2][2] *
phit[2][2];


         pestp[0][0] = pestp[0][0] + MODELVARIANCE;
/*
 Calculate Kalman Gain
*/
#ifndef TEST
         gain[0] = (phi[0][0] * pestp[0][0] +
                    phi[0][1] * pestp[1][0] +
                    phi[0][2] * pestp[2][0] ) / (pestp[0][0] + MEASUREMENTVARIANCE);

         gain[1] = (phi[1][0] * pestp[0][0] +
                    phi[1][1] * pestp[1][0] +
                    phi[1][2] * pestp[2][0] ) / (pestp[0][0] + MEASUREMENTVARIANCE);

         gain[2] = (phi[2][0] * pestp[0][0] +
                    phi[2][1] * pestp[1][0] +
                    phi[2][2] * pestp[2][0] ) / (pestp[0][0] + MEASUREMENTVARIANCE);

#endif


/*
 Update state and state covariance
*/
         est[0] = estp[0] + gain[0] * (pressure - estp[0]);
         est[1] = estp[1] + gain[1] * (pressure - estp[0]);
         est[2] = estp[2] + gain[2] * (pressure - estp[0]);

         pest[0][0] = pestp[0][0] * (1.0 - gain[0]);
         pest[0][1] = pestp[1][0] * (1.0 - gain[0]);
        pest[0][2] = pestp[2][0] * (1.0 - gain[0]);
         pest[1][0] = pestp[0][1] - gain[1] * pestp[0][0];
         pest[1][1] = pestp[1][1] - gain[1] * pestp[1][0];
        pest[1][2] = pestp[2][1] - gain[1] * pestp[2][0];
         pest[2][0] = pestp[0][2] - gain[2] * pestp[0][0];
         pest[2][1] = pestp[1][2] - gain[2] * pestp[1][0];
        pest[2][2] = pestp[2][2] - gain[2] * pestp[2][0];

/*
 Output
*/

#ifdef DEBUG
```

```
        printf("%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf\n",
        time, pressure, est[0], est[1], est[2], sqrt(pest[0][0]),
        sqrt(pest[1][1]), sqrt(pest[2][2]), gain[0], gain[1], gain[2]);
#else
        if( liftoff == 0)
        {
                if(est[1] < -5.0)
                {
                        liftoff = 1;
                        printf("Liftoff detected at time: %f\n", time);
                }
        }
        else
        {
                if(est[1] > 0)
                {
                        printf("Apogee detected at time: %f\n", time);
                        exit(0);
                }
        }
#endif

        last_time = time;

        }
}
```