

```

1 /*-----Basic Features-----*/
2 #include "contiki.h"
3 #include "dev/light-sensor.h"
4 #include "lib/list.h"
5 #include "lib/memb.h"
6 #include <stdio.h> /* For printf() */
7
8 #define BUFFER_SIZE 12
9 #define LOW_ACTIVITY_THRESHOLD 1000
10 #define HIGH_ACTIVITY_THRESHOLD 2000
11 #define SAX_FRAGMENTS 4
12
13 /* Helper functions */
14 static void print_float(float number)
15 {
16     int integer_part = (int)number;
17     int decimal_part = (int)((number - integer_part) * 1000);
18     printf("%d.%02d", integer_part, decimal_part);
19 }
20
21 float read_light_sensor(void)
22 {
23     float V_sensor = 1.5 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC) / 4096;
24     // ^ ADC-12 uses 1.5V_REF
25     float I = V_sensor / 100000; // xm1000 uses 100kohm resistor
26     float light_lx = 0.625 * 1e6 * I * 1000; // convert from current to light intensity
27     return light_lx;
28 }
29
30 /* Use linked list for sensor data */
31 struct sensor_data
32 {
33     struct sensor_data *next;
34     float light;
35 };
36
37 LIST(sensor_list);
38 MEMB(sensor_mem, struct sensor_data, BUFFER_SIZE);
39
40 static void add_sensor_data(float light)
41 {
42     struct sensor_data *new_data;
43
44     if (list_length(sensor_list) >= BUFFER_SIZE)
45     {
46         struct sensor_data *oldest = list_pop(sensor_list);
47         memb_free(&sensor_mem, oldest);
48     }
49
50     new_data = memb_alloc(&sensor_mem);
51     if (new_data == NULL)
52     {
53         printf("Memory allocation failed!\n");
54         return;
55     }
56
57     new_data->light = light;
58     list_add(sensor_list, new_data);
59 }
60
61 static float calculate_avg()
62 {
63     struct sensor_data *item;
64     float sum = 0.0;
65     int count = 0;

```

```

66     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
67     {
68         sum += item->light;
69         count++;
70     }
71     return (count == 0) ? 0.0 : sum / count;
72 }
73
74 static float calculate_ssd(float avg)
75 {
76     struct sensor_data *item;
77     float ssd = 0.0;
78     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
79     {
80         float diff = item->light - avg;
81         ssd += diff * diff;
82     }
83     return ssd;
84 }
85
86 static float sqrt_approx(float ssd)
87 {
88     float error = 0.001; // Error tolerance for Babylonian method
89     float x = ssd;        // Initial guess for square root
90     float difference;
91     int i;
92
93     if (ssd == 0)
94     {
95         return 0.0; // No variance
96     }
97
98     for (i = 0; i < 50; i++)
99     { // Babylonian method
100         x = 0.5 * (x + ssd / x);
101         difference = x * x - ssd;
102         if (difference < 0)
103         {
104             difference = -difference;
105         }
106         if (difference < error)
107         {
108             break;
109         }
110     }
111     return x;
112 }
113
114 static float calculate_std()
115 {
116     float avg = calculate_avg();
117     float ssd = calculate_ssd(avg);
118     return sqrt_approx(ssd);
119 }
120
121 void perform_sax(char sax_output[SAX_FRAGMENTS])
122 {
123     float avg = calculate_avg();
124     float std = calculate_std();
125     struct sensor_data *item;
126     int i, j;
127
128     // normalize the time series
129     float normalized_data[BUFFER_SIZE];
130     int idx = 0;
131     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))

```

```

132     {
133         normalized_data[idx] = (item->light - avg) / std;
134         idx++;
135     }
136
137     // perform PAA
138     float fragment_means[SAX_FRAGMENTS] = {0};
139     int fragment_size = BUFFER_SIZE / SAX_FRAGMENTS;
140
141     for (i = 0; i < SAX_FRAGMENTS; i++) {
142         float sum = 0;
143         for (j = 0; j < fragment_size; j++) {
144             sum += normalized_data[i * fragment_size + j];
145         }
146         fragment_means[i] = sum / fragment_size;
147     }
148
149     // convert symbols with gaussain breakpoints
150     char alphabet[4] = {'A', 'B', 'C', 'D'};
151     float breakpoints[3] = {-0.67, 0, 0.67};
152
153     for (i = 0; i < SAX_FRAGMENTS; i++) {
154         float z = fragment_means[i];
155         if (z <= breakpoints[0]) {
156             sax_output[i] = alphabet[0];
157         }
158         else if (z <= breakpoints[1]) {
159             sax_output[i] = alphabet[1];
160         }
161         else if (z <= breakpoints[2]) {
162             sax_output[i] = alphabet[2];
163         }
164         else {
165             sax_output[i] = alphabet[3];
166         }
167     }
168 }
169
170 static void aggregate_and_report()
171 {
172     struct sensor_data *item = list_head(sensor_list);
173     float std = calculate_std();
174     float avg = calculate_avg();
175     char sax_output[SAX_FRAGMENTS];
176
177     // Print the original buffer
178     printf("B = ");
179     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
180     {
181         print_float(item->light);
182         if (list_item_next(item) != NULL)
183         {
184             printf(", ");
185         }
186     }
187     printf("\n");
188
189     // Print the standard deviation
190     printf("StdDev = ");
191     print_float(std);
192     printf("\n");
193
194     // Determine the activity level and aggregation
195     if (std < LOW_ACTIVITY_THRESHOLD)
196     {
197         printf("Aggregation = 12-into-1\n");

```

```

198     printf("X = [");
199     print_float(avg);
200     printf("]\n");
201 }
202 else if (std < HIGH_ACTIVITY_THRESHOLD)
203 {
204     printf("Aggregation = 4-into-1\n");
205     printf("X = [");
206     int count = 0;
207     float sum = 0.0;
208     item = list_head(sensor_list);
209     while (item != NULL)
210     {
211         sum += item->light;
212         count++;
213         if (count == 4)
214         {
215             print_float(sum / 4);
216             sum = 0.0;
217             count = 0;
218             if (list_item_next(item) != NULL)
219             {
220                 printf(", ");
221             }
222         }
223         item = list_item_next(item);
224     }
225     printf("]\n");
226 }
227 else
228 {
229     printf("Aggregation = 1-into-1\n");
230     printf("X = [");
231     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
232     {
233         print_float(item->light);
234         if (list_item_next(item) != NULL)
235         {
236             printf(", ");
237         }
238     }
239     printf("]\n");
240 }
241 }
242
243 // Perform SAX transformation and print
244 perform_sax(sax_output);
245 printf("SAX = [");
246 int i;
247 for (i = 0; i < SAX_FRAGMENTS; i++)
248 {
249     printf("%c", sax_output[i]);
250     if (i < SAX_FRAGMENTS - 1)
251     {
252         printf(", ");
253     }
254 }
255 printf("]\n");
256 }
257
258 /*-----*/
259 PROCESS(sensor_reading_process, "Sensor reading process");
260 AUTOSTART_PROCESSES(&sensor_reading_process);
261 /*-----*/
262 PROCESS_THREAD(sensor_reading_process, ev, data)
263 {

```

```

264     static struct etimer timer;
265     static int sample_counter = 0;
266     static int k = 12; // number of samples before aggregation
267
268     PROCESS_BEGIN();
269     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
270
271     SENSORS_ACTIVATE(light_sensor);
272
273     while (1)
274     {
275         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
276
277         float light = read_light_sensor();
278         add_sensor_data(light);
279         sample_counter++;
280
281         if (sample_counter >= k)
282         {
283             aggregate_and_report();
284             sample_counter = 0;
285         }
286
287         etimer_reset(&timer);
288     }
289
290     PROCESS_END();
291 }

```

```

1 /*-----Advanced Features-----*/
2 #include "contiki.h"
3 #include "dev/light-sensor.h"
4 #include "dev/sht11-sensor.h"
5 #include "lib/list.h"
6 #include "lib/memb.h"
7 #include <stdio.h> /* For printf() */
8
9 #define PI 3.14159f
10 #define LN2 0.69315f
11 #define BUFFER_SIZE 12
12
13 /* Helper functions for sensor reading and printing */
14 static void print_float(float number)
15 {
16     int integer_part = (int)number;
17     int decimal_part = (int)((number - integer_part) * 1000);
18     printf("%d.%02d", integer_part, decimal_part);
19 }
20
21 float read_light_sensor(void)
22 {
23     float V_sensor = 1.5 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC) / 4096;
24     // ^ ADC-12 uses 1.5V_REF
25     float I = V_sensor / 100000; // xm1000 uses 100kohm resistor
26     float light_lx = 0.625 * 1e6 * I * 1000; // convert from current to light intensity
27     return light_lx;
28 }
29
30 float get_temp_sensor(void)
31 {
32     // For simulation sky mote
33     int tempADC = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM);
34     float temp = 0.04 * tempADC - 39.6; // skymote uses 12-bit ADC, or 0.04 resolution
35
36     // For xm1000 mote
37     // int tempADC = sht11_sensor.value(SHT11_SENSOR_TEMP);
38     // float temp = 0.01*tempADC-39.6; // xm1000 uses 14-bit ADC, or 0.01 resolution
39
40     return temp;
41 }
42
43 /* Use linked list for sensor data */
44 struct sensor_data
45 {
46     struct sensor_data *next;
47     float value;
48 };
49
50 LIST(light_list);
51 LIST(temp_list);
52 MEMB(light_mem, struct sensor_data, BUFFER_SIZE);
53 MEMB(temp_mem, struct sensor_data, BUFFER_SIZE);
54
55 static void add_sensor_data(float value, list_t lst, struct memb *mem)
56 {
57     struct sensor_data *new_data;
58
59     if (list_length(lst) >= BUFFER_SIZE)
60     {
61         struct sensor_data *oldest = list_pop(lst);
62         memb_free(mem, oldest);
63     }
64
65     new_data = memb_alloc(mem);
66
67     if (new_data == NULL)
68     {
69         printf("Memory allocation failed!\n");
70         return;
71     }
72
73     new_data->value = value;
74     list_add(lst, new_data);

```

```

73     list_add(list, new_data);
74 }
75
76 // Math helper functions for approximation
77 static float sqrt_approx(float ssd)
78 {
79     float error = 0.001; // Error tolerance for Babylonian method
80     float x = ssd;       // Initial guess for square root
81     float difference;
82     int i;
83
84     if (ssd == 0)
85     {
86         return 0.0; // No variance
87     }
88
89     for (i = 0; i < 50; i++)
90     {
91         x = 0.5 * (x + ssd / x);
92         difference = x * x - ssd;
93         if (difference < 0)
94         {
95             difference = -difference;
96         }
97         if (difference < error)
98         {
99             break;
100         }
101     }
102     return x;
103 }
104
105 // Math functions from basic features
106 static float calculate_avg(list_t lst)
107 {
108     struct sensor_data *item;
109     float sum = 0.0;
110     int count = 0;
111     for (item = list_head(lst); item != NULL; item = list_item_next(item))
112     {
113         sum += item->value;
114         count++;
115     }
116     return (count == 0) ? 0.0 : sum / count;
117 }
118
119 static float calculate_ssd(float avg, list_t lst)
120 {
121     struct sensor_data *item;
122     float ssd = 0.0;
123     for (item = list_head(lst); item != NULL; item = list_item_next(item))
124     {
125         float diff = item->value - avg;
126         ssd += diff * diff;
127     }
128     return ssd;
129 }
130
131 static float calculate_std(list_t lst)
132 {
133     float avg = calculate_avg(lst);
134     float ssd = calculate_ssd(avg, lst);
135     return sqrt_approx(ssd);
136 }
137
138 static float calculate_manhattan_dist(list_t light_list, list_t temp_list)
139 {
140     struct sensor_data *light_item = list_head(light_list);
141     struct sensor_data *temp_item = list_head(temp_list);
142     float dist = 0.0f; // Use float with f suffix
143
144     while (light_item != NULL && temp_item != NULL)
145     {
146         float diff = light_item->value - temp_item->value;

```

```

147     dist += (diff < 0) ? -diff : diff; // Inline abs for float
148     light_item = list_item_next(light_item);
149     temp_item = list_item_next(temp_item);
150 }
151 return dist;
152 }
153
154 static float calculate_correlation(list_t light_list, list_t temp_list)
155 {
156     struct sensor_data *light_item = list_head(light_list);
157     struct sensor_data *temp_item = list_head(temp_list);
158
159     float avg_x = calculate_avg(light_list);
160     float avg_y = calculate_avg(temp_list);
161     float std_x = calculate_std(light_list);
162     float std_y = calculate_std(temp_list);
163
164     if (std_x == 0 || std_y == 0)
165     {
166         // correlation is undefined if either std is zero
167         return 0.0;
168     }
169
170     float numerator = 0.0;
171     while (light_item != NULL && temp_item != NULL)
172     {
173         float x = light_item->value;
174         float y = temp_item->value;
175
176         numerator += (x - avg_x) * (y - avg_y);
177
178         light_item = list_item_next(light_item);
179         temp_item = list_item_next(temp_item);
180     }
181     return numerator / (std_x * std_y);
182 }
183
184 // STFT Implementation
185 typedef struct
186 {
187     float real;
188     float imag;
189 } complex_t;
190
191 // helper functions for linked list
192 float list_get(list_t lst, int index)
193 {
194     struct sensor_data *element = (struct sensor_data *)list_head(lst);
195     int curr_index = 0;
196     while (element != NULL && curr_index < index)
197     {
198         element = (struct sensor_data *)list_item_next(element);
199         curr_index++;
200     }
201     if (element == NULL)
202     {
203         printf("Index %d is out of bounds. \n", index);
204         return 0.0;
205     }
206     return element->value;
207 }
208
209 // define memory pool
210 MEMB(chunk_pool, float, 4);
211
212 // math helper functions for advanced features
213 float sine_approx(float x)
214 {
215     float term = x; // First term
216     float result = term;
217     int sign = -1;
218     int n;
219

```



```

220 // Use 5 terms for approximation
221 for (n = 3; n <= 9; n += 2)
222 {
223     term = term * x * x / (n * (n - 1)); // Compute next term
224     result += sign * term;                // Add/subtract the term
225     sign = -sign;                        // Alternate sign
226 }
227 return result;
228 }
229
230 float cosine_approx(float x)
231 {
232     float term = 1.0f; // First term
233     float result = term;
234     int sign = -1;
235     int n;
236
237     // Use 5 terms for approximation
238     for (n = 2; n <= 8; n += 2)
239     {
240         term = term * x * x / (n * (n - 1)); // Compute next term
241         result += sign * term;                // Add/subtract the term
242         sign = -sign;                        // Alternate sign
243     }
244     return result;
245 }
246
247 float log_approx(float x)
248 {
249     if (x <= 0.0f)
250         return 0.0f; // Guard against invalid input
251
252     float result = 0.0f;
253     float term = (x - 1.0f) / (x + 1.0f);
254     float term_squared = term * term;
255     float numerator = term;
256     int n;
257
258     // Use first 4 terms of series expansion
259     for (n = 1; n <= 7; n += 2)
260     {
261         result += numerator / n;
262         numerator *= term_squared;
263     }
264
265     return 2.0f * result;
266 }
267
268 void fft(complex_t *data, int fft_size)
269 {
270     int i, j, len;
271
272     // Bit-reversal permutation
273     for (i = 0, j = 0; i < fft_size; i++)
274     {
275         if (i < j)
276         {
277             // Swap data[i] and data[j]
278             complex_t temp = data[i];
279             data[i] = data[j];
280             data[j] = temp;
281         }
282         int m = fft_size / 2;
283         while (m >= 1 && j >= m)
284         {
285             j -= m;
286             m /= 2;
287         }
288         j += m;
289     }
290
291     // Iterative FFT computation
292     for (len = 2; len <= fft_size; len *= 2)
293     {

```

```

293
294     float angle = -2.0f * PI / len;
295     complex_t wlen = {cosine_approx(angle), sine_approx(angle)};
296
297     for (i = 0; i < fft_size; i += len)
298     {
299         complex_t w = {1.0, 0.0};
300         for (j = 0; j < len / 2; j++)
301         {
302             complex_t u = data[i + j];
303             complex_t v = {
304                 w.real * data[i + j + len / 2].real - w.imag * data[i + j + len / 2].imag,
305                 w.real * data[i + j + len / 2].imag + w.imag * data[i + j + len / 2].real};
306
307             data[i + j].real = u.real + v.real;
308             data[i + j].imag = u.imag + v.imag;
309             data[i + j + len / 2].real = u.real - v.real;
310             data[i + j + len / 2].imag = u.imag - v.imag;
311
312             // Update twiddle factor
313             complex_t w_new = {
314                 w.real * wlen.real - w.imag * wlen.imag,
315                 w.real * wlen.imag + w.imag * wlen.real};
316             w = w_new;
317         }
318     }
319 }
320 }
321
322 // Helper function to print STFT results as a table
323 void print_stft_results(int chunk_index, complex_t *data, int chunk_size)
324 {
325     int j;
326     printf("Chunk %-3d", chunk_index);
327     for (j = 0; j < chunk_size; j++)
328     {
329         print_float(data[j].real);
330
331         printf(" + ");
332         print_float(data[j].imag);
333         printf("i ");
334     }
335     printf("\n");
336 }
337
338 static void perform_stft(list_t lst, int chunk_size, int hop_size)
339 {
340     memb_init(&chunk_pool);
341
342     int signal_length = list_length(lst);
343     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
344     printf("signal length %d and %d chunks\n", signal_length, num_chunks);
345     int i, j;
346     for (i = 0; i < num_chunks; i++)
347     {
348         float *chunk = (float *)memb_alloc(&chunk_pool);
349         if (chunk == NULL)
350         {
351             printf("Chunk allocation failed\n");
352             return;
353         }
354
355         int start_index = i * hop_size;
356         // printf("Start index: %d\n", start_index);
357         for (j = 0; j < chunk_size; j++)
358         {
359             int signal_index = start_index + j;
360             // printf("Signal index: %d\n", signal_index);
361             if (signal_index < signal_length)
362             {
363                 chunk[j] = list_get(lst, signal_index);
364             }
365             else
366             {
367                 chunk[j] = 0.0f; // just in case, not needed in our case.

```

```

367     }
368 }
369 complex_t data[chunk_size];
370 for (j = 0; j < chunk_size; j++)
371 {
372     data[j].real = chunk[j];
373     data[j].imag = 0.0f;
374     /*-----DEBUGGING-----*/
375     // printf("CHUNK VALUE: \n");
376     // print_float(data[j].real);
377     // printf("\n");
378     // print_float(data[j].imag);
379     // printf("\n");
380     /*-----DEBUGGING-----*/
381 }
382 fft(data, chunk_size);
383
384 /*-----DEBUGGING-----*/
385 print_stft_results(i, data, chunk_size);
386 /*-----DEBUGGING-----*/
387
388 memb_free(&chunk_pool, chunk);
389 }
390 }
391
392 void compute_power_spectrum(complex_t *chunk, float *power_spectrum, int chunk_size)
393 {
394     int k;
395     for (k = 0; k < chunk_size; k++)
396     {
397         power_spectrum[k] = chunk[k].real * chunk[k].real +
398                             chunk[k].imag * chunk[k].imag;
399     }
400 }
401
402 static float compute_spectral_entropy(list_t lst, int chunk_size, int hop_size)
403 {
404     int signal_length = list_length(lst);
405     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
406
407     if (num_chunks <= 0)
408         return 0.0f;
409
410     // Allocate memory for power spectrum calculations
411     float power_spectrum[chunk_size];
412     float avg_power_spectrum[chunk_size];
413     float pdf[chunk_size];
414
415     // Initialize average power spectrum
416     int i;
417     for (i = 0; i < chunk_size; i++)
418     {
419         avg_power_spectrum[i] = 0.0f;
420     }
421
422     // Process each chunk
423     for (i = 0; i < num_chunks; i++)
424     {
425         complex_t chunk_data[chunk_size];
426         int j;
427         for (j = 0; j < chunk_size; j++)
428         {
429             int signal_index = i * hop_size + j;
430             if (signal_index < signal_length)
431             {
432                 chunk_data[j].real = list_get(lst, signal_index);
433                 chunk_data[j].imag = 0.0f;
434             }
435             else
436             {
437                 chunk_data[j].real = 0.0f;
438                 chunk_data[j].imag = 0.0f;
439             }
440         }
441     }

```

```

440     }
441
442     fft(chunk_data, chunk_size);
443
444     // Compute power spectrum for this chunk
445     compute_power_spectrum(chunk_data, power_spectrum, chunk_size);
446
447     // Add to average power spectrum
448     for (j = 0; j < chunk_size; j++)
449     {
450         avg_power_spectrum[j] += power_spectrum[j];
451     }
452 }
453
454 // Compute final average
455 float total_power = 0.0f;
456 for (i = 0; i < chunk_size; i++)
457 {
458     avg_power_spectrum[i] /= num_chunks;
459     total_power += avg_power_spectrum[i];
460 }
461
462 // Compute PDF and entropy
463 float entropy = 0.0f;
464 if (total_power > 0.0f)
465 {
466     for (i = 0; i < chunk_size; i++)
467     {
468         if (avg_power_spectrum[i] > 0.0f)
469         {
470             pdf[i] = avg_power_spectrum[i] / total_power;
471             entropy -= pdf[i] * log_approx(pdf[i]);
472         }
473     }
474 }
475
476 return entropy;
477 }
478
479 static void aggregate_and_report()
480 {
481     struct sensor_data *light_item = list_head(light_list);
482     struct sensor_data *temp_item = list_head(temp_list);
483     printf("X (Light Sensor Readings) = [");
484     for (light_item = list_head(light_list); light_item != NULL; light_item = list_item_next(light_item))
485     {
486         print_float(light_item->value);
487         if (list_item_next(light_item) != NULL)
488         {
489             printf(", ");
490         }
491     }
492     printf("]\n");
493     printf("Y (Temperature Sensor Readings) = [");
494     for (temp_item = list_head(temp_list); temp_item != NULL; temp_item = list_item_next(temp_item))
495     {
496         print_float(temp_item->value);
497         if (list_item_next(temp_item) != NULL)
498         {
499             printf(", ");
500         }
501     }
502     printf("]\n");
503
504     printf("Manhattan Distance: ");
505     print_float(calculate_manhattan_dist(light_list, temp_list));
506     printf("\n");
507
508     printf("Correlation: ");
509     print_float(calculate_correlation(light_list, temp_list));
510     printf("\n");
511
512     printf("Performing STFT with ");
513     perform_stft(light_list, 4, 2);

```

```

514     printf("\n");
515
516     printf("Spectral Entropy: ");
517     print_float(compute_spectral_entropy(light_list, 4, 2));
518     printf("\n");
519 }
520 /*-----*/
521 PROCESS(sensor_reading_process, "Sensor reading process");
522 AUTOSTART_PROCESSES(&sensor_reading_process);
523 /*-----*/
524 PROCESS_THREAD(sensor_reading_process, ev, data)
525 {
526     static struct etimer timer;
527     static int sample_counter = 0;
528
529     static int k = 12; // number of samples before calculation
530
531     PROCESS_BEGIN();
532     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
533
534     SENSORS_ACTIVATE(light_sensor);
535     SENSORS_ACTIVATE(sht11_sensor);
536
537     while (1)
538     {
539         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
540
541         float light = read_light_sensor();
542         float temp = get_temp_sensor();
543         add_sensor_data(light, light_list, &light_mem);
544         add_sensor_data(temp, temp_list, &temp_mem);
545
546         sample_counter++;
547         if (sample_counter >= k)
548         {
549             aggregate_and_report();
550             sample_counter = 0;
551         }
552
553         etimer_reset(&timer);
554     }
555
556     PROCESS_END();
557 }

```