COM3023 Coursework Submission

Hallam Saunders (URN: 6788550)

November 17, 2025

Contents

1 Demonstration Screenshots

1.1 12-into-1 Aggregation

```
B = [ 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, 915.527, ]
StdDev = 0.0
Aggregation = 12-into-1.
X = 915.527
```

Figure 1: 12-into-1 aggregation indicating little activity.

1.2 4-into-1 Aggregation

```
B = [ 1627.349, 1524.353, 1833.343, 1988.983, 1911.163, 1705.169, 1627.349, 1780.700, 1858.520, 1988.983, 1858.520, 1677.703, ]
StdDev = 143.675
Aggregation = 4-into-1.
X = [ 1743.507, 1756.95, 1845.932, ]
```

Figure 2: 4-into-1 aggregation indicating some activity.

1.3 No Aggregation

```
B = [ 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 3794.860, 6738.281, 2117.156, 7667.541, ]
StdDev = 1327.374
No aggregation needed.
X = [ 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 4389.953, 3794.860, 6738.281, 2117.156, 7667.541, ]
```

Figure 3: No aggregation indicating high activity.

2 Source Code

2.1 Pre-Demonstration

Below is the source code for basic functionality required for the coursework, before the code demonstration task was implemented. Note that it is arranged such that functions can be viewed in their entirety and do not break over pages.

```
#include "contiki.h"
#include "dev/light-sensor.h"
#include "dev/sht11-sensor.h"
#include <stdio.h>

// Macros for buffer size and sample interval etc
#define BUFFER_SIZE 12
#define SAMPLE_INTERVAL (CLOCK_CONF_SECOND / 2)

// Square root
static int square_root_max_iterations = 20;
static float square_root_precision = 0.001f;

// Deviation thresholds
static float low_deviation_threshold = 100.0f;
static float high_deviation_threshold = 400.0f;

// Light buffer
float light_buffer[BUFFER_SIZE];
static int count = 0;
static int k = 12;

// ===== CUSTOM PRINTING =====
int d1(float f)
{
  // Integer part of the float
  return((int)f);
}

unsigned int d2(float f)
{
  // Find decimal part of the float
  if (f>0)
    return(1000*(f-d1(f)));
  else
    return(1000*(d1(f)-f));
}

void printFloat(float f)
{
  // Using the above functions, print a float
  printf("%d.%d", d1(f), d2(f));
}
```

```
void printCollection(float collection[], int degree)
{
  // Iterate through the given collection of floats (e.g light_buffer) and
  ↪  print it using the above function
  printf("[ ");
  int i = 0;
  for (i = 0; i < degree; i++) {
    printFloat(collection[i]);
    printf(", ");
  }
  printf("]\n");
}

// ===== MEASUREMENTS =====
float getLight(void)
{
  float V_sensor = 1.5 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC)/4096;
                // ^ ADC-12 uses 1.5V_REF
  float I = V_sensor/100000;          // xm1000 uses 100kohm resistor
  float light_lx = 0.625*1e6*I*1000; // convert from current to light
  ↪  intensity
  return light_lx;
}

// ===== BUFFER MANAGEMENT =====
void updateBuffer(void)
{
  // Record a new value to the buffer at the current count position
  float light = getLight();
  light_buffer[count] = light;
  count++;
}
```

```
// ===== CALCULATIONS =====
float calculateSquareRoot(float value)
{
  if (value <= 0) return 0; // Catch-all for negatives or 0

  // Calculate the square root using Babylonian method like seen in the labs
  // Initial value should be somewhat close to the real value (value/2)
  float difference = 0.0;
  float x = value / 2.0f;
  int i = 0;
  for (i = 0; i < square_root_max_iterations; i++) {
    float new_x = 0.5f * (x + value / x);
    difference = new_x - x;

    // If
    if (difference < square_root_precision && difference > -square_root_precision) break;

    x = new_x;
  }
  return x;
}

float calculateMean(float collection[], int start_index, int end_index)
{
  float total = 0;
  int value_count = end_index - start_index;

  int i = start_index;
  for (i = start_index; i < end_index; i++) {
    total += collection[i];
  }
  float mean = total / value_count;
  return mean;
}

float calculateStandardDeviation(float collection[], float mean, int start_index, int end_index)
{
  // Calculate the standard deviation using the functions I defined above
  int value_count = end_index - start_index;
  float variance_sum = 0;

  int i = start_index;
  for (i = start_index; i < end_index; i++) {
    float deviation = collection[i] - mean;
    variance_sum += deviation * deviation;
  }
  float variance = variance_sum / value_count;
  float standard_deviation = calculateSquareRoot(variance);
  return standard_deviation;
}
```

```
// ===== UTILITY FUNCTIONS =====
int findAggregation(float std_dev)
{
  // From the standard deviation, find which of the aggregation values should
  ↪  be used
  // The specification defines three types of aggregation: every 12 values
  ↪  (full), every 4, and every 1 (no aggregation)
  if (std_dev < low_deviation_threshold) {
    printf("Aggregation = 12-into-1.\n");
    return 12;
  }

  if (std_dev < high_deviation_threshold) {
    printf("Aggregation = 4-into-1.\n");
    return 4;
  }

  printf("No aggregation needed.\n");
  return 1;
}
```

```
void processCollection(float collection[])
{
  // Take in a collection of floats (e.g: light_buffer) and do the following:
  // 1. Print the whole collection.
  // 2. Calculate the standard deviation.
  // 3. Decide upon an aggregation degree based on that standard deviation.
  // 4. Perform aggregation and print.

  // Print buffer
  printf("B = ");
  printCollection(light_buffer, count);

  // Find standard deviation
  float mean = calculateMean(light_buffer, 0, count);
  float standard_deviation = calculateStandardDeviation(light_buffer, mean, 0,
   ↪  count);
  printf("StdDev = ");
  printFloat(standard_deviation);
  printf("\n");

  // Print that aggregate collection with averages calculated using mean
  int degree = findAggregation(standard_deviation);

  printf("X = ");

  if (degree == 1) printCollection(light_buffer, count); // For 1-into-1, the
   ↪  array doesn't change

  if (degree == 4) {
    // If we need 4-into-1, we need to aggregate every 4 values into one
     ↪  average
    int aggregate_count = BUFFER_SIZE / degree; // If we want to aggregate n
     ↪  values into 1, we will end up with (total / n) aggregate results

    float aggregate_buffer[aggregate_count];

    int i = 0;
    for (i = 0; i < aggregate_count; i++)
    {
      aggregate_buffer[i] = calculateMean(light_buffer, (i * degree), ((i + 1)
       ↪  * degree));
    }

    printCollection(aggregate_buffer, aggregate_count);
  }

  if (degree == 12) printFloat(mean); // For 12-into-1 the entire array is
   ↪  averaged, already did this earlier so we can save some computation by
   ↪  reusing

  printf("\n\n");
}
```

```c
/*---------------------------------------------------------------------------*/
PROCESS(coursework_process, "Coursework");
AUTOSTART_PROCESSES(&coursework_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(coursework_process, ev, data)
{
  static struct etimer timer; // Initialise a timer

  PROCESS_BEGIN();
  SENSORS_ACTIVATE(light_sensor); // Initialise the light sensor
  etimer_set(&timer, SAMPLE_INTERVAL); // Rate of 2 readings per second

  while(1) {
    // Wait for timer event to do anything
    PROCESS_WAIT_EVENT();

    if (ev == PROCESS_EVENT_TIMER) {
      updateBuffer();

      // If we have filled the buffer, we need to process
      if (count == BUFFER_SIZE) {
        processCollection(light_buffer);
        count = 0; // Reset the counter so we start overwriting values
        ↪  according to FIFO
      }

      // Restart the timer
      etimer_reset(&timer);
    }
  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

2.2 Post-Demonstration

Below is the source code after implementing the code demonstration task.