Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Benchmarking Estimators of Harvestable Energy on a Low-Power System

Semester Thesis

Francisco Andrés Dumont Duvauchelle
fdumontd@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Stefan Draskovic
Reto Da Forno
Prof. Dr. Lothar Thiele

July 11, 2020

Acknowledgements

I am very thankful to my supervisors, Stefan Draskovic and Reto Da Forno. They helped me a lot into organizing things despite the situation with the corona virus. Part of the hardware needed to be set correctly in order to work from home and they helped me a lot with this issue. Both of them also gave feedback every time possible and the success of this semester project is certainly thanks to their active involvement.

At one point of the project I encountered difficulties in setting the RocketLogger properly. Stefan Draskovic contacted Lukas Sigrist, whom is no longer studying nor working in ETH but was willing to help me out, because he was the one involved in the RocketLogger development. My thanks to Lukas Sigrist for his help.

I also want to thank the ETH computer and electric engineering department. They provided all things needed for me to work. This includes a laptop, because I did not have one myself.

Abstract

The report is a about the construction of a benchmark to evaluate different algorithms used to predict the availability of energy in the near future. These algorithms are meant to be used on microcontrollers, in this specific case the DPP DevBoard with the microcontroller MSP432P401R.

The report goes though different chapters explaining in general terms the benchmark implementation. It starts with some context for the project motivation, followed by a design overview of the setup and algorithms. Then the different implementation levels are described, more specific information can be found in the annexes. There is also an evaluation chapter, which goes through the functionality results of each one of the algorithms and also makes a comparison between them. Lastly, missing things and possible improvement in the project are mentioned.

At the end there are annexes, but the purpose of them is to explain in details about the installation, preparation and implementation. It is highly recommended to read them if you are going to use or intent to expand the functionally of this tool.

Contents

Chapter 1

Introduction

1.1 Context and background

In this day and age, embedded systems are everywhere. All our electronics have them and we find new ways to use them in order to solve problems. Though regardless of the implementation, they all use energy to function. Some get their energy from batteries or are connected directly to the grid. But since energy harvesting of unconventional sources has experienced a lot of research, there is a growing interest in using them as well. Among the unconventional sources solar, wind and vibrations can be highlighted. The problem with these sources is how variable and unreliable they are.It can be said, all days are created equally important, but some have more wind or sunlight than others.

This feature of unconventional energy sources presents a trade off. If you want to be sure the system will have always energy available to perform its functionality, you could have a huge battery or a large solar panel connected. This solution despite possible is not practical in many cases. Thus when there is not a big harvester present in the system, it is advantageous to have an energy harvesting prediction algorithm helping in the scheduling decisions of the embedded system.

To start any approach to the problem at hand, we need to look into some of the theoretical available solutions. Several papers have been written about possible algorithms for estimators. But regardless of how good the calculations on paper are, at some point they have to be tested and evaluated in the real world. And here is where this project attempts to make your live easier by developing a useful tool.

1.2 Goal of the project

The goal of this project is the development of a benchmark and later test it by evaluating different energy harvesting prediction algorithms for low power systems. This benchmark includes a functional evaluation of high-level simulation

in the computer (python) and a low-level implementation for the microcontroller TI MSP432P401R. Ultimately, by using this tool, the reader will be able to implement his own algorithms.

1.3 Mathematical frame and algorithms

Three algorithms were implemented in this project, to test the benchmark. All the algorithms are well known in the state of art solutions and are described in the following chapters. Many of these definitions are later used in the coding of the implementation.

1.4 Remarks

Throughout the report and annexes many aspects are mentioned and due to the different implementation levels, it is easy to lose track of what is being discussed. In order to facilitate the understanding, the report uses different colors to highlight different aspects.

Only some colors appear first in the report and others are added in the annexes. In the annex A these additional colors are introduced. The colors being used in the report are the following:

- color `violet` relates to anything to do with the low-level implementation. For the report, the evaluation section 4 uses this color for variable names in the low-level implementation code.
- color `blue` is for database files or directories
- color `cyan` highlights things to keep in mind. For example a value, software names or command lines.

Chapter 2

Design overview

2.1 Hardware

Our setup uses several components, and they are described in this chapter. The general design can be seen in the figure 2.1, which is explained below. The algorithms are implemented in two different parts.

First, a low-level implementation of the prediction algorithm is to be implemented on an embedded system, in this case the DPP DevBoard [1]. The DPP DevBoard here receives data by a UART port, executes the algorithm and sends the prediction back through the same channel.

On the other hand, the high-level implementation is done in a general purpose computer. Here the user inputs both a high-level and a low-level implementation of the estimation algorithm, and gives the measured harvested energy data set. Then the benchmark does several steps automatically. The chronological steps are the following:

- Prepare the different resources (compilation / activation)
- Sending data to or receiving data form DPP DevBoard
- Start / Stop the power measurement device (RocketLogger) [2]
- Simulate the algorithm on the computer
- Saves the data for future analysis

All of these processes are addressed in more detail in the chapter 3 and annex C. Here, general information about some of the specific hardware being used is given.

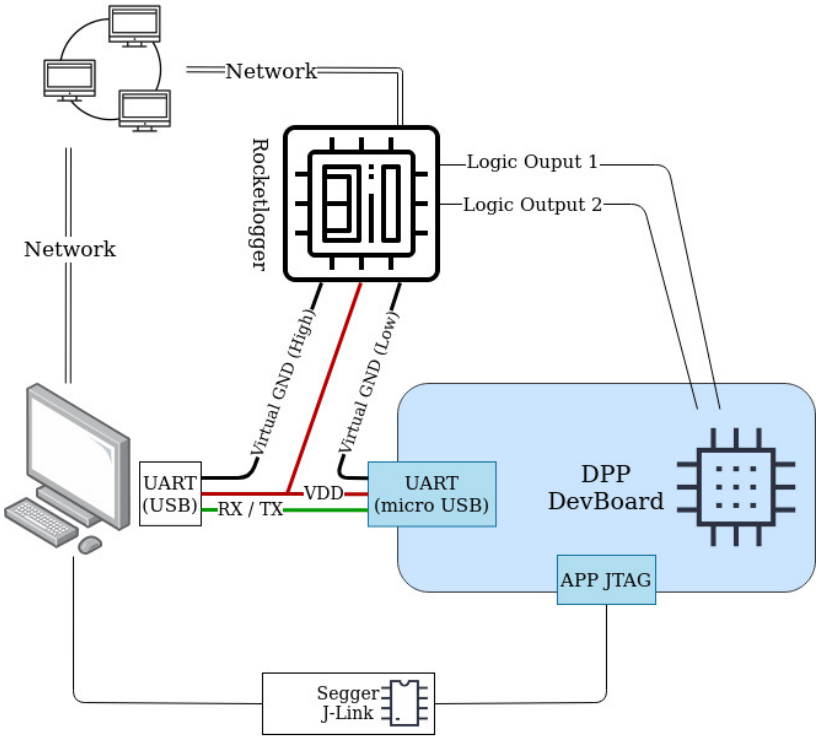

Figure 2.1: setup overview

RocketLogger: The RocketLogger is a device developed at the Computer Engineering Group at ETH Zurich to measure voltage and current with high precision [3]. It is important that the RocketLogger is connected to the local network, because that way the device can be controlled through a command line interface and results can be retrieved via SSH.

DPP DevBoard: The DPP DevBoard is a hardware device developed at the Computer Engineering and Networks Laboratory (TIK) of ETH with a microcontroller MSP432P401R [1]. Among the most important features being used are the UART communication port and two logical outputs for the RocketLogger measurements. In order to flash the MSP432P401R, the debugger hardware Segger J-Link EDU Mini [4] is used.

Computer: The computer used is Linux Ubuntu 20.04. Most of the high-level implementation is done in Python 3, Python 3.6.9 to be specific. Also the corresponding .C files for the microcontroller MSP432P401R are compiled into one BIN file the Code Composer Studio 9.3.0 software.

2.2 Energy harvesting prediction algorithms

2.2.1 Exponentially Weighted Moving-Average (EWMA)

The algorithm EWMA is discussed in the paper *"Power Management in Energy Harvesting Sensor Networks"* [5]. The core principle is to exponentially average the incoming measurement with a historical saved value. The algorithm can be described in the following way:

$$\alpha \in [0, 1]$$

$$i \in \mathbb{N}, i \in [1, slots]$$

$$x_h(i) \text{ historical value from slot i}$$

$$x(i) \text{ measurement from slot i}$$

$$x_h(i) = \alpha \times x_h(i-1) + (1-\alpha) \times x(i)$$

2.2.2 Weather-Conditioned moving Average (WCMA)

WCMA was published in the paper *"Prediction and Management in Energy Harvested Wireless Sensor Nodes"* [6]. In this case, the prediction involves more parameters than the previous algorithm. The parameter included in this case is called GAP_k, which measures the energy conditions of the present day relative to the previous days. Some definitions are needed before describing the GAP_k factor.

Matrix E of size DxN is used to save the recollected energy measurements of the last D days, where we have:

$$D \text{ number of days saved}, N \text{ day slots}$$

$$E(j, i) \text{ energy stored of slot j form day i}$$

$$j \in \mathbb{N}, j \in [1, N]$$

$$i \in \mathbb{N}, i \in [1, D]$$

Vector M_D is the average of the last D days for each slot:

$$M_D(d, n) = \frac{\sum_{l=d-1}^{d-D} E(l, n)}{D}$$

Vectors V and P have K elements each. On one hand, the elements of the vector V represent the quotient between a sample and the mean of previous D days in

that slot. On the other hand, in vector P the elements will be weighted values. With these concepts in mind, the GAP_K can be declared.

$$V = [v_1, v_2, v_3, ...v_K]$$

$$\text{with } k \in [1, K] \text{ and } v_k = \frac{E(d, n-K+k-1)}{M_D(d, n-K+k-1)}$$

$$P = [p_1, p_2, p_3, ...p_K]$$

$$\text{with } k \in [1, K] \text{ and } p_k = \frac{k}{K}$$

$$GAP_k = \frac{V \times P}{\sum P}$$

The final prediction with all the parameters is the following:

$$E(d, n+1) = \alpha \times E(d, n) + (1-\alpha) \times GAP \times M_D(d, n+1)$$

2.2.3 Pro-Energy

In the paper *"Pro-Energy: a novel energy prediction model for solar and wind energy-harvesting wireless sensor network"* [7] the algorithm Pro-Energy was introduced. The paper described multiple options, but in this semester project only two variants are implemented.

Pro-Energy is an algorithm, which saves energy profiles and combines this information with the incoming measurement to make a prediction. These profiles are saved in a E matrix of size NxD, which represent D different profiles. Among these profiles one is selected and combined with the parameter α:

D days profiles saved, N day slots

E_i^d energy stored in slot i for profile d

$$j \in \mathbb{N}, j \in [1, N]$$

$$i \in \mathbb{N}, i \in [1, D]$$

K number of previous observations in profile to be considered

C_i measurement in slot i of the current day

$$E^d = \min_{E^d \in E} \sum_{i=t-K}^{t} \frac{|C_i - E_i^d|}{K}$$

$$\hat{E}_{t+1} = \alpha \times C_t + (1-\alpha) \times E_{t+1}^d$$

Profiles can be updated in order to have the most recent data available to make predictions. This feature can be inactive or active. In case of inactivity, the profiles have to be calculated and defined in advanced, the criteria used for this will depend on the developer. On the other hand, if the feature is active the profiles are updated according to two options.

The first option is to replace the profile that is older than A days of the current day. The second option needs to calculate the mean absolute error (MAE) value of each saved profile with the current day's profile. Once the MAE is calculated, the one with the lowest (MAE) is compared to a threshold. If that threshold is passed, that selected profile is replaced with the current day.

Chapter 3

Implementation

3.1 High-level implementation

As mentioned before, the high-level implementation is ran in a computer and does several processes. In this chapter the implementation will be discussed in general terms, but a detailed information about them can be found in the Annex C. It is recommended to look the Annex, because that information allows the user to be able to adapt the code and include new desired algorithms.

The high-level implementation can be broadly separated into two different stages. The first stage involves what the user has to give as inputs before running the scripts and the second stage is the automated process coded in the different scripts.

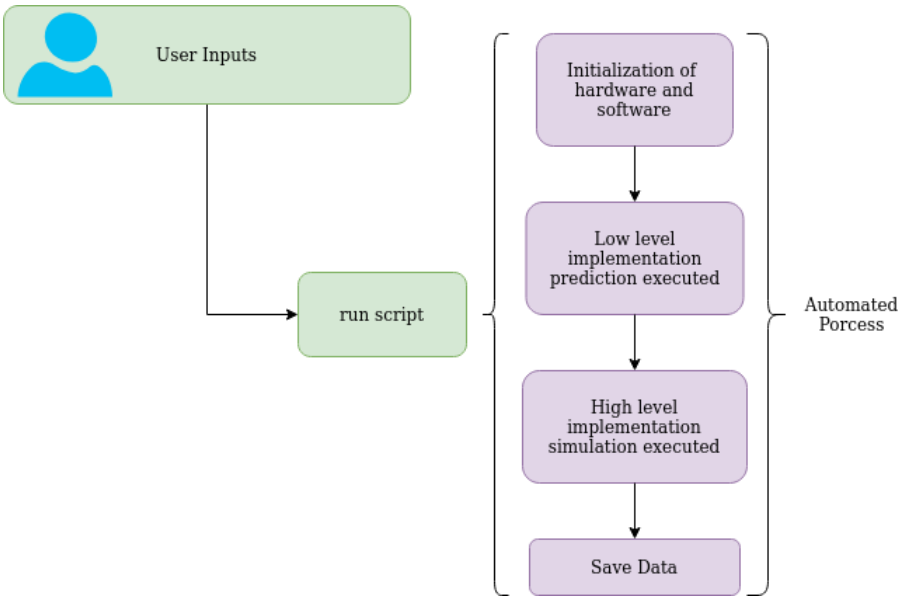

Figure 3.1: general high-level implementation

3.1.1 User actions

There is no user interface, thus the has to write the inputs for the benchmark in the scripts themselves. The inputs written in the scripts are related to four mayor aspects: algorithms parameters, database selection, RocketLogger configuration and number of days to simulate. The annex in section C.4 has full descriptions about each one of the inputs. Once the inputs are given, the script just has to be ran.

3.1.2 Automated process

The automated process starts when the user runs the scripts. Detailed information can be found in the Annex C, but for the purpose of this section the processes will be generalised. The figure 3.1 highlights four process and the comments will be done on them.

Initialization

This first step of the processes is setting the all the hardware and software correctly. Here below is a list of the most important settings that need to be done.

- Database selection: a database for sunlight and another for wind measurements can be selected.
- Algorithm and its parameters selection: choose between EWMA, WCMA and ProEnergy. If a new algorithms is implemented, several changes need to be made in order for the benchmark to do the process automatically. Otherwise changes in the files need to be done manually.
- Edit and compile files of low-level implementation: the files for the low-level implementation are also edited in this phase, but again if the algorithm is not included in the high-level implementation, the user needs to change everything manually.
- Flash the DPP DevBoard: tools from SEGGER are used for this purpose.
- Set RocketLogger parameters: as the RocketLogger is connected to the local network, the setup is able to set the device correctly for the measurement on the DPP DevBoard.
- Send to DPP DevBoard the initial array / matrix values: the DPP DevBaord needs the initial historical values and the high-level implementation provides them through the UART Channel.

Low-level implementation execution

At this point the prediction calculation of low-level implementation can take place. First, the high-level implementation sends a start command to the RocketLogger. Immediately after that data is send to the DPP DevBoard and the computer waits for the prediction value. This will go on until the last day of the simulation is finished. At that point the RocketLogger is stopped by another command and the process can continue with the next step.

High-Level implementation simulation

The high-level simulation is ran. In this case only the computer is involved, thus no coordination with external hardware is needed.

Save data

This is the last step of the processes. The data from the high-level simulation, low-level prediction and RocketLogger measurements are saved in a predefined folder for further analysis.

3.2 Low-level implementation

The target of the low-level implementation is the DPP DevBoard. It was already mentioned that the editing of the files happens in the high-level implementation process, unless the algorithm is not among the ones recognized by the benchmark. In this section two aspects will be commented: the file structure and the control flow on the DPP DevBoard. Further information can be found in the Annex D, it is highly recommended to take a look at it.

3.2.1 File structure

The low-level implementation is separated in two types of files. The first of them are the common files, which have general variables and the control flow of the implementation. These files are used by every algorithm and call the respective functions from the specific algorithm files depending on the stage of the control flow. The second group of files are algorithm specific, thus their activation depends on the user inputs at the high-level implementation.

3.2.2 Control flow

Two different processes can be identified in the control flow. The figure 3.2 represents these two generalised decisions to choose from.

The process starts by always waiting for data to be received from the computer. Once something has been received the DPP DevBoard decides what process to perform. If the DPP DevBoard has just recently being flashed, it has no historical values in the memory array. So the first action taken is copying the data received via UART to the memory. Once this is done, the DPP DevBoard performs the second action, which is making a prediction with the received data. The DPP DevBoard stays in this configuration even after the last day of the simulation is through.

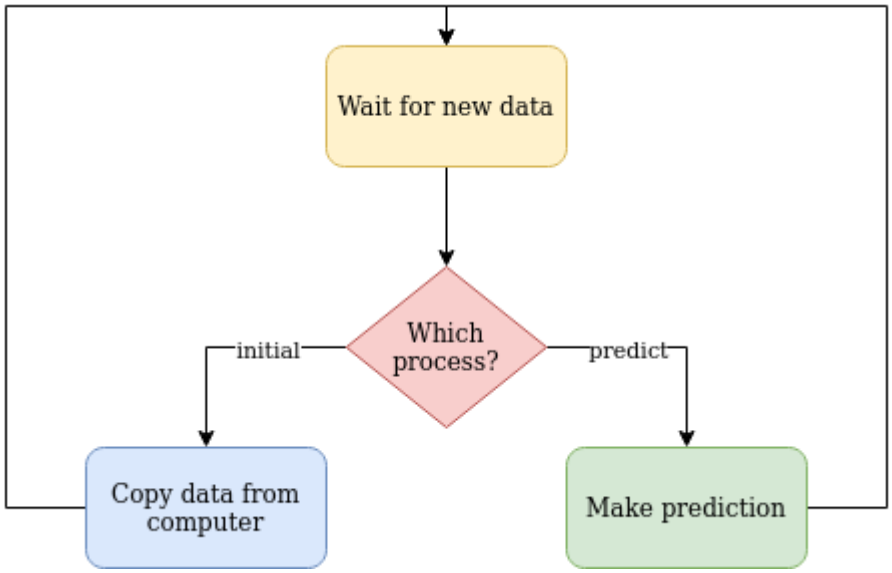

Figure 3.2: general control flow for low-level implementation

Chapter 4

Evaluation

The benchmark makes it easy to plot graphs describing the performance of any algorithm under test, which can help in their development and understanding.

There are some python scripts and code that output the following graphs. Three different aspects are shown by these representations. The first one enables the user to compare the high-level and low-level implementation as an easy check on the low-level functionality. The next one is information about about the DPP DevBoard predictions and measurements of the RocketLogger. Lastly, a comparison between all the current algorithms is done.

The user is encouraged to look into the code of the graphs if he wants to add his own representations.

4.1 Algorithm individual performance

In this section four graphs are shown for every algorithm. The parameters used for each one of this predictions are given, but as the graphs are pretty much the same, some explanations are given on each of them. It is important to note that the sections shown in the figures are selected from a 30 day simulation (usually among the best examples).

The first graph (figures 4.1, 4.5, 4.9 and 4.13) shows a comparison between the high-level simulation prediction, DPP DevBoard prediction and the actual measurement from the database. Below it, the percentage and absolute error between the predictions and the measurement are displayed.

The following graphs (figures 4.2, 4.6, 4.10 and 4.14) presents three things. The x axis is the time measured by the machine computer or the RocketLogger. In the upper part the DPP DevBoard prediction values and the time they arrive at the computer are plotted. Right below it, is a sub-graph that plots the time execution for the prediction calculation in the DPP DevBoard and the time measured by the RocketLogger. Finally the lowest one is the power consumption of

the DPP DevBoard through time. It is important to note that the time measurements of the computer and the RocketLogger might have a shift, that is the reason for using the `PORT 5 PIN 1` as an indicator of where exactly does the whole prediction process starts and ends. More information about the ports used in the DPP DevBoard can be found in the Annex D.3

The third graph (figures 4.3, 4.7, 4.11 and 4.15) considers more data than the previous two. As said before these results are from a 30 day simulation, what these figures consider are those 30 days results. The upper and middle graph have the same y-axis (number of prediction executions in the DPP DevBoard), but the x-axis is different. One is the time used for a prediction execution in the DPP DevBoard and the other is the energy consumption in the time interval of the prediction execution in the DPP DevBoard. The graph in the lower part combines the energy consumed and time execution of the prediction calculation, the expected thing is to see the points on a common line.

Finally the last figure (figure 4.4, 4.8, 4.12 and 4.16) puts the prediction calculation with its execution time / energy consumption, instance by instance.

4.1.1 EWMA

The only parameter set in this example was `ALPHA`, with a value of `0.5`. Some comments can be made. From figure 4.1 there is a clear shift between the predictions and measurements, and that is due to the fact that the prediction is made with the previous historical measurement and the current received measurements. In the figure 4.2 and 4.4 it can be seen how the execution time and power consumption is greater during sunlight hours, because the calculation involves more than only zeros.

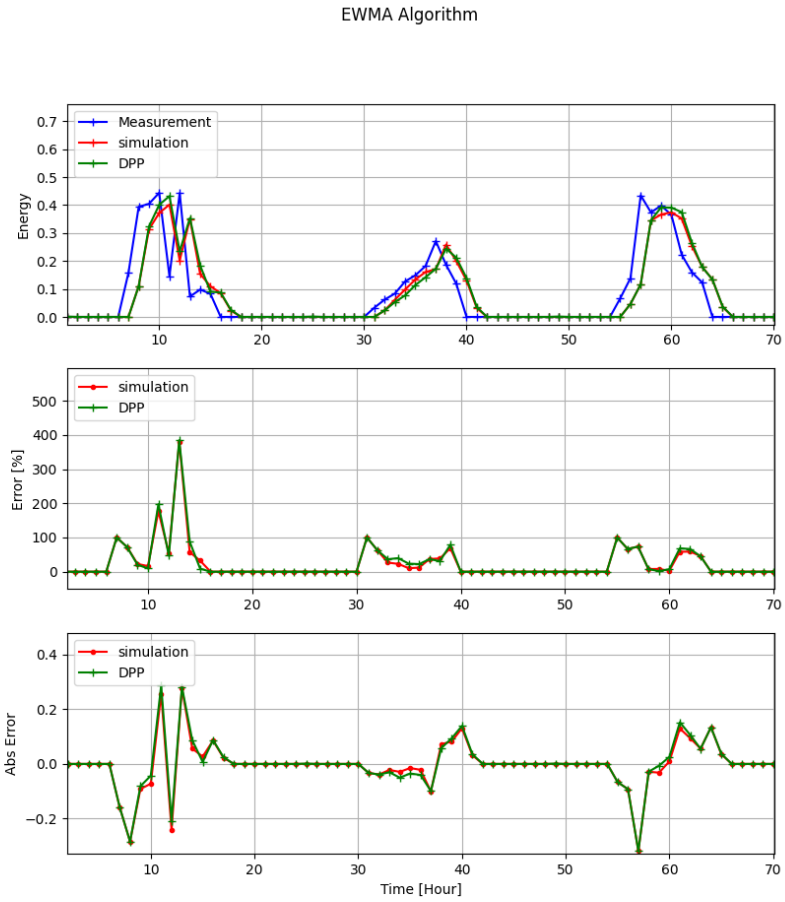

Figure 4.1: EWMA comparison between DPP DevBoard and simulation

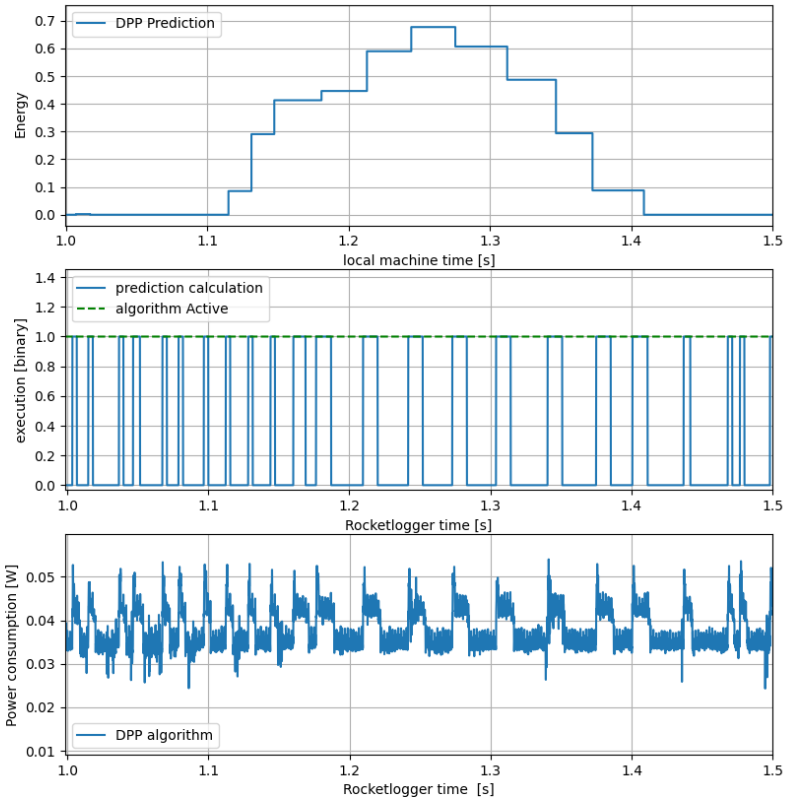

Figure 4.2: EWMA execution time and power consumption of during prediction

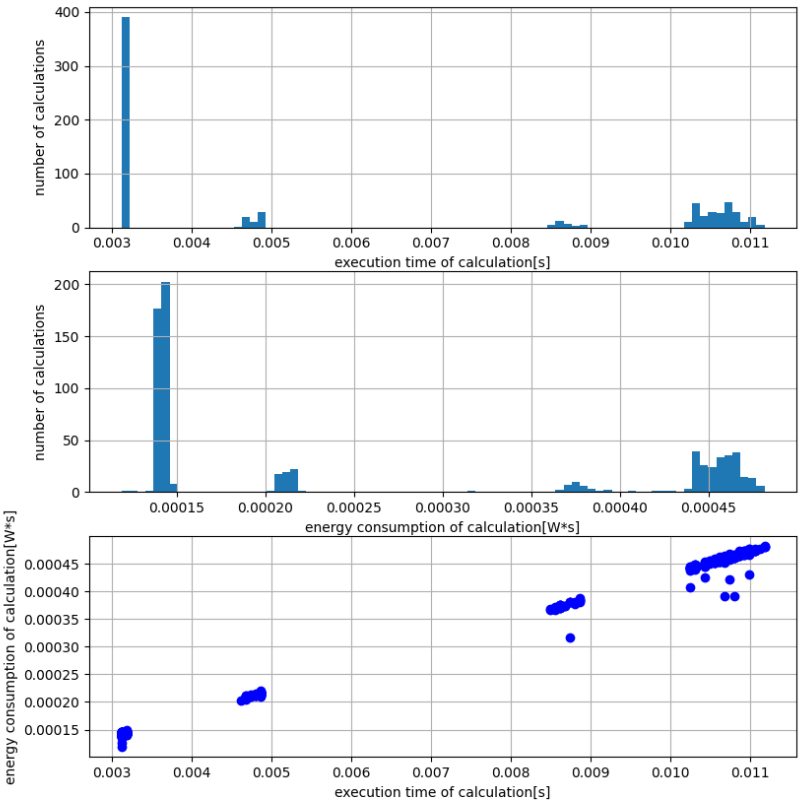

Figure 4.3: EWMA execution time and energy consumption occurrences

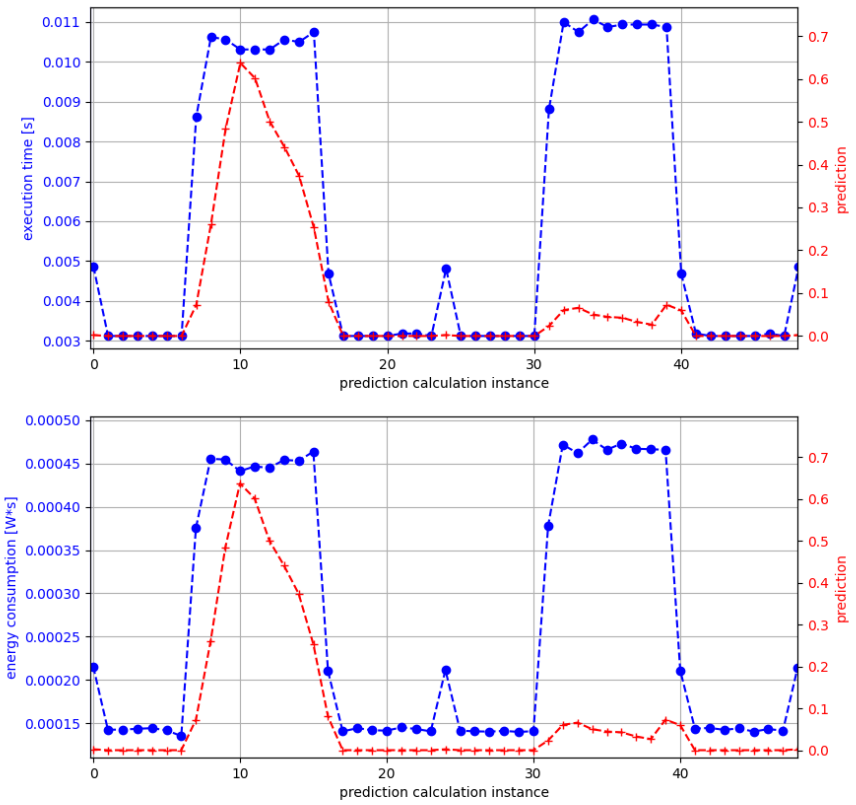

Figure 4.4: EWMA prediction compared to energy consumption or execution time

4.1.2 WCMA

These are the settings for the WCMA used in this case: `ALPHA` = `0.7`, `K` = `6` and `D` = `5`. In this case it can also be observed the shift as in the EWMA algorithm. Regarding the execution time and the energy consumption, the last time slot of the day is the greatest. The reason for it is simple, on the last slot of the day the algorithm has to shift and update the saved measurements in the E matrix.

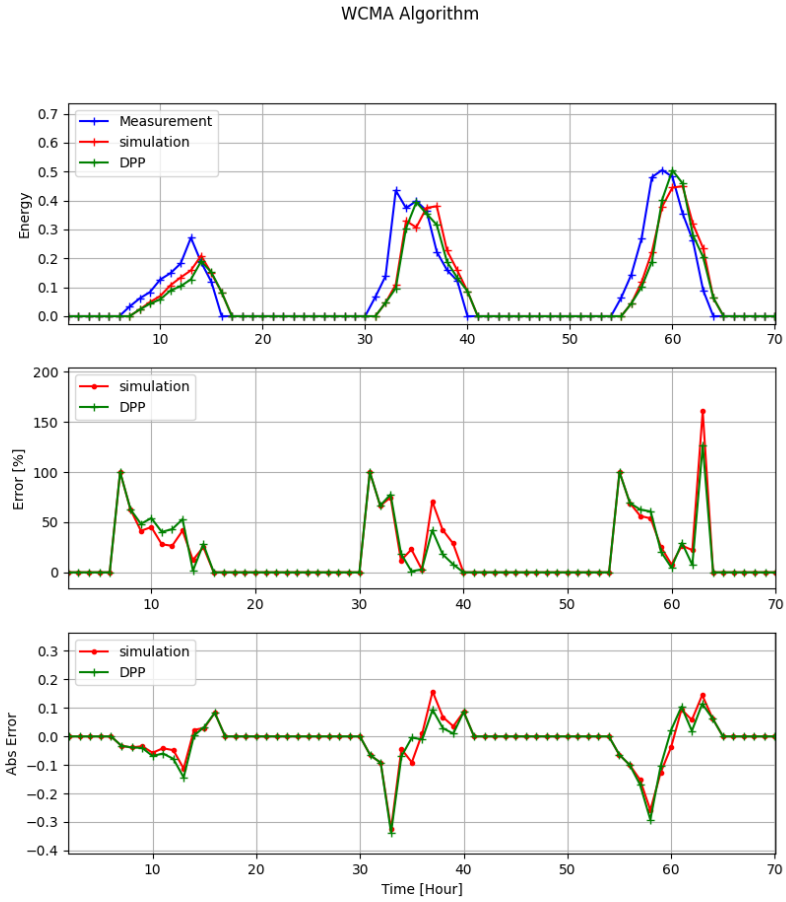

Figure 4.5: WCMA comparison between DPP DevBoard and simulation

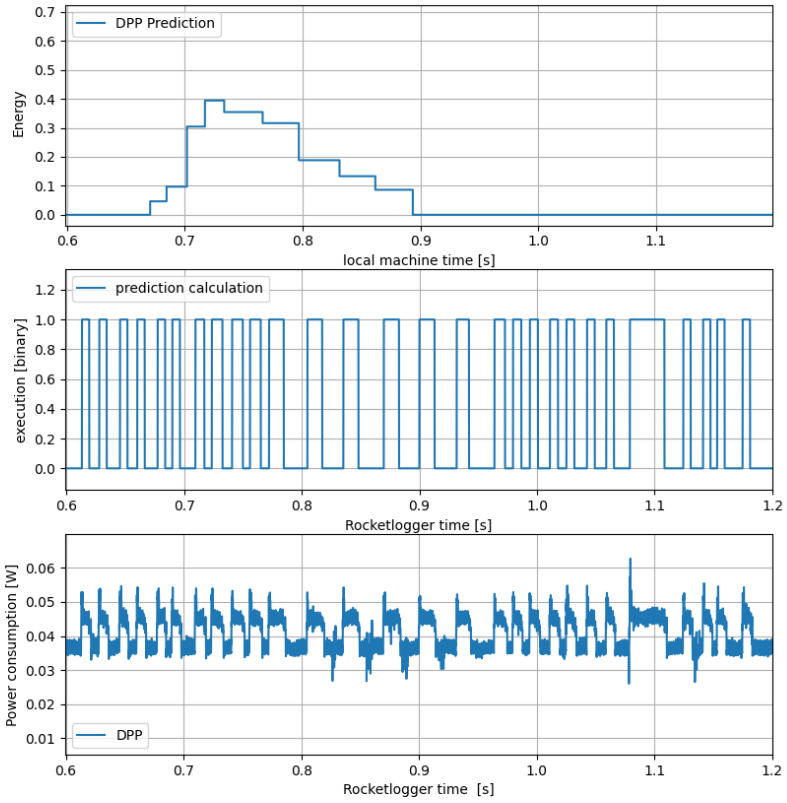

Figure 4.6: WCMA execution time and power consumption of during prediction

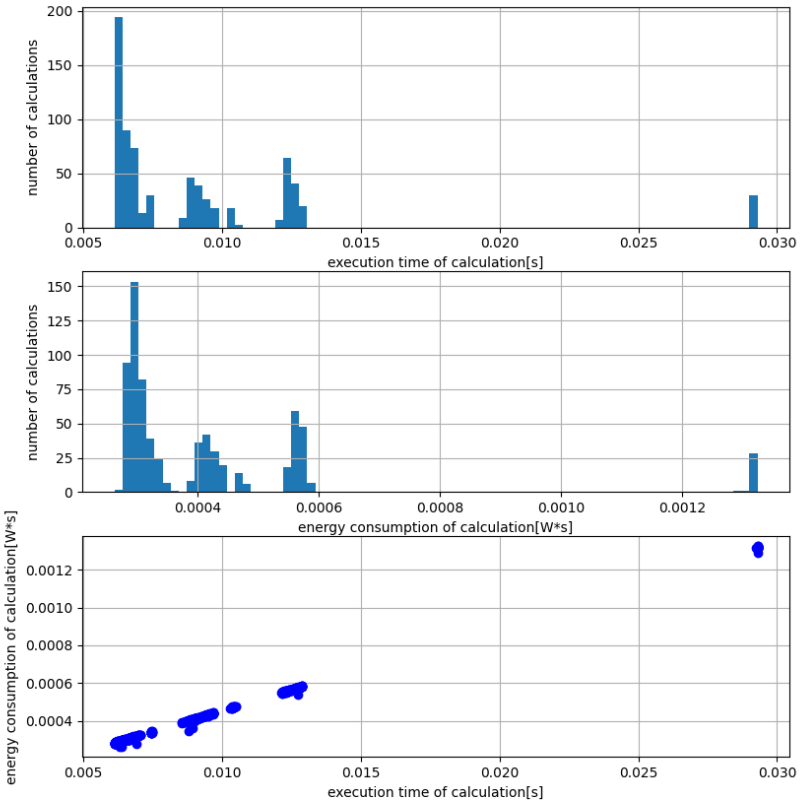

Figure 4.7: WCMA execution time and energy consumption occurrences

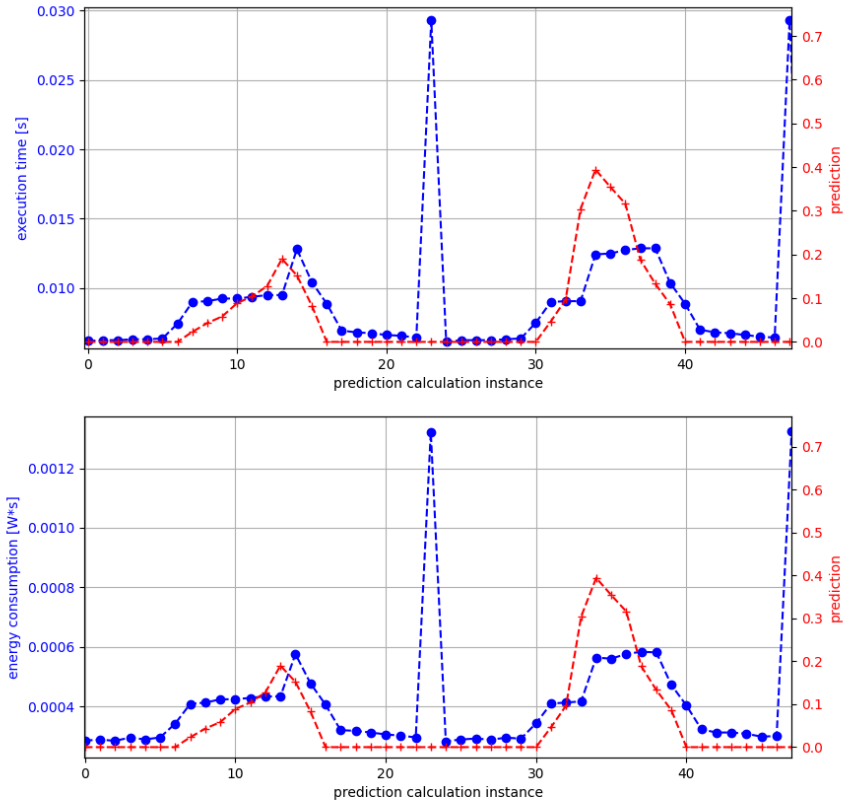

Figure 4.8: WCMA prediction compared to energy consumption or execution time

4.1.3 ProEnergy active update

The active update means that the saved profiles are updated under certain conditions and that is why all the parameters of the algorithm are considered. The setting of them is: `ALPHA` = `0.6`, `K` = `7`, `D` = `10`, `A` = `20` and `MAE_THRESHOLD` = `500`.

In the case of the ProEnergy algorithm, there is no shift present as in the WCMA and EWMA due to what the prediction considers, in this case the profile value of the next slot together with received data. In this cases the greatest energy consumption and time execution is at the end of the day, because it has to update the profiles. Though there is something interesting to point out, which can be seen in the figure 4.12 clearly. The energy consumption and execution time increases till the 8th slot of the day, the explanation for this is the following. In the calculation prediction the last `K` (7 in this case) slots are considered. But before the 8th slot the algorithm can't consider slots of other profiles, thus less than `K` slots will be considered, making the prediction calculation less resource intensive.

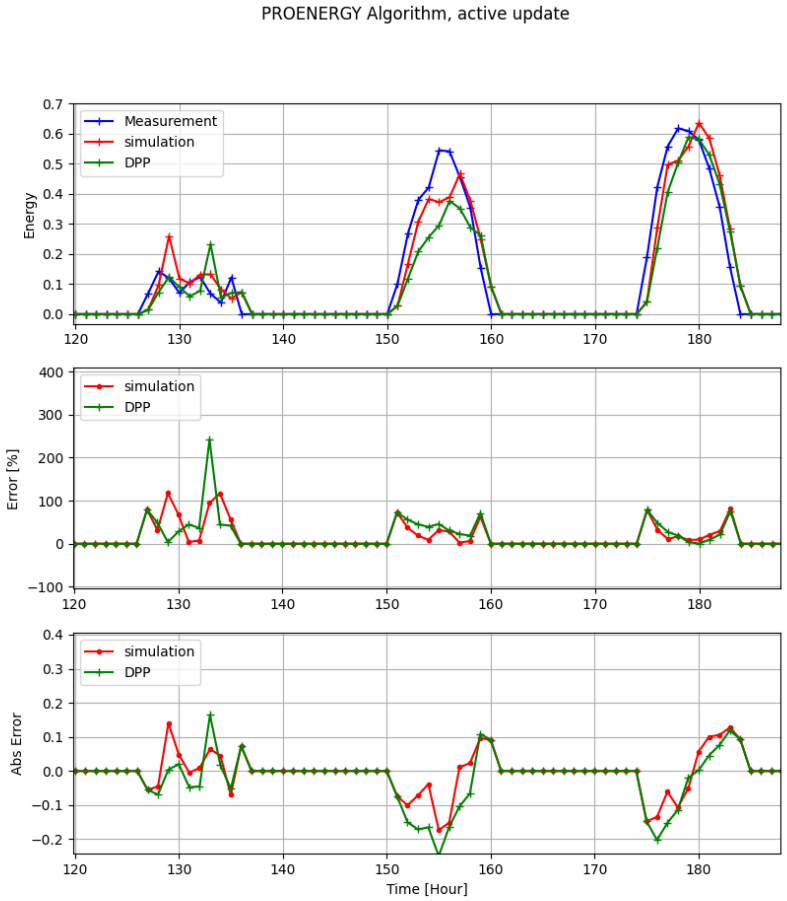

Figure 4.9: ProEnergy active update comparison between DPP DevBoard and simulation

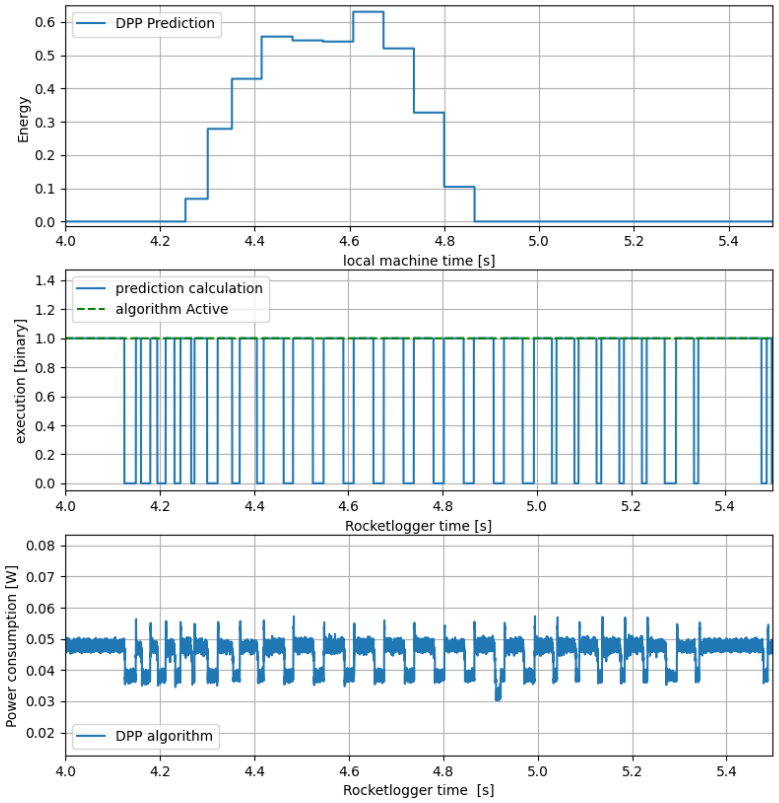

Figure 4.10: ProEnergy execution time and power consumption of during prediction

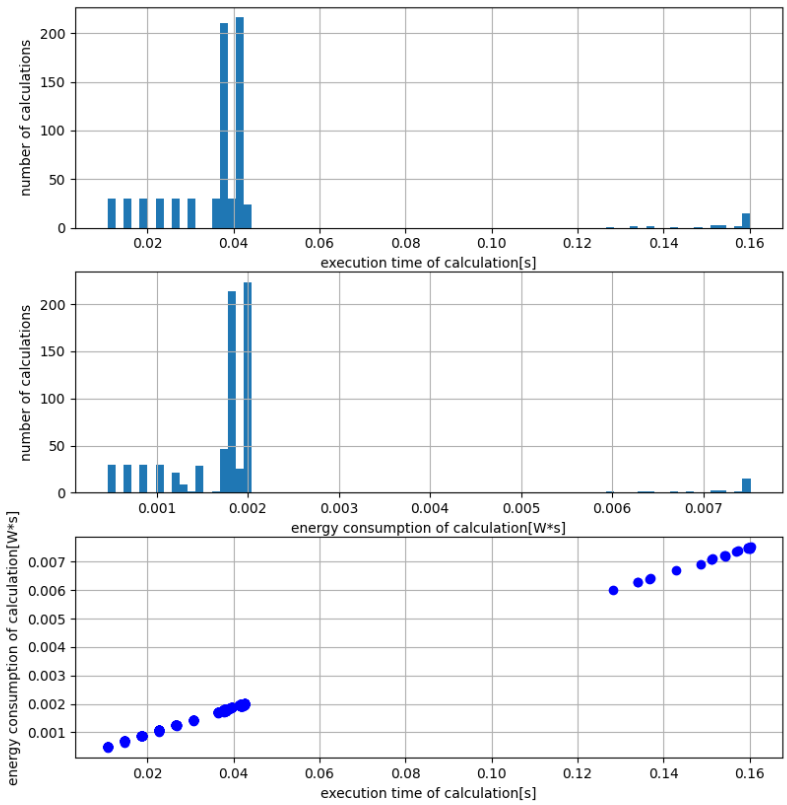

Figure 4.11: ProEnergy active execution time and energy consumption occurrences

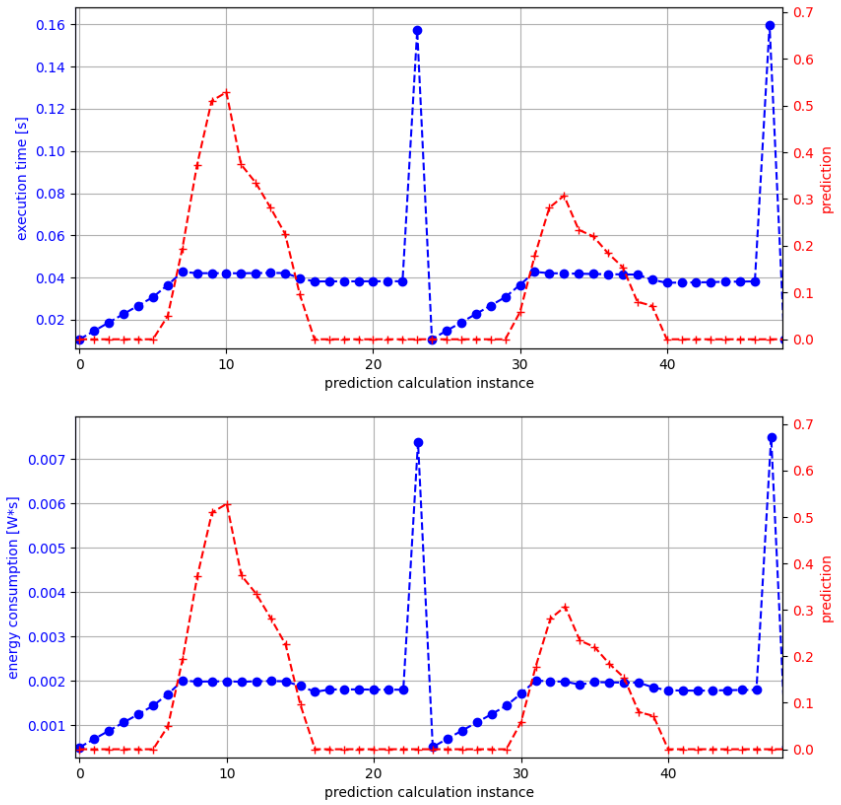

Figure 4.12: ProEnergy active prediction compared to energy consumption or execution time

4.1.4 ProEnergy no active update

This variant of the ProEnergy doesn't update the profiles, instead it takes one full year, selects certain days and creates `D` amount of fix profiles. All other parameter are the same, but many of them won't be considered, like `A` and the `MAE_THRESHOLD`.

At first sight similar conclusions as in the active case can be made, with the exception that this time at the end of the day no big execution time and energy consumption is used.

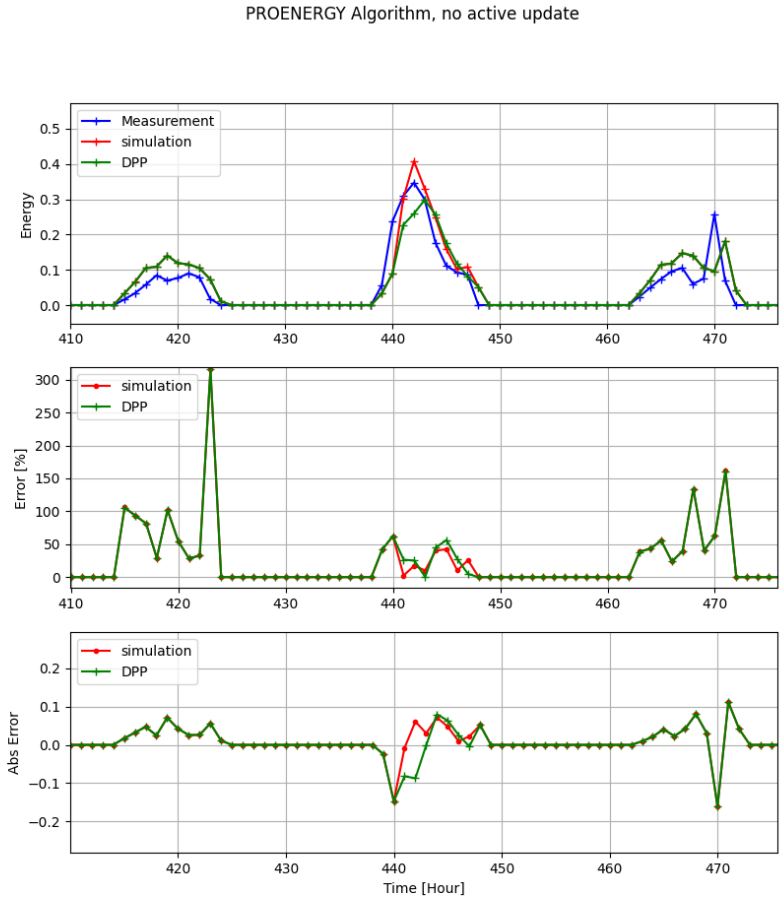

Figure 4.13: ProEnergy no active update comparison between DPP DevBoard and simulation

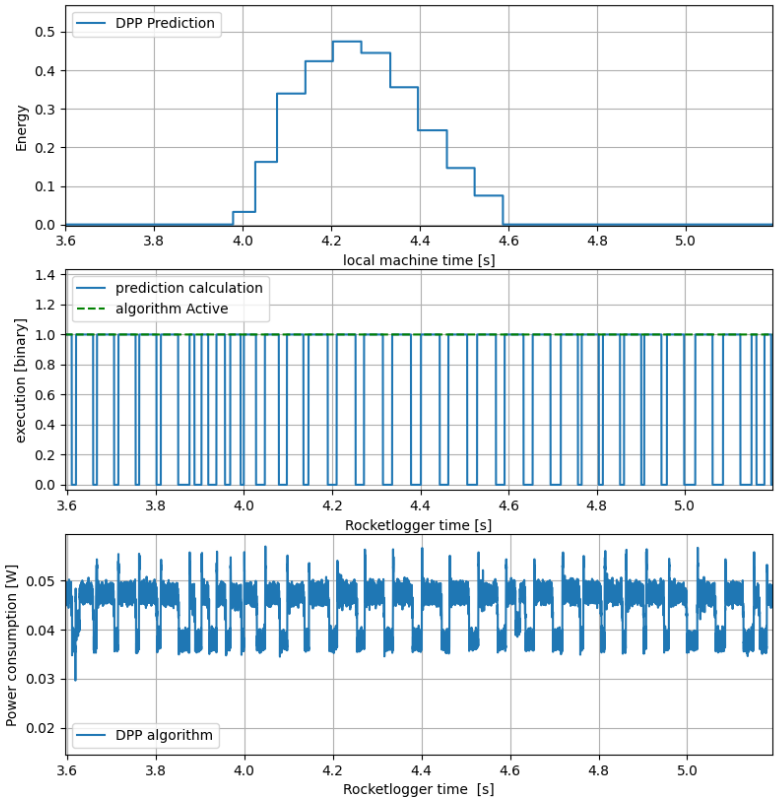

Figure 4.14: ProEnergy no active update execution time and power consumption of during prediction

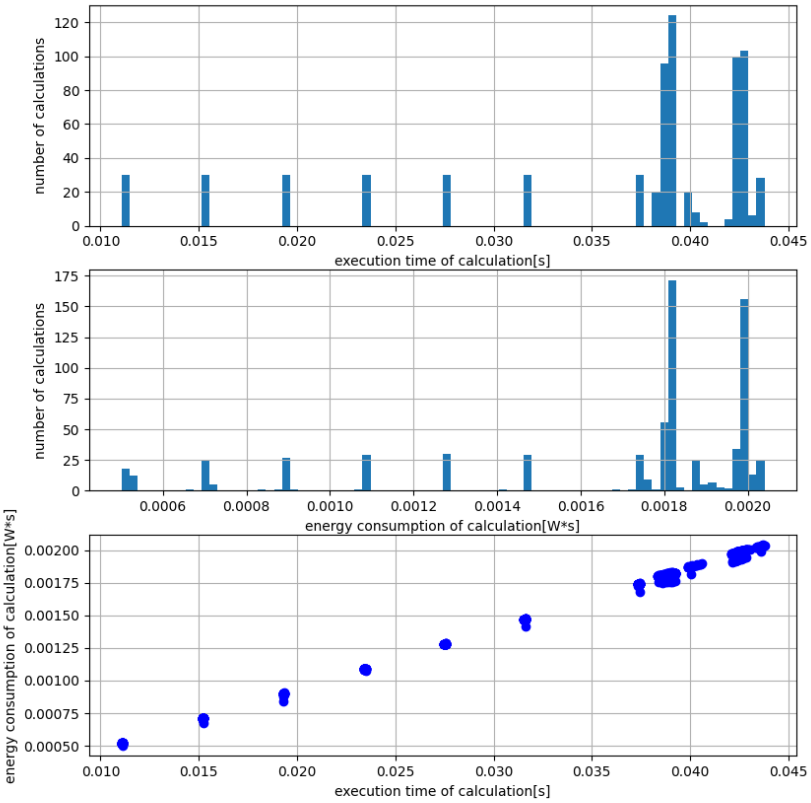

Figure 4.15: ProEnergy no active update execution time and energy consumption occurrences

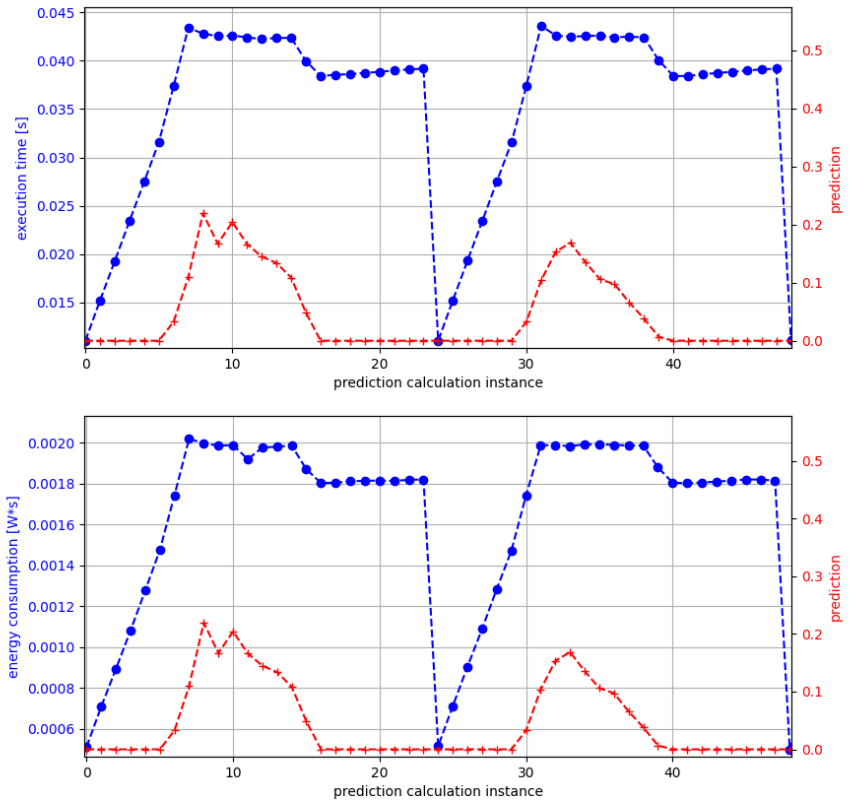

Figure 4.16: ProEnergy no active update prediction compared to energy consumption or execution time

4.2 Algorithms comparisons

This is an attempt to compare these different algorithms by using these 30 day measurements, thus please do not draw definitive conclusions for the algorithm performances, there are many variables that can be changed and are open for analysis. Despite that, some general comments can be made.

The energy consumption and execution time is higher for the ProEnergy algorithm than the other two. EWMA and WCMA are not that far apart when it comes to resource usage.

When is comes to the predictions errors, the absolute difference is similar between algorithms and simulation, but the ProEnergy in the computer has a significant better performance than in the DPP DevBoard implementation. On the percentage error these differences can be better appreciated. The high-level simulations performed better than the DPP DevBoard implementation, specially the ProEnergy active update.

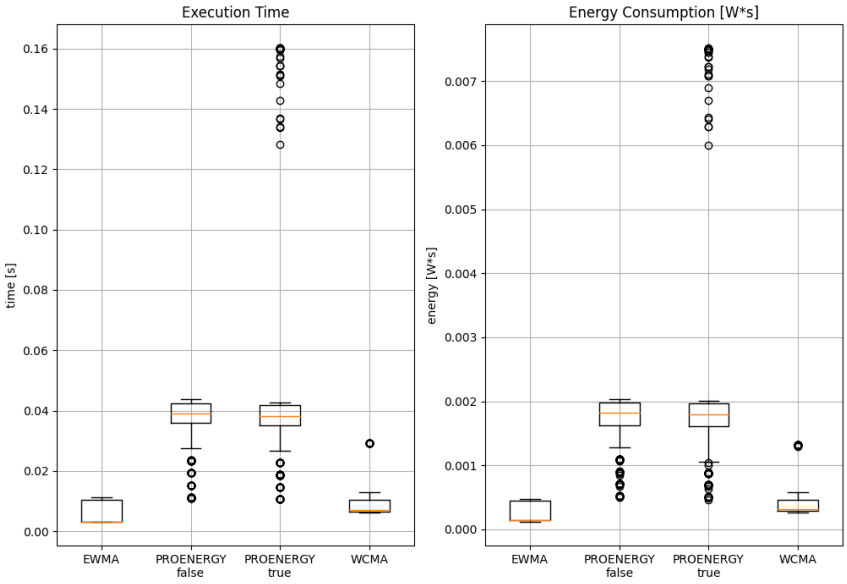

Figure 4.17: algorithm comparison: execution time and energy consumption

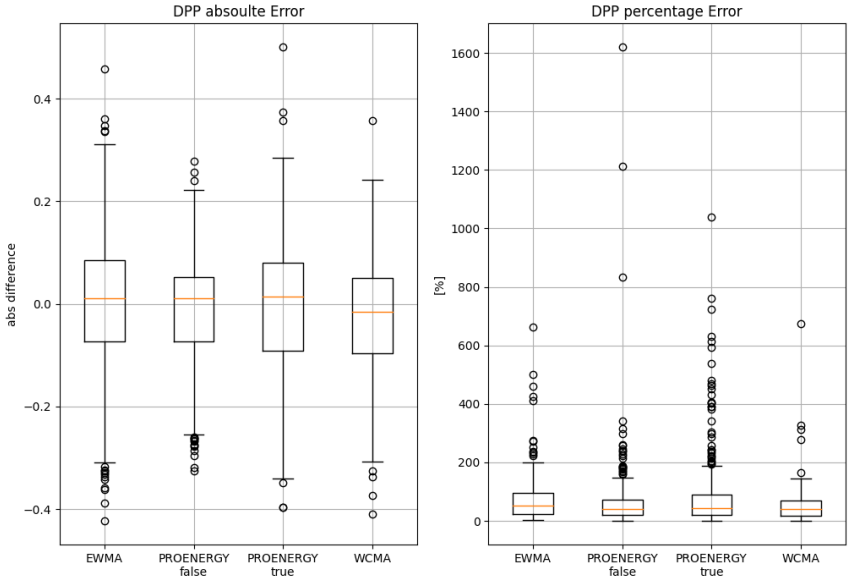

Figure 4.18: algorithm comparison: DPP DevBoard prediction absolute and percentage error

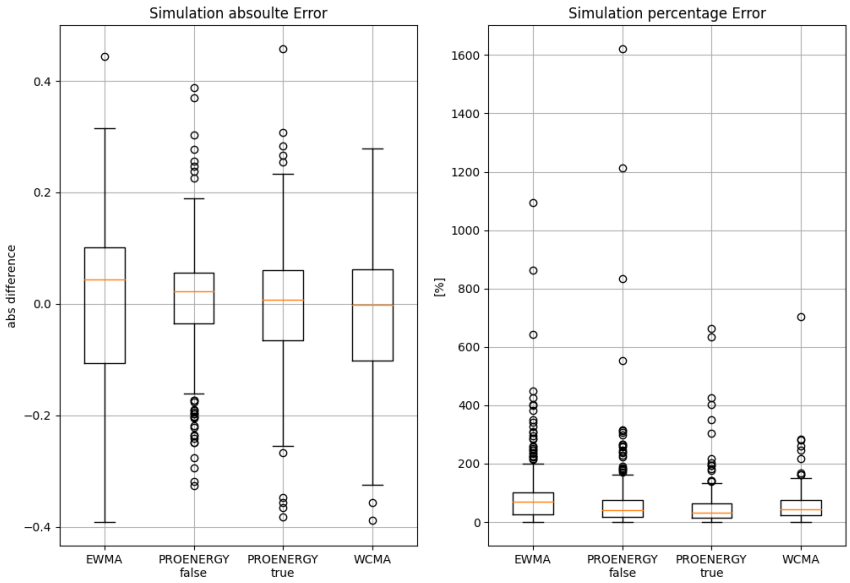

Figure 4.19: algorithm comparison: simulation prediction absolute and percentage error

Chapter 5

Missing points and future improvements

Due to the time constrictions of the project and human nature, there are for sure some errors and improvements that can be made. This is just a short list of personal appreciations.

5.1 Coding errors

The code has been reviewed and the implementation works, but there is a possibility that some chucks of code are unnecessary or can be written in a more efficient way.

5.2 Available data

There are two possible improvements regarding the database. The whole project used only two files (`solar_data.xls` and `wind_data.xls`) as database and its column structure in the DataFrame is simplified to [`"Year", "Month","Day", "Hour", "<country abbreviation>"`]. Thus one restriction is how the data is loaded and changed in order to satisfy the previously mentioned columns.

On the other hand the setup has no easy way to select certain specific days to simulate. For example, if the database has data in a interval between january and december, you can't currently choose to simulate july. The only way to do it right now is to simulate all the way till the end of july and later select the specific interval. The other method would be to eliminate all the data of the database before the starting of the wanted interval, that is not practical.

5.3 Algorithm unknown behavior

In some cases the simulation performs way better than the DPP DevBoard prediction, ProEnergy for instance. The exact reason of why this happens is unknown. Here are some ideas, but a more precise analysis is needed to have a definitive conclusion.

Assuming that the all the coding was done right, it is possible that the precision of some more digits make a significant difference in ProEnegry algorithm when it comes to select the profile for the prediction.

The other option is a bug in the low-level implementation code and annoyingly messes up the calculation under certain conditions.

5.4 User interface

In order to make changes or insert new features / algorithms, the user has to do all that manually. It would be a very comfortable thing to have some kind of user interface for at least editing the manual inputs and run the whole process.

Chapter 6

Conclusion

In its current state this benchmark is good enough to do quick appreciations of a energy harvesting estimator algorithm destined for microcontrollers and it allows further analysis, because after the simulation is done the data is saved in several CSV files. The results from these three algorithms show that it is functional and it has potential to do more extensive analysis.

The high-level implementation in the computer manages and coordinates the whole process which includes the DPP DevBoard and the RocketLogger. Additionally it executes a simulation, which is more accurate and also serves as a good comparison for the DPP DevBoard's functionality.

The low-level implementation is divided in two structures. One that is common to everyone and another specific for the chosen algorithm. This was done in this manner so that a future algorithms could be implemented without having to change many things.

There is clearly room for improvement and expansion of features for this tool. I personally hope that this can facilitate the work of other fellow engineers and possibly some scientists too.

Appendix A

Additional remarks for the Annexes

In previous section 1.4 some of the colors were introduced. As the annexes go into more details, more colors are used to identify levels of the implementation. Here is a remainder of old colors and a statement about the new ones.

- color `red` is used to point out things related with python implementation
- color `orange` highlights BASH scripts or files
- color `violet` relates to anything to do with the low-level implementation on the DPP DevBoard
- color `blue` is for database files or directories
- color `cyan` highlights things to keep in mind. For example a value, software names or command lines

Appendix B

Necessary installations and how to use

B.1 Python

The python version used is `Python 3.6.9` and here is a list of libraries that need to be installed:

- pandas. Terminal command: `pip install pandas`
- numpy. Terminal command: `pip install numpy`
- matplotlib. Terminal command: `pip install matplotlib`
- os. Terminal command: `pip install os`
- re. Terminal command: `pip install re`
- math. Terminal command: `pip install math`
- serial. Terminal command: `pip install serial`
- binascii. Terminal command: `pip install binascii`
- datetime. Terminal command: `pip install datetime`
- struct. Terminal command: `pip install struct`
- time. Terminal command: `pip install time`

B.2 Texas Instrument: Code Composer Studio

The version of CCS used is `9.3.0` and the installation has to consider the inclusion of the `Segger Debugger` and `MSP432` files.

Depending on the computer's operating system, the eclipse file installed by the CCS will by a little different. This file is used in the script `BuildAndFlash.sh`, line 17. The command is written for a Linux machine, if another operating system is used please look up this reference [8].

Below here are some screen shoots of how the project properties are set (those are important to set the generation of the bin file for example):

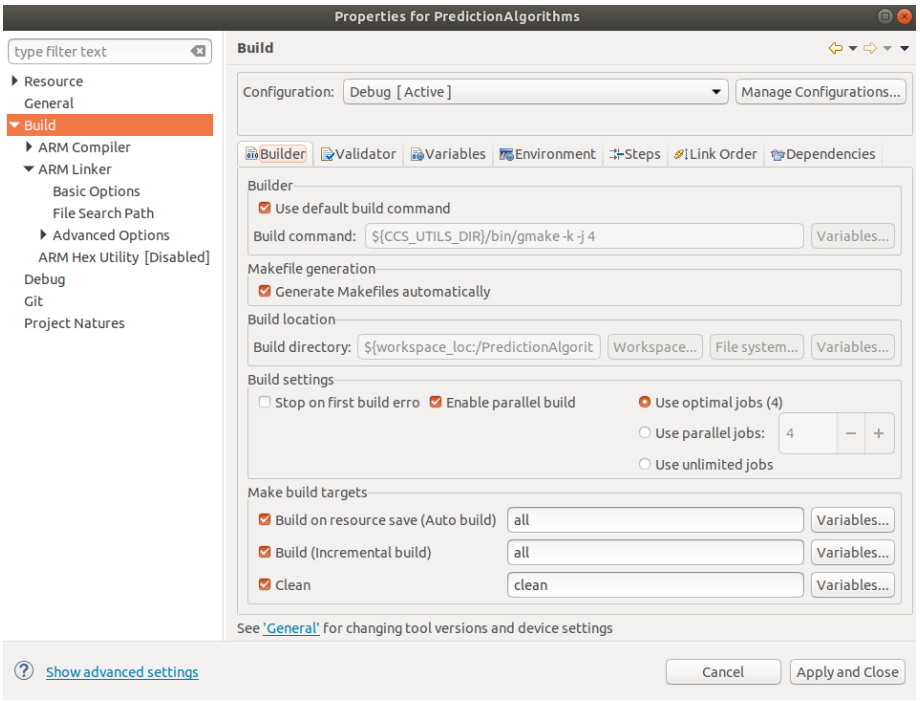

Figure B.1: CCS properties, build

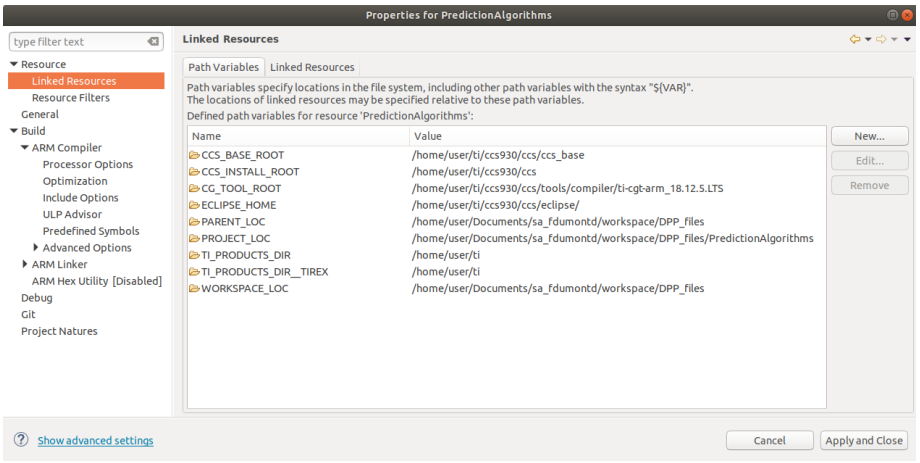

Figure B.2: CCS properties, linked resources

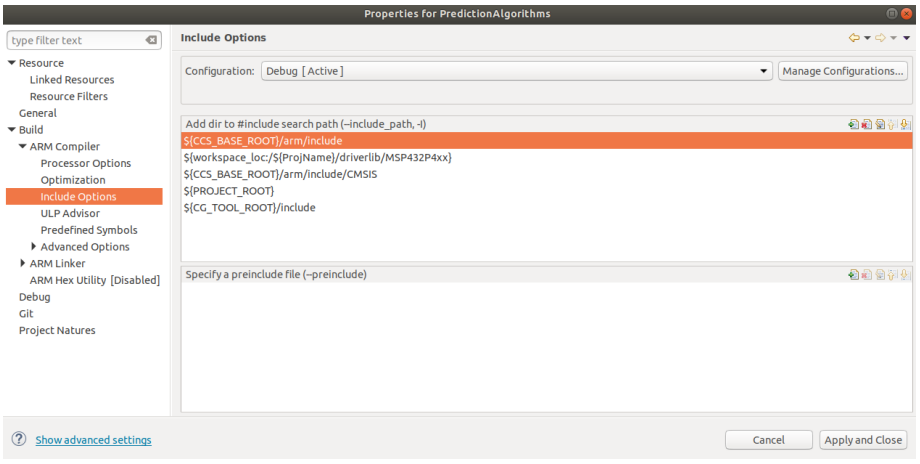

Figure B.3: CCS properties, resources include options

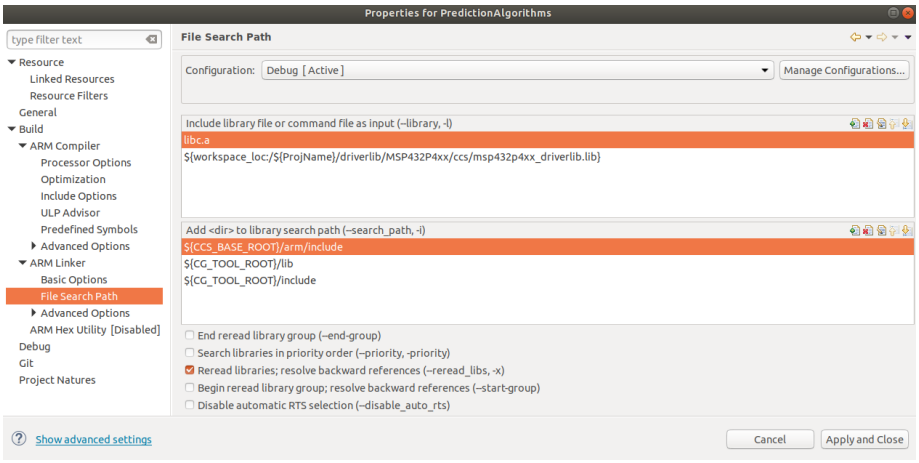

Figure B.4: CCS properties, file search path

B.3 RocketLogger

Most likely the user will be given a functioning RocketLogger, but if that isn't the case instructions about how to configure the hardware can be found here [2]. In the script `RocketLoggerProcess.sh` the commands `ssh` use two parameters that will be quickly comment here.

The first of them is the IP of the RocketLogger in the local network. There are multiple ways to get the IP, but this command in terminal will help you too: `sudo nmap -v 192.168.1.1-255` or `for i in {1..255}; do host 192.168.1.$i;done`.

The second parameter is the path to the `rsa` key of the RocketLogger. The default path for the Linux operating system is `/.ssh/RocketLogger.default_rsa` (RocketLogger.default_rsa is the default file after the RocketLogger's configuration)

B.4 Gitlab

The pull of the protect is done the same way as any other, but here it is highlighted the use of `Git Large File Storage (lfs)`. The database files are huge and in order to save space in the archive `lfs` is used. A list of commands are presented below:

- `git lfs install`: install the resource.

- `git lfs track "*.xls"`: find the database used.
- `git lfs track "*.xls" --lockable`: see the lock files
- `git lfs lock <complete file path>`: to lock the file
- `git lfs unlock <complete file path>`: ro unlock the file

B.5 Setup

This is a step by step on how to connect the all the hardware, photos below.

1. `RocketLogger local connection`: connect the LAN cable to the local router. Keep in mind that the cable mus not have a ground connection.

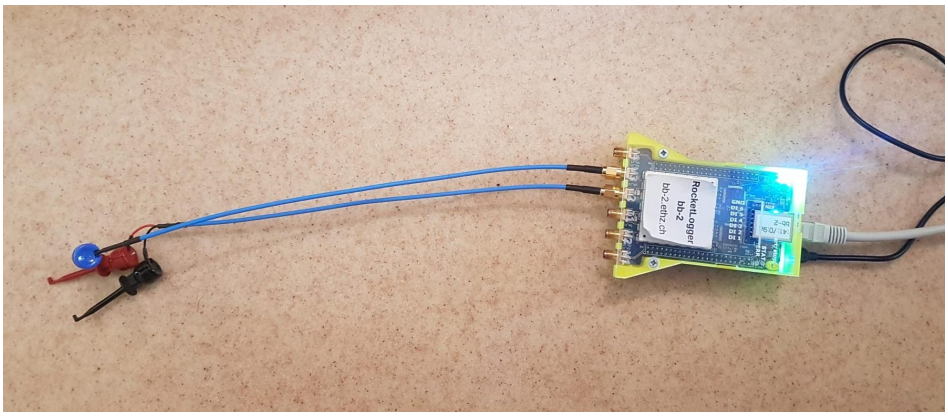

Figure B.5: setup, connect RocketLogger to the local net

2. `Connect SEGGER Mini EDU and mini USB`: the mini USB is own made, most likely you will have to do your own. The GND connection is cut and connected in this way.

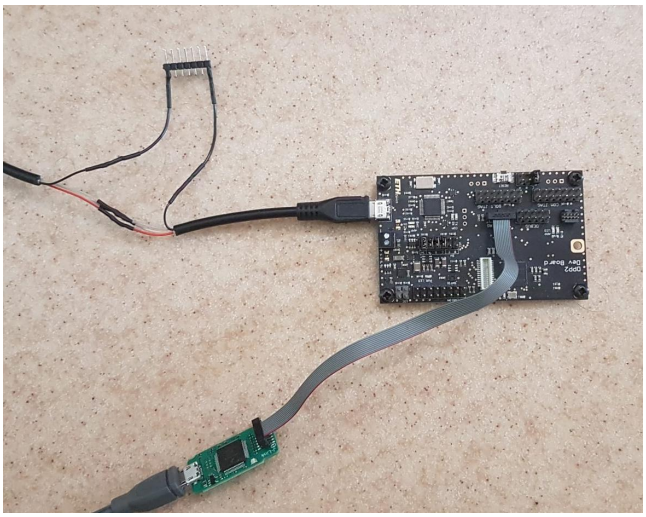

Figure B.6: setup, connect SEGGER EDU Mini and mini USB

3. `Connect logic pins`: The logic inputs ID 1 and ID 2 muss be connected with extension header's pins 2 and 4 respectively. These pins can be change if the user decides to.

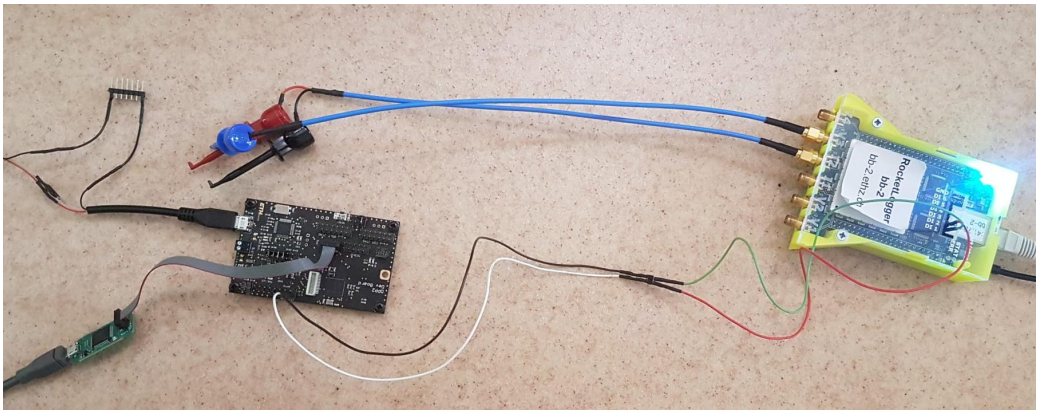

Figure B.7: setup, connect logic inputs with outputs

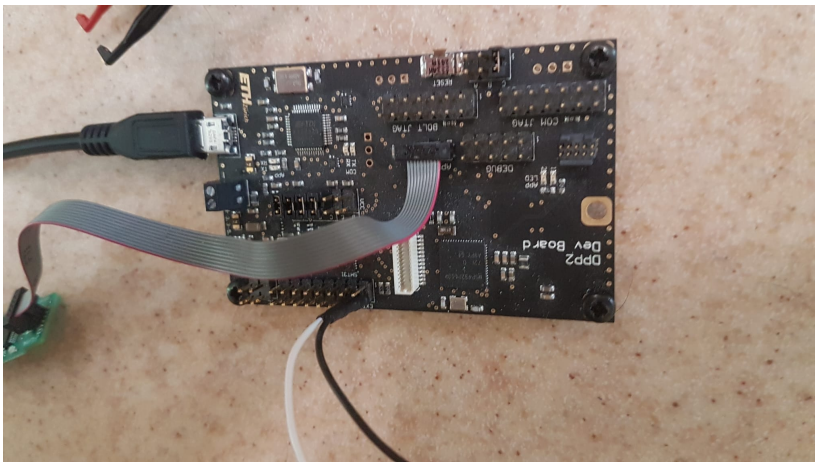

Figure B.8: setup, zoom into logic output's connection with the RocketLogger

4. `Voltage measurement connection`: the voltage connection is done in with the jumper `3V` of the header connector `VCC`

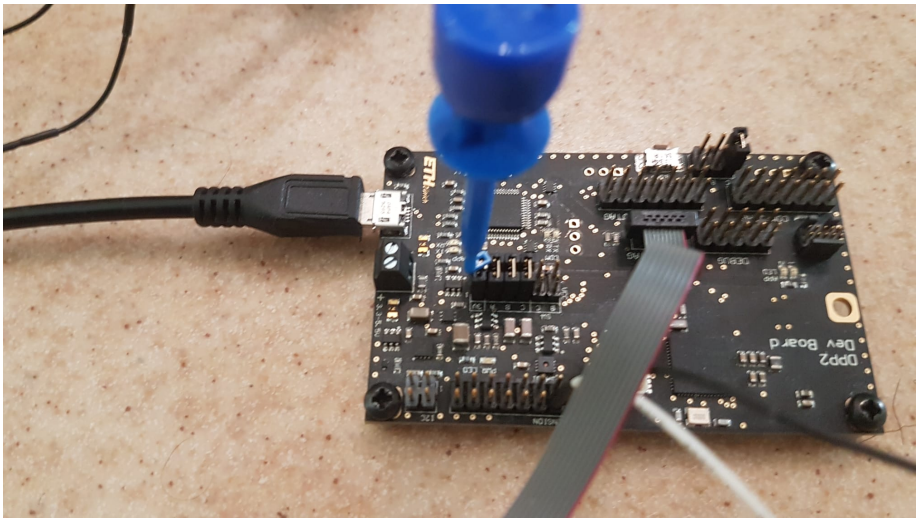

Figure B.9: setup, voltage measurement's connection

5. `Current measurement connection`: the current is connected as described in the `low side measure` option, more detail in the RocketLogger Wiki [2]. In this case the `LO` is black and the `HI` is red.

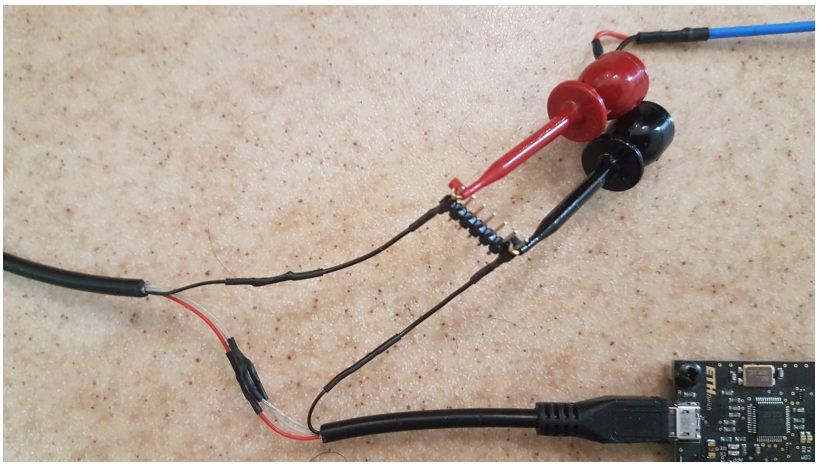

Figure B.10: setup, current measurement's connection

6. `Connect the USB connectors to the computer`:

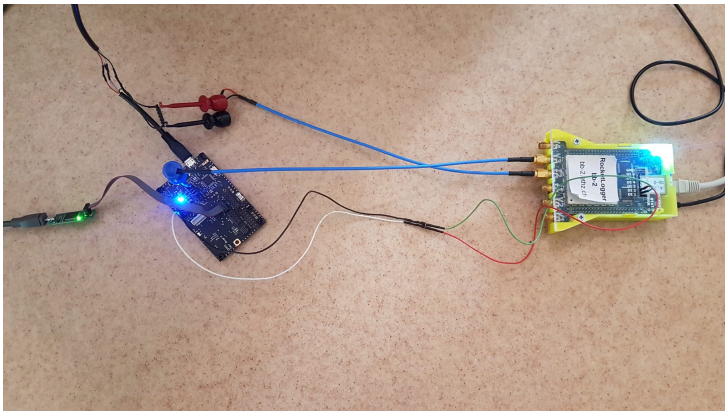

Figure B.11: setup, whole functioning connection

Appendix C

High-level implementation

This annex is meant to do a more in dept understanding of the high-level implementation and to state exactly what the user inputs for each script are.

C.1 Data acquisition

C.1.1 General functionality

The data acquisition script is called `data_acquisition.py`. In the script four objects are written to read data form specific directories. The four classes are `Solar`, `Wind`, `RocketLogger` and `SimulationAndDPP`.

The first two classes read data from the files `solar_data.xls` and `wind_data.xls`. Those databases include several locations, but the DataFrame is filtered to contain only one. The initial columns of this DataFrame are [`"Year", "Month","Day", "Hour", "<country abbreviation>"`].

After a simulation is performed, data from the RocketLogger is saved in the `[custom_path]/workspace/data/RocketLogger` directory and the class `RocketLogger` reads it and the fills missing time values for the DataFrame. It is important to note that the RocketLogger doesn't write all the timestamps of its measurements, only the first one. Obviously the measurement rate is also needed and that is given in the RocketLogger's measurement file.

Lastly `SimulationAndDPP` reads the output files of a certain simulation and stores the information on two DataFrames, `self.DPP_data` and `self.Sim_data`. These variables are later used for evaluation purposes.

C.1.2 Data directories

All of these directories can be found in the path `[custom path]/workspace/data` and have different data on them.

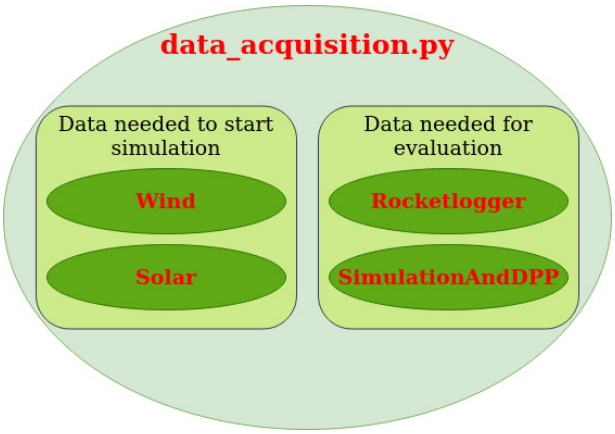

Figure C.1: classes defined in data_acquisition.py

`FilteredData` : the data saved here comes form the script `evaluation.py` and has information about the interval where the prediction was done in the DPP DevBoard. The first two columns `start index` and `stop index` help to find the exact row in the database been used. Then it has a column `time`, which indicates the time of the execution in the DPP DevBoard in seconds, followed by a column `energy` that gives the energy consumed by the DPP DevBoard in that same interval. The other columns are just the percentage and absolute error of the predictions of the DPP DevBoard and the simulation.

`SimulationAndDPP` : the script `simulate.py` at the end saves the data from the DPP DevBoard predictions and Simulation predictions in two different files, one with the prefix `DPP_` and another with the prefix `simulation_`. The DPP DevBoard files have two extra columns indicating the time at which the data was sent to the DPP DevBoard and at which time the DPP DevBoard's prediction was received.

`RocketLogger` : the RocketLogger's measurement data is saved in this directory. There is only one file that is important to keep, `deepSleep.csv`. That file is the measurement of the DPP DevBoard on deep sleep. With that information it is possible to isolate the consumption of the MSP432P401R during the prediction calculation.

`solar-data` **and** `wind-data` : here the databases for the simulations are placed. If another file is used, keep in mind that significant changes need to be done in the script `data_acquisition.py`.

C.2 Algorithm coding structure

As stated before, in the project there is a high-level implementation for the algorithms too. The specific code for each one of them is in the `predictors.py` script. Each class defined there (with their corresponding methods) emulates one of the algorithms, thus the names `EWMA`, `WCMA` and `ProEnergy` were given. The structure of these classes is quite similar, but of course the code on each one of them is different. Many variables' names on these classes were taken from the papers [7], [5] and [6] respectively.

The general structure of these classes consists in three substructures. The first one is getting parameter's value and calculate the initial matrix / vector. The second substructure consists on methods that perform the prediction and updates other elements. The third substructure focus on the simulation of the algorithm itself, which means in a loop it goes through the rows of the DataFrame with the actual measurements and uses the second substructure to get the predictions and save them. In this last subsection the relative and absolute error also calculated and saved. The general overview of this script is shown in figure C.2, which has the three substructures and the methods for each predictor.

If a new algorithm is going to be introduced to the benchmark, it is recommended to write the high-level implementation of it in this script, `predictors.py`.

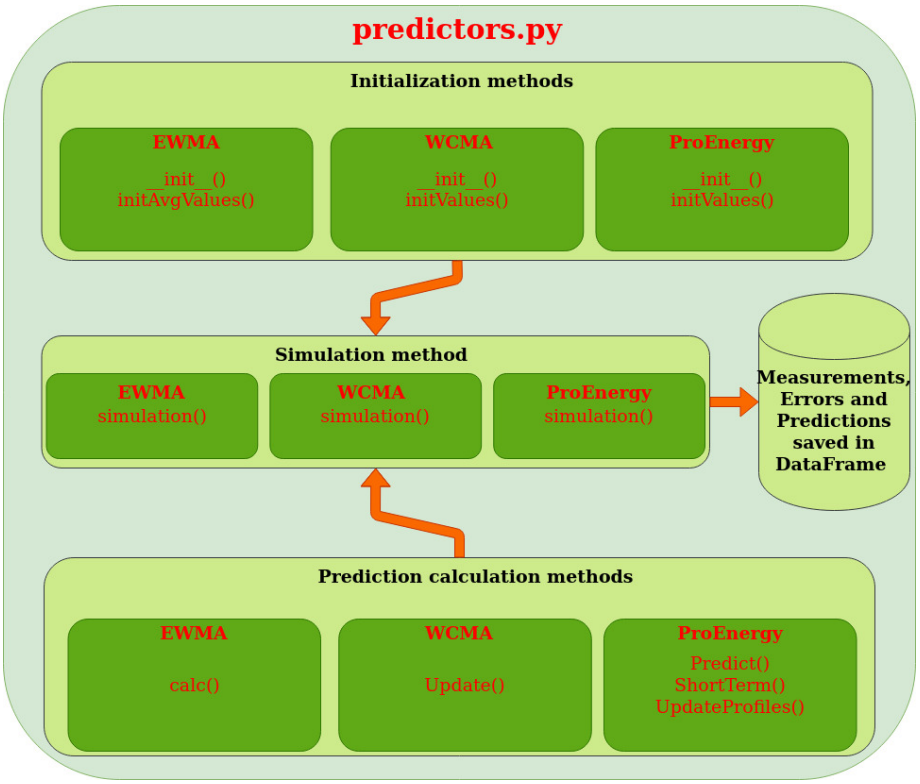

Figure C.2: structure of predictors.py

C.3 Communication with DPP DevBoard and its measurements

The machine used to run this setup has to edit and load the corresponding files of the DPP DevBoard. Then communication has to happen in order to give the measurements to the DPP DevBoard and receive the predictions made by it. Additionally to this, the RocketLogger has to be set correctly and started before the DPP DevBoard begins with its predictions. As soon as the DPP DevBoard predictions are done, the simulation on the high-level implementation has to start and later all the data has to be saved. This whole process is mainly coded in the scripts `DPP_simulation.py` and `simulate.py`, but uses other scripts for purposes that will be explain in the following subsections.

The two scripts mentioned before have the following roles to fulfill. On one hand the script `DPP_simulation.py` has a class defined where the processes for the high-level implementation and the DPP DevBoard are coded. On the other hand `simulate.py` is used to define the different initial inputs for the `DPP_simulation.py`'s class, organize the general process and the launch of the

setup.

C.3.1 Initialization

The class defined in the DPP_simulation.py is called SimulationAndDPP (same as the one in data_acquisition.py). The method __init__() receives parameters, paths and directory names in the form of dictionaries that are used to perform the simulation. All of these dictionaries are filled in the script simulate.py. This method is also in charge of reading the data from the database, the initialization of the selected algorithm and the preparation of the DPP DevBoard.

Initialization in DPP_simulation.py :
The method __init__() uses also other method from class SimulationAndDPP, these are ReadData(), InitializeSimulation() and PreparationDPP().

The first, ReadData(), uses features of the script data_acquisition.py to read the data from the solar or wind database. Following the order, the method InitializeSimulation() selects the algorithm to be used in the high-level implementation with the corresponding parameters given by the initial input. And finally, the third method used, PreparationDPP(), runs several things with the objective to edit the files for the DPP DevBoard correspondingly, build the BIN file and flash it to the DPP DevBoard. This will be looked at with some more detail in the subsection C.3.2.

Initialization configuration in simulate.py :
In this script the corresponding inputs for the class SimulationAndDPP and for the RocketLogger are defined.

In regard to the SimulationAndDPP's input, they are divided into 3 dictionaries: ccs_dict, algorithm_dict and data_dict. The first,ccs_dict , of them describes paths the CCS project directories and the BIN file generated after the project is build. The dictionary algorithm_dict has the algorithm selection and its parameters. This values will be used for the setting of the high-level simulation and DPP DevBoard files. Lastly, the third dictionary, data_dict, defines the path to the database and the country selected to filtered. After the definition of these dictionaries, the amount of days to be simulated are also coded.

The settings of the RocketLogger are coded in the dictionary RocketLogger_dict. Among those inputs the most important (or at least the most likely to be changed) are the path to the rsa key for the ssh connection and the IP given to the RocketLogger. There is also a variable called use_RocketLogger with the purpose to activate or disregard use of the RocketLogger, of course if deactivated the voltage and current measurement

will be missing.

The specifics about the elements of the dictionary is described in the section C.4. It is highly recommended to look at it so that user knows how to set the benchmark correctly.

C.3.2 File preparation and flashing for DPP DevBoard

This aspect is coded in the method `PreparationDPP` in the class `SimulationAndDPP`. The first thing done in this method is the popper editing of the DPP DevBoard files, those are: `config.h`, `config.h`, `ewma.h`, `wcma.h` and `proenergy.h`. It is important to note that last three files only are edited if a particular algorithm is selected.

Following that the file `commands.jlink` is edited to have the appropriated `BIN` file path written on it. This file is used by the utility `J-Link Commander` [9] later. At the end of this method the bash script `BuildAndFlash.sh` is ran. This script receives the CCS directories, builds the CCS project, generates `BIN` file and flashes it to the DPP DevBoard with the utility `J-Link Commander`. It is important to note that the physical connection between the board and the computer for the flashing is done through the hardware `J-Link EDU Mini`.

C.3.3 RocketLogger

The RocketLogger has to be started before the predictions are made. Once finished, this hardware is stopped and the data saved to the corresponding directory `[custom_path]/workspace/data/RocketLogger`. All this can be controlled thanks to the available `SSH` commands of the RocketLogger. They are used by the scripts `RocketLoggerProcess.sh` and `RocketLogger_function.py`.

The script `rocketlogggerProcess.sh` has all the `SSH` commands, but just some of them will be used depending on the given parameters when the script is called. This commands can be summarized as:

- Stetting parameters
- Start measurement
- Stop measurement
- Copy data from measurement

On the other hand `RocketLogger_function.py` defines a class `RocketLoggerFunctions` with methods that represent the same ideas described in the paragraph above, but it gives the correct parameters to the `RocketLoggerProcess.sh`'s execution

As mentioned before, the script simulate.py structures the whole process with the corresponding input, thus it uses the methods from RocketLoggerFunctions to set the RocketLogger, start / stop its measurement at the correct time and get the data.

C.3.4 Control flow

Many parts of the process are described above separately, but none of them describes the control flow of the whole execution in computer. Thus this part gives that overview which is depicted in the figure C.3. The main structure of the control flow can be seen in the script simulate.py, though as seen before this script will use methods from others scripts to performed the needed tasks.

First, the user has to set some parameters into the file simulate.py. All of those parameters are organized in dictionaries, which are given to the class SimualtionAndDPP.

Then the script has to be executed and the first step is the reading of the database, the initialization of the algorithm in the high-level implementation and the correct flashing of the DPP DevBoard. In this step there are a lot of things involved as seen in sections before, but some of them will be mentioned again. In the initialization of the algorithm the class SimulationAndDPP form the script DPP_simulation.py is used. The algorithm is selected and the initial memories' values of the vector or matrix are calculated, this values are sent to the DPP DevBoard in a later step. Right after that, the .C files for the DPP DevBoard are edited according to the manually given inputs. Then the bash script BuildAndFlash.sh is used to create the BIN file and flash it to the DPP DevBoard.

If the RocketLogger is going to be used (the most likely case), their parameters have to be set and that is done via a SSH connection. Among these parameters are the sampling rate, the activation of certain channels and the format in which the data has to be saved.

After that, the first communication between the DPP DevBoard and the computer through UART takes place in order to send the initial values of the memory vector or matrix. From the editing of the .C files for the DPP DevBoard, the chip already knows the order and amount of data that will be sent for the memory vector or matrix. Thus the machine sends a value through UART to the DPP DevBoard and waits till an acknowledgement is received. Once all the values have been sent and acknowledged, this step is considered finished.

Before starting with the prediction calculations of the DPP DevBoard, the RocketLogger measurements needs to be started. For that the method start form the class RocketLoggerFunctinos is used. It is basically a SSH command sent to the RocketLogger in order to start measuring.

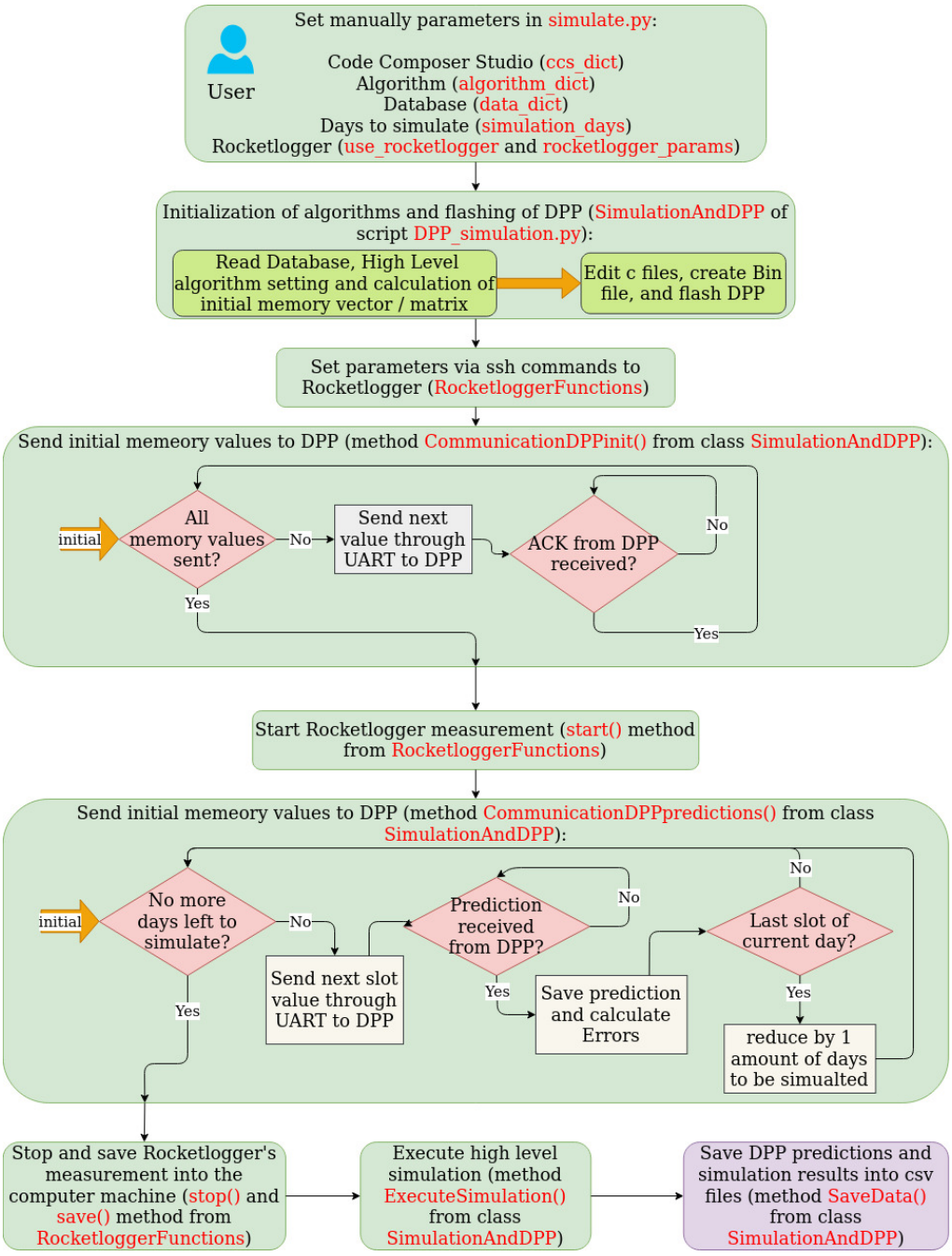

Figure C.3: computer control flow

At this point the data can be sent to the DPP DevBoard in order to do their predictions. The data is taken form the selected database from before. In the database each row represents a time slot for a certain day and they are ordered

in chronological order. Thus a loop that goes through each row is used and it will stop once the days left to simulate become zero. On each row the following information will be sent through UART to the DPP DevBoard: the number of the slot and the harvested energy value. The computer will wait for a response from the DPP DevBoard, which contains the prediction value. Each time one prediction is received the absolute and the relative error are calculated, once this is finished the loop continues with the next row of the database.

When the days to simulate are finished, the RocketLogger is stopped and the data saved to the computer by the methods `stop` and `save` respectively.

At this point the execution of the simulation by the computer is performed. The method used for this is `ExecuteSimulation` from the class `SimulationAndDPP`. The simulation is a similar loop through rows in the database, but of course there is no interaction between DPP DevBoard and the computer.

The last step is saving the prediction's data of the DPP DevBoard and simulation into `CSV` files in the respective directories. The method `SaveData` from the class `SimulationAndDPP` is used for this purpose.

C.4 Comments and user inputs for python scripts

C.4.1 Script: DPP_simulation

There are two comments to be made here about the format in which data is been sent to the DDP DevBoard and the editing of the .C files.

The format used to send the data is in three Bytes. The first Byte determines the slot number of the data. If this is the last data to be sent (because this is the last slot of the last simulation day) the last bit of the Byte is set to 1. The other two bytes are the data from the database. The data from the database is multiplied by a factor of `10000` before putting it into the Bytes. It is done this way to just send an integer number, the factor might need to be changed depending on other databases.

Regarding the editing of the .C files, obviously this needs to be adapted if new algorithms want to be tested. First it reads the file, saving it into an array. Then in loop it finds the line with the declaration of the variable and writes the value of that variable. Once all of the variables have been modified, the array of the file is used to rewrite the file. By doing the editing this way, the lines of the variables do not have to specified nor remembered.

C.4.2 Script: simulate.py

This script organizes the whole high-level implementation and at the end generates some additional graphs comparing the high-level with the low-level implementation.

Below are the default inputs and how they can be changed, though some of them don't need any alteration.

`ccs_dict`

- `"bin file path"`: path to bin file generated by CCS, no need to be changed
- `"eclipse path"`: this is the complete path to the eclipse file. This needs to be changed to the local computer of the user.
- `"workspace"`: no need to change this.
- `"project name"`: no need to change it.

`algorithm_dict` Data not used by the algorithm is simply be disregarded.

- `"algorithm"`: this is the selection of the algorithm. The possible values are `"EWMA"`, `"WCMA"` and `"PROENERGY"`
- `"repetitions"`: this is the amount of times the prediction calculation will be done before sending it to the computer. The boundaries for this variable is [1, 255]
- `"day slot"`: this is determined by the database been used. No need to be changed if `solar_data.xls` or `wind_data.xls` is selected.
- `"alpha"`: parameter for the algorithm. This depends on the user and the value is in the interval [0, 1].
- `"K"`: parameter describes the amount of previous slots to be considered in the prediction. The maximal value it can take is the amount of slots in a day.
- `"D"`: parameter destines to determine how many days are saved in the WCMA or ProEnergy E matrix. In the case of EMWA this value will determine how many days are considered in the database to calculate the initial historical values.
- `"A"`: parameter of ProEnergy for active update. Determines the age the profile must have to be replaced.

- "mae threshold": this value is the MAE threshold in ProEnergy to update profiles. Its interval is between [0, 65535]. Keep in mind, the value written to the high-level implementation is a factor 0,0001 of what is given here.
- "active update": this value determines if active update is used or not in ProEnergy algorithm. The values it can take are {"true", "false"}.

data_dict

- "directory": selects the directory of the database to be used. Has two values {"solar-data", "wind-data"}.
- "type": described the type of data. Currently not been used, but is has two values Has two values {"solar", "wind"}.
- "location": this denotes the location to be used from the database. There are many to chose from, the one has default is "AL".

RocketLogger_params

- "format": format in which the measuring file is saved.
- "rate": rate of the measurements. Other options are described in [2]
- "channels": active channels. All the options can be found in [2]. Important to consider that if these are changed, other parameters will need to be changed as well in the evaluation part.
- "filename": the name given to the measurement. No need to change it.
- "ssh_file": path to the rsa key file. The user will have to set this right depending on the operating system been used.
- "ip": IP of the RocketLogger given by the local network.

simulation_days : amount of days to be simulated. It just has to be an integer.

use_RocketLogger : determines if the RocketLogger is used or not. It can take two values {True, False}.

If all of these parameters are set correctly then the script can be ran in a terminal with the python command.

C.4.3 Script: evaluation.py

As mentioned before this script takes the raw data and outputs graphs that gave an individual evaluation of the algorithm. Here the inputs will be given and explained.

`filename` : suffix of the file's names to be considered in the directory `[custom path]/data/SimulationAndDPP`. Just as a reminder, the whole name has the prefix `Simulation_` or `DPP_`.

`I_ds` : this is the name of channel used in the RocketLogger for the deep sleep's current measurement. The default file used is `deepSleep.csv` and needs to be changed if another hardware than the DPP DevBoard is used.

`V_ds` : this is the deep sleep's voltage measurement channel in the file `deepSleep.csv`. It need to be changes if another files is been used.

`I` : name of the channel measuring the current of the algorithm test. If channels in the RocketLogger are modified, this setting as to be changed as well.

`V` : name of the channel measuring the voltage of the algorithm test. If channels in the RocketLogger are modified, this setting as to be changed as well.

`LogIn_exe` : this is the name of the RocketLogger's logic input measuring the calculation prediction's execution time.

`LogIn_alg` : this is the name of the RocketLogger's logic input measuring the whole interval in which calculations are been done by the DPP DevBoard. This variable helps in correcting the time shift between the RocketLogger and computer.

C.4.4 Script: comparison_evaluation.py

The script combines the info from all other algorithms. It has to be mentioned that, before this script is ran, the script `evaluation.py` for each one of the algorithms had to be ran. Otherwise the files in the directory `[custom path]/workspace/FilteredData` will not be there. The only input the user has to give is `files`, which is an array with strings denoting the names of the files to be considered.

Appendix D

Low-level implementation

It has already being mentioned that the low-level implementation refers to everything related to the implementation that will be flashed to the DPP DevBoard. All of these files can be found in the CCS project directory [custom_path]/workspace/DPP_files/PredictionAlgorithms. In this annex the structure of the implementation, important aspects of it and the control flow inside the DPP DevBoard is explained in more detail.

D.1 Structure composition

The first thing to mentioned here is that the Texas Instrument's driver library for the MSP432P401R is included. All of those files can be found in the directory [custom_path]/workspace/DPP_files/PredictionAlgorithms/driverlib. In the implementation of course not all of the drivers are used, but we can mention at least the most important ones: UART, CS, GPIO and INTERRUPT.

Now the specific implementation files for this tool can be discussed. These are the files with their respective headers: config.c, config.h, main.c, main.h, ewma.c, ewma.h, wcma.c, wcma.h, proenergy.c and proenergy.h.

The files with the names main and config are common for every algorithm. On the other hand, the content of the rest of the files will only be included if that particular algorithm was selected.

config files: The config.h header has some general global variables used for every algorithms, those include clock frequencies, clock dividers, algorithm selection (ALGORITHM (which will takes values EWMA, WCMA, PROENERGY or OTHER)), amount of calculation repetition for each prediction and number of time slots per day. The corresponding .C file config.c has four coded functions with the following objectives: clock signal setting (clock_init()), uart port configuration

(uart_init()), gpio setting (gpio_init()) and flash memory configuration for the respective clock used (flash_init()).

main files: In this header file the driver libraries for the implementation are included and some global variables, four of them are Boolean tokens (Token_calc, init_values, Start_prediction and End_prediction), one is an array of three bytes to save the data / prediction (comm_data[3]) and the last one helps in the receiving of data to select which of the three bytes is being received (missing_bytes). The file main.c has the control flow of how the processing and events in the chip happen. More information about this will be given in the subsection D.3.

algorithm files: The files referred here are ewma, wcma and proenergy. Their headers and the structure are similar. The variable ALGORITHM in the file config.h selects the respective file which their content will be considered by the use of macro statements. The header has the specific global variables for each algorithm, those are also mentioned in the subsection 2.2. The .C files have at least three functions with he same name that serve the same objectives. Those are initialization_algorithm(), initialization_values() and predict(). initialization_algorithm() initializes the global variables / arrays from the header file with some default values. The function initialization_values() writes down the initial values of vectors / matrix memory sent by the computer in the correct order. And lastly predict() which is indented to do the actual prediction once the real data is being sent by the computer.

D.2 Important variables

Some of theses variables were already mentioned, but more specific information about them is needed.

D.2.1 Common configuration

ALGORITHM : This variable is coded as macro in the file config.h and its only purpose is the activation of the code from the selected algorithm. The values it can take are EWMA, WCMA, PROENERGY or OTHER (which have the values 1, 2, 3 and 4 respectively). This variable is changed by the editing process of files from the high-level implementation (described in C.3.2)

REPETITIONS : Prediction calculations on the DPP DevBoard are too fast for the RocketLogger to measure with accuracy. Thus in order to be able to do so, each prediction calculation has to be repeated several times and this variable

specifies the amount of repetition times. This variable is defined in the file config.h and can also be edited by the high-level implementation (described in C.3.2).

DAY_SLOTS : As mentioned before the day for the algorithms is divided into time slots. In the case of this setup, due to how the database is structured, the amount of slots per day is 24. Obviously, this can change depending on the database being used. This variable, located in config.h, can also be edited by the high-level implementation process (described in C.3.2)

Token_calc : Once data has being received through the UART communication channel, the MSP432P401R activates by an interrupt and this Boolean global variable (Token_calc) is set to True and the functions initialization_values() or predict() executes. Once that is finished the variable is set to False again. The declaration of this variable is in main.h.

init_values : This is also declared in the file main.h and its purpose is to select which one of the functions (initialization_values() or predict()) should be executed. It starts with a value True, which allows the function initialization_values() to execute if something has being received through UART. Once all the initial values for the global arrays for that particular algorithm have been set, the init_values is set to False and the function predict() executes next time something is received from the computer.

Start_prediction **and** End_prediction : The RocketLogger has some logical inputs that are used in this instance for identifying the execution time of each prediction and the period in which calculations are done. To use that feature, the MSP432P401R needs certain GPIO ports to be programmed accordingly. The port and pin used for identifying the period of time in which the prediction's calculations are taking place is assigned to PORT 5, PIN 1. The two Boolean variables Start_prediction and End_prediction are used to set this pin Low or High depending on their values. The declaration of these variables are in the file main.h.

D.2.2 Algorithm specific

In the mathematical frame subsection 2.2 many variables were mentioned for each algorithm and those names will also be used in the coding.

EWMA algorithm : The algorithm uses only one specific parameter, ALPHA, and an array containing the historical values of each time slot,

hist_slot[DAY_SLOTS]. Both of them are declared in the header file ewma.h. The variable ALPHA can be edited by the high-level implementation.

WCMA algorithm : There are more variables involved and many of them use the same name as described in the mathematical section. Here are the variables and arrays presented: ALPHA, K, D, E[D+2][DAY_SLOTS], slot_mean[DAY_SLOTS], slot_mean_Shift[DAY_SLOTS] and current_day. All of them are declared in the header file wcma.h, but only the first three can be directly edited in the high-level editing process (subsection C.3.2). The purpose of many of them is clear from the mathematical algorithm, with the exception of the last three. slot_mean[DAY_SLOTS] has the mean values of the each time slot from the last D days, without including the current day. In the same sense slot_mean_Shift[DAY_SLOTS] has the average value of the days between the two days and D+1 days ago. And lastly the variable current_day helps the process of setting the initial values in the E matrix (done in the function initialization_values()).

PROENERGY algorithm : This last algorithm includes all of the following variables that will be declared in the file proenergy.h: ALPHA, K, D, ACTIVE_UPDATE, A, MAE_THRESHOLD, E[D+1][DAY_SLOTS], age_profile[D+1], current_age and current_day. The first six can be edited by the high-level process (subsection C.3.2) and the explanation of the first seven can be found in the mathematical subsection (2.2). The array age_profile[D+1] saves the age of each saved profile, the order in this array represents the order in which the profiles are saved in the matrix E. And the last two variables current_age and current_day serve a support role.

D.3 Processing flow

The processing flow of the DPP DevBoard is shown in the figure D.1 and with the previous concepts in mind it should be easier to understand what is happening here.

It starts with the flashing and initialization of the several parameters. Once this is done it goes directly to LPM0 and it stays there till something is received through the UART that sets the microcontroller in run mode.

The function ReceiveData() is linked to the interrupt that is activated once something has been received through UART. What it does is saving the bytes into the array comm_data[]. Two things happen when the three bytes are received. It is checked if Start_prediction is True, which would set to High the PORT 5 PIN 1 indicating that the prediction calculations have started. The second thing is setting the boolean variable Token_calc to True, which allows to

enter some executions in the while infinite loop.

Now that Token_calc is True, the while loop gets the measurement from the array comm_data[] and it checks if this is the last data sent for prediction calculations. This is shown in the most significant bit of last byte of comm_data[].

After that depending on the boolean variable init_values, the microcontroller saves the values to the memory array (initialization_values()) or makes a prediction calculation (predict()). In the first process, as said before, values are saved to the memory arrays and once all of them have been received, init_values will be set to False and Start_predictions to True. On the other hand, if init_values was already False the microcontroller executes the prediction calculation, but in the interval in which the calculation is taking place, the PORT 5 PIN 0 is set to High (this is measured by the RocketLogger and allows the calculation of the execution time). The prediction calculation is repeated the amount of times described in the variable REPETITIONS, but only in the very last one the prediction value is saved in comm_data[].

Among the last things done in the while loop before going to LPM0 is the setting to False of Token_calc and the execution of the function SendData(). This last function does two things, first checks if End_prediction is True, in that case PORT 5 PIN 1 it put to Low, and the data is sent to the computer via UART.

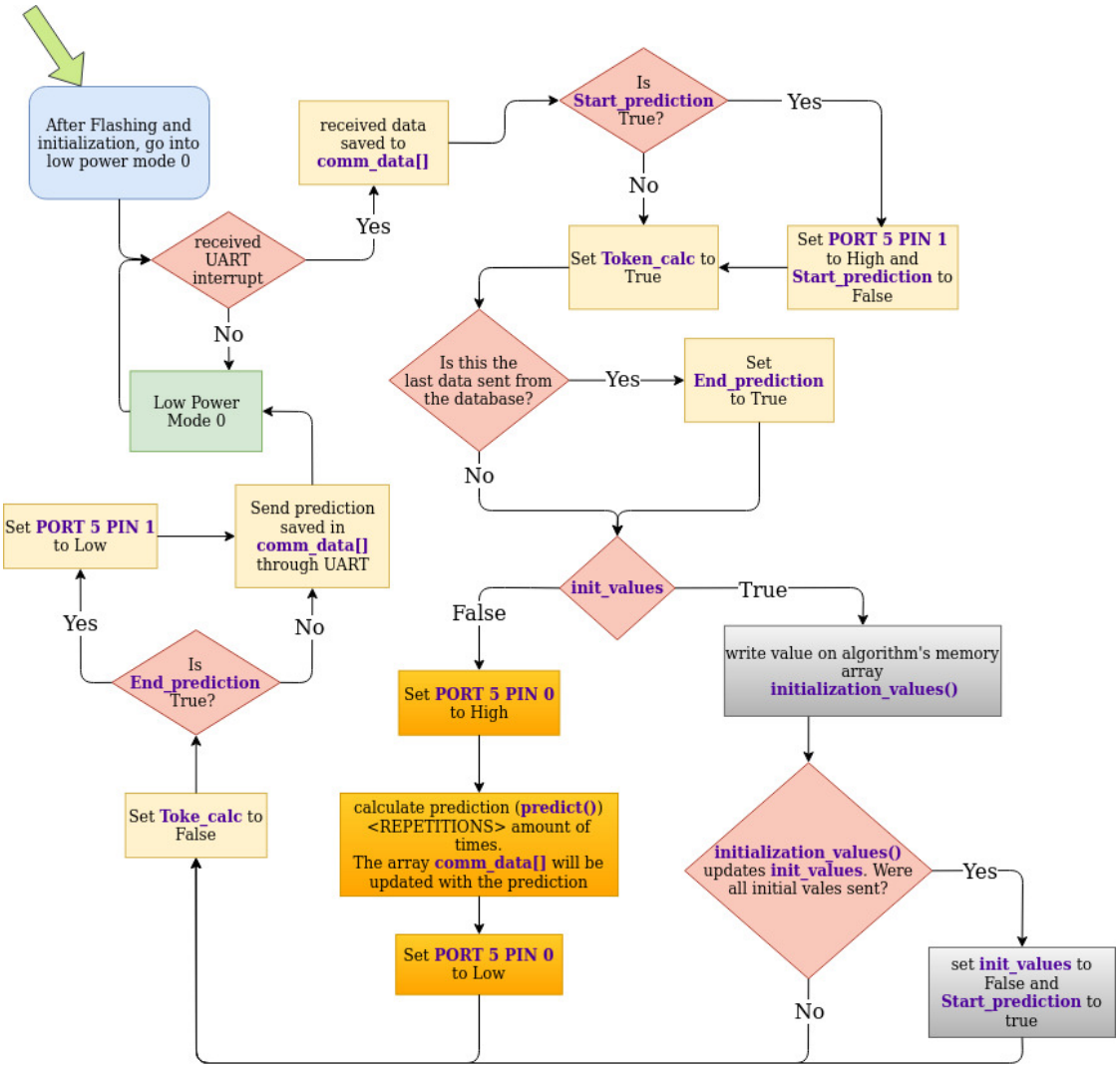

Figure D.1: DPP development board control flow

D.4 Include new algorithm

The following section explains what has to be taken into account in order to include a new algorithm into the low-level implementation. Thus two parts are discussed, changes in the common files and coding considerations for the algorithm specific files.

D.4.1 Common files

In the shared files not much has to be changed, just some minor things muss be included

Configuration files

The `config.h` needs to have a line with the name of the of the algorithm and assign a number to it. For Example :

```
#define SUPERDUPERALG 5
```

Main files

There are some lines to be included in the header file `main.h`. Depending on the value of `ALGORITHM` a header lib needs to be included. For example the lines of be included:

```
//Algorithms
#if ALGORITHM == EWMA
#include "ewma.h"
#elif ALGORITHM == WCMA
#include "wcma.h"
#elif ALGORITHM == PROENERGY
#include "proenergy.h"
#elif ALGORITHM == SUPERDUPERALG
#include "SUPERDUPERALG.h"
#endif
```

D.4.2 Algorithm specific files

It is important that the names, inputs and outputs format of the functions `initialization_algorithm()`, `initialization_values()` and `predict()` are maintained, because they are used in the file `main.c`.

`initialization_algorithm()` : doesn't receive any parameters and doesn't return any either.

`initialization_values()` : return `true` or `false` depending if the all the values of the memory arrays/ matrix have been received. The inputs are: the number of the slot (`idx`) which is the first byte of the array `comm_data[]`; the measurement data (`measurement`) which is the combination of the second and third bytes of the array `comm_data[]`; and the address of `comm_data[]`.

`predict()` : does not return anything and it has 4 inputs. The first one is the slot number (`idx`). The second is the data from the database (`input`). The third one is the number of the repetition (`rep`). And lastly the address to the array `comm_data[]` (`*communication_data`).

In the header file and the .C file of the new algorithm do not forget to include the macro command:

```
#include "config.h"
#if ALGORITHM == SUPERDUPERALG
...
//rest of code
...
#endif
```

Otherwise in the compilation of the projects errors will arise due to multiple declaration of the same functions and variables.

Bibliography

[1] ETH Zurich Computer Engineering Group. *Gitlab DPP Wiki*, 2020 (accessed April 1, 2020). `https://gitlab.ethz.ch/tec/public/dpp/-/wikis/home`.

[2] ETH Zurich Computer Engineering Group. *Gitlab Rockelogger Wiki*, 2020 (accessed April 1, 2020). `https://gitlab.ethz.ch/tec/public/rocketlogger/-/wikis/home`.

[3] ETH Zurich. *Rockelogger ETH Zurich Page*, 2020 (accessed April 1, 2020). `https://rocketlogger.ethz.ch/`.

[4] SEGGER. *Segger J-Link Mini EDU Debbuger*, 2020 (accessed April 1, 2020). `https://www.segger.com/products/debug-probes/j-link/models/j-link-edu-mini/`.

[5] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power management in energy harvesting sensor networks. *ACM Trans. Embed. Comput. Syst.*, 6(4):32–es, September 2007. ISSN 1539-9087. doi: 10.1145/1274858.1274870. URL `https://doi.org/10.1145/1274858.1274870`.

[6] J. Recas Piorno, C. Bergonzini, D. Atienza, and T. Simunic Rosing. Prediction and management in energy harvested wireless sensor nodes. In *2009 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology*, pages 6–10, 2009.

[7] A. Cammarano, C. Petrioli, and D. Spenza. Pro-energy: A novel energy prediction model for solar and wind energy-harvesting wireless sensor networks. In *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*, pages 75–83, 2012.

[8] Texas Instrument. *TI ecplise command line*, 2020 (accessed April 1, 2020). `http://software-dl.ti.com/ccs/esd/documents/ccs_projects-command-line.html#working-with-ccs-linux`.

[9] SEGGER. *J-Link Commander*, 2009 (accessed April 1, 2020). `https://www.segger.com/products/debug-probes/j-link/tools/j-link-commander/`.