# ESS: Exercise Set 7

## Optimization

Question 1:

A common operation in embedded systems is to compute an average over a window (a moving average) to smooth out noise and spikes from sensor data. There are a number of ways of computing this.

Assume we are trying to compute an average of a set of `uint8_t` numbers over a window of 4 samples. We compute the average, and then move the index of the window one sample to the right, discarding the oldest sample. Typically, this is implemented by using a FIFO (first-in first-out) buffer of length equal to the window length. Compare the operation of the following four different approaches, where `index` is the index into the FIFO buffer and `array` is the data in the buffer.

FLOAT_AVE: Upcast to floating point

```
float average = ((float)array[index] +
                 (float)array[index-1] +
                 (float)array[index-2] +
                 (float)array[index-3])/4.0f;
```

FIXED_AVE: Fixed point moving average

```
uint8_t average = (array[index] +
                   array[index-1] +
                   array[index-2] +
                   array[index-3])/4;
```

The following two algorithms are optimized, using the fact that the only thing that changes from each window is to discard the oldest sample and add the newest sample.

FIFO1: Optimized FIFO

```
static uint8_t average = 0;
uint8_t average = average + array[index]/4 - array[index-3]/4;
```

And this super optimized example saves an extra divide:

FIFO2: Optimized FIFO

```
static uint8_t average = 0;
uint8_t average = average + (array[index] - array[index-3])/4;
```

Compute the output for these four algorithms using the following stream of sensor data:

```
 uint8_t array[] = {0,0,0,0,1,2,3,4,2,4,5,7,7,2,1,0,0,0,0};
```

Which algorithm works best? Can you suggest some ways of making some of them work better?

Solution:

Here is a sample implementation, showing how each one works.

```
#include <stdint.h>
#include <stdio.h>

// input data
uint8_t array[] = {0,0,0,0,1,2,3,4,2,4,5,7,7,2,1,0,0,0,0};

// floating point moving average
float float_average(uint8_t index)
{
   float average = ((float)array[index] +
    (float)array[index-1] +
    (float)array[index-2] +
    (float)array[index-3])/4.0f;
   return average;
}

// fixed point moving average
uint8_t fixed_average(uint8_t index)
{
   uint8_t average = (array[index] +
    array[index-1] +
    array[index-2] +
    array[index-3])/4;
   return average;
}

// fixed point, fifo, method 1
uint8_t fifo1_average(uint8_t index)
{
   static uint8_t average = 0;
   average = average + array[index]/4 - array[index-3]/4;
   return average;
}

// fixed point, fifo, method 2
uint8_t fifo2_average(uint8_t index)
{
   static uint8_t average = 0;
   average = average + (array[index]- array[index-3])/4;
   return average;
}


int main(void)
{
    uint8_t index;
    float float_ave;
    uint8_t fixed_ave;
    uint8_t fifo1;
    uint8_t fifo2;
    for (index = 4; index < sizeof(array);index++)
    {
            float_ave = float_average(index);
            fixed_ave = fixed_average(index);
            fifo1 = fifo1_average(index);
            fifo2 = fifo2_average(index);
            printf("Index %d Float_Ave %g Fixed ave %d Fifo1 %d Fifo2 %d\n",
                index,float_ave,fixed_ave,fifo1,fifo2);
    }
    return 0;
}
```

The output from the various algorithms is shown below.

```
>> output
Index 4 Float_Ave 0.25 Fixed ave 0 Fifo1 0 Fifo2 0
Index 5 Float_Ave 0.75 Fixed ave 0 Fifo1 0 Fifo2 0
Index 6 Float_Ave 1.5 Fixed ave 1 Fifo1 0 Fifo2 0
Index 7 Float_Ave 2.5 Fixed ave 2 Fifo1 1 Fifo2 0
Index 8 Float_Ave 2.75 Fixed ave 2 Fifo1 1 Fifo2 0
Index 9 Float_Ave 3.25 Fixed ave 3 Fifo1 2 Fifo2 0
Index 10 Float_Ave 3.75 Fixed ave 3 Fifo1 2 Fifo2 0
Index 11 Float_Ave 4.5 Fixed ave 4 Fifo1 3 Fifo2 1
Index 12 Float_Ave 5.75 Fixed ave 5 Fifo1 3 Fifo2 1
Index 13 Float_Ave 5.25 Fixed ave 5 Fifo1 2 Fifo2 1
Index 14 Float_Ave 4.25 Fixed ave 4 Fifo1 1 Fifo2 0
Index 15 Float_Ave 2.5 Fixed ave 2 Fifo1 0 Fifo2 255
Index 16 Float_Ave 0.75 Fixed ave 0 Fifo1 0 Fifo2 255
Index 17 Float_Ave 0.25 Fixed ave 0 Fifo1 0 Fifo2 255
Index 18 Float_Ave 0 Fixed ave 0 Fifo1 0 Fifo2 255
```

Not only does FIFO2 not work correctly, it suffers from an underflow, causing it to wrap around from 0 to 255. This can be extremely dangerous in safety critical systems: for example in an insulin pump, this could represent the concentration of blood sugar, reporting it as extremely high, instead of the true value.

To make the fixed point versions work better, one technique is to scale (multiply) the input by at least the window size before computing the average, to avoid truncation. At the end of the computation, divide by the window length again to scale back.

The main difference between the first two approaches and the last two (FIFO1, FIFO2) is that the latter maintains a history or state that persists between calls. The first two approaches are an example of an FIR (finite impulse response) filter whereas the last two are IIR (infinite impulse response) filters. IIR filters are often more computationally efficient, but can suffer from numerical/precision issues, such as we saw here.

Question 2:

An embedded device without a floating point unit needs to calculate $y = \sin(x)$. $x$ is an unsigned integer, with a scaled range from $-\pi$ to $+\pi$. The output $y$ is also an unsigned integer, with a scaled range from -1 to +1. $x$ is represented by an 8 bit unsigned integer (`uint8_t`). $y$ is also represented by an 8 bit unsigned int.

(a) For each domain ($x$, $y$) work out the mapping from real world values to fixed point values. What is the precision of each domain?

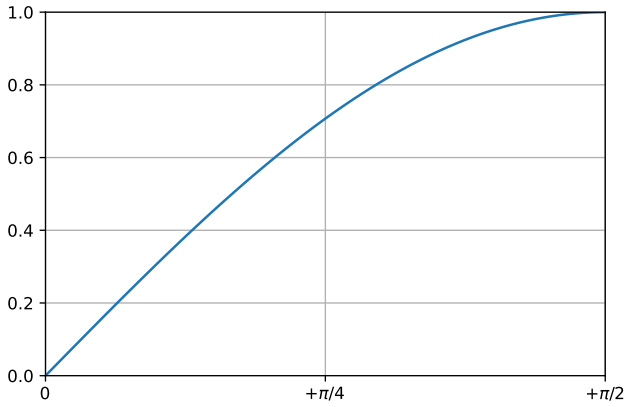
Solution:

The first thing to notice is that we do not need to store the full sine wave mapping, only the first quadrant. This is because the other quadrants can easily be computed from the first quadrant.

Thus, the input domain would range from $0$ to $\frac{\pi}{2}$, corresponding to a digital value from $0$ to $255$. Thus, the resolution is $\frac{\pi}{2} \div 255 = 0.00615$ rad/LSB

$\approx 0.35°$.

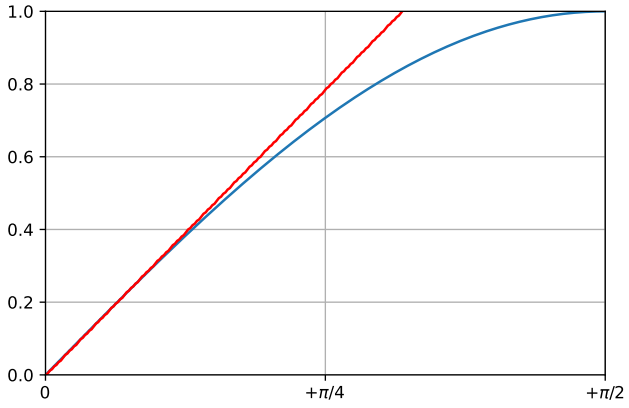
The output domain would range from 0 to 1. Thus the output resolution would be $1 \div 255 = 0.0039$.




(b) Taylor series approximation One technique for approximating the function is to use a Taylor series linearization. The simplest Taylor approximation is to use the approximation $\sin(x) \approx x$, which holds for small values of $x$. Show, with the aid of a diagram, how this could be used as a first approach at approximating the sine curve.

Solution:

As can be seen, until about $\frac{\pi}{4}$, this is a pretty reasonable approximation.




(c) Implicit Lookup table An alternative solution is to precalculate a table (e.g. in a spreadsheet) and then simply put into the microcontroller as a constant array e.g.

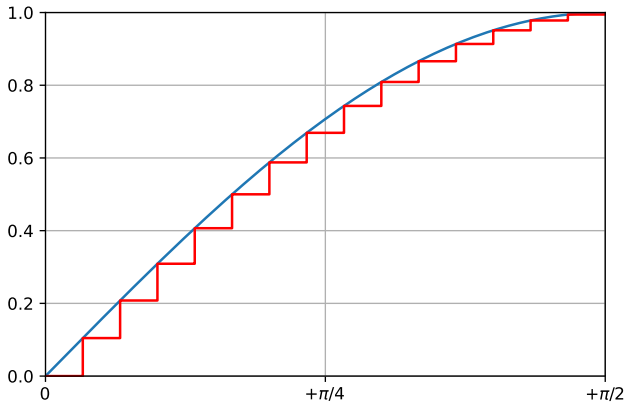
```
static const uint8_t lookup[] = {127,255,127,0};
```

To find the correct value, we just find the index in the table. Generally, the table is stored as a power of 2 to make lookup easy. For example, if the table

has 256 entries, then the $x$ value is the index into the table. If the table only has 128 entries, then the $x$ value has to be divided by 2 to index the table. For a table with 16 entries, show the approximated sine wave.

Solution:

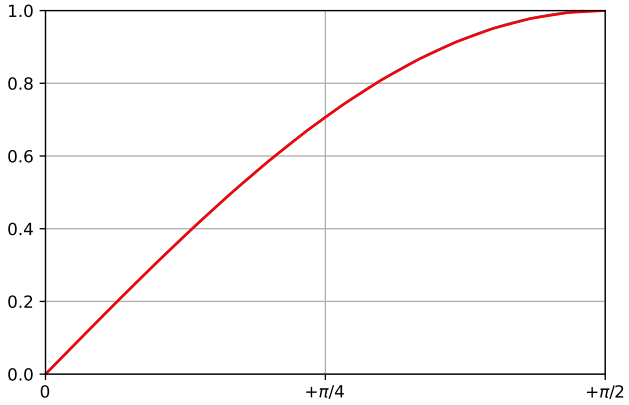
The lookup, even with just 16 points, approximates the curve quite well, however, just before the lookup index, the error can be quite large.




(d) Linearization We can combine the principles of the taylor series approximation with the lookup table. If we are trying to estimate the $y$ value for an $x$ coordinate that lies between two values in the table, we can interpolate between the two points. The simplest way of doing this is to simply take the average. Show that for a small lookup table (e.g. 16 entries), the overall error using linearization is smaller than without.

Solution:

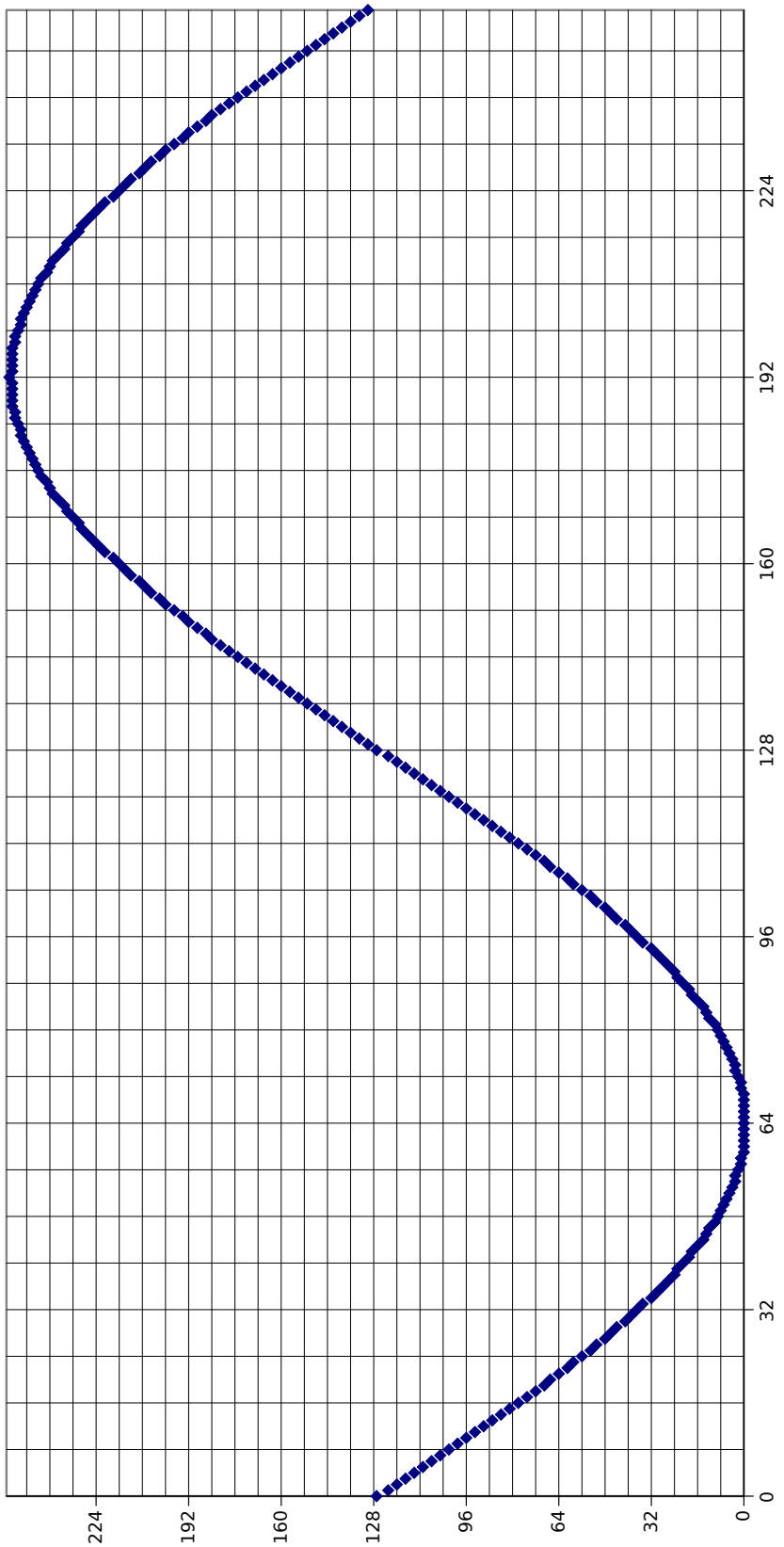
We can see that interpolation makes the curve really accurate.




(e) Staggered lookup table

Instead of having a equally spaced index into the table, each entry can be stored

as a pair of co-ordinates $(x, y)$. More densely spaced coordinates are stored when the curve cannot be interpolated well, e.g. at the peaks and troughs of the sine curve. Where the curve can be interpolated well (e.g. the relatively linear region between peak and trough) then store fewer coordinates. With an aid of a diagram show how to spread out coordinates to maximize the reconstruction accuracy.


0
32
64
96
128
160
192
224
224
192
160
128
96
64
32
0