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?

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?

(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.

(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.

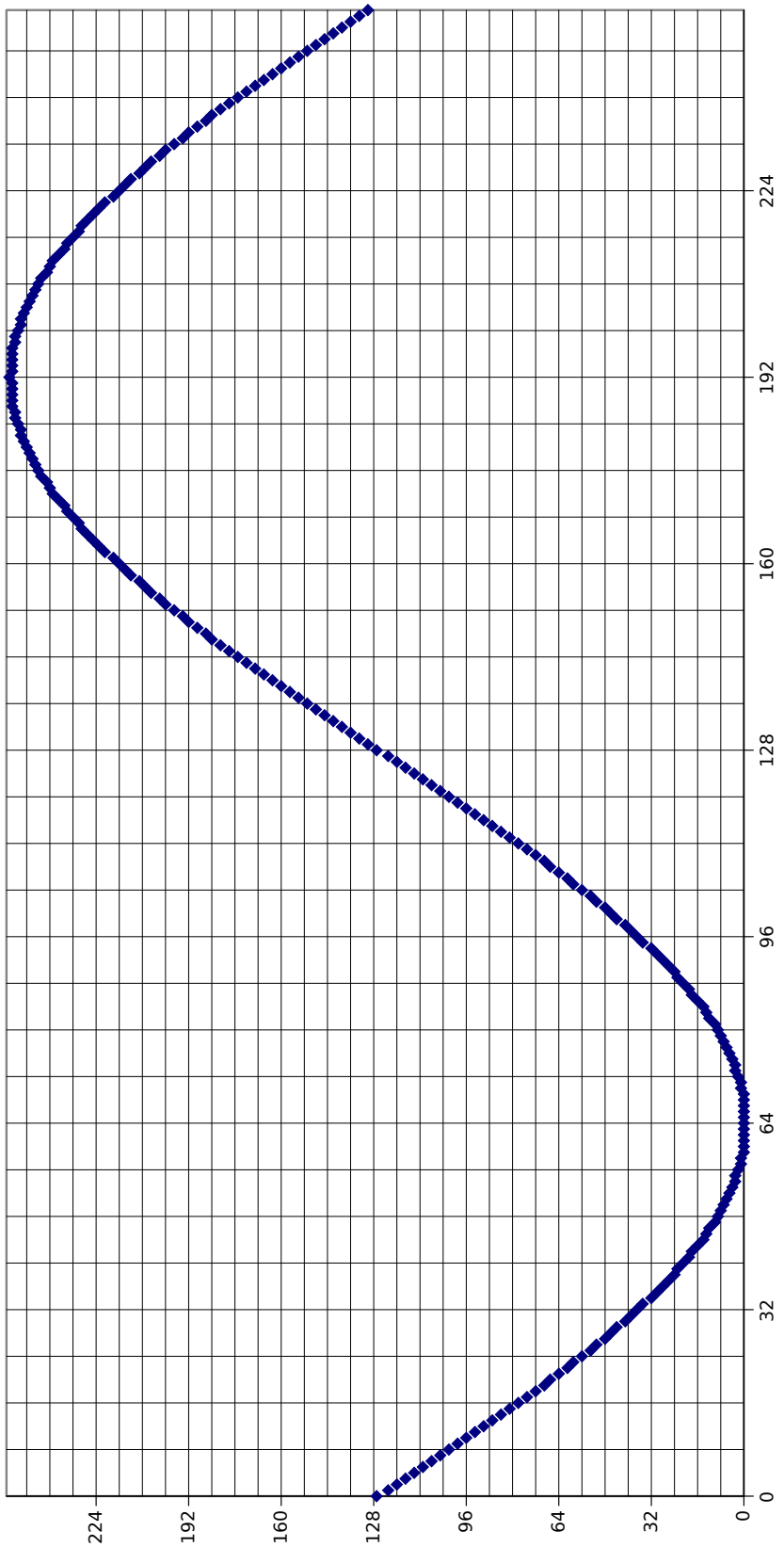
```
static const uint8_t lookup[] = {127,255,127,0};
```

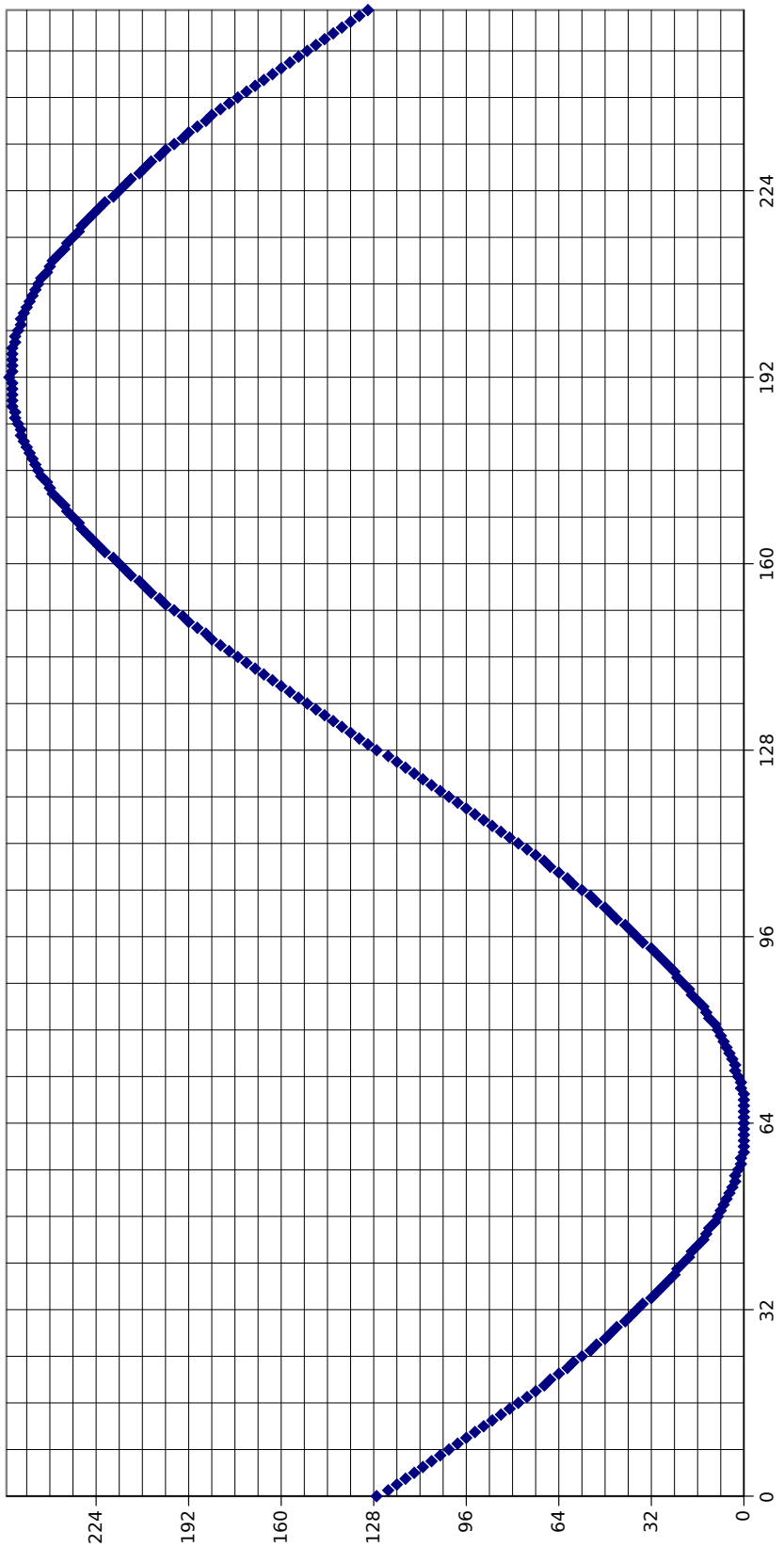
To find the correct value, we just find the index in the table. Generally, the table is stored as a power of 2 to make lookup easy. For example, if the table has 256 entries, then the x value is the index into the table. If the table only has 128 entries, then the x value has to be divided by 2 to index the table. For a table with 16 entries, show the approximated sine wave.

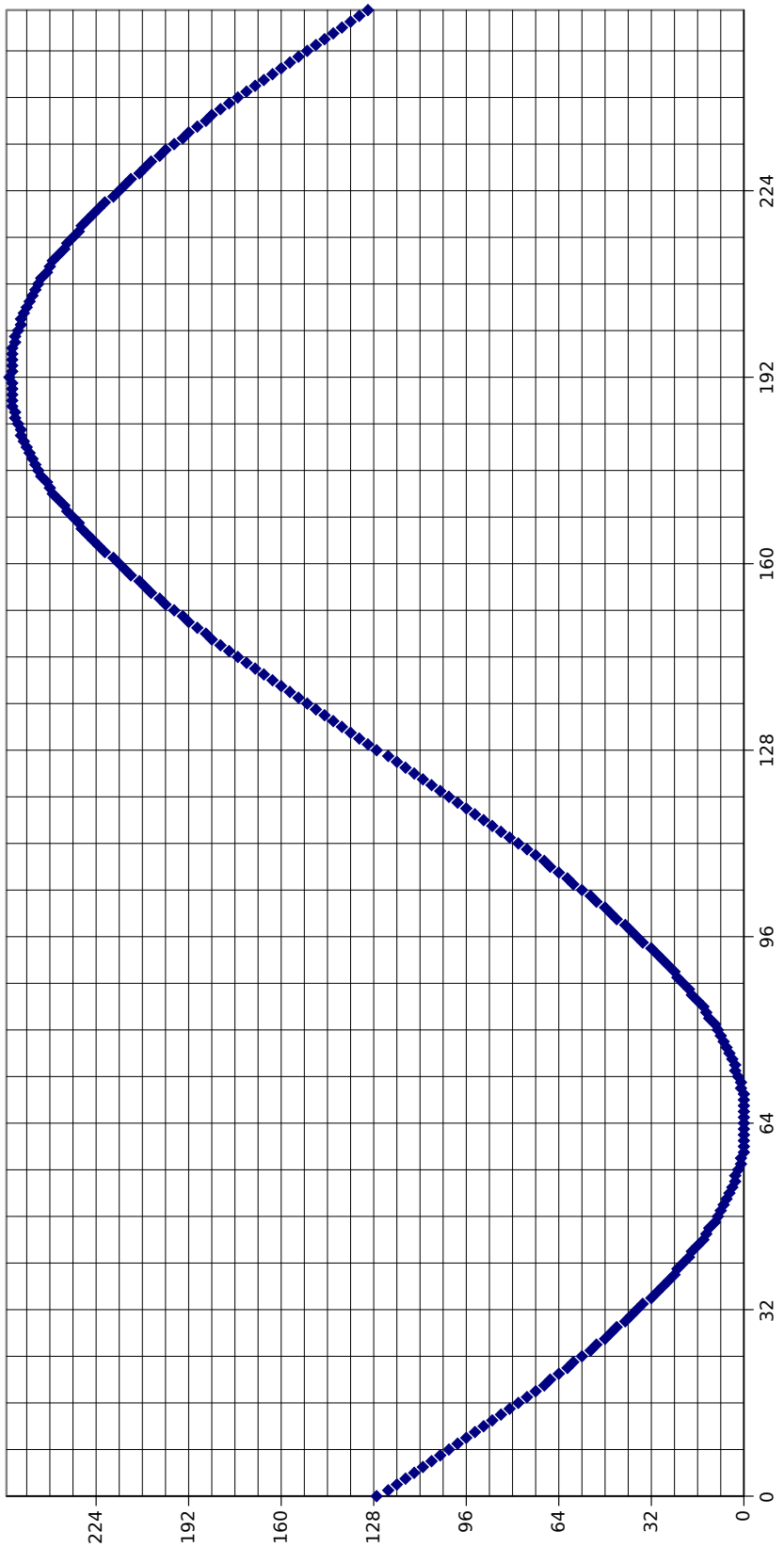
(d) **Linearization** We can combine the principles of the taylor series approximation with the lookup table. If we are trying to estimate the y value for an x coordinate that lies between two values in the table, we can *interpolate* between the two points. The simplest way of doing this is to simply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

(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 co-ordinates. 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


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


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