Sensor Signals for Interaction

This is a brief tutorial on algorithms for processing sensor signals. When developing interactive technologies that are off of the desktop and out in the real world where people live, we need tools for recognizing behavior from a variety of sensors. Many of these sensors produce a sequence of numeric values over time. This tutorial describes a series of techniques for performing the recognition at interactive speeds.

Signal processing vs. interaction

The topics covered in this tutorial traditionally fall under signal processing. Some of the techniques described are drawn from signal processing. Many are modifications or downright software hacks that approximate some of these concepts. The goal here is that we care about interaction rather than the mathematics of signals. We want something simple that will indicate when some event has occurred.

All of these techniques assume that machine learning algorithms will be involved. Therefore we will focus on various kinds of features that can be fed into a machine learning algorithm to perform the recognition. This discussion is independent of the machine learning algorithm to be used. Decision trees, naïve Bayes, nearest neighbor or linear classifiers are all simple algorithms that can use the features described here to perform the classification.

What we are most interested in are features of the signal that can be used to distinguish the events that we are trying to recognize. If a feature will distinguish, we use it. Otherwise we want our machine learning algorithm to discard it.

Many times with these kinds of signals we want to perform the recognition on small microcontrollers that are close to the sensor and then propagate the recognition (rather than the full signal) back to some interaction controller. These small microcontrollers allow us to place lots of cheap sensors throughout the environment. However, these microcontrollers have limited processing speed and limited memory. This severely restricts the kinds of computations that we can perform as part of the recognition.

This tutorial assumes that you know how to program and you have a math background up through basic linear algebra.

Types of signals

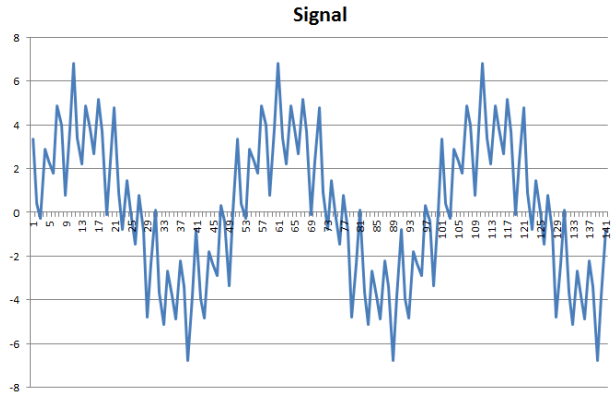

Figure 1

Figure 1 shows a simple signal of the type that you might receive from a sensor. In most cases there will be an analog pin on your microcontroller that you can read periodically. Successive readings of the pin form a signal like that shown above. Such signals might come from microphones, accelerometers, light sensors, twist, pressure or vibration sensors. There are many, many sensors that would generate a sequence of sampled values.

From such a signal we might want to recognize striking an object, walking, running, talking, scratching, tapping, bending or a variety of other human activities. Figures 2 and 3 show two other signals that will be useful to our discussion. Figure 2 is a step signal. These rarely occur in actual use because the real world is not that clean. However, a step signal is helpful in understanding how our various filters and features behave. Figure 3 is the kind of signal that might be caused when an object is struck or tapped. This will add a little realism to our recognition discussions.

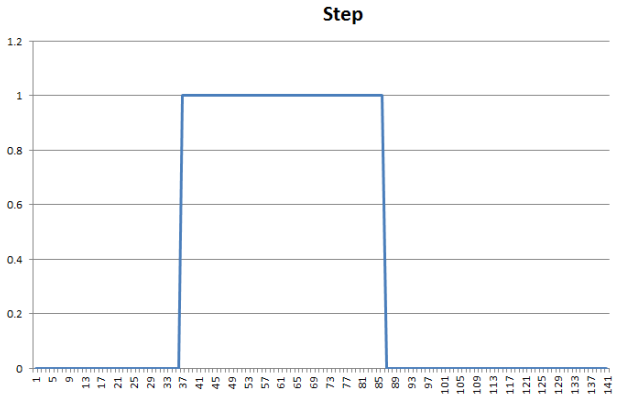

Figure 2 – Step signal

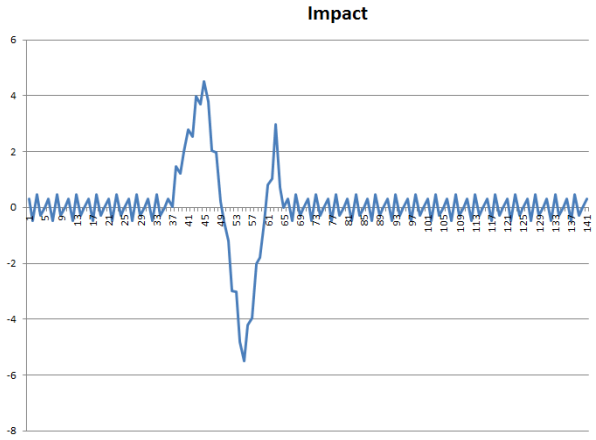

Figure 3 – an impact

Implementation

For handling signal sampling in microcontrollers see the implementation hints for sampling rate (to get samples of accurate time duration) and for signal memory (to use ring buffers to handle small memory resources).

Obtaining features for recognition

There are two basic approaches for obtaining a set of features from a signal: windowing and incremental. When we talk about a signal we generally look at is as an array S of numeric values. In theory the array is infinite but we generally only look at the recent history of S.

Windowing

We take a window between time S[t] and time S[t-w] (where w is the width of the window in time). We then take all of the samples in this window and generate a set of features that describes the signal in that window. Figure 4 shows a window that captures all of the information about our impact. The problem, however, is to automatically pick the correct window to get the information that we need. For example, figure 5 shows a window of the same size that is offset relative to the impact. This window does not pick up all of the signal information that will help us to correctly detect the impact. Figure 6 shows what happens if the window that we select is too narrow. Again we miss a lot of the information about the impact.

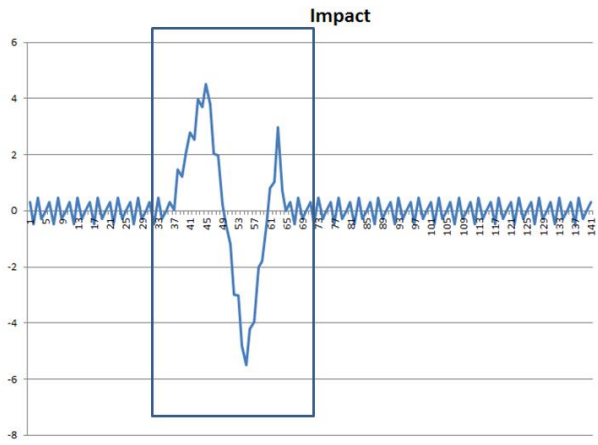

Figure 4 – good window

The problem, however, is to automatically pick the correct window to get the information that we need.

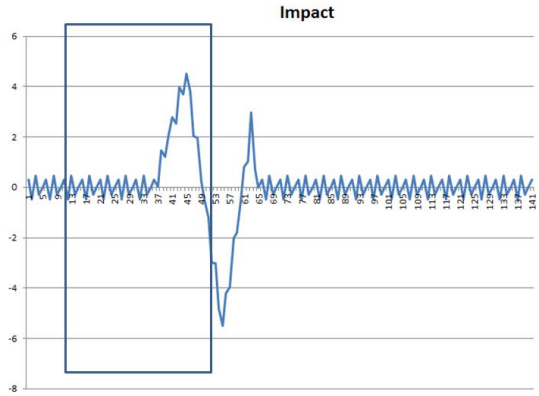

Figure 5 – offset window

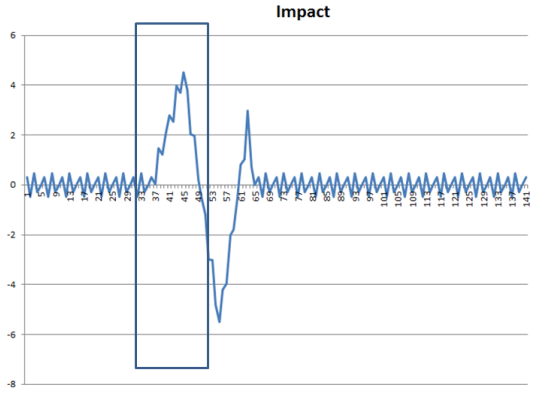

Figure 6 – narrow window

The window placement is less of a problem if we are trying to detect an ongoing behavior such as talking, walking or whistling middle C. For such ongoing activities that are similar across the activity, the window placement is not much of an issue. For example in figure 6 the noisy region to the right of the impact can be detected by a window placed anywhere in that region because it is relatively similar.

A simple technique for window placement is some kind of a threshold value. For example in figure 6, none of the background rises above an amplitude of 2. If we sample a value above 2 then we can start the window back a ways in time and then make it wide enough to cover the kinds of events that we are interested in.

Incremental

The incremental strategy is to update our feature set with each sample so that our features are always available. We typically keep a history of features over a short period of time. Using this we can regularly perform our recognition and report a class as soon as the features match. This essentially behaves as if we are continuously applying windows across the recent history. This eliminates the problem of finding the right place to start a window. We can use a history of large width initially and then let our machine learning algorithm determine what features actually matter and only keep a history wide enough for those features.

Filters

Historically the processing of signals was performed by special circuits called filters. Filters are designed to separate various frequencies of the signal. There are low-pass filters that get rid of high frequency information, high-pass filters that get rid of the low frequency information and band-pass filters (a combination of low-pass and high-pass) that select for frequencies in a certain range. We will not be using the special circuits, but these are easily replaced by two very simple algorithms (high-pass and low-pass). These filter algorithms lend themselves readily to incremental implementations.

Low pass filters

Low pass filters are used to get rid of high frequency noise to produce a simpler signal that is easier to work with. For signal values S[i] we compute a corresponding filter value F[i]. For low pass filters the formula is:

$$F[i] = \alpha\, S[i] + (1 - \alpha)\, F[i-1]$$

In essence the filter value is a blend of the current signal value and the previous filter value. This is for values of α between 0 and 1. For example if α=1 the filter value becomes the signal value. The smaller the value of α the wider the area that the signal is blended across. The filter value becomes a mixture of ever increasing amounts of the earlier filter values. This averaging effect of the blend eliminates high frequency information from the filtered signal. The lower the value of α the lower the frequencies that pass through the filter.

Figure 7 shows the result of applying low-pass filters to the signal from figure 1 using various values for α. An α=1 is included to show the original signal. Notice that as α gets smaller the signal gets smoother. By α=0.1 the primary frequency is all that remains and for α=0.01 the primary low frequency is all that is

left. All of the noise has been smoothed away. Also note that as values of α decrease, the peaks of the signal move forward (to the right) in time. This is because the averaging effect all comes from the left (earlier in time).

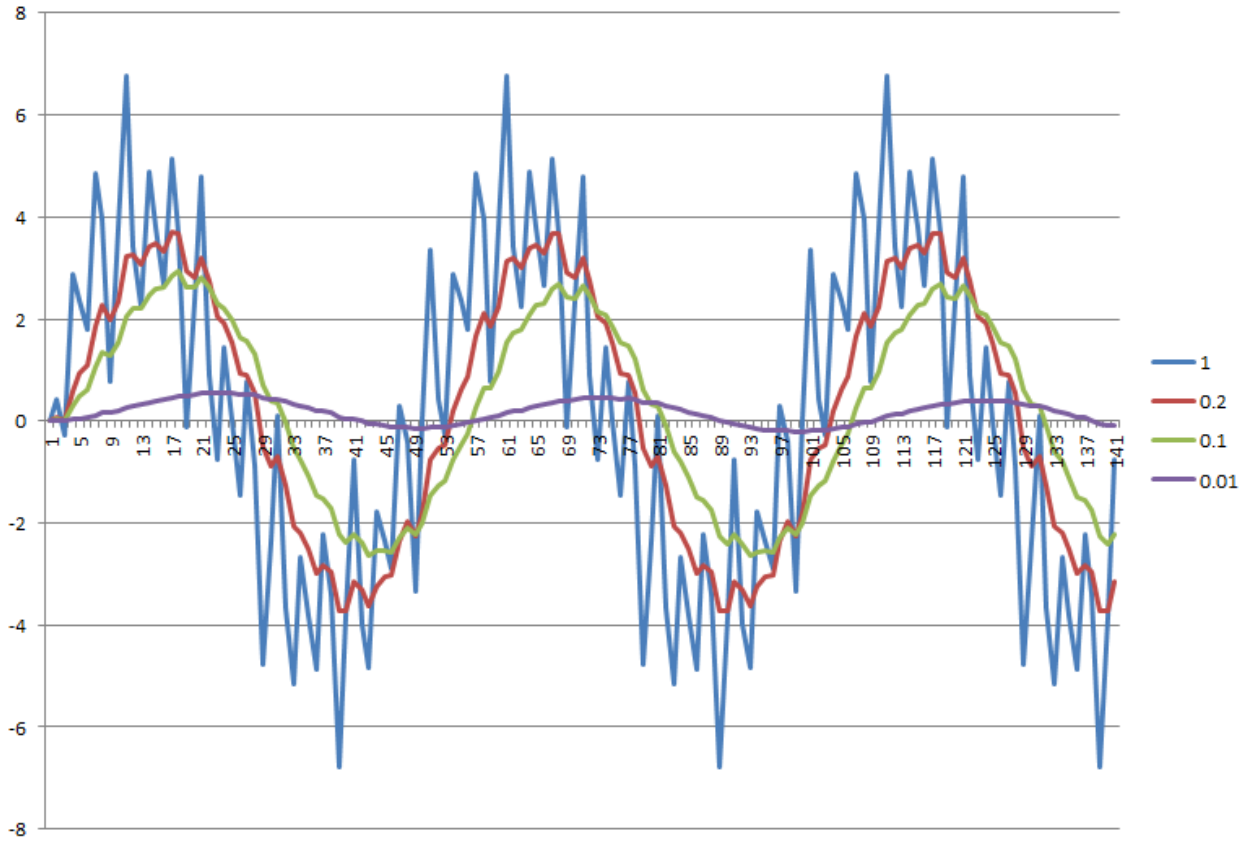

Figure 7 – Low-pass filters

Our other two example signals give some idea as to how these filters treat signals we are likely to care about. Notice in figure 8 that the smaller the value of α, the slower the filter is to respond to the signal.

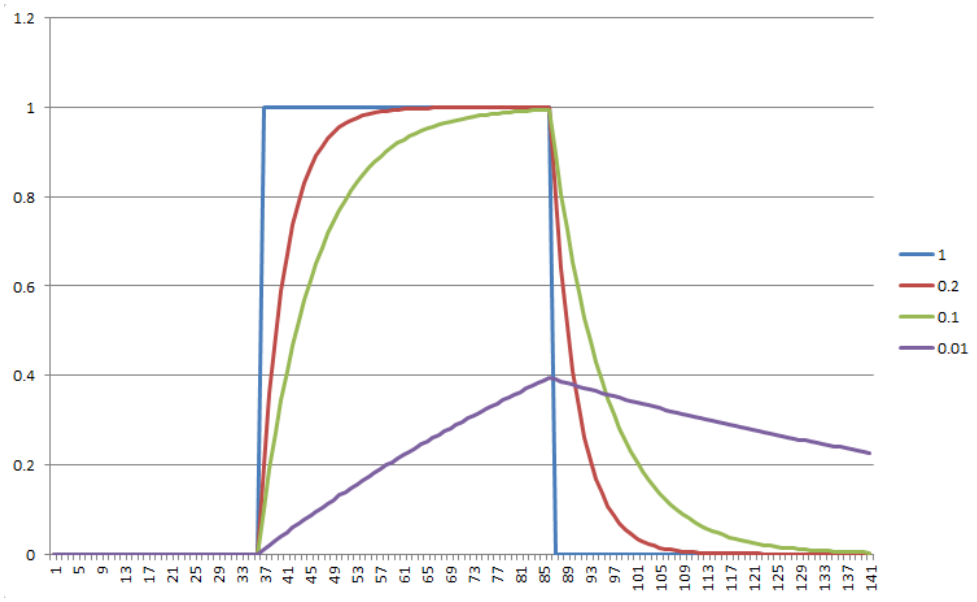

Figure 8 – Low-pass filter applied to square function

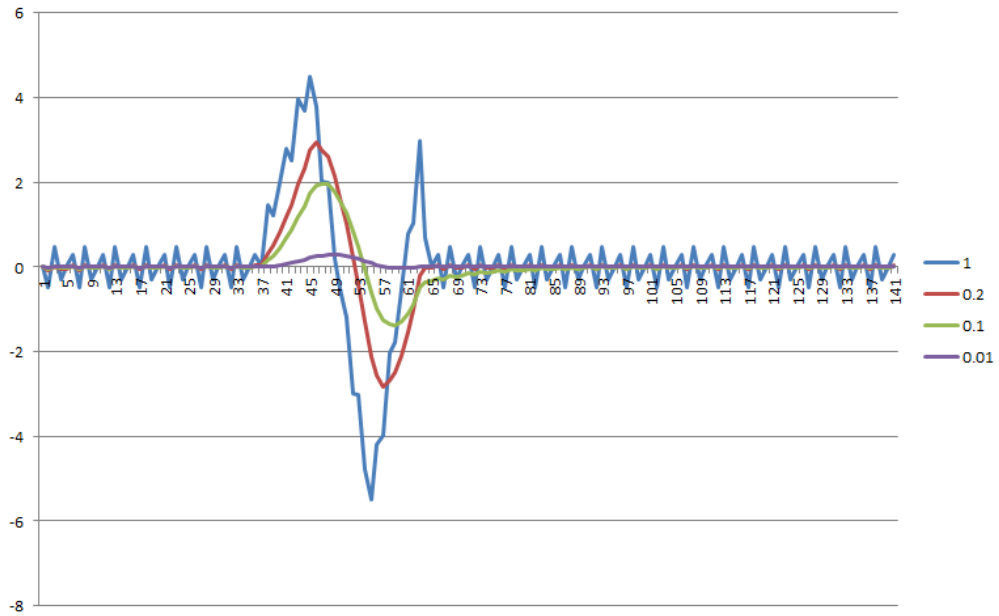

Figure 9 – Low-pass filter applied to an impact signal

In figure 9 the filters are applied to our impact signal. Notice that with an α=0.2 the noise has been eliminated from the signal and there is a clear peak and valley. The secondary peak has also been eliminated. This filtered version of the signal would give us two very clear features to detect when this impact has occurred. Filtering out the high frequency noise has simplified the process.

Notice also that for α=0.01 the filtered signal is almost flat. Even lower values would give us very close to an average. This can be really helpful when the average is not around zero. There are features that are

best produced by subtracting the signal from the average. Getting a nice running average is possible using a low-pass filter with an α value much closer to zero.

High-pass filters

Sometimes it is the high frequencies that we want. For example we may want people's speech without the rumbling of the air conditioning. High-pass filters are calculated in a very similar fashion to low pass.

$$F[i] = \alpha (F[i-1] + S[i] - S[i-1])$$

The intuition here is that we are taking the difference between successive values of S to blend in with the filter value from the past. Figure 10 is a repeat of our signal from figure 1 for comparison. Figure 11 shows the high-pass filter for various values of α.

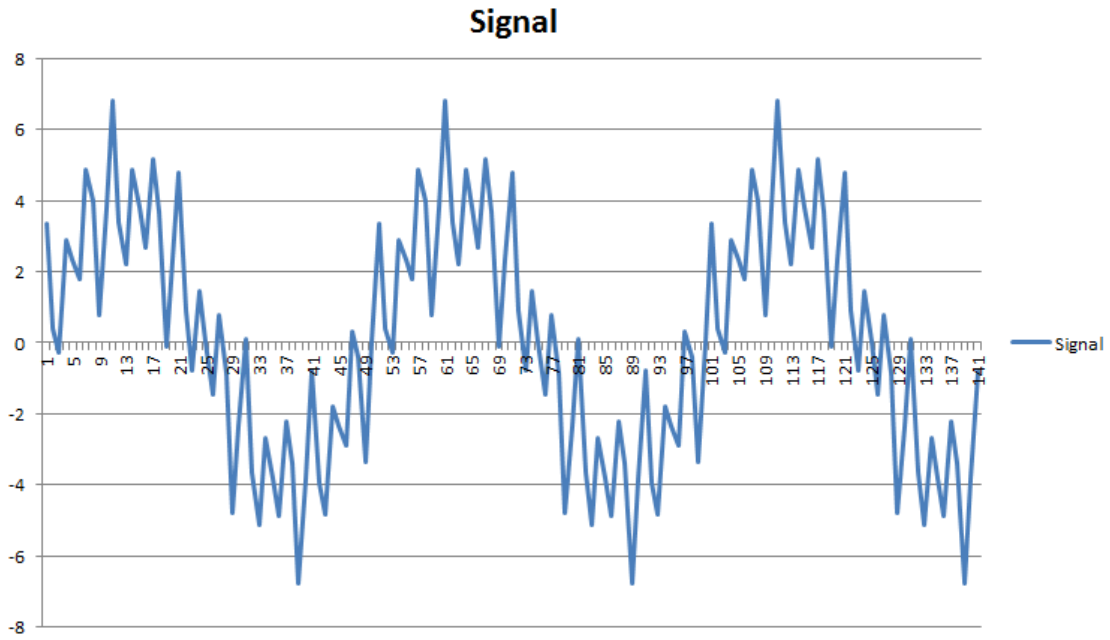

Figure 10 – Original signal

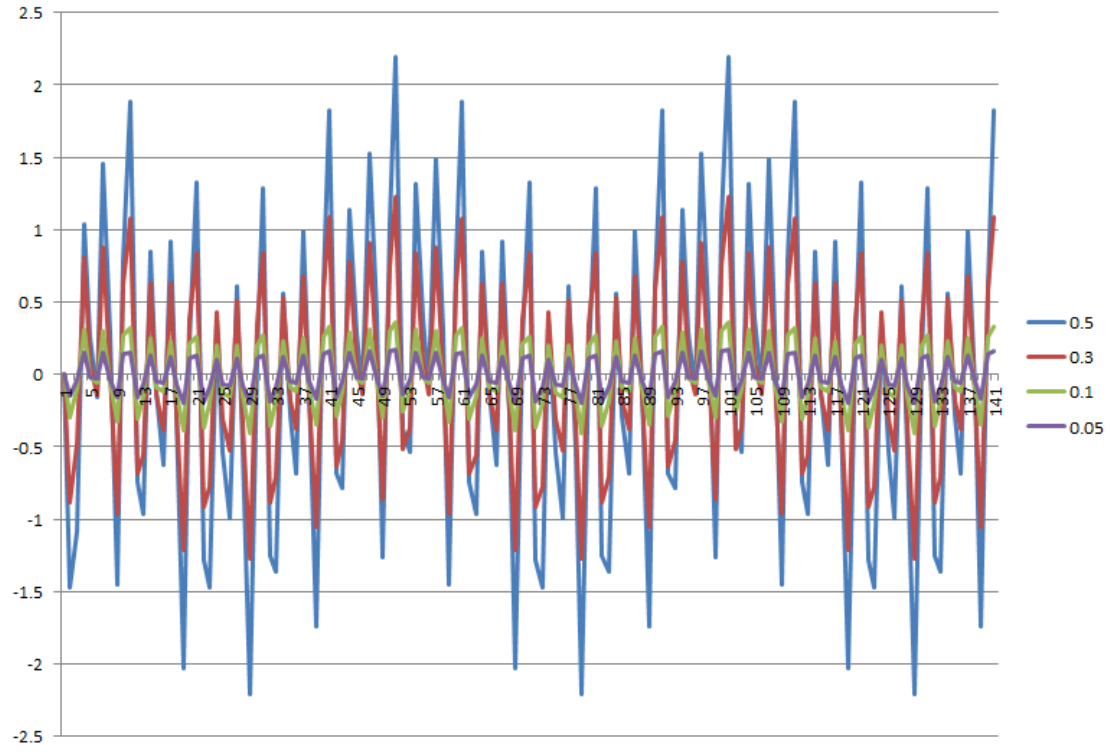

Figure 11 – High-pass filter of the signal

Notice in figure 11 as the values of α decrease the large up and down effect of the original signal is eliminated and what remains is the higher frequency signal.

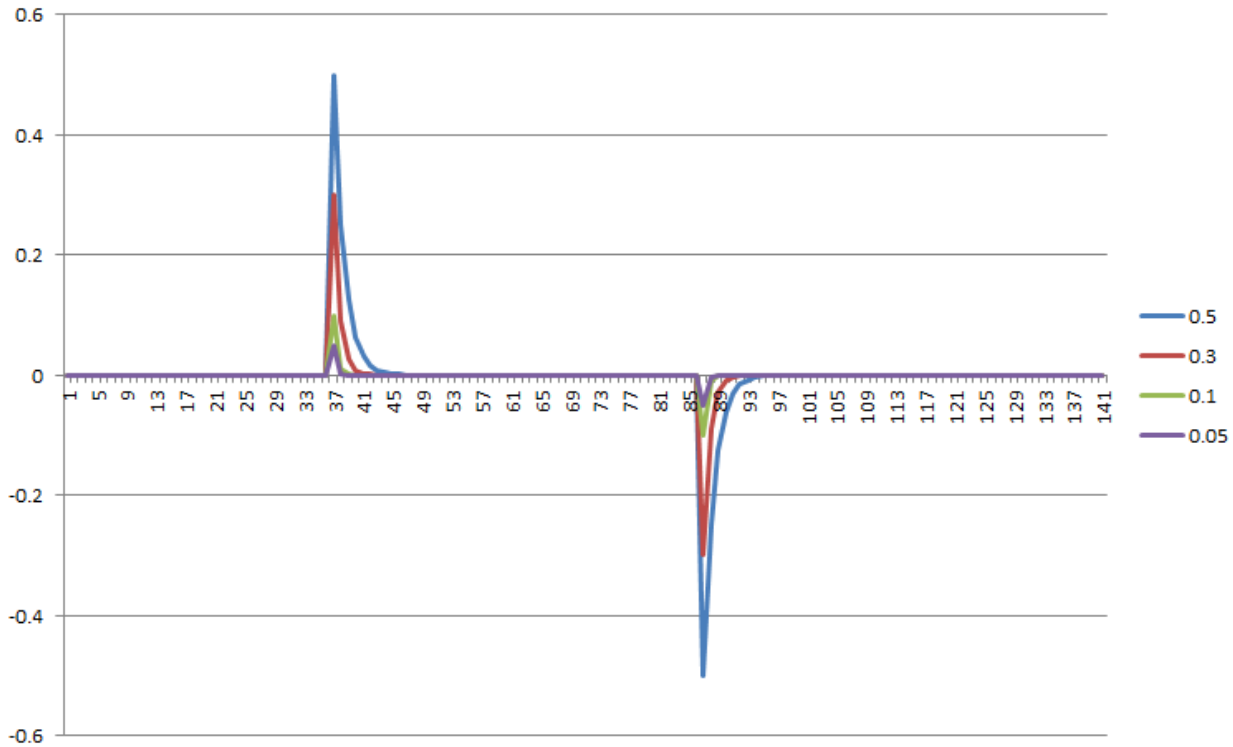

Figure 12 – High-pass filter of the square wave

In figure 12, where the high-pass filter is applied to the square wave, the rising and falling edges are strongly accentuated. The high-pass filter shows us where strong change is occurring.

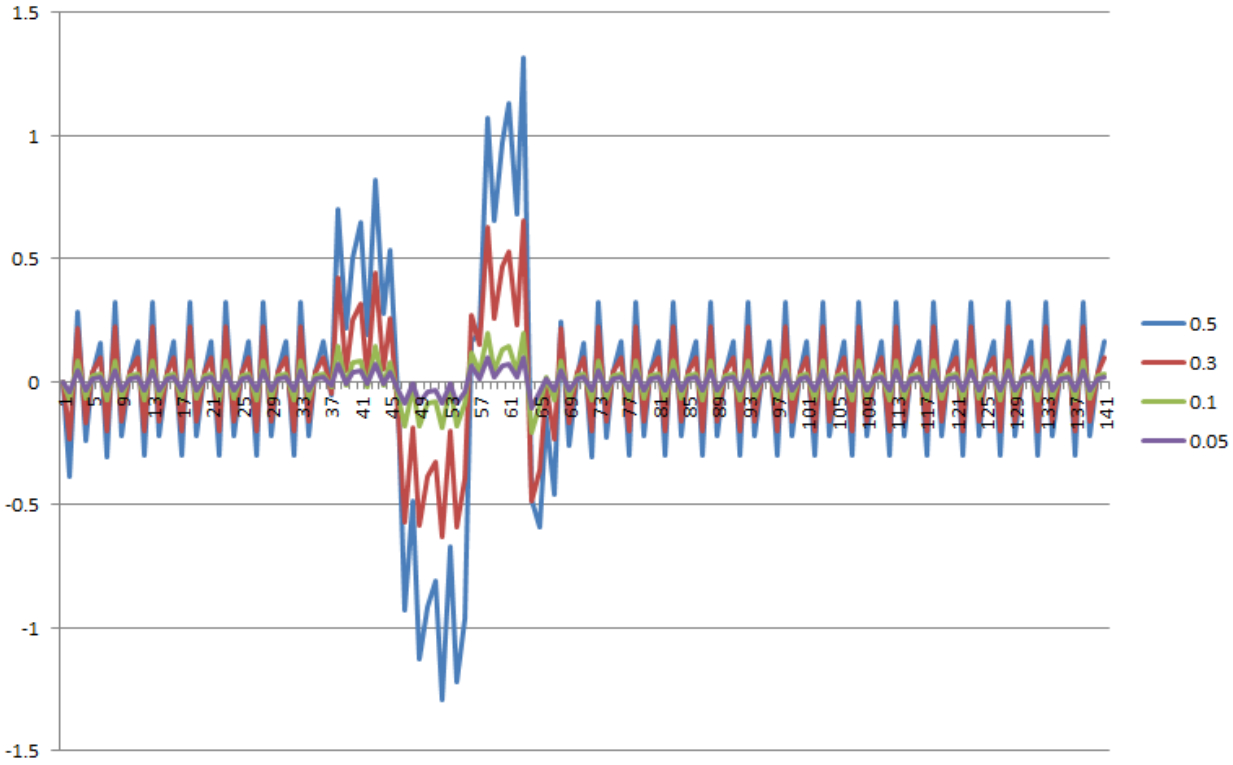

Figure 13 – High-pass filter applied to the impact signal

Notice in figure 13 that as lower values of α are used, the actual impact almost disappears. The high frequency noise is all that remains.

Implementation hints

To implement low-pass and high-pass filters, see the implementation hints on fixed point arithmetic (cheap microcontrollers do not like to do floating point) and on ring buffer storage of the filtered signals.

Signal integrals

A rich source of incremental features are signal integrals. Remember that the integral of a function is simply the infinite resolution sum of the values of the function. Note also that the definite integral between A and B is the integral at B minus the integral at A. Our signals do not have infinite precision. They are sampled at discrete intervals. Our integral devolves into a simple sum. There are many useful features that can be derived from the sum of signal values across some region. By storing the integral (sum) of a signal rather than the signal itself, we can get the sum of any historic period in constant time.

If we have a signal S[i] we can compute its integral as

I[i] = S[i] + I[i-1]

If I[-1]=0 then I[i] is the sum of all signal samples from 0 to i. As you can see, I[i] is trivial to compute if done incrementally. Suppose that we wish the sum of all signals starting at A and ending at B. This is computed as:

sumAB=I[B]-I[A-1]

if I[B] contains the sum of all signals since the beginning of time and I[A-1] contains the sum of all signals since before A then sumAB will contain the desired value. Note that the time to compute sumAB is constant no matter how far apart A and B might be. This is really important for fast computation of signal features. An alternative representation would be time offset O with width W (in samples). Then the computation is:

sumOW=I[O]-I[O-W].

Typically we would build several features from various offsets back into time and various values of W. Note that all such features share the same storage of I. Unlike filters, which must have a separate ring buffer for each value of α, integral features share the same storage.

There is a problem, however. We cannot continue to sum up signal values for very long before we overflow the number of bits available. The key to this is in the ring buffer implementation. In our ring buffer we fill it up until we reach the last space in the buffer. In the integral version of the ring buffer we restart the sum at zero when we start back at the 0 index of the buffer. Thus the number of sums never exceeds the number items in the buffer.

Figure 14 shows three cases of the integral ring buffer that resets its sum each time it restarts the buffer. For case 1 the computation of sumOW is:

sumOW = I[lastIndex-O] – I[lastIndex-O-W]

For case 2, the offset has wrapped around the end of the buffer. In this case the computation is:

newO = BUFFER_SIZE + (lastIndex-O)

sumOW=I[newO]-I[newO-W]

Case 3 is a little more complicated. In this case the window W that we want to sum up is crossing the wrap around boundary. The computation is:

sumOW = I[lastIndex-O] + I[BUFFER_SIZE-1] – I[BUFFER_SIZE+(lastIndex-O-W)]

As long as O+W never exceeds the buffer size, one of these three cases will always yield the sum of all values across W.

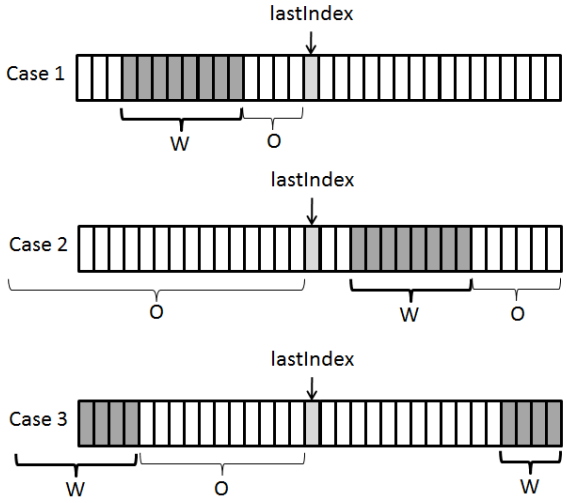

Figure 14 – Integral Signal Ring Buffer

In some cases, we may not want the integral of the original signals, but rather the integral of one of the filters. We can also use combinations of integral features to recognize more complex situations.

Suppose we are trying to recognize the impact signal shown in figure 15. In this case we will sum two regions that are W wide and then subtract them from each other.

sumA = I[W]-I[2W] sumB = I[0]-I[W]

feature = A-B

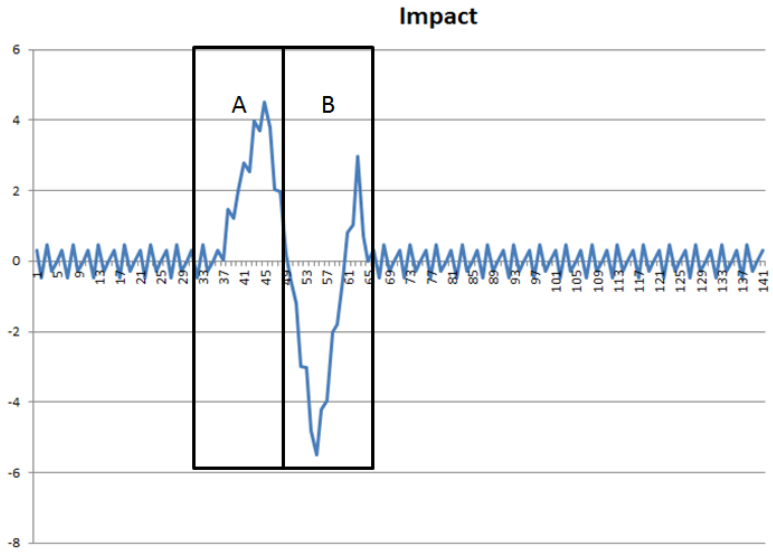

Figure 15 – Two Integral Regions to detect the Impact signal

This feature will be very positive because it is positive in region A where the signal is high and then it subtracts B where the signal would sum to a negative number. This will yield a very large positive number on this signal. We would pick a value for W based on our observations of the signal. Or we could generate multiple versions of this feature with different values of W and then let the machine learning algorithm select the best feature choice.

Other integrals

We don't always need to integrate over the signal itself. We can use other functions as the basis for our integral. For example we sometimes want to get the "energy" of a signal. In figure 15 if we sum most of the signal it will sum to zero because the signal varies between positive and negative values. That is not very interesting. Even our impact region sums to close to zero. Instead we can use:

$$Energy[i] = (S[i]-mean)^2 + Energy[i-1]$$

The "mean" term is introduced because some signals like that in 15 are not centered around zero. This integral accentuates strong signal values regardless of the direction from the mean.

We might also measure the "roughness" across a region using the integral:

$$Rough[i]=(S[i]-S[i-1])^2 + Rough[i-1]$$

This integral sums up the squares of the differences between successive samples. Where there are lots of sharp changes this feature will have a high sum.

If we integrate the signal and the square of the signal we will have enough information to compute the standard deviation of any region of the signal.

Zero crossings

Another variation of an integral is the zero crossing. We can define our zero crossing function as:

$$zeroCross(I) = if (s[i-1]<mean \&\& s[i]>=mean) then 1 else 0$$

This function simply returns 1 when the signal crosses from below the mean to above the mean. In many cases this mean is 0, which is the reason for the name. Our integral then is:

$$zeroInt[i] = zeroCross(i) + zeroInt[i-1]$$

This integral simply counts the number of zero crossings that have occurred. Using this integral we can determine the number of times the signal crossed zero for any range of samples. The count of the number of times a signal crosses zero is a simple substitute for the primary frequency of the signal.

Implementation

See the implementation hints for integral buffers for computing and storing these kinds of features.

Windowing

The approaches that we have described so far are all incremental. Another common technique is windowing. In this technique a set of samples (a window) is chosen and a variety of features are

computed from that sample signal. Possible features include the number of zero crossings and the total energy across the window. A common feature for this treatment is the Fourier analysis. A Fourier transform will convert a signal into its component frequencies. The strength of the various frequencies are used as features. This is particularly useful where we are not trying to recognize events but classify what is going on in the signal. The problem with Fourier analysis is that it uses complex arithmetic and can frequently consume all of the processor power of a microcontroller. Most modern PCs will handle this calculation just fine.

Implementation Hints

Signals

There are two issues when dealing with signals via a microcontroller. They are sampling rate and signal memory.

Sample Rate

The sample timing problem arises because there is so much variability in the amount of time it takes to sample and process a signal. What you want is an array of signal values that occur at steady, regular intervals. Frequently this is handled by an interrupt or special timer circuits but not all microcontrollers have these. One of the ways to get a cheap microcontroller is to reduce all of the frills. So we assume that we have the ability to read from an analog-to-digital pin on a regular basis and that we have access to a clock that at least has microsecond (one millionth of a second) accuracy.

The following code uses the Arduino convention of a *setup* procedure that is called once when the processor resets and a *loop* procedure that is repeatedly called as often as the processor can get around the loop. This code computes the average of all the samples that occur within SAMPLE_TIME. Once SAMPLE_TIME has expired it calls the procedure *handleSample*. The *handleSample* procedure is where all of the code goes that does whatever is necessary with the average sample. This would be where the saving, feature extraction, recognition and other processing code would go.

This code also deals with the problem that clocks (particularly ones that count millionths of seconds), regularly overflow their integer storage. This code detects that and correctly accounts for it. This code is easily modified to read, average and sample multiple signals.

```
int sampleCount;
int sampleSum;
unsigned int sampleStartTime;
const int SAMPLE_TIME=1000;

void setup()
{     sampleCount=0;
      sampleSum=0;
      sampleStartTime=clockMicros();
}
void loop()
{     int value = analogRead(PinNumber);
      unsigned int curTime = clockMicros();
      var time;
      if (curTime>sampleStartTime)
      {     time = curTime-sampleStartTime; }
      else
      {     // the clock has rolled over its maximum time
            time =curTime+(CLOCK_MAX_TIME-sampleStartTime);
      }

      if (time>=SAMPLE_TIME)
      {     int sample = sampleSum/sampleCount;
            handleSample(sample);
            sampleCount=0;
            sampleSum=0;
            sampleStartTime = curTime;
      }
      else
      {     sampleCount++;
            sampleSum+=value;
      }
}
```

Signal memory

Microcontrollers generally have their own RAM on the chip, which simplifies building systems but limits what can be remembered for recognition. The first approach to managing limited memory is to set SAMPLE_TIME to be as large as possible and still see the desired signal properties. The less often you sample, the less memory you need to store a signal covering a given amount of time.

In all of our implementation discussions we will address time as the number of samples preceding current time. S[0] is current time. If SAMPLE_TIME is 1000 then S[4] is 4 milliseconds into the past. Be careful here because positive sample indices move in negative time. We are looking back into history, not forward into the future.

The data structure for signal memory is a simple ring buffer. This buffer has a fixed size and accepts data until it is full and then starts over by writing over the oldest data. As long as our features do not require more samples than the buffer size, the required historical signals will always be available.

```
int buffer[BUFFER_SIZE];
int lastIndex=BUFFER_SIZE;
void addSample(int sample)
{     lastIndex++;
      if (lastIndex>=BUFFER_SIZE)
            lastIndex=0;
      buffer[lastIndex]=sample;
}
int getSample(int offset)
{     int sampleIndex = lastIndex-offset;
      if (sampleIndex<0)
            sampleIndex=BUFFER_SIZE+sampleIndex;
      return buffer[sampleIndex];
}
```

The procedure *addSample* will add a sample to the ring buffer. The procedure *getSample* will return the sample value at the specified sample offset into the past.

Low-pass and high-pass filters

These filters have two implementation issues. The first is that floating point arithmetic is generally slow on microcontrollers with integer numbers being much preferred. The second is that every filter for each value of α must have its own buffer for storing the filtered signal. If you are using lots of filters, this can add up.

Fixed point arithmetic

Microcontrollers are steadily improving but they still tend to have slow floating point. Our low and high-pass filters require multiplication by α, which is a value between 1.0 and 0.0. We can avoid floating point by using fixed-point arithmetic. Fixed point arithmetic involves taking a fractional number and multiplying it by a factor F, where F is a positive integer and usually a power of 2. Multiply and divide by a power of 2 is just a shift left or shift right operation, which is very fast. Multiplying by F will move part of the fractional information into the integer portion of the number. For example, with α=0.001, we can use F=1000. Then the scaled value of α will be 1 and can use integer arithmetic.

For our purposes we can assume a constant factor F for all of our operations. The basic operations are as follows:

- Float to Fixed – fixedInt = float * F
- Fixed to Float – float = fixedInt/(float)F
- Adding/subtract – If both arguments are encoded with the same F then integer add and subtract are all that is necessary.
- Multiply – If A and B both use the same F then the product is (A*B)/F.

Therefore, all we need to do is select an appropriate value for F and then work in integer space rather than floating point. The simplest choice is F=1/α. This makes the step of multiplying by α into an integer multiply by 1.

Low-pass filter storage

Note that we will need a separate buffer for each value of α. In many cases the current value of the filter is all we need. Because the low-pass filter has eliminated many of the high-frequency components we can generally use a lower sampling rate (longer sampling time) to store the filtered result. This can result in smaller storage. The stored filter is a simple modification of the ring buffer technique for signals. Samples are converted to fixed point and then the low-pass filter computation applied. If F is a power of 2 then many multiplies and divisions by F can be replaced by shift operations.

```
int buffer[BUFFER_SIZE];
int lastIndex=BUFFER_SIZE;
int fixedAlpha = alpha * F;
int fixedAlphaMinus = (1*F)-fixedAlpha
void addSample(int sample)
{     int fixedLast = buffer[lastIndex];
      lastIndex++;
      if (lastIndex>=BUFFER_SIZE)
            lastIndex=0;
      int fixedSample = sample * F;
      buffer[lastIndex]=(fixedAlpha * fixedSample)/F +
            (fixedAlphaMinus*fixedLast)/F;
}
int getSample(int offset)
{     int sampleIndex = lastIndex-offset;
      if (sampleIndex<0)
            sampleIndex=BUFFER_SIZE+sampleIndex;
      return buffer[sampleIndex]/F;
}
```

High-pass filter storage

High-pass filters also require a separate buffer for each filter with a different value of α. However, because only the high frequencies are retained, generally a shorter buffer with much less history is adequate. This is because the wavelengths of higher frequencies are shorter. The code is as follows.

```
int buffer[BUFFER_SIZE];
int lastIndex=BUFFER_SIZE;
int lastSample=0;
int fixedAlpha = alpha * F;
void addSample(int sample)
{     int fixedLast = buffer[lastIndex];
      lastIndex++;
      if (lastIndex>=BUFFER_SIZE)
            lastIndex=0;
      int fixedSample = sample * F;
      buffer[lastIndex]=
            (fixedAlpha * (fixedLast+fixedSample-lastSample))/F;
      lastSample=fixedSample;
}
int getSample(int offset)
{     int sampleIndex = lastIndex-offset;
      if (sampleIndex<0)
            sampleIndex=BUFFER_SIZE+sampleIndex;
      return buffer[sampleIndex]/F;
}
```

Integral features

The integral buffer is a simple modification of the ring buffer for storing signals. For the variations on the function being integrated, replace *sample* in the *addSample* procedure with the value that you really want to integrate.

```
int buffer[BUFFER_SIZE];
int lastIndex=BUFFER_SIZE;
void addSample(int sample)
{     int lastInt = buffer[lastIndex];
      lastIndex++;
      if (lastIndex>=BUFFER_SIZE)
      {     lastIndex=0;
            buffer[0] = sample; //restart the sum
      }
      else
            buffer[lastIndex]=sample+lastInt;
}
int getSum(int offset, int width)
{     int sampleIndex = lastIndex-offset;
      if (sampleIndex<0)
            sampleIndex=BUFFER_SIZE+sampleIndex;
      if ((sampleIndex - width)>0)
      {     return buffer[sampleIndex]-buffer[sampleIndex-width];}
      else
      {     int bottomSum=buffer[sampleIndex];
            int topWidth = lastIndex-offset-width;
            int topSum = buffer[BUFFER_SIZE-1] -
                  buffer[BUFFER_SIZE-1+topWidth];
            return topSum+bottomSum;
      }
}
```