

COM3023 Coursework Submission

Hallam Saunders (URN: 6788550)

November 17, 2025

Contents

1	Demonstration Screenshots	3
1.1	12-into-1 Aggregation	3
1.2	4-into-1 Aggregation	3
1.3	No Aggregation	3
1.4	Cooja Simulation	3
2	Source Code	4
2.1	Pre-Demonstration	4
2.2	Post-Demonstration	10
	List of Figures	11

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.

1.4 Cooja Simulation

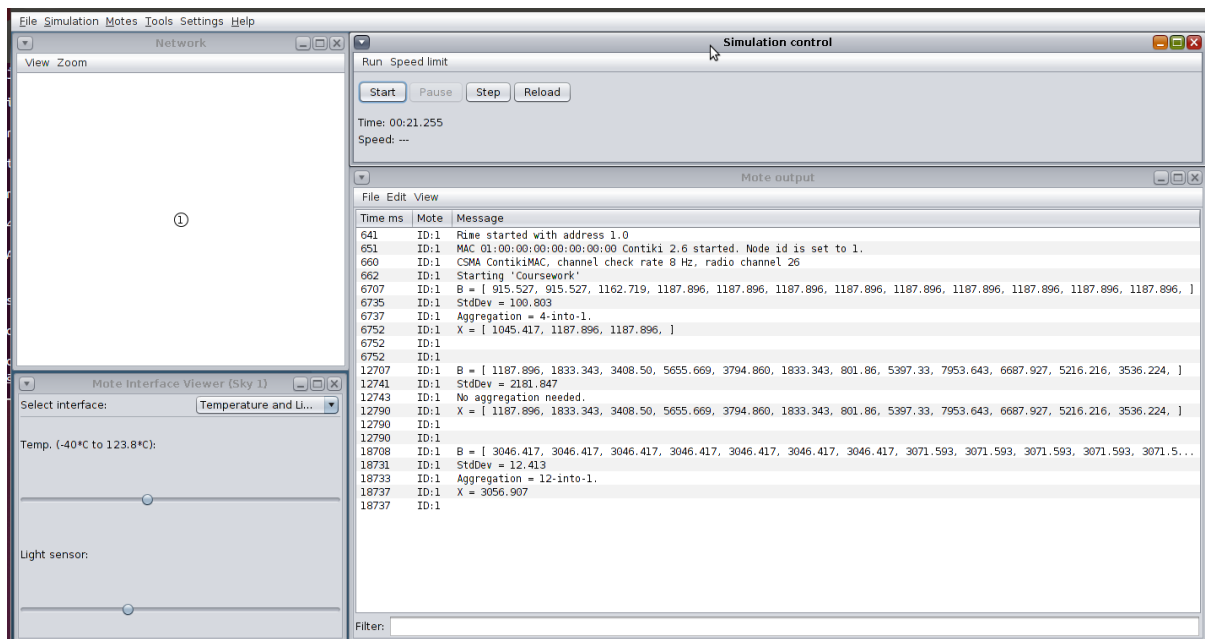


Figure 4: Cooja simulation demonstrating the full output produced.

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.

```
1  #include "contiki.h"
2  #include "dev/light-sensor.h"
3  #include "dev/sht11-sensor.h"
4  #include <stdio.h>
5
6  // Macros for buffer size and sample interval etc
7  #define BUFFER_SIZE 12
8  #define SAMPLE_INTERVAL (CLOCK_CONF_SECOND / 2)
9
10 // Square root
11 static int square_root_max_iterations = 20;
12 static float square_root_precision = 0.001f;
13
14 // Deviation thresholds
15 static float low_deviation_threshold = 100.0f;
16 static float high_deviation_threshold = 400.0f;
17
18 // Light buffer
19 float light_buffer[BUFFER_SIZE];
20 static int count = 0;
21 static int k = 12;
22
23 // ===== CUSTOM PRINTING =====
24 int d1(float f)
25 {
26     // Integer part of the float
27     return((int)f);
28 }
29
30 unsigned int d2(float f)
31 {
32     // Find decimal part of the float
33     if (f>0)
34         return(1000*(f-d1(f)));
35     else
36         return(1000*(d1(f)-f));
37 }
38
39 void printFloat(float f)
40 {
41     // Using the above functions, print a float
42     printf("%d.%d", d1(f), d2(f));
43 }
```

```

44 void printCollection(float collection[], int degree)
45 {
46     // Iterate through the given collection of floats (e.g light_buffer) and
47     ↪ print it using the above function
48     printf("[ ");
49     int i = 0;
50     for (i = 0; i < degree; i++) {
51         printFloat(collection[i]);
52         printf(", ");
53     }
54     printf("]\n");
55 }
56 // ===== MEASUREMENTS =====
57 float getLight(void)
58 {
59     float V_sensor = 1.5 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC)/4096;
60     // ^ ADC-12 uses 1.5V_REF
61     float I = V_sensor/100000; // xm1000 uses 100kohm resistor
62     float light_lx = 0.625*1e6*I*1000; // convert from current to light
63     ↪ intensity
64     return light_lx;
65 }
66 // ===== BUFFER MANAGEMENT =====
67 void updateBuffer(void)
68 {
69     // Record a new value to the buffer at the current count position
70     float light = getLight();
71     light_buffer[count] = light;
72     count++;
73 }

```

```

74 // ===== CALCULATIONS =====
75 float calculateSquareRoot(float value)
76 {
77     if (value <= 0) return 0; // Catch-all for negatives or 0
78
79     // Calculate the square root using Babylonian method like seen in the labs
80     // Initial value should be somewhat close to the real value (value/2)
81     float difference = 0.0;
82     float x = value / 2.0f;
83     int i = 0;
84     for (i = 0; i < square_root_max_iterations; i++) {
85         float new_x = 0.5f * (x + value / x);
86         difference = new_x - x;
87
88         // If difference is within precision bounds, we can stop
89         if (difference < square_root_precision && difference >
            ↪ -square_root_precision) break;
90
91         x = new_x;
92     }
93     return x;
94 }
95
96 float calculateMean(float collection[], int start_index, int end_index)
97 {
98     float total = 0;
99     int value_count = end_index - start_index;
100
101     int i = start_index;
102     for (i = start_index; i < end_index; i++) {
103         total += collection[i];
104     }
105     float mean = total / value_count;
106     return mean;
107 }
108
109 float calculateStandardDeviation(float collection[], float mean, int
    ↪ start_index, int end_index)
110 {
111     // Calculate the standard deviation using the functions I defined above
112     int value_count = end_index - start_index;
113     float variance_sum = 0;
114
115     int i = start_index;
116     for (i = start_index; i < end_index; i++) {
117         float deviation = collection[i] - mean;
118         variance_sum += deviation * deviation;
119     }
120     float variance = variance_sum / value_count;
121     float standard_deviation = calculateSquareRoot(variance);
122     return standard_deviation;
123 }

```

```

124 // ===== UTILITY FUNCTIONS =====
125 int findAggregation(float std_dev)
126 {
127     // From the standard deviation, find which of the aggregation values should
128     ↪ be used
129     // The specification defines three types of aggregation: every 12 values
130     ↪ (full), every 4, and every 1 (no aggregation)
129     if (std_dev < low_deviation_threshold) {
130         printf("Aggregation = 12-into-1.\n");
131         return 12;
132     }
133
134     if (std_dev < high_deviation_threshold) {
135         printf("Aggregation = 4-into-1.\n");
136         return 4;
137     }
138
139     printf("No aggregation needed.\n");
140     return 1;
141 }

```

```

142 void processCollection(float collection[])
143 {
144     // Take in a collection of floats (e.g: light_buffer) and do the following:
145     // 1. Print the whole collection.
146     // 2. Calculate the standard deviation.
147     // 3. Decide upon an aggregation degree based on that standard deviation.
148     // 4. Perform aggregation and print.
149
150     // Print buffer
151     printf("B = ");
152     printCollection(light_buffer, count);
153
154     // Find standard deviation
155     float mean = calculateMean(light_buffer, 0, count);
156     float standard_deviation = calculateStandardDeviation(light_buffer, mean,
157         ↪ 0, count);
158     printf("StdDev = ");
159     printFloat(standard_deviation);
160     printf("\n");
161
162     // Print that aggregate collection with averages calculated using mean
163     int degree = findAggregation(standard_deviation);
164
165     printf("X = ");
166
167     if (degree == 1) printCollection(light_buffer, count); // For 1-into-1, the
168         ↪ array doesn't change
169
170     if (degree == 4) {
171         // If we need 4-into-1, we need to aggregate every 4 values into one
172         ↪ average
173         int aggregate_count = BUFFER_SIZE / degree; // If we want to aggregate n
174         ↪ values into 1, we will end up with (total / n) aggregate results
175
176         float aggregate_buffer[aggregate_count];
177
178         int i = 0;
179         for (i = 0; i < aggregate_count; i++)
180         {
181             aggregate_buffer[i] = calculateMean(light_buffer, (i * degree), ((i +
182                 ↪ 1) * degree));
183         }
184
185         printCollection(aggregate_buffer, aggregate_count);
186     }
187
188     if (degree == 12) printFloat(mean); // For 12-into-1 the entire array is
189         ↪ averaged, already did this earlier so we can save some computation by
190         ↪ reusing
191
192     printf("\n\n");
193 }

```

```

187  /*-----
    ↪  */
188  PROCESS(coursework_process, "Coursework");
189  AUTOSTART_PROCESSES(&coursework_process);
190  /*-----
    ↪  */
191  PROCESS_THREAD(coursework_process, ev, data)
192  {
193      static struct etimer timer; // Initialise a timer
194
195      PROCESS_BEGIN();
196      SENSORS_ACTIVATE(light_sensor); // Initialise the light sensor
197      etimer_set(&timer, SAMPLE_INTERVAL); // Rate of 2 readings per second
198
199      while(1) {
200          // Wait for timer event to do anything
201          PROCESS_WAIT_EVENT();
202
203          if (ev == PROCESS_EVENT_TIMER) {
204              updateBuffer();
205
206              // If we have filled the buffer, we need to process
207              if (count == BUFFER_SIZE) {
208                  processCollection(light_buffer);
209                  count = 0; // Reset the counter so we start overwriting values
                     ↪ according to FIFO
210              }
211
212              // Restart the timer
213              etimer_reset(&timer);
214          }
215      }
216
217      PROCESS_END();
218  }
219  /*-----
    ↪  */

```

2.2 Post-Demonstration

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

List of Figures

1	12-into-1 aggregation indicating little activity.	3
2	4-into-1 aggregation indicating some activity.	3
3	No aggregation indicating high activity.	3
4	Cooja simulation demonstrating the full output produced.	3