Wireless Sensor Networks - Assignment 1

author: Kristoffer Plagborg Bak Sørensen

auid: 649525

date: 30-09-2022

repository: github.com/kpbs5/wireless_sensor_networks_assignment_1

Introduction

> This report contains the implementation of the Discrete Cosine Transform compression algorithm using `contiki-ng` for the Telos B Mote microcontroller. First the theory behind the algorithm is explained. Then some of the methodology used to implement and test the algorithm is presented. A set of experiments with different parameters have been conducted, the results of which are presented and concluded upon.

Theory

The DCT-II transform of a signal $\mathbf{x}$ of length L is defined as:

$$y_k = \sqrt{\frac{2-\delta(k)}{L}} \sum_{n=0}^{L-1} x_n \cos\left(\frac{\pi}{L}\left(n+1/2\right)k\right)$$

where y_k is the k^{th} DCT coefficient off the signal, $\mathbf{x}$, for a particular k. The function $\delta(k)$ is defined as:

$$\delta(k) = \begin{cases} 1 & k = 0, \\ 0 & elsewhere \end{cases}$$

The DCT transformation can be written as a matrix product with a vector:

$$\mathbf{y} = \mathbf{H}\mathbf{x}$$

where $\mathbf{H} \in \mathbb{R}^{L\times L}$ with the k, n th entry given us:

$$H[k,n] = \sqrt{\frac{2-\delta(k)}{L}} \cos\left(\frac{\pi}{L}(n+1/2)k\right)$$

The compression works on an input signal $\mathbf{x}$ of length N. The signal is splitted into N/L disjoint subsequences all of even size. This means that the choice of L

must be selected such that it satisfies $N \bmod L = 0$. For each subsequence $\mathbf{x_i}$ the DCT transform is applied:

$$\mathbf{y_i} = \mathbf{H}\mathbf{x_i}$$

The first M coefficients of $\mathbf{y_i}$ is retained, where $M < L$:

$$\mathbf{y_i}' = \mathbf{y_i}[0, 1, \ldots, M-1]$$

The compressed signal $\mathbf{y}'$ is all $\mathbf{y_i}'$ concatenated together in order:

$$\mathbf{y}' = [\mathbf{y_0}', \mathbf{y_1}', \ldots, \mathbf{y_{L-1}}']$$

To decompress the compressed signal the signal is first splitted into M disjoint subsequences of even size. Each subsequence $\mathbf{y_i}'$ is right padded with $L-M$ zeros, such that it has a size of $L - M + M = L$.

$$\mathbf{y_i} = [\mathbf{y_i}', 0_0, 0_1, \ldots, 0_{L-M}]$$

Then $\mathbf{y_i}$ is multiplied with the inverse matrix of $\mathbf{H}$ to get $\mathbf{x_i}$:

$$\mathbf{x_i} = \mathbf{H}^{-1}\mathbf{y_i}$$

The decompressed signal $\mathbf{x}$ is all $\mathbf{x_i}$ concatenated together in order:

$$\mathbf{x}' = [\mathbf{x_0}', \mathbf{x_1}', \ldots, \mathbf{x_{L-1}}']$$

The compression ratio is defined as L/M. By changing L and M higher or lower levels of compression can be achieved. A higher compression ratio will result in smaller memory usage to store the data, and using less time and energy to transmit it. At the cost of having to use more time to compress the data, and likely lose some information when decompressing, as the DCT transform is **lossy** compression algorithm.

Methodology

Computing the H Matrix

In each experiment configuration, the matrix H has been precomputed and embedded in the binary as a static array. The script `./gen_H.py` generates the matrix and transpiles it to a static `c` array. e.g. for $L = 8$:

```c
static const float H[8][8] = {
    { 0.354f, 0.354f, 0.354f, 0.354f, 0.354f, 0.354f, 0.354f, 0.354f },
    // ...
    { 0.098f, -0.278f, 0.416f, -0.490f, 0.490f, -0.416f, 0.278f, -0.098f }
};
```

Test Signal

To test the compression algorithm an artificial signal was generated and used. The signal is given by:

$$x[n] = 5\cos(2n) + \cos(8n - \pi/2) + \mathrm{wgn}(0, 0.1), \quad 0 \leq n < 10$$

where $\mathrm{wgn}(0, 0.1)$ is *white gaussian noise* with 0 mean and 0.1 variance.

The signal was generated using `python` (see `./gen_time_series_signal.py`) and transpiled to `c` code, to easily test different signals.

```python
def gen_test_signal(N: int) -> np.ndarray:
    noise = np.random.normal(0, 0.1, N)
    signal = (
        5 * np.cos(np.linspace(0, 10, N) * 2)
        + np.cos(np.linspace(0, 10, N) * 8 - np.pi / 2)
        + noise
    )
    return signal
```

This signal was chosen as it is periodic, contains more than one frequency, and contains and small amounts of white noise. All of which are characteristics found in signals that are of interest for real applications.

Evaluating the Reconstruction Accuracy of the Decompression Process

For each experiment, the *mean squared error* is used as a performance metric to evaluate how close the decompressed signal is to the original signal.

$$\mathrm{mse}(\mathbf{x}, \mathbf{y}') = (y' - x)^2/N$$

where N is the number of data points in the signal.

Measuring Execution Time

For each experiment, the time it takes to run the `dct()` algorithm is measured. The `c` code below shows how the time interval is computed, using the `contiki-ng` library.

```c
#include "contiki.h"
// ...

clock_init(); // init timer lib

clock_time_t t_start, t_end, dt_compression;

t_start = clock_time();
dct(test_signal);
t_end = clock_time();
dt_compression = t_end - t_start;

// ...

LOG_INFO("DT in clock ticks: %ld\n", dt_compression);
LOG_INFO("arch platforms CLOCK_SECOND: %ld\n", CLOCK_SECOND);
LOG_INFO("DT in seconds: %ld\n", dt_compression / CLOCK_SECOND);
```

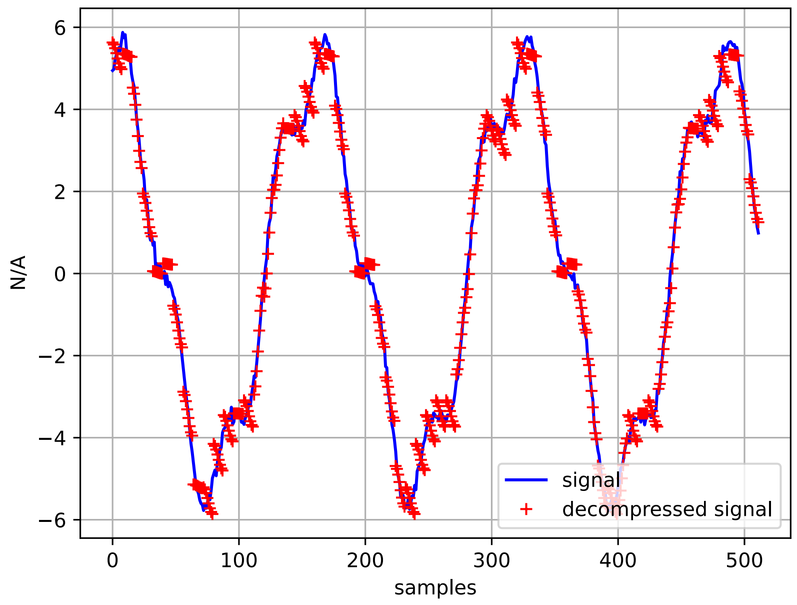
`CLOCK_SECOND` is an architecture specific macro that expands to the number of *ticks* that correspond to a second.

Experiments and Results

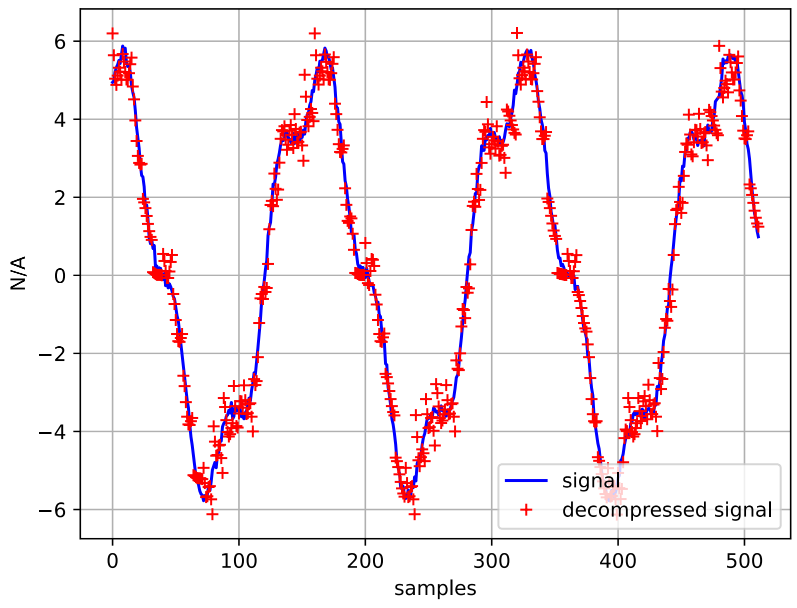
Different combinations of the parameters N, L, M have been tested. For each combination, the `./dct.c` code has been compiled and uploaded onto the Telos B Mote where it runs and decompresses the test signal. The decompressed signal is outputted to the `stdout` stream captured by the console. The compressed data is then decompressed with the `./idct.py` script on the authors host computer. The original and reconstructed signalis then plotted on a graph together with the MSE score for each.

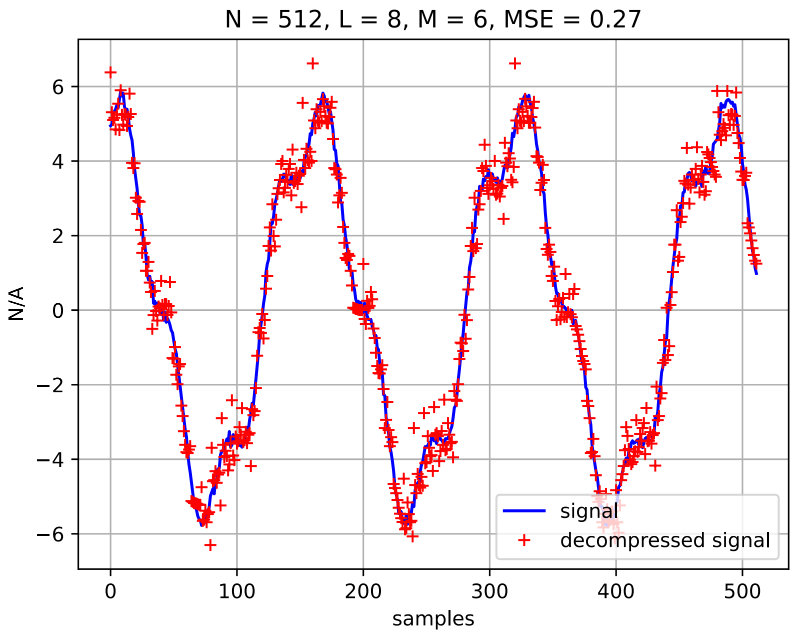
For $L = 16$ the compiled binary became to big to fit into the avialable segment of the Telos B Mote's ROM, resulting in a compiler error. To alleviate this the test signal was cut in half $512/2 = 256$ for when $L = 16$.

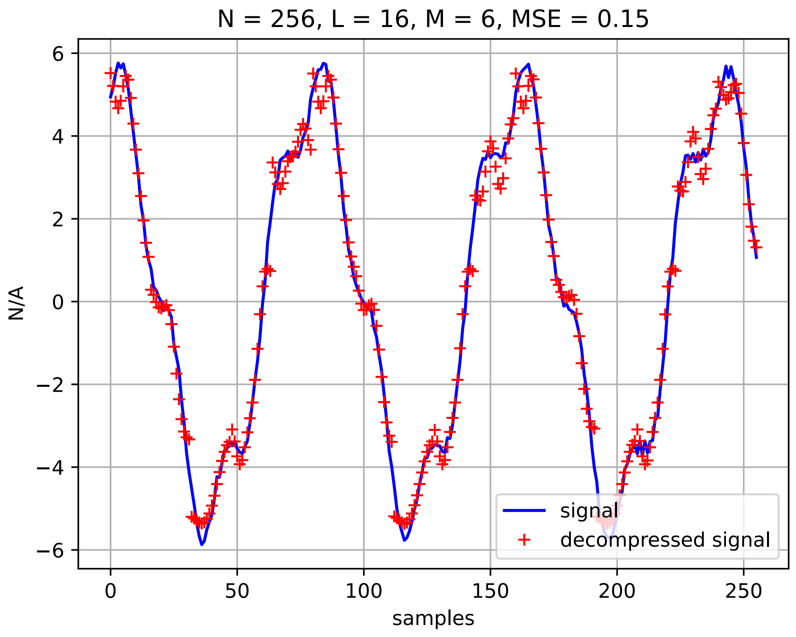
N = 512, L = 8, M = 2, MSE = 0.13

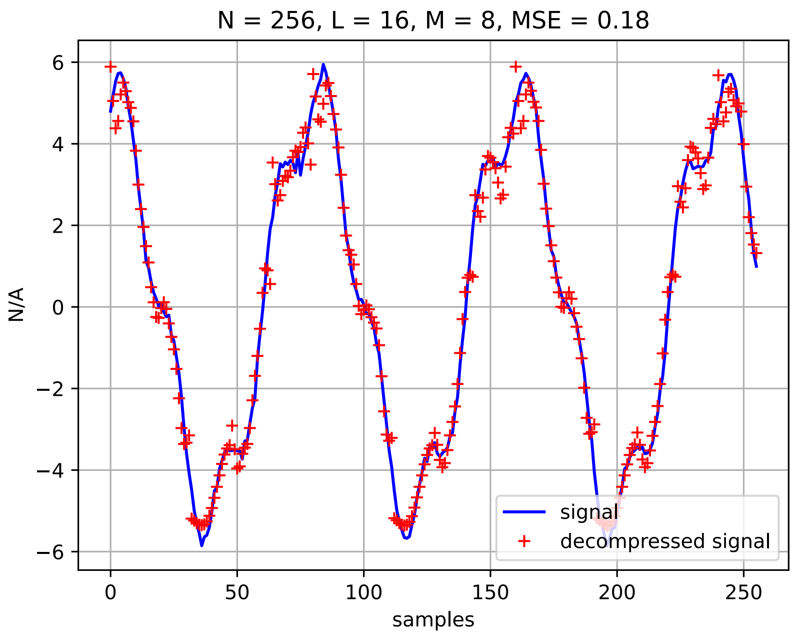


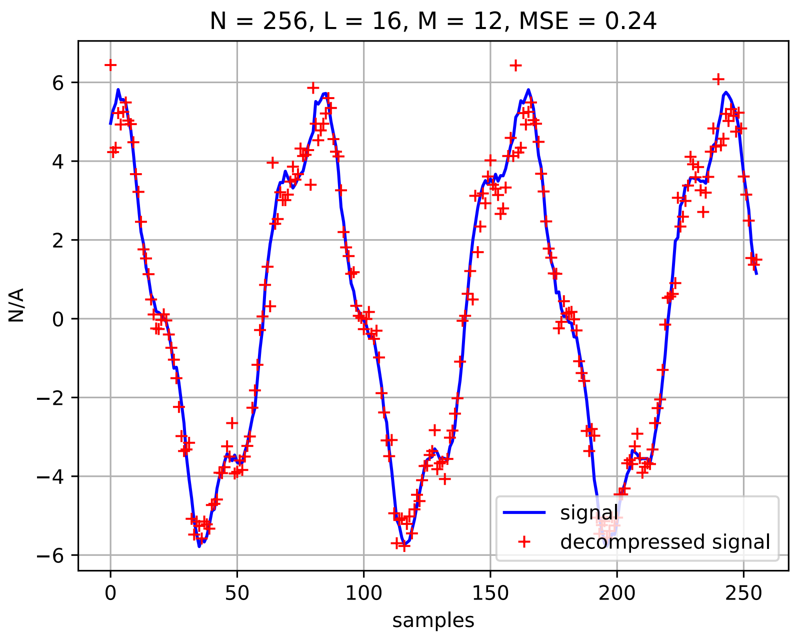
N = 512, L = 8, M = 4, MSE = 0.20




N = 512, L = 8, M = 6, MSE = 0.27
N/A
samples
signal
decompressed signal
6
4
2
0
−2
−4
−6
0
100
200
300
400
500


N = 256, L = 16, M = 6, MSE = 0.15
N/A
samples
signal
decompressed signal
6
4
2
0
−2
−4
−6
0
50
100
150
200
250





Time Measurements

N	L	M	dt in arch platform ticks	dt in seconds
256	16	12	307	2.398
256	16	6	307	2.398
256	16	8	307	2.398
512	8	2	308	2.406
512	8	4	309	2.414
512	8	6	309	2.414

Conclusion

The achieved results does not match well with the behavior expected based on the theory presented. For both $L = 8$ and $L = 16$ the observed behavior is that a lower M result in a lower reconstruction error. But reducing M would increase the compression ratio L/M, which should result in a worse reconstruction as fewer coefficients are kept. This might be due to errors in the implementation, or the choice of test data. The test signal only has two significant components, which can explain why a low M like $M = 2$ can give a good reconstruction as only the coefficients of the two most significant frequencies are necessary to construct the signal. For all test configurations it actually seems like a higher M causes more noise in the reconstruction which could be due to more coefficients of the noise components are kept. Testing with other signals would help with determining if the observed results are general for the Discrete Cosine Transform.

Regarding the time measurements they are all very close to being the same. This might indicate that the computational work required for the different combinations of the parameters are insignificant. But I suspect it might have to with the lack of granularity of the `contiki-ng` timer API, as $L = 16$ would result in a H matrix with $16^2 = 256$ and $L = 8$ in a H matrix with $8^2 = 64$ with is 4 timers smaller. This difference should be noticeable. Regardless of the size of H and the selection of M, it seems likeky to presume that precomputing H beforehand is a good tradeoff compared to computing it every time a new signal has to be compressed as each entry in the matrix requires a call to the `cos()` and `sqrt()` routines, both of which are costly in terms of CPU cycles, compared to addition and multiplication instructions. Even more so on a resource constrained environment like the Telos B Mote where it is highly desirable to reduce the amount of computation required to save on energy.

Intructions for how to replicate results

Build binary and upload to Telos B Mote:

```
make TARGET=sky MOTES=/dev/ttyUSB0 dct.upload login
```

The `./Makefile` assumes that the repository is placed in a subdirectory of your `contiki-ng` installation. e.g. `contiki-ng/<subdir>/<repo>`

Clean build:

```
make distclean
```

Generate plots:

```
# the plots will ge generated in folder of each respective
# experiment e.g. ./experiments/N511-L8-M2/plot.svg
./gen_plots_for_every_experiment.py ./experiments
```

Generate report:

```
pandoc ./report.md -o ./report.pdf
```