Project in visualization

Peder Refsnes
peder.refsnes@student.uib.no

June 12, 2008

1 Preface

This project was about visualizing real-time data streams with graphs. The real-time data comes from a noisy source and it is therefore necessary to apply filters to the streams to makethem more useful for the observer. One of the goals of the project was to code several such filters. The filters were coded on top of the EnlightenMe framework. It uses a combination of C++ and python. C++ is a very fast language, but a bit low-level compared to newer object-oriented, high-level languages. For instance, there is no garbage collection. Python is a small modular scripting language that is dynamically typed[1]. This results in a language suited for rapid-prototyping since the code is very compact, but since it's an interpreted language, as opposed to C++ which is compiled, it lacks the performance of C/C++. Everything presented in this paper was coded in python.

This short paper is divided up into three sections, one for each of the tasks the project specified. The first was writing the filters. These are made from basic mathematics formulas to manipulate the real-time data in an useful way. Second is the time buffer. The time buffer adds support for variable time kernel size. Depending on what you are looking for, different time kernel sizes are useful. Last is the implementation of a variance graph using OpenGL. This displays the mean and variance of a given input set and presents them in an easy-to-grasp way.

2 Filters

The different filters all use a synthetically generated data set with normal distributed noise added. For the example screen shots in this paper three different data sets were used, two variations over a sine curve and a constant value with a high degree of noise added. The first is a sine curve with normal distributed noise added. This is used for one of the Kalman filter examples. The second is a multiscale sine curve generated from the formula:$f(t) = \sin(i/50) + 0.5 \times \sin(i/10) + 0.1 \times \sin(i)$ This data set is used to illustrate the mean and variance filter. The third, with the highest amount of normal distributed noise added, is used to show the variance graph.

One thing to note about the screen shots, the graphs do not overlap correctly in the mean and variance graphs. This is because the output value is calculate over **past** values and therefore lag behind the present value. Possible solutions to this problem would be to delay the real-time data graph until the graphs are properly aligned. Another would be to calculate the filter output values over probable future values. This is not yet supported. Still, the screen shots shows that the output values are correctly calculated and that the filters are working as they should.

Kalman Filter

The Kalman Filter was the hardest one to implement. Used a paper explaining how the filter is applied to get it working. [2] The basic idea of the Kalman Filter is that you anticipated what the next value will be given the previous. In practice this removes a lot of noise from the data since it is reasonable to assume that the next value will fall close

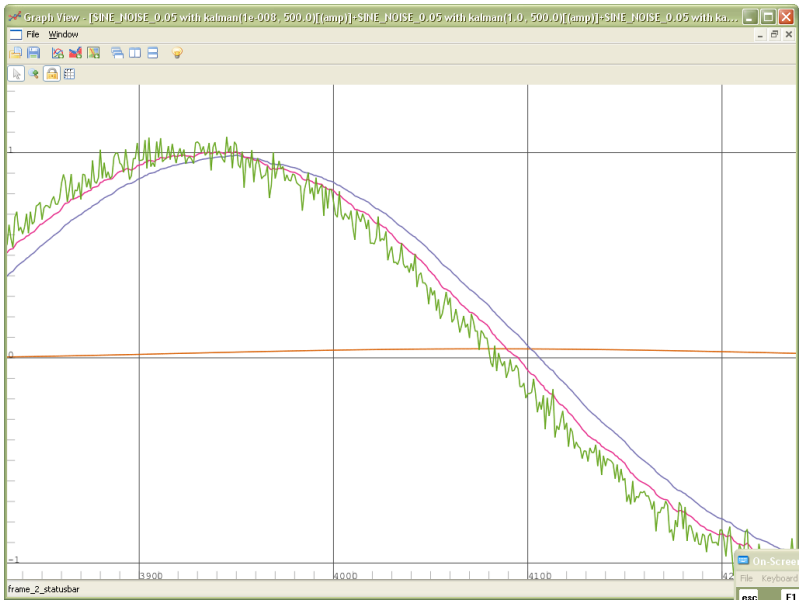

Figure 1: A sine wave with normal distributed noise added and 3 different Kalman filters being applied on it. The green graph represents the actual data values, while the pink, orange and blue are Kalman filtered streams. In the case of the orange filter a high degree of process noise is assumed and this all but cancels out the data variance.

to the previous, this assumption often holds true which is why the filter can be useful. Kalman filtering works in two stages, time update and measurement update. First the filter estimates what the value will be, it projects forward in time. This is the time update stage. In the second stage, measurement update, the filter looks at the actual, but noisy, value that is passed in and alters the estimate based on this. How much the estimate is altered can be tweaked by adjusting two method parameters, Q and R. These values represent process variance noise, or actual noise, and measurement noise, noise from the measuring equipment. In more sophisticated implementations these variables can take the form of two matrices which can change values from measurement to measurement, but here they are constants. A screen shot of the Kalman filter is shown in figure 1.

Variance Filter

The variance filter was a much more straight forward filter to implement. The variance is the squared distance from the expected, or mean, value. The filter uses a two-pass algorithm and is shown in Algorithm 1. This filter is useful for showing the scale or degree of value spreading. Figure 2 shows a basic example of a variance filter

Algorithm 1 Two-pass algorithm for calculating variance.

$n = 0$
$sum1 = 0$
for x in $data$ **do**
  $n = n + 1$
  $sum1 = sum1 + x$
end for
$mean = sum1/n$
$sum2 = 0$
for x in $data$ **do**
  $sum2 = sum2 + (x - mean)^2$
end for
$variance = sum2/(n - 1)$

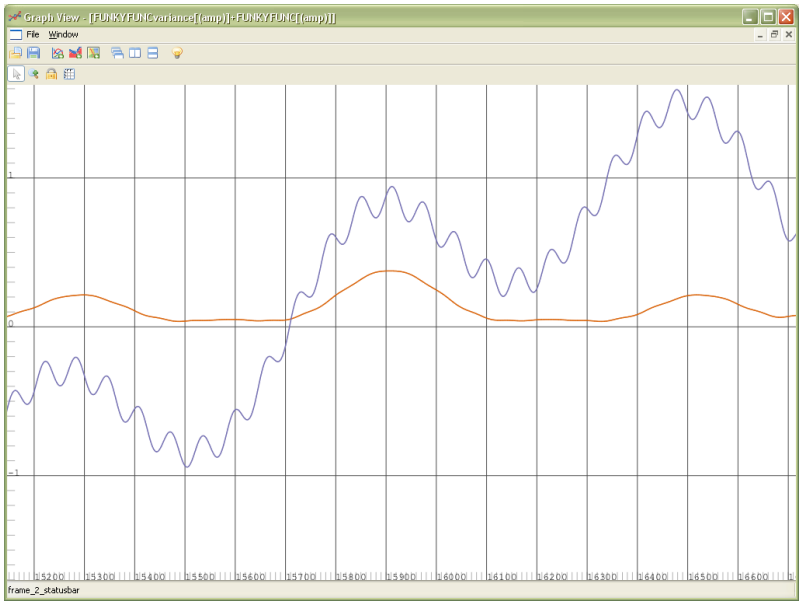

Figure 2: A variance filter applied to a multiscale sine curve. The filtered stream is in orange. The variance of a graph increased when values increase rapidly over a short time frame.

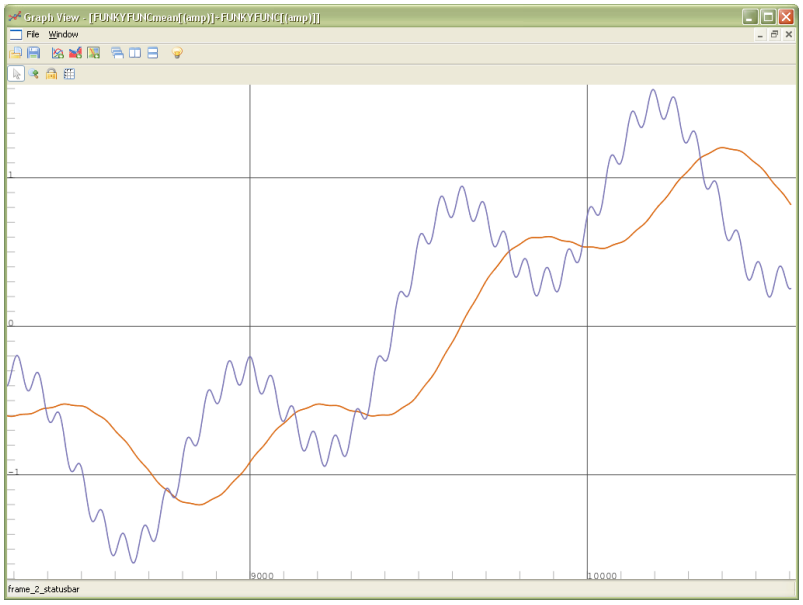

Figure 3: Mean filter applied to the same function as in figure 2. The blue colored graph shows the output of a multiscale sine wave and the orange colored graph shows the filter output.

Mean Filter

The mean filter adds values over a given time kernel and outputs the average. This was a very easy filter to implement, but it's also a very useful filter. There are many times where seeing the overall change in values is important and you don't want to see all the deviations from moment to moment. Be aware of the difference between a Kalman filter and the mean filter. Where the Kalman filter tries to ignore most of the short time variance, the mean filters adds these values. This is a huge difference even if the graphs can appear to be similar. Figure 3 gives an example of a mean filter. Pseudo-code for the algorithm is shown in Algorithm 2.

Algorithm 2 Algorithm for calculating mean value.

$n = 0$
$sum1 = 0$
for x in $data$ **do**
  $n = n + 1$
  $sum1 = sum1 + x$
end for
$mean = sum1/n$

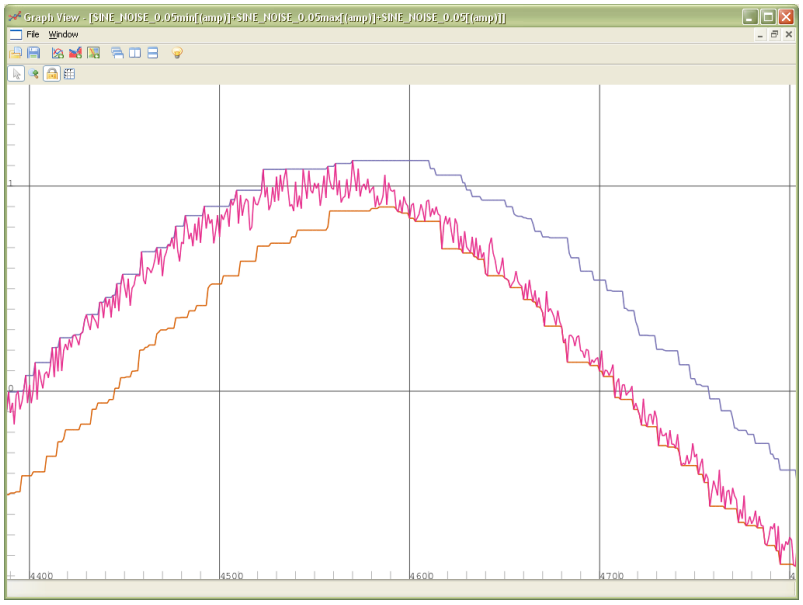

Figure 4: An example of the Min/Max filter applied to a sine wave with normal distributed noise. The pink colored graph shows the data set output while the blue and orange colored graphs are maximum and minimum filter output over the time kernel size respectively.

Min/Max Filter

The Min/Max filter shows the minimum and maximum values over a given time kernel. Actually it's a either maximum or minimum filter so if you want both you have to make two filters. The method takes a static MIN or MAX to specify which one to use. Figure 4 show an example of a Min/Max filter.

3 Time Buffer

The data provided as input is supposed to be real-time. It's therefore difficult to know how much of the earlier data you want to use to calculate the value of the filter output. The solution we came up with for this was to use a ring buffer. A ring buffer keeps adding values until it is full, then it replaces the next value with the oldest one. The first iteration of the buffer used an array of static length, defined at the time of filter creation, to regulate what data span to calculate over. This works well for the synthetic data set used for testing, it provided data at an even interval. The problem with this approach is that this may not always be the case. If a burst of values comes in in a short time frame and then additional values ticks in with a longer time spans between them, you get a disjunction between time and the graphical representation of the data. The solution to this problem was to wrap the data with a time-stamp when appended to the buffer. This interval is given as the time delta since program start up. Each time a value gets appended the buffer is checked for old values surpassing the cut-off time (given as argument in seconds). This lets the

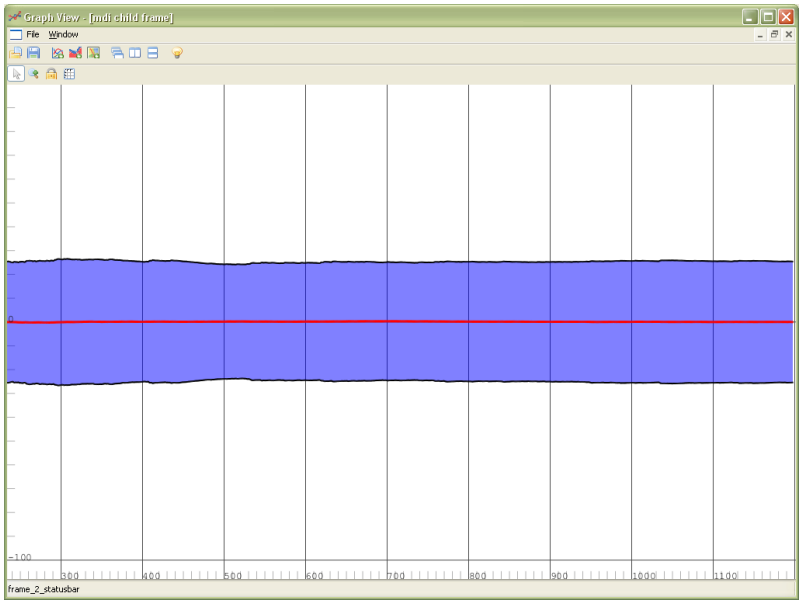

Figure 5: Variance filter with a time buffer set to 4000 seconds. The graph stabilizes and it's easy to see the overall tendencies of the data

user examine the same data in different ways depending on what is of interest. Figure 6 shows a graph with time kernel, the time over which a filter output value is calculated, set to 4 seconds. With this kernel any changes in the data are clearly visible. Figure 5 show the same data set, but with a time kernel set to 4000 seconds (or roughly 1 hour and 7 minutes). Here any change in the data over a short time frame is lost, but it's easier to see the overall variance of the data.

4 OpenGL

The last part of the project was writing a "Variance Graph". The graph takes two arguments, the output of a mean filter and the output of a variance filter and uses OpenGL to represent the data. The graph draws four different primitives, three instances of GL_LINE_STRIP and one GL_TRIANGLE_STRIP. The three line strips are used to show the mean value and outline the upper and lower borders of the variance, The triangle strip colors the area of the variance to give it a better overview. At first I implemented the graph as a series of vertex placements between glBegin() and glEnd(). This is not a very effective way to render since the instruction has to be passed back and forth over the bus for each call. Later rewrote it as a vertex array. All vertexes to be drawn are appended in an array and sent together over the bus for rendering. This eliminates a lot of the overhead associated with multiple function calls. Figure 6 and 5 show screen shots of the variance graph.

Even though switching to a vertex array increased the performance significantly it's

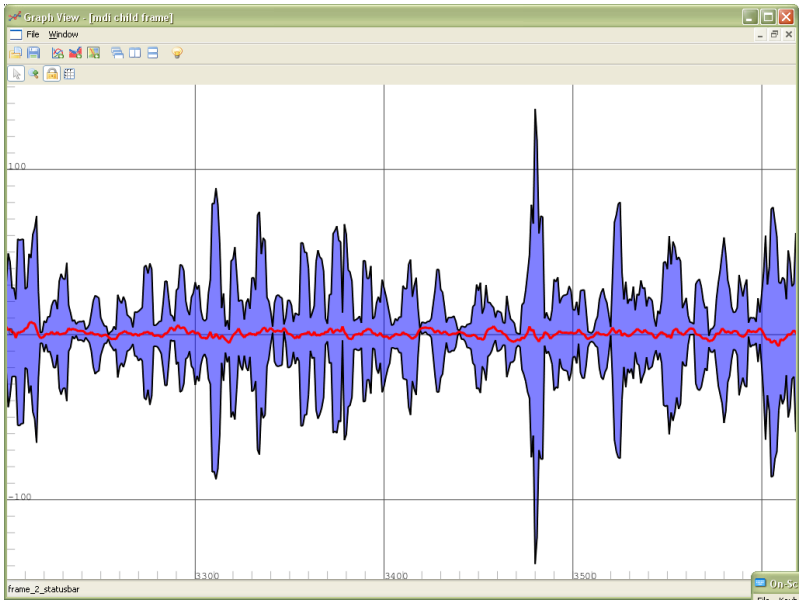

Figure 6: Variance filter with a time buffer set to 4 seconds. Any changes in the data from moment to moment is clearly visible in the graph.

still far from fast. One thing slowing it down is the fact that the entire graph is rendered, not just the part showed on screen. This has a larger and larger impact on performance the longer it runs and eventually ends program execution with an overflow. Another way it could be optimized is transferring the OpenGL calls down to C++ code. I used PyOpenGL to do render calls and although it's an excellent set of bindings, issuing the expensive OpenGL calls from python adds a bit of a performance penalty. Still, the point of this project was not to optimise the code, which would be sort of a waste of time since the EnlightenMe framework it depends on is still a work in progress, but to get it to work, demonstrate the filter design and actually displaying something useful.

5 Still to be done.

This project added filters, a time buffer and graph visualization to the EnlightenMe framework. These features are not all completely in compliance with the target feature set of the framework. The project description several more filters to implement . This includes a derivation filter, sampling frequency filter and one for normal distribution to name some. Having more filters would make one of the EnlightenMe frameworks planned features more useful, to pipe filters together. Also, the performance of the variance graph is not optimized and so runs poorly on anything but high-end systems. It is however reasonable to delay this optimization since the framework is still a work in progress and the features implemented in this project might get altered.

References

[1] The Python Programming Language
http://www.python.org

[2] An introduction to Kalman filtering
http://www.cs.unc.edu/ welch/kalman/kalmanIntro.html

[3] Algorithms for calculating variance
http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance