

Internet of Things Assignment Code

Aggregation Algorithm

Faiz Ahmad Khan

University of Surrey

2021/2022

Contents:

- Section-1: Libraries (Line 3)
- Section-2: Constants (Line 11)
- Section-3: Functions, Structures & Global Variables (Line 23)
- Section-4: Start Processes (Line 119)
- Section-5: Sensor Reading Process (Line 126)
- Section-6: Advanced Feature (Line 208)
- Section-7: Aggregation Process (Line 267)

```

1 /*----- IOT Assignment -----*/
2
3 //----- Section-1: Libraries
4 #include "contiki.h"
5 #include "dev/light-sensor.h"
6 #include "dev/sht11-sensor.h"
7 #include <stdio.h>
8
9 /*-----*/
10
11 //----- Section-2: Constants
12 #define Advance_Feature 1
13 /* 1- Run advance feature
14    0- Skip advance feature */
15 #define BUFFER_SIZE 12
16 #define K 12
17 // For Cooja Sim
18 #define Light_Lower_Threshold 400
19 #define Light_Upper_Threshold 1000
20
21 /*-----*/
22
23 //----- Section-3: Functions, Structures & Global Variables
24 static process_event_t Buffer_full_event; // Event for passing full Buffers
25
26 // Buffer Structure
27 typedef struct Buffer {
28     float Array[BUFFER_SIZE];
29     int Length;
30     float Mean;
31     float SD;
32 } Buffer;
33
34 // Read Temperature function
35 float getTemperature(void)
36 {
37     int tempData;
38     tempData = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM); // For Cooja Sim
39     float so_temp = tempData;
40     // transfer function
41     float temp = (0.04 * so_temp) - 39.6;
42     return temp;
43 }
44
45 // Read Light function
46 float getLight(void)
47 {
48     float lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
49     // transfer function
50     float V_sensor = (1.5 * lightData) / 4096;
51     float I = V_sensor / 100000;
52     float light_lx = 0.625 * 1e6 * I * 1000;
53     return light_lx;
54 }
55
56 // Calculate mean function
57 float getmean(float arr[],int length)
58 {
59     float sum = 0;
60     int i;
61     for(i=0; i<length; i++)
62     {
63         sum += arr[i];
64     }
65     float mean;
66     mean = sum / length;
67     return mean;
68 }
69
70 // Calculate Standard Deviation function
71 float getSD(float arr[],int length,float mean)
72 {
73     int i;
74     float deviation;
75     float sqdev = 0 ;
76     for(i=0; i<length; i++)
77     {
78         deviation = arr[i] - mean;
79         sqdev += deviation*deviation;
80     }
81     float SD;
82     SD = sqrt(sqdev/length);
83     return SD;
84 }
85
86 // Calculate square root function
87 float sqrt(float num)
88 {
89     float sqrt = num/2;
90     float t = 0;
91     while(sqrt != t)
92     {
93         t = sqrt;
94         sqrt = (num/t + t) / 2;
95     }
96     return sqrt;
97 }
98
99 // Convert Float to Int function
100 int d1(float f) // Integer part
101 {
102     return((int)f);
103 }
104
105 unsigned int d2(float f) // Fractional part
106 {
107     if (f>0)
108     {
109         return(100*(f-d1(f)));
110     }
111     else
112     {
113         return(100*(d1(f)-f));
114     }
115 }

```

```

115 }
116
117 /*-----*/
118
119 //----- Section-4: Start Processes
120 PROCESS(sensor_reading_process, "Sensor reading process");
121 PROCESS(aggregation_process, "Aggregation process");
122 AUTOSTART_PROCESSES(&sensor_reading_process, &aggregation_process);
123
124 /*-----*/
125
126 //----- Section-5: Sensor Reading Process
127 PROCESS_THREAD(sensor_reading_process, ev, data)
128 {
129     static struct etimer timer;
130
131     PROCESS_BEGIN();
132
133     // Set timer
134     etimer_set(&timer, CLOCK_SECOND/2);
135
136     // Activate sensors
137     SENSORS_ACTIVATE(light_sensor);
138     SENSORS_ACTIVATE(sht11_sensor);
139
140     // Allocate global event number
141     Buffer_full_event = process_alloc_event();
142
143     // Buffers
144     static struct Buffer Light_Buffer;
145     static struct Buffer Temp_Buffer;
146
147     static int counter = 0;
148     while(1)
149     {
150         PROCESS_WAIT_EVENT_UNTIL(ev=PROCESS_EVENT_TIMER);
151
152         // Take readings and print
153         float temp = getTemperature();
154         float light_lx = getLight();
155         Light_Buffer.Array[counter] = light_lx;
156         Temp_Buffer.Array[counter] = temp;
157
158         printf("\nLight = %.2u lux\n", d1(light_lx), d2(light_lx));
159         printf("Temp = %.2u C\n ", d1(temp), d2(temp));
160
161         counter++;
162         if (counter == K)
163         {
164             // Get Length of Buffer
165             Light_Buffer.Length = sizeof(Light_Buffer.Array)/sizeof(Light_Buffer.Array[0]);
166             Temp_Buffer.Length = sizeof(Temp_Buffer.Array)/sizeof(Temp_Buffer.Array[0]);
167
168             int i;
169
170             // Print Light Bufffer
171             printf("\nLight Buffer = [ ");
172             for(i=0; i<K; i++)
173             {
174                 printf("%.2u", d1(Light_Buffer.Array[i]), d2(Light_Buffer.Array[i]));
175                 if(i<(K-1))
176                 {
177                     printf(", ");
178                 }
179             }
180             printf(" ] ");
181
182             // Print Temperature Buffer
183             printf("\nTemperature Buffer = [ ");
184             for(i=0; i<K; i++)
185             {
186                 printf("%.2u", d1(Temp_Buffer.Array[i]), d2(Temp_Buffer.Array[i]));
187                 if(i<(K-1))
188                 {
189                     printf(", ");
190                 }
191             }
192             printf(" ] \n");
193
194             // Get mean
195             Light_Buffer.Mean = getmean(Light_Buffer.Array, K);
196             Temp_Buffer.Mean = getmean(Temp_Buffer.Array, K);
197
198             // Get Standard Deviation
199             Light_Buffer.SD = getSD(Light_Buffer.Array, K, Light_Buffer.Mean);
200             Temp_Buffer.SD = getSD(Temp_Buffer.Array, K, Temp_Buffer.Mean);
201
202             // Print Standard Deviation
203             printf("\nStandard Deviation of Light = %.2u ", d1(Light_Buffer.SD), d2(Light_Buffer.SD));
204             printf("\nStandard Deviation of Temperature = %.2u \n", d1(Temp_Buffer.SD), d2(Temp_Buffer.SD));
205
206             /*-----*/
207
208             //----- Section-6: Advance Feature
209             /* To find Linear Regression using Least Squares
210             and the Pearson Correlation Coefficient.
211             y = mx + c
212             y -> Dependant Variable (Temperature)
213             x -> Independant Variable (Light)
214             m -> Slope
215             c -> Intercept
216
217             r = Covariance(x,y) / SD(x) * SD(y)
218             Covariance(x,y) -> Mean(x*y) - Mean(x) * Mean(y)
219             r -> Pearson correlation coefficient */
220
221             if(Advance_Feature == 1)
222             {
223                 printf("\nAdvance Feature\n");
224                 // Linear Regression
225                 float m;
226                 float c;
227                 float num = 0; // numerator
228                 float den = 0; // denominator
229                 for(i=0; i<K; i++)

```

```

231         num += (Light_Buffer.Array[i] - Light_Buffer.Mean) * (Temp_Buffer.Array[i] - Temp_Buffer.Mean);
232         den += (Light_Buffer.Array[i] - Light_Buffer.Mean) * (Light_Buffer.Array[i] - Light_Buffer.Mean);
233     }
234     m = num / den;
235     c = Temp_Buffer.Mean - m * Light_Buffer.Mean;
236     printf("Slope (m) = %.2f\n", d1(m), d2(m));
237     printf("Intercept (c) = %.2f\n", d1(c), d2(c));
238
239     // Pearson correlation coefficient
240     float r;
241     float xy[K];
242     for(i=0; i<K; i++)
243     {
244         xy[i] = Light_Buffer.Array[i] * Temp_Buffer.Array[i];
245     }
246     float mean_xy; // Mean of Product of Vectors
247     mean_xy = getmean(xy,K);
248     float cov; // Covariance
249     cov = mean_xy - (Light_Buffer.Mean * Temp_Buffer.Mean);
250     r = cov / (Light_Buffer.SD * Temp_Buffer.SD);
251     printf("Pearson correlation coefficient (r) = %.2f\n", d1(r), d2(r));
252 }
253
254 /*-----*/
255
256 // Send Buffer to Aggregation Process
257 process_post(&aggregation_process, Buffer_full_event, &Light_Buffer);
258 counter = 0;
259 }
260 etimer_reset(&timer);
261 }
262 PROCESS_END();
263 }
264
265 /*-----*/
266
267 //----- Section-7: Aggregation Process
268 PROCESS_THREAD(aggregation_process, ev, data)
269 {
270     PROCESS_BEGIN();
271
272     while(1)
273     {
274         PROCESS_WAIT_EVENT_UNTIL(ev == Buffer_full_event); //Wait for Buffer to fill
275         Buffer Light_Buffer = *(Buffer *)data; // Cast to a Buffer pointer
276
277         // High Activity No Aggregation
278         if(Light_Buffer.SD > Light_Upper_Threshold)
279         {
280             printf("\nAggregation = No Aggregation\n");
281             // Print Output Buffer
282             printf("Light_X = [ ");
283             int i;
284             for(i=0; i<K; i++)
285             {
286                 printf("%.2f", d1(Light_Buffer.Array[i]), d2(Light_Buffer.Array[i]));
287                 if(i<(K-1))
288                 {
289                     printf(", ");
290                 }
291             }
292             printf(" ] \n");
293         }
294
295         // Some Activity 4-into-1 Aggregation
296         else if(Light_Buffer.SD > Light_Lower_Threshold && Light_Buffer.SD < Light_Upper_Threshold)
297         {
298             printf("\nAggregation = 4-into-1\n");
299             // For K less than 4, aggregate entire buffer
300             if(K<4)
301             {
302                 printf("4-into-1 Aggregation not possible!");
303                 printf("\nAggregation = K-into-1\n");
304                 float Output_Buffer[1];
305                 Output_Buffer[0] = Light_Buffer.Mean;
306                 // Print Output Buffer
307                 printf("Light_X = [ ");
308                 printf("%.2f ] \n", d1(Output_Buffer[0]), d2(Output_Buffer[0]));
309             }
310             // For K equal to 4 or greater
311             else
312             {
313                 int Q = K / 4; // Quotient
314                 int R = K % 4; // Remainder
315                 float sum;
316                 float mean;
317                 int j = 0;
318                 int i;
319                 // For values of K, divisible by 4
320                 if(R == 0)
321                 {
322                     float Output_Buffer[Q];
323                     while(j<K)
324                     {
325                         // Mean of 4 values set to appropriate slot in output buffer
326                         sum = Light_Buffer.Array[j]+Light_Buffer.Array[j+1]+Light_Buffer.Array[j+2]+Light_Buffer.Array[j+3];
327                         mean = sum / 4;
328                         Output_Buffer[j/4] = mean;
329                         j = j+4;
330                     }
331                     // Print Output Buffer
332                     printf("Light_X = [ ");
333                     for(i=0; i<Q; i++)
334                     {
335                         printf("%.2f", d1(Output_Buffer[i]), d2(Output_Buffer[i]));
336                         if(i<(Q-1))
337                         {
338                             printf(", ");
339                         }
340                     }
341                     printf(" ] \n");
342                 }
343                 // For values of K, not divisible by 4
344                 else

```

```

345 {
346     // Adjust Output Buffer to include extra readings within Light Buffer
347     Q = Q + 1;
348     float Output_Buffer[Q];
349     while(j<(K-R))
350     {
351         // Mean of 4 values set to appropriate slot in output buffer
352         sum = Light_Buffer.Array[j]+Light_Buffer.Array[j+1]+Light_Buffer.Array[j+2]+Light_Buffer.Array[j+3];
353         mean = sum / 4;
354         Output_Buffer[j/4] = mean;
355         j = j+4;
356     }
357     if(R == 1)
358     {
359         // Insert final value of Light Buffer into Output Buffer
360         Output_Buffer[Q-1] = Light_Buffer.Array[K-1];
361     }
362     else if(R == 2)
363     {
364         // Insert mean of Last 2 values of Light Buffer into Output Buffer
365         sum = Light_Buffer.Array[K-1] + Light_Buffer.Array[K-2];
366         mean = sum / R;
367         Output_Buffer[Q-1] = mean;
368     }
369     else if(R == 3)
370     {
371         // Insert mean of Last 3 values of Light Buffer into Output Buffer
372         sum = Light_Buffer.Array[K-1] + Light_Buffer.Array[K-2] + Light_Buffer.Array[K-3];
373         mean = sum / R;
374         Output_Buffer[Q-1] = mean;
375     }
376     // Print Output Buffer
377     printf("Light_X = [ ");
378     for(i=0; i<Q; i++)
379     {
380         printf("%d.%u",d1(Output_Buffer[i]), d2(Output_Buffer[i]));
381         if(i<(Q-1))
382         {
383             printf(", ");
384         }
385     }
386     printf(" ] \n");
387 }
388 }
389 }
390
391 // No Activity K-into-1 Aggregation
392 else
393 {
394     printf("\nAggregation = K-into-1 \n");
395     float Output_Buffer[1];
396     Output_Buffer[0] = Light_Buffer.Mean;
397     // Print Output Buffer
398     printf("Light_X = [ ");
399     printf("%d.%u ] \n",d1(Output_Buffer[0]), d2(Output_Buffer[0]));
400 }
401 printf("-----\n");
402 }
403 PROCESS_END();
404 }
405
406 /*-----*/

```