⌨️

Advanced Feature Implementation

Data Structure Selection 🔗

Defining sensor data and the `complex_t` datatype for complex numbers

```
struct sensor_data
{
    struct sensor_data *next;
    float value;
};

typedef struct
{
    float real;
    float imag;
} complex_t
```

The Contiki list API calls

```
void *list_head(list_t list);
void *list_item_next(void *item);
int list_length(list_t list);
void list_add(list_t list, void *item);
void *list_pop(list_t list);
```

Initialization of lists and memory pools

```
LIST(light_list);
LIST(temp_list);
MEMB(light_mem, struct sensor_data, BUFFER_SIZE);
MEMB(temp_mem, struct sensor_data, BUFFER_SIZE);
```

For my advanced feature implementation, I've chosen Contiki's List Library to track sensor readings. Just like any linked list implementation, each item on the list contains a value and a pointer to the next item. I've defined `light_list` and `temp_list` as `list_t` types using the `LIST()` macro. It is important to note that **the module using the list allocates memory for both the list handle and its items**. The list library doesn't handle memory allocation or store the lists' state. Therefore, we must allocate memory using the `memb` module. I've already set the buffer size for this memory allocation.

On the left are the five functions in the Contiki list API that I've used to implement the advanced features. I've also implemented a `list_get` function to get the specific item on a list given the index. It is a helper function meant for my `perform_stft` and `compute_spectral_entropy` function.

Memory Allocation 🧠

The Contiki memory allocation API calls

```
MEMB(name, structure, num);
void memb_init(struct memb *m);
void *memb_alloc(struct memb *m);
int memb_free(struct memb *m, void *ptr);
```

According to the Contiki Memory Allocation doc, the use of `malloc` is discouraged. **Instead, the `memb memory block allocator` is most frequently used.** Throughout my code, I have used the following four API calls to allocate memory for temperature readings, light sensor readings, and each chunk of my STFT transformations (5 total chunks, with each chunk containing 4 samples).

Floating Point Precision Loss 🔢

Give that single-precision floats have approximately 7 decimal digits of precision. Take the `calculate_manhattan_distance` function for instance, large numbers in the thousands from our light sensor readings might get rounded, leading to **precision loss**. As `dist` accumulates large differences, precision errors can compound causing **accumulation errors**. Moreover, the disparity in scales between light sensor and temperature readings can **cause smaller values to be "swamped" by numerical errors**.

Professor Elzanaty suggested **normalizing the data** to improve numerical stability for floating-point calculations. However, Dr. Foh confirmed that we don't need to account for unusually high light sensor values, so I've decided not to implement this approach.

Method Return Type and Chunking 🪓

To further optimize memory usage, I've intentionally used `void` returns in most of my implementations. This **in-place modifications** reduces memory copying operations and avoids creating additional memory allocation for return values. This is evident in functions like `fft`, `compute_power_spectrum` and `perform_stft`, which are all relatively resource-heavy processes. The tradeoff is that the code becomes less functional-programming-friendly, but I think these are acceptable tradeoffs for the performance and reliability benefits.Moreover, I used Contiki's memory pool management to **allocate and free a single chunk buffer for each window of data in my SFFT implementation**. This chunking approach, **combined with in-place FFT calculations** and immediate memory deallocation after processing each chunk, ensures minimal memory footprint and prevents fragmentation.

Exclusion of Math Libraries 📐

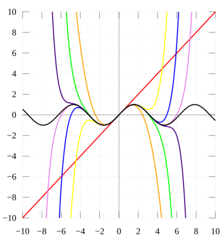
The `f` suffix ensures the values are stored as floats rather than doubles

```
#define PI 3.14159f
#define LN2 0.69315f
```

Math approximation methods

```
1. sqrt_approx: Babylonian method
2. sine_approx: Taylor series (5 terms)
3. cosine_approx: Taylor series (5 terms)
4. log_approx: Series expansion (4 terms)
```

Order 5 approximation should suffice for inputs in [-π/2, π/2]



Despite being able to include the math library by adding a `-lm` flag to the Makefile's `TARGET_LIBFILES` variables. I have opted not using `math.h` to avoid external library dependencies as well as lowering the memory footprint. As a result, I've implemented four math helper functions using various approximation methods: `sqrt_approx`, `sine_approx`, `cosine_approx`, and `log_approx`. The specific approximation methods for each function are detailed in the code block above. For the absolute value in the `manhattan distance` calculation, I've opted for a simple conditional instead of `fabs`.

Spawning Another Process 🧵

I am also considering spawning a separate process to maintain sensor sampling rate by defining a custom event called `PROCESS_EVENT_DATA_READY` and use it with `process_post()` to signal between processes.

```
process_post(&signal_processing_process, PROCESS_EVENT_DATA_READY, NULL);
```

But again, for my current application, **I don't see a compelling reason to split the process given that i have relatively infrequent sensor reading (2 readings / second) and FFT is only computed every 12 samples. Moreover, the STFT and power spectrum calculations need the complete buffer of sensor readings.**

Screenshot

Results of the advanced feature implementation