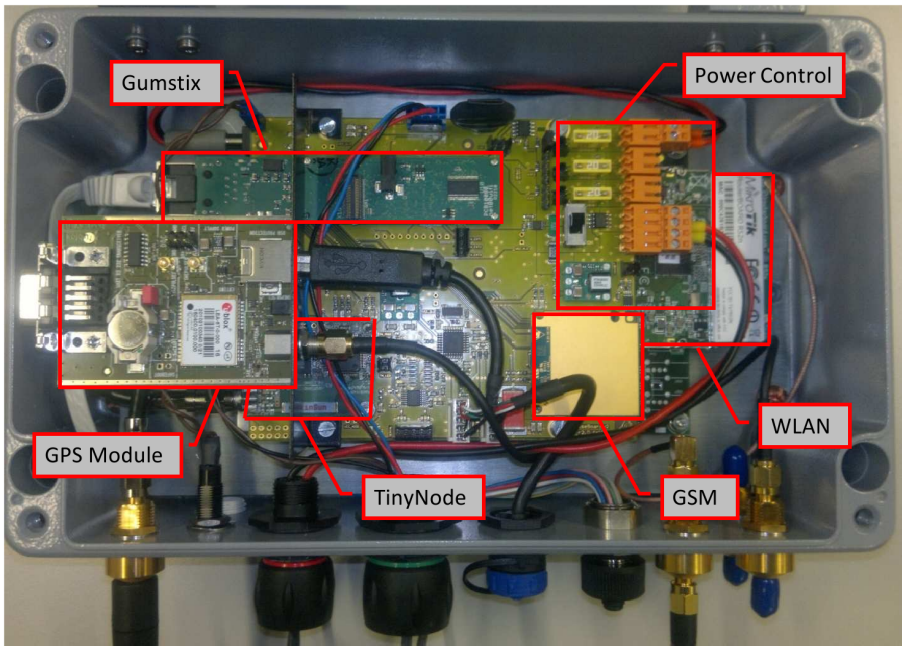
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and
Networks Laboratory

Spring semester 2013

SEMESTER THESIS

Power Management for the CoreStation Platform



Author:
Aljaž Dobnikar

Supervisor:
Bernhard Buchli

Professor:
Dr. Lothar Thiele

Acknowledgement

This project was a new experience for me in several ways. It was my first encounter with real python programming, getting to know APIs for the weather forecast for the first time, and I have never worked with such an extensive and elaborate piece of software as BackLog. I took part in a truly big project and have seen how hard it is for such a project to fit together and for everyone to work together towards a common goal.

My greatest thanks go to my supervisor Bernhard Buchli, for guidance and help with every problem I had during this project. He responded to my every email, no matter how late, and gave me all the information, data and ideas I needed along the way. I would never have finished without his assistance. I would also like to thank Tonio Gsell for helping me set up my computer workspace and explaining BackLog and Python code to me numerous times. Thanks to Professor Dr. Lothar Thiele for making this thesis possible.

Zürich, 21. June 2013

Aljaž Dobnikar

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and
Networks Laboratory

SEMESTER THESIS

for
Mr. Aljaž Dobnikar

Power Management for the CoreStation Platform

Main Reader: Bernhard Buchli, ETZ G75
Professor: Prof. Lothar Thiele, ETZ G87

Issue Date: 18. March 2013
Submission date: 30. June 2013

Goal

The aim of this thesis is to implement a dynamic power management (DPM) module for the CoreStation platform [1]. The DPM module should be generic with respect to the scheduling algorithm, such that different algorithms can be easily integrated and evaluated. For this purpose, the backlog software stack [2] is first modified, so that stand-alone operation on a desktop environment is possible for experimentation and evaluation of the algorithms. After successful completion, the DPM module will be integrated into the backlog framework, so that it can be used on the CoreStation platform.

A rough outline of the tasks:

1. Configure backlog so that it runs on a desktop PC.
2. Implement a simple Plugin.
3. Implement a plugin wrapper that provides the necessary information to simulate the plugin (read energy input from file, provide battery SoC based on energy input and consumption, write output to file).
4. Implement ”exponentially weighted moving average” (EWMA).
5. Implement ”weather-conditioned moving average” (WCMA).
6. Implement interface to weather forecast service.
7. Use weather forecast (WF) service to compute future WLAN duty-cycle.
8. Implement necessary functionality to write new schedule based on computed duty-cycle.
9. Evaluate WF, EWMA, WCMA.

Tentative Timeplan

A semester thesis is to be completed within 14 weeks [3]. The following tentative schedule is to be followed, and will only be modified with the approval of the supervisors.

Week	Date	Description
1	18.03. - 24.03.	Set up working environment
		Get to know BackLog
2	25.03. - 31.03.	Read relevant literature
3	01.04. - 07.04.	Design flexible, dynamic power management architecture
4	08.04. - 14.04.	Implement PM with EWMA
5	15.04. - 21.04.	Implement PM with WCMA
		Give initial presentation at TIK group meeting
6	22.04. - 28.04.	Implement PM with forecast
7	29.04. - 05.05.	Implement PM with forecast
8	06.05. - 12.05.	Implement PM for long-term sustainability
9	13.05. - 19.05.	Implement PM for long-term sustainability
X	20.05. - 26.05.	Reserved
10	27.05. - 02.06.	Simulate and evaluate with trace data
11	03.06. - 09.06.	Simulate and evaluate with trace data
12	10.06. - 16.06.	Write report
13	17.06. - 23.06.	Write report
14	24.06. - 30.06.	Give final presentation

Requirements

- 20 hours per week minimum (but quality over quantity).
- Weekly status meetings (on demand).
- Weekly status email containing progress of the week, outcome, problems encountered and solution proposal, work for following week.
- In addition to the above, the institute's guidelines apply [3].
- Sign plagiarism statement [4].

Relevant literature

Note that this list may not be exhaustive!

1. **Power/Energy Simulator:** N. Ferry, et. Al., "Power/Energy Estimator for Designing WSN Nodes with Ambient Energy Harvesting Feature," in EURASIP Journal on embedded systems, 2011.
2. **EWMA vs. WCMA comparison:**
   http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5184785
3. **EWMA:**
   - C. Vigorito, D. Ganesan, and A. Barto, "Adaptive control of duty cycling in energy-harvesting wireless sensor networks," in IEEE Conf. in Sensor, Mesh and Ad Hoc Communications and Networks (SECON'07), 2007.
   - A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor Networks," in ACM Transactions on Embedded Computing Systems (TECS'07) , vol. 6, no. 4, Sep. 2007.
4. **EWMA background:**
   - D. R. Cox, "Prediction by exponentially weighted moving averages and related methods," Royal Statistical Society , vol. 23, no. 2, pp. 414–422.
   - J. S. Hunter, "The exponentially weighted moving average," Quality Technology , vol. 18, no. 4, pp. 203–207, 1986
5. **WCMA:**
   http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5172412&tag=1
6. **Weather forecast:**
   - Paper: http://www.sciencedirect.com/science/article/pii/S0959152409001231

- API: http://stackoverflow.com/questions/3363052/best-weather-apis-free-for-commercial-use
- http://blog.programmableweb.com/2009/04/15/5-weather-apis-from-weatherbug-to-weather-channel/
- http://www.phpgangsta.de/mit-wetterdaten-arbeiten-wetter-com-api

Zürich, 18. March 2013

Bernhard Buchli

Contents

List of Figures

List of Tables

Abstract

The semester thesis reports on the findings of simulating a power management algorithm for the CoreStation platform. The goal of power management is to provide a reliable system with zero uncontrolled power outages, because of critical battery state of charge. Exponentially weighted moving average and weather-conditioned moving average prediction algorithms are employed for predicting harvested solar energy. A professional weather forecast service is implemented and investigated with the purpose of improving long-term stability. The power management algorithm reduces power consumption with calculating and updating the wireless LAN duty cycle in regular time intervals. Simulation results show that the choice of hardware, the battery capacity and solar panel power, plays a predominant role in system reliability, and determines the effectiveness of any software efforts.

Chapter 1

Introduction

Wireless sensor networks like the X-Sense project [5] are growing in size and complexity. Wireless sensors may operate at varying sample and data rates, the network might require user interaction. Base stations that connect these sensors together and gather their data must therefore have diverse and powerful functionality, but this often comes at the price of high energy consumption. The wireless base station of interest to this thesis, the *CoreStation platform* [1], has to operate under constrained conditions, yet still deliver sufficient functionality while avoiding sporadic power outages.

Figure 1.1: Locations of CoreStation deployments and positioning on the mountain face.

1.1 Motivation

The *CoreStation platforms* in the X-Sense project are deployed on various locations in the swiss mountains, e.g. Jungfraujoch and Matterhorn, at high altitudes (Figure 1.1). They rely only on a battery and solar energy harvesting for power. Harvested solar energy is dependent on weather conditions, mainly on the amount of illumination that the solar panel is exposed to. But the weather is often unpredictable and changes rapidly, and so does the amount of harvested energy. Consequently the battery *state of charge* (SoC)

fluctuates rapidly and unpredictably as well.

We would like our system to run unattended and uninterrupted, even with limited power. We can not rely on the battery and solar panel alone to provide just enough power for our system to be energy neutral. Some sort of power management software could be put into place to respond to current energy conditions and reduce the system's power consumption when needed, to avoid power outages.

1.2 Goals

The goal of the thesis was to implement a dynamic power management module for the CoreStation platform. The module first had to be simulated on a desktop PC to allow for fast testing. Two energy prediction algorithms, EWMA and WCMA, had to be implemented, as well as an interface to a professional weather forecast service, through an API (application programming interface). The power management module then had to use them and calculate the future wireless LAN duty cycle, as this is the largest power consumer on the whole platform. A new schedule had to be written for the platform, with the newly calculated duty cycle, and updated in the next time period. The final step was to evaluate the results of individual components.

1.3 Thesis overview

The semester thesis report is organized as follows: Chapter 2 offers the theoretical background on EWMA and WCMA prediction algorithms. Chapter 3 shows how the algorithms are implemented and how parameters of WCMA are optimized for performance. This chapter also explains the implementetion of a professional weather forecast service, using an API, and calculation of a weather scaling factor. Chapter 4 talks about power management, lists different approaches to duty cycle calculation and reviews the results of simulations with these approaches. The evaluation of results in the thesis is written in chapter 5, and chapter 6 gives conclusions about the thesis and an outlook on further work. Segments of code from energy prediction algorithms, parameter optimization, weather API and power management are located in the appendix.

Chapter 2

Theory

Power management software running on a real system must not only give output to the system, but also take input from it, such as measurements of the state the system is in. But measurements only give us the current state, the condition the system (mainly the battery charge) is in right at this moment. If we want to save battery charge by reducing power consumption, we need to set the duty cycle of our biggest power consumer, the WLAN, for the next time interval based on measurements done in the current time interval.

It would make sense to introduce prediction algorithms at this point, since this is exactly what they do. A prediction algorithm predicts an output variable quantity in the next time interval based on measurements or inputs from the current time interval. We will look at two prediction algorithms, that are going to be used to predict harvested solar energy.

It must be noted, that we are making a simplification. We are assuming that we know the harvested energy and can measure it accurately, but in reality, the CoreStation has no dedicated hardware to measure energy taken from the solar panel. The only way we can calculate harvested solar energy is with battery SoC. Calculations can be incorrect, when battery SoC reaches 100% and saturates, but solar energy is still being harvested. This energy goes to waste, as it does not recharge the battery. The system should recognize the real amount of energy being harvested however, so a power management algorithm can set the appropriate power usage of the system. If we know the power consumed by the station and the difference in the battery state of charge, we can calculate the amount of energy harvested in the current time interval

$$\begin{aligned} SoC(n) - SoC(n-1) = \delta SoC = E_g - P_{consumed} \cdot T_{interval} \\ E_g = \delta SoC + P_{consumed} \cdot T_{interval} \end{aligned} \tag{2.1}$$

where the battery states of charge $SoC(t)$ and $SoC(t-1)$, the difference in battery state of charge δSoC, and the harvested (or generated) solar energy E_g are energy values in (Wh), $P_{consumed}$ is the platform power consumption in (W) and $T_{interval}$ is the time interval in (h).

2.1 Exponentially Weighted Moving Average

Exponentially Weighted Moving Average, or EWMA, has long been used [6] for data prediction because of its simplicity and relatively good accuracy. When implemented on a platform like the CoreStation, it has low complexity and low computational intensity since it is run only once per time interval, which can be once per 30 minutes up to once per day.

Equation 2.2 explains how it works:

$$E_{pred}(n+1) = \alpha E_{meas}(n) + (1-\alpha)E_{pred}(n) \tag{2.2}$$

$E_{pred}(n+1)$ is the energy prediction for the following time interval, $E_{meas}(n)$ is the current measurement (true value) and $E_{pred}(n)$ is the energy prediction for the current time interval, which we made in the previous interval. All value units are (Wh). α is a scaling factor, which determines how much the current value or past predictions influence our current prediction. It can have any value in the interval $[0, 1]$. For a large α, past predictions influence less, and we value the current measurement more, but for a smaller α, we value past predictions more and the current measurement less.

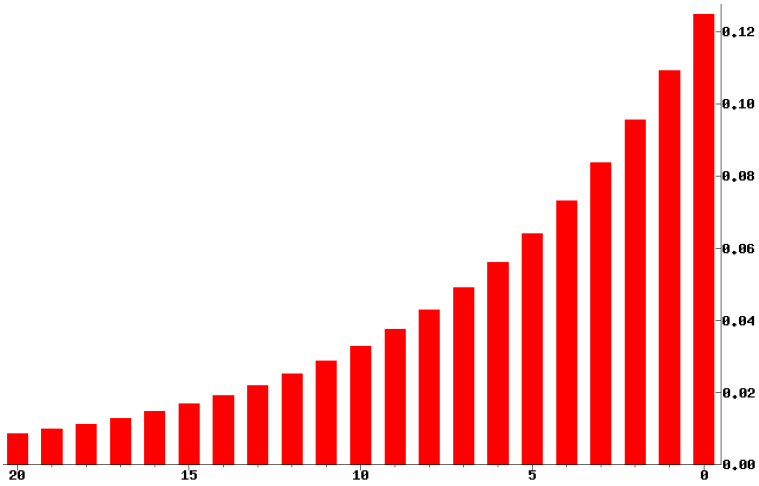


Figure 2.1: Graph, showing the weights of samples for successively older time intervals.

Figure 2.1 shows the weights of past predictions and how much they influence the current prediction. With each time interval, the sample contributes less to the current EWMA prediction. Because of this, the predictions are slow to respond to quick changes, such as night/day solar energy changes.

2.2 Weather-Conditioned Moving Average

Weather-Conditioned Moving Average, or WCMA, takes into account also weather changes [7]. It is based on EWMA, but is more complex and computationally intense, when implemented. It shows faster responses to night and day changes and even adapts sunrise

and sunset times and seasonal variations. Figure 2.2 and Equations (2.3) demonstrate, how the algorithm works.

$$E(d, n+1) = \alpha \cdot E(d, n) + (1-\alpha) \cdot GAP_k \cdot M_D(d, n+1)$$

$$\begin{aligned} M_D(d, n+1) &= \frac{\sum_{i=d-1}^{d-D} E(i, n+1)}{D} & GAP_k &= \frac{V \cdot P}{\sum P} \\ P &= [p_1, \ldots, p_k, \ldots, p_K] & p_k &= \frac{k}{K} \\ V &= [v_1, \ldots, v_k, \ldots, v_K] & v_k &= \frac{E(d, n-K+k)}{M_D(d, n-K+k)} \end{aligned} \tag{2.3}$$

The variables d and n signify the day, or sample in the day, respectively. d is the current day, n is the current sample, so *(d,n+1)* would mean a sample in the current day and the next time interval (current prediction). There are *N-samples* in a day and we keep a history of at least *D-days* and *K-previous samples* for calculations. Therefore, WCMA needs $(N * D + K - 1)$ historical samples to predict a value in the next time interval.

Contrary to EWMA, WCMA uses only real values, past measurements, to calculate the prediction in the next time interval. $E(d, n+1)$ is the predicted energy value, shown in green in Figure 2.2, $E(d, n)$ is the current energy measurement, shown in brown, both in (Wh) α is a scaling factor, similar to the one in EWMA. $M_D(d, n+1)$ is a mean value of the samples at $(n+1)$ for the past days $(d-1)$ to $(d-D)$. The innovation in WCMA is the additional scaling factor GAP_k, which gives weather information about the current day. It compares samples $(n-K+1)$-to-(n) in the current day d, to the mean value of the same samples in previous days. This is done by *vector* V and its values v_k. These tell us if, the weather today is better than the previous days, there will be more harvested energy and the factor GAP_k will be greater than 1, but if we are getting less energy, the factor is less than 1. The vector P weighs the samples according to relevance by time.

The WCMA algorithm has four parameters, that we need to consider :

1. Scaling factor α: value from 0 to 1.
2. Number of days in the calculation D. In our demonstration, D=4.
3. Number of previous samples K. In our demonstration, K=4.
4. Number of samples per day N. Our system is set up for hourly values, N=24.

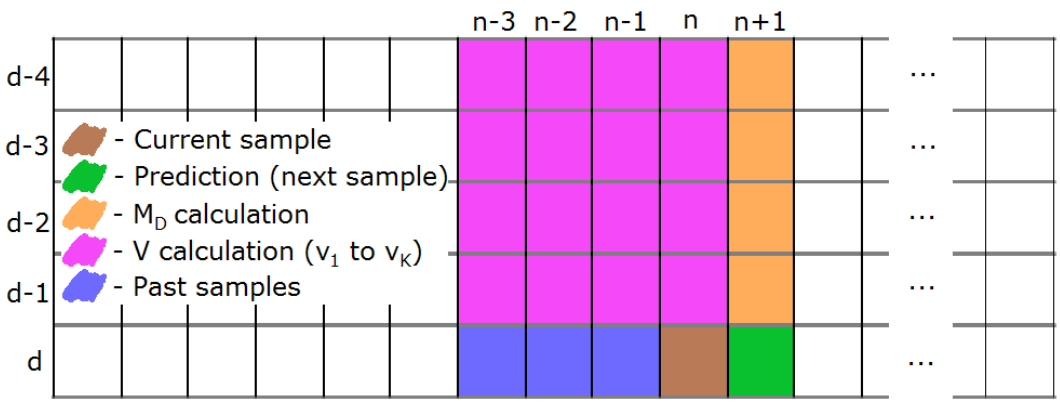


Figure 2.2: A schematic showing the working of WCMA. The white boxes represent measured energy values for individual time intervals. There are $N = 24$ samples per day and the algorithm needs a history of values for $D = 4$ days and $K = 4$ samples. n is the current sample of the current day d.

Chapter 3

Energy prediction

In the previous chapter, two energy prediction algorithms have been introduced. Now, let us see them being used on simulated harvested energy data. This data is a set of harvested energy values (in Wh) in one hour intervals over 12 years. Night and day values are clearly distinguishable, as are seasonal variations. Figure 3.1 shows a subset of this data for three days in mid-March. The real values of the harvested energy are shown in blue, these represent the ground truth. EWMA prediction is shown in red and WCMA is shown in green.

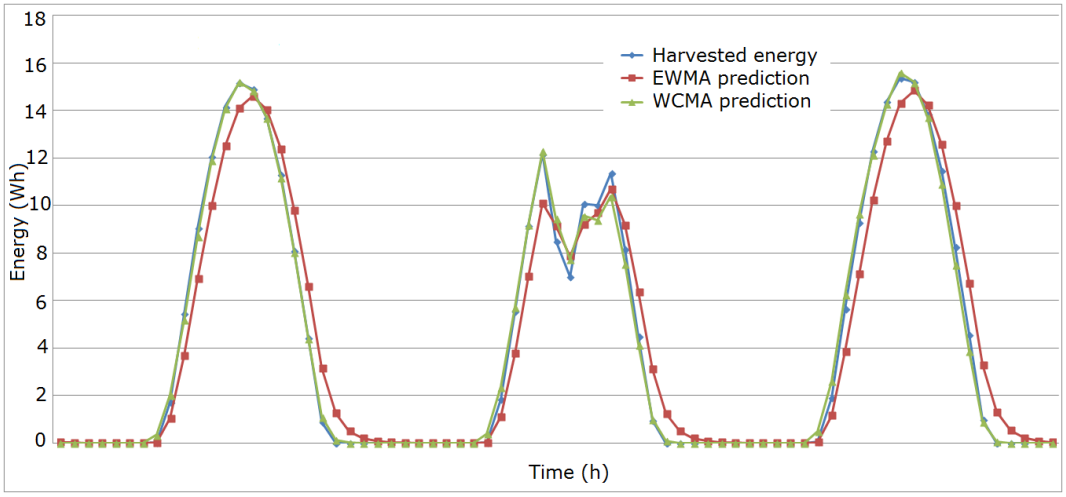


Figure 3.1: EWMA and WCMA prediction algorithms plotted against measured values of harvested energy. Samples are spaced in one hour time intervals, units are Wh.

3.1 EWMA

The only parameter that we must set is the scaling factor α. After looking at related work in articles and publications [7, 8, 9, 10, 11], an $\alpha = 0.6$ was chosen. The outline of the plot

matches the ground truth very well, but the actual values vary from the measurements like they are ”lagging behind”. The curve looks like it is shifted to the right. The benefit of implementing this method on a real system is its simplicity and low computational intensity, even though we have mismatch.

3.2 WCMA

This prediction method shows in Figure 3.1 that it responds to a fast changing data set superiorly to EWMA. It has less ”lag effect”, because is uses a diurnal correlation of energy samples and is able to determine the approximate value of the harvested energy based on previous days. This however means a long initialization period, as the first prediction can only be made once we have a sufficient $(N * D + K - 1)$-number of values. If we are predicting hourly values, it takes the full $D - days$ plus $K - hours$ (in our case 99 hours), but if we are only running predictions for daily values, this means 99 days and might not be acceptable for power management.

We need to consider this trade off: since WCMA is complex and computationally intense, making only daily energy predictions might be a good way to save valuable computing power, but the algorithm has proven to be less successful, since there is no more diurnal correlation between samples, and therefore D and N have no more meaning. Making several observations per day, in our case once per hour, gives much more precise predictions, even more precise than EWMA, but we are possibly straining the resource constrained system which has only limited computing time available for power management.

There are also three parameters to consider and optimize when implementing WCMA on a system: D, K, and α. For our system, N is set to 24 in all cases.

3.2.1 Parameter optimization

The optimization process in this thesis is similar to the one described in article [7]. The error function is given by Equation (3.1).

$$Err = \frac{1}{N} \sum_{i=1}^{N} abs(\frac{E_{Real}}{E_{Pred}} - 1) \tag{3.1}$$

Err is the relative error, E_{Real} and E_{Pred} are the ground-truth and WCMA predictions respectively, in (Wh).

We calculate the mean daily error, discarding ”night values”, for each day of the year. *Article* [7] defines *night values* as energy values that are less than 10% of the maximum. We define *night values* as values that do not exceed 10% of the maximum energy value *for that day*, since energy values of sunny and cloudy days can vary by as much as a factor of 3, and so can summer and winter days. Therefore it's important to only discard night values that correspond to energy values for the individual day. The sample count N is reduced accordingly.

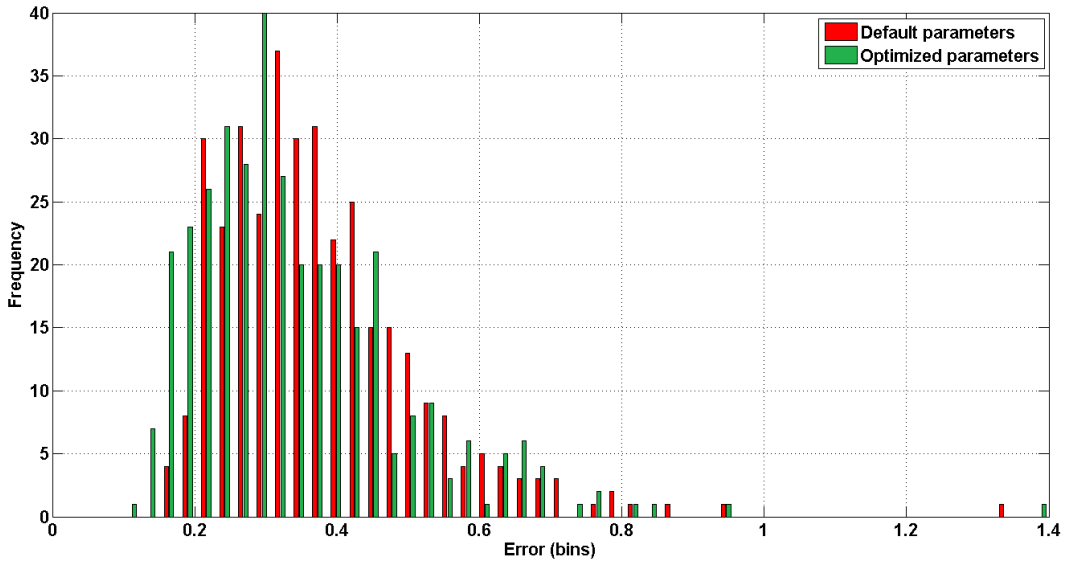


Figure 3.2: Histogram of relative errors between measured values and WCMA predictions for two different parameter sets. "Default" parameters, that article [7] claims are optimal are $\alpha = 0.7$, $K = 3$, $D = 4$. "Optimized" parameters are a result of the optimization process performed during this thesis and are $\alpha = 0.5$, $K = 1$, $D = 9$. $N = 24$ in both cases and the histogram bins are of width 0.025.

If we look at the histogram of relative errors (Figure 3.2), we can see, that the optimization of parameters based on the smallest *maximum error* among the parameter combinations might not be the best option, as claimed in article [7], since there is only one occurrence of such a large error, and we would like to minimize the "overall" error of the WCMA prediction. Therefore in this thesis, a minimum of the *mean error* among the parameter combinations determines the optimal parameters.

Histogram in Figure 3.2 shows the comparison between relative errors of our optimized parameters in green, and the parameters optimized in the article, in red.

In the first step of the optimization, we look at mean error values for combinations of parameters α from 0.1 to 1 in intervals of 0.1, and D from 1 to 10, with the default $K = 3$. The results are plotted in Figure 3.3. The optimized α is 0.5 and the optimal D is 5. Since the values for different D are very similar, we make another run with this parameter, together with K.

The results of the second step are plotted in Figure 3.4. Here, we look at error values for combinations of D from 2 to 11 and K from 1 to 10, with optimal $\alpha = 0.5$. We can see that the function has a minimum at $K = 1$ and $D = 9$. These are our final optimized parameters, with which we will use the WCMA prediction algorithm in the power management software.

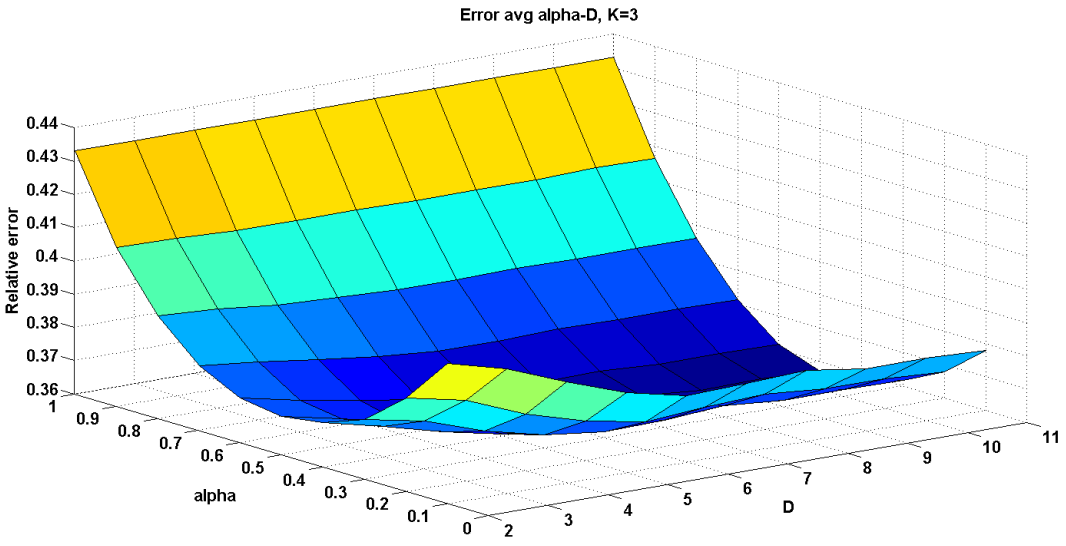


Figure 3.3: 3-D plot of relative error size for combinations of WCMA parameters α and D with constant $K = 3$.

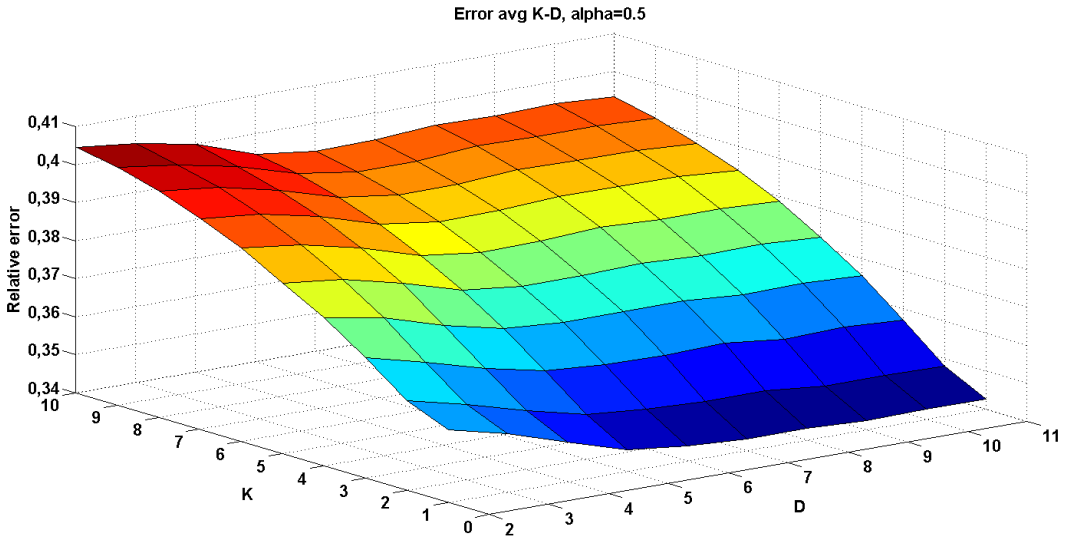


Figure 3.4: 3-D plot of relative error size for combinations of WCMA parameters K and D with constant $\alpha = 0.5$.

3.3 Weather Forecast

Since the prediction algorithms are based on historical values, they provide only limited accuracy in calculating the harvested energy of the next time interval. More importantly, they are accurate on a short time scale. But this does not guarantee the long term reliability that we require for our autonomous system. A professional weather forecast should improve this by providing the system with information on weather conditions and consequently harvested energy for several days in advance.

In this project, we used *OpenWeatherMap.com* weather forecast service and web API to provide our system with weather data such as precipitation, cloudiness and temperature. The API offers a more precise 5-day forecast with data in three hour intervals. It also offers a more general 14-day forecast with daily data. The service offers weather forecast from a multitude of weather stations around the world, for any location specified by geographic coordinates (longitude and latitude).

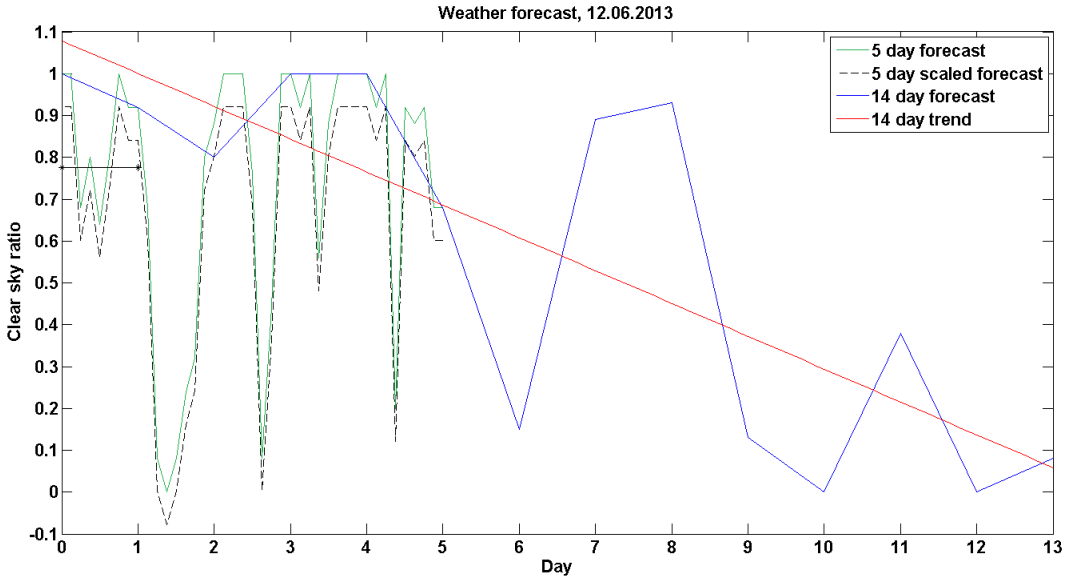


Figure 3.5: Plot showing OpenWeatherMap weather forecast data for July 12th, 2013. A 14-day forecast for sky clarity percentage ($= 100\% - \%cloudiness$) is plotted in blue and a linear fit is applied to it in red. A 5-day forecast is plotted in green and the scaled 5-day forecast in dashed black. The solid black line indicates the value of the weather scaling factor (WSF).

Since we introduced a weather forecast in trying to improve the long term stability of our system, we try to determine the progression of our harvested energy in the following days and incorporate this knowledge into our decision making process in the power management. We can determine the energy available for harvesting with one particular weather parameter: cloudiness. A simple approach was taken in this thesis, where 40% clear-sky ratio (the same as 100%"*minus*"60%*cloudiness*) means that only 40% of the

energy can be harvested on that day, compared to the full energy on a sunny day with 100% clear sky.

Figure 3.5 is a plot demonstrating both the 14-day clear-sky forecast (in blue) and the 5-day clear-sky forecast (in green). The harvested energy progression, or *trend*, is determined by simply applying a linear fit to the 14-day forecast (red in Figure 3.5). This figure shows a negative trend, which means the weather is going to be worse in a couple of days and the power management should start conserving energy despite having a possible surplus of harvested energy with the current WLAN duty cycle at the moment, because the harvested energy is going to decrease with the weather conditions.

The 5-day forecast shows a more precise picture of the weather conditions and the potentially available harvested energy. We introduce the *weather scaling factor*, or *WSF*, which is calculated as a mean of the values for the following day, from the 5-day forecast and then scaled by the 14-day trend. The final WSF is plotted with a solid black line with stars.

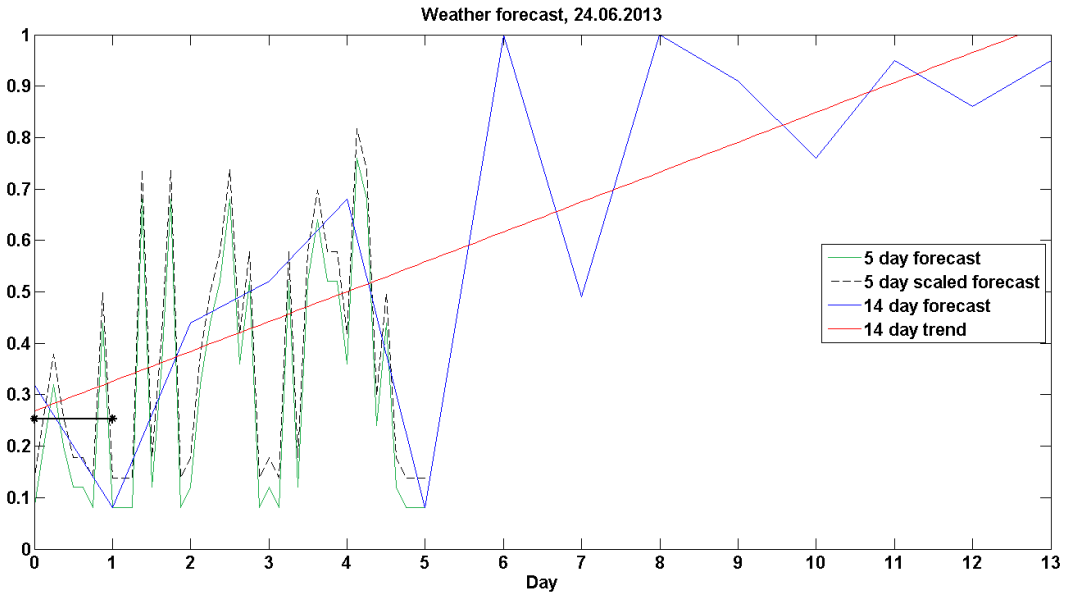


Figure 3.6: Plot showing OpenWeatherMap weather forecast data for July 24th, 2013. A 14-day forecast for sky clarity percentage ($= 100\% - \%cloudiness$) is plotted in blue and a linear fit is applied to it in red. A 5-day forecast is plotted in green and the scaled 5-day forecast in dashed black. The solid black line indicates the value of the weather scaling factor (WSF).

Figure 3.6 is an example, where the 14-day trend is positive and therefore the 5-day forecast and WSF are scaled up. The WSF is ultimately used to correct the duty cycle of the WLAN.

Because of time constraints, the weather forecast could not be tested effectively. Contrary to harvested energy, which can be generated or old recorded values could be used

for simulation purposes, a weather forecast needs to be real, not generated, otherwise our power management has little meaning and is dependent on the generated forecast. It would take a long time to test the weather forecast with real CoreStation platforms and real data, and would require additional changes to the BackLog core functionality. This surpasses the scope of this semester thesis, so the weather forecast could not be evaluated.

Chapter 4

Power management

Predicting the harvested energy is only a building block in the entire power management. What is missing, is the calculation of the WLAN duty cycle, which will actually reduce the power consumption and drainage of the battery. WLAN was chosen for duty cycling, because it is the most power-hungry part of the CoreStation. A simple approach to modeling and simulating the CoreStation was employed.

For purposes of this thesis, the system consists of three parts. The battery, acting as a buffer, has a nominal voltage of $12V$, a nominal capacity of $34Ah$ but the energy available is only eighty percent of the maximum: $80\% * 12V * 34Ah = 326.4Wh$. The CoreStation is the power consumer and it is split into power consumed by the WLAN, $P_{WLAN} = 12V * 250mA = 3W$, and the power consumed by the station itself, reffered to as "base power", $P_{base} = 12V * 200mA = 2.4W$. The solar panel provides power to the battery.

Calculating the duty cycle of the WLAN was not as trivial as first assumed. Several approaches were tested to see which one performs best. The criteria were: highest average WLAN duty cycle, which gives the most functionality to the system, and uninterrupted operation without power outages. Here are short descriptions of differences in duty cycle calculation for the power management versions. Full code is in Appendix A.

- Version 1.0:
  The harvested energy was used in the WLAN duty cycle calculation. CoreStations in our systems do not have hardware for energy measurements and therefore do not have access to the harvested energy. Only we do for simulation purposes.
- Version 2.0:
  The harvested energy is calculated from the battery state of charge and used to get duty cycle.
- Version 3.0:
  Same DC calculation as in 2.0, but the maximum DC is not 100%, rather the current percentage of SoC (unless this is smaller than minimal).

- Version 4.0:
  Same DC calculation as in 2.0. The system is shut down during the night time, when the harvested energy is zero. This reduces critical battery shutdown, but does not eliminate it. The overall system down-time is always over 50%, thus functionality is unacceptably reduced by concept. This approach was simulated, as it is sometimes being used in practice on real systems.

- Version 2.1:
  Same as 2.0, but only change the duty cycle once per day. This stabilizes the rapid DC toggling present in other versions with hourly DC changes.

- Version 2.2:
  Same as 2.1, but the DC can only change in finite increments, smaller than 25%.

There are several parts to all versions of the power management. The system is simulated in different conditions to observe the behavior of the algorithm.

- No wlan - Used as comparison for all versions. There is no WLAN power consumption, only CoreStation base consumption. This determines the best case scenario in terms of power requirement and drainage.

- No duty cycle - WLAN is on all the time (100% duty cycle). Also for reference and comparison. This determines the worst case scenario in terms of power requirement.

- Simple duty cycling - Energy conditions are thought to be constant. No prediction is used. The error we are making is the same as the change in conditions from the last time interval to the current one.

- EWMA predictions - Duty cycle is calculated using this prediction algorithm.

- WCMA predictions - Duty cycle is calculated using this prediction algorithm.

To simulate conditions on a real system, there is a critical battery shutdown at $SoC = 5\% * E_{bat,max} = 16,32Wh$ and a reconnect hysteresis at 60% SoC. There is also a minimum duty cycle for the WLAN of 10%, because the system needs some functionality and must be observable and controllable.

Table 4.1 gathers results of all the power management versions. We observe the number of hours, the system spends in shutdown and the average duty cycle while the system is on-line.

Version 2.1 of our power management has the best results. The duty cycle is only changed once per day. This was implemented, because the duty cycle was toggling rapidly in other versions and changing the battery state of charge more violently than necessary. This approach stabilizes the power consumption over one day, while also reducing the computational intensity from the prediction algorithms and providing better long-term results.

Version	No WLAN	100% DC	Simple DC	EWMA DC	WCMA DC
1.0	1302 (14.8%)	3874 (44.1%)	2133 (24.3%) 38.5%	2056 (23.4%) 37.2%	2019 (23.0%) 36.0%
2.0	1302 (14.8%)	3874 (44.1%)	2021 (23.0%) 37.0%	2047 (23.3%) 38.3%	2109 (24.0%) 38.3%
3.0	1302 (14.8%)	3874 (44.1%)	1816 (20.7%) 31.4%	1826 (20.8%) 32.7%	1888 (21.5%) 32.7%
4.0	4104/0	4760/656	4269/165 56.6%	4214/110 55.3%	4150/46 52.2%
2.1	1302 (14.8%)	3874 (44.1%)	1962 (22.3%) 48.0%	1783 (20.3%) 45.9%	1873 (21.3%) 47.1%
2.2	1302 (14.8%)	3874 (44.1%)	1960 (22.3%) 48.6%	1807 (20.6%) 46.6%	1908 (21.7%) 47.8%

Table 4.1: Results of different power management script versions. Numbers in the top left corners are hour counts (and percentages in parentheses) of the CoreStation being forced into critical battery shutdown. The total number of hours in the year is $24 * 366 = 8784$. Version 4.0 is different and shows rather the total number of hours off-line and the number of hours off-line because of critical battery respectively, separated by a forward slash. The percentages in the bottom right corners of cells are the average duty cycle values throughout the year.

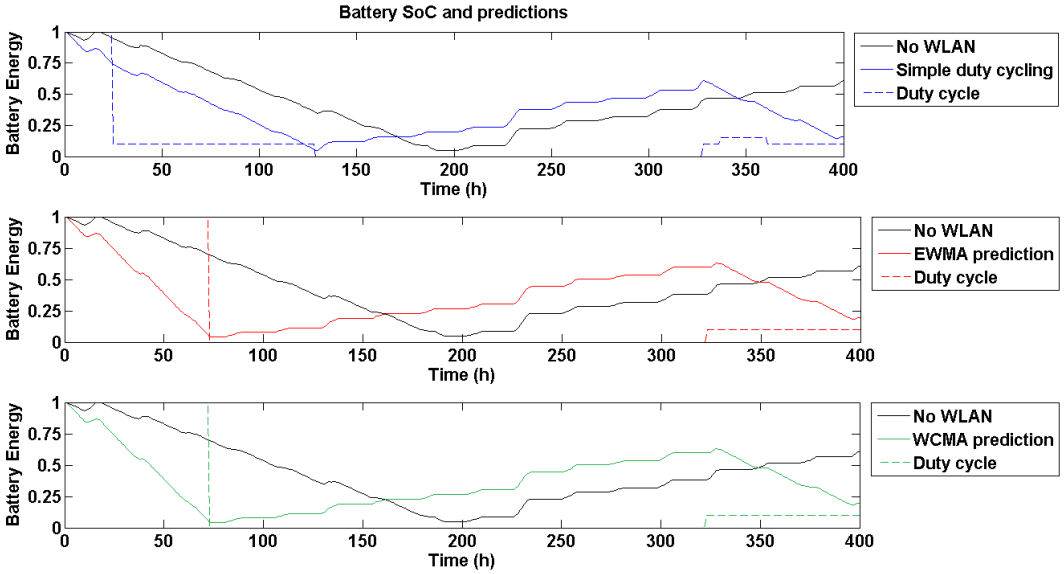


Figure 4.1: Battery state of charge (in Wh) in solid lines. Simple duty cycling calculated based on current conditions and not on prediction (blue), EWMA prediction duty cycling (red), WCMA duty cycling (green) and only base power consumption, with no WLAN (black, all subplots). Dashed lines represent duty cycle values for respective approaches. Time axis shows the time of year (in hours) starting from January 1st (hour 0).

But because we are dealing with daily changes, the WCMA prediction algorithm takes a very long time to initialize, several months in fact. While waiting for historical samples to accumulate, we use EWMA prediction for this time period rather than leaving the system without power management. Figure 4.1 shows all three duty cycling approaches and the "no WLAN" plot in black, for comparison. WCMA is the same as EWMA until it is fully initialized. No matter what the duty cycling approach though, the battery inevitably drains below critical.

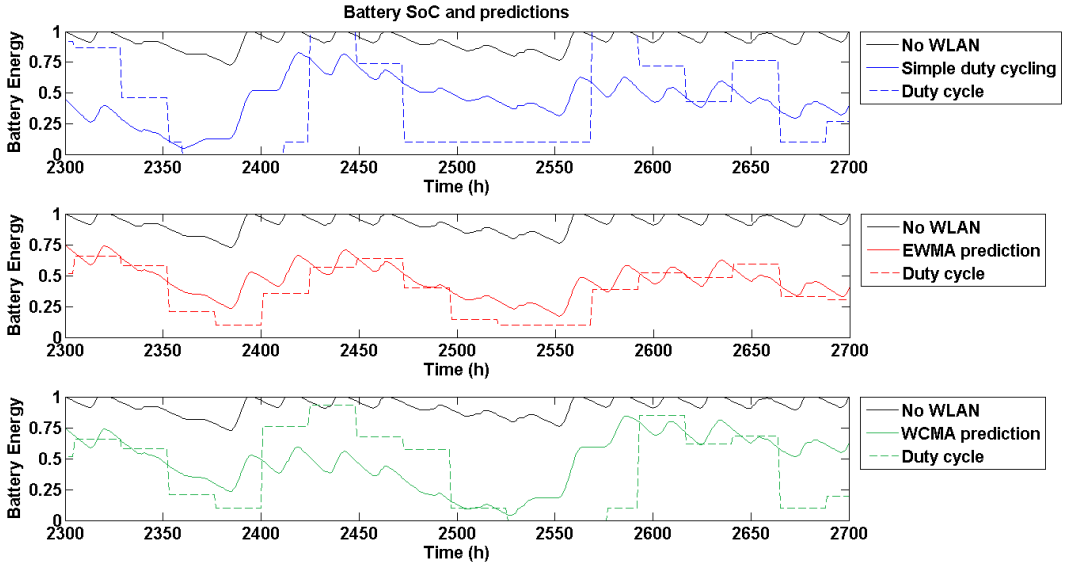


Figure 4.2: Battery state of charge (in Wh) in solid lines. Simple duty cycling calculated based on current conditions and not on prediction (blue), EWMA prediction duty cycling (red), WCMA duty cycling (green) and only base power consumption, with no WLAN (black, all subplots). Dashed lines represent duty cycle values for respective approaches. Time axis shows the time of year (in hours) starting from April 6th (hour 2304).

Figure 4.2 shows the same simulations, but in the $2300th$ hour (April 6th). WCMA is fully initialized on the $100th$ day and starts to differ from EWMA. It can be observed that WCMA performs worse than EWMA and actually forces the system into shutdown. This might be due to the fact, that WCMA prediction loses diurnal sample correlation when calculating only daily values, and performs worse than EWMA.

None of the power management versions could achieve zero power outages. Even with no WLAN, our base station consumes so much power, that the battery inevitably drains below the critical point and the system forces shutdown. This means that our system is incapable under any circumstances of running uninterrupted. The next step was to determine, what kind of system would in fact be able to run uninterrupted. In other words, how large does the battery need to be, to support such power consumption, and what is the *maximum, long-term, average duty cycle* of the WLAN, that our solar panel can support in a long time-span, provided the battery has enough capacity.

This was explored in power management version 2.3. It is similar to v2.1, but has an over-sized battery of $7500Wh$, which is just enough to buffer the seasonal changes in harvested energy. V2.3 simulates results over 12 years of harvested energy data, so the long-term effects on the system can be observed. The duty cycle is updated once per day, but is upward-limited to the average duty cycle value of the last 30 days, not to 100%. This decision was made, because the system can only be stable over a long period (many years), if the cumulative energy consumption is lower or equal to the cumulative harvested energy of the system. If we subtract the constant base consumption from the 12-year average of the harvested energy, we are left with the energy available for the duty-cycled WLAN. In our system, this is equal to $P_{remaining} = 0.6W$ of power. Because WLAN requires $3W$ of power, the maximum, long-term, average duty cycle is exactly 20%.

Figures 4.3, 4.4, 4.5 show simulation results for power management version 2.3. The normalized battery state of charge, as a result of a constant duty cycle of $DC = 20\%$, is shown in blue (Figure 4.3). EWMA calculated duty cycle (black) and the resulting battery state of charge (red) are shown in Figure 4.4. WCMA calculated duty cycle (black) and the resulting battery state of charge (green) are shown in Figure 4.5. The average EWMA duty cycle is 17.11% and 16.58% for WCMA.

These two results indicate that there is a possibility to have a working power management algorithm, but only if our system is augmented by increasing the battery size dramatically. EWMA performs better than WCMA, because it has a higher average duty cycle, but both average duty cycles are below 20%, which is the upper-bound for our system. The prediction algorithms do make a difference on the system, as can be seen by the seasonally changing duty cycle. The system tries to conserve energy during the winters by decreasing the duty cycle to the minimum, and exploits the excess energy in the summer, which is apparent by the rising of the duty cycle above 20%. This means the battery does not drain as low as it does with the constant duty cycle approach in the winter, stabilizing the SoC more.

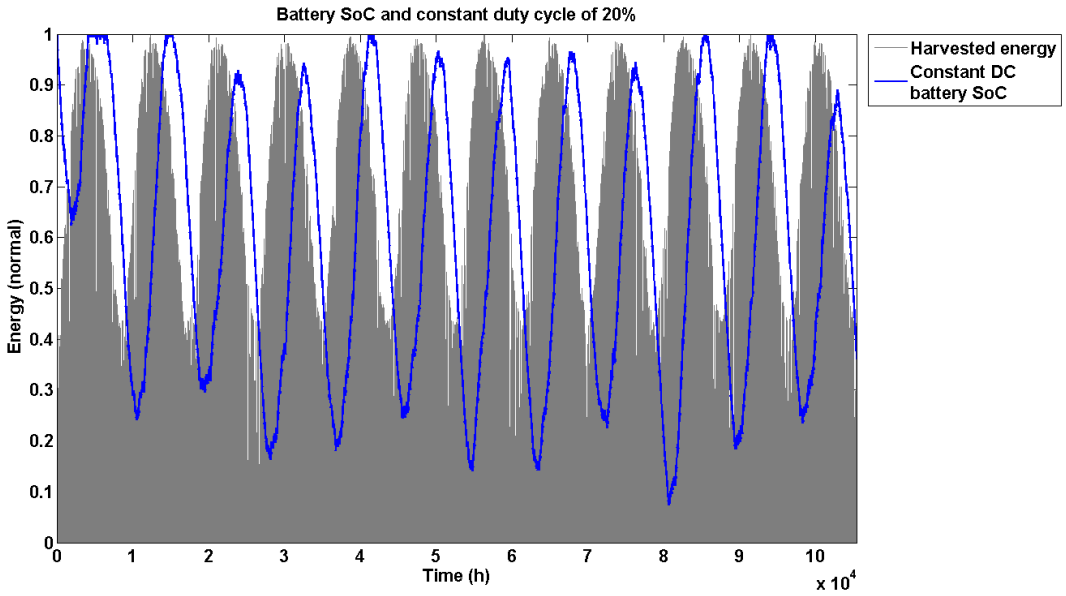


Figure 4.3: Battery state of charge for a constant WLAN duty cycle of 20% (blue) and harvested energy data for 12 years (grey), both normalized. The battery capacity is changed to 7500Wh.

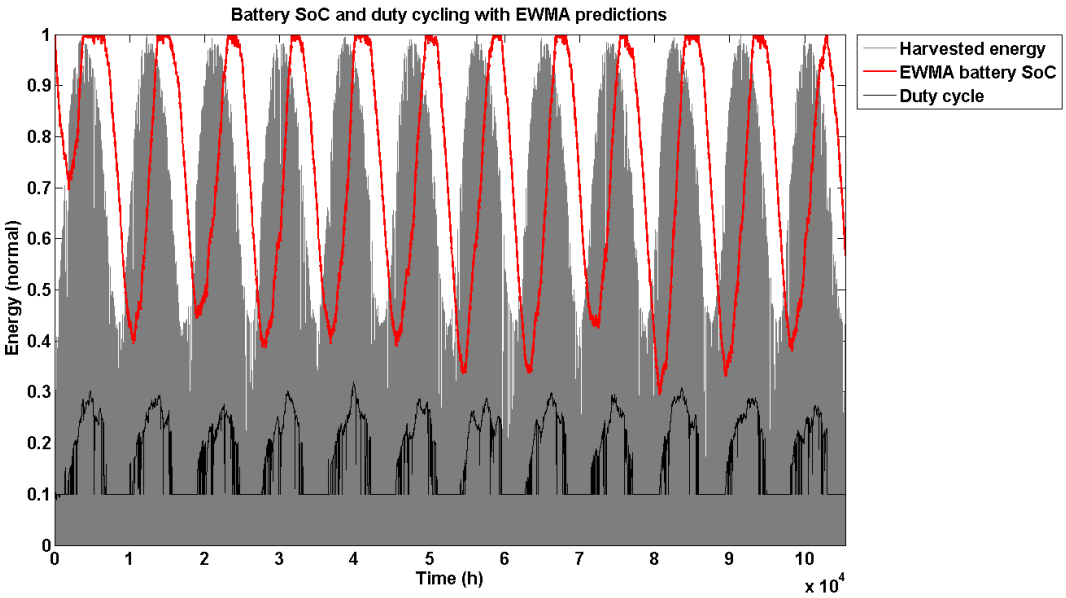


Figure 4.4: Battery state of charge for EWMA duty-cycling (red) and harvested energy data for 12 years (grey), both normalized. The black curve shows the duty cycle value. The battery capacity is changed to 7500Wh.

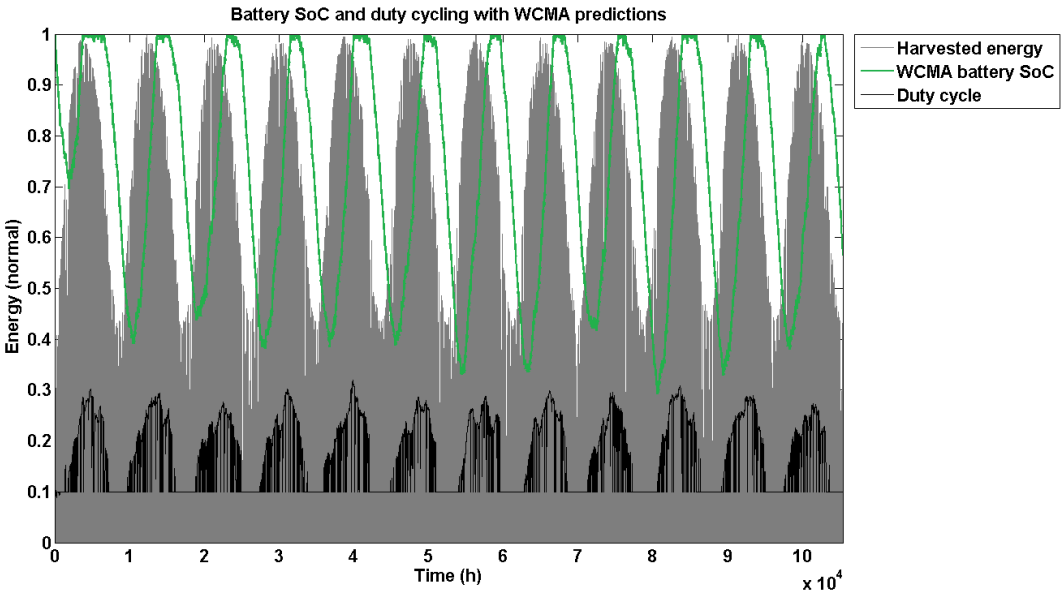


Figure 4.5: Battery state of charge for WCMA duty-cycling (green) and harvested energy data for 12 years (grey), both normalized. The black curve shows the duty cycle value. The battery capacity is changed to 7500Wh.

Chapter 5

Evaluation

Both prediction algorithms are fairly accurate when using and predicting hourly values of harvested energy. The mean value of the relative error for WCMA with optimized parameters is 34.2%, the mean for WCMA with default parameters is 37.5%, and for EWMA it is 53.9%. The error histogram is shown in Figure 5.1. We can see that WCMA performs better and that parameter optimization improves performance, although by 3.3%, but this improvement does not necessarily impact the entire power management at all.

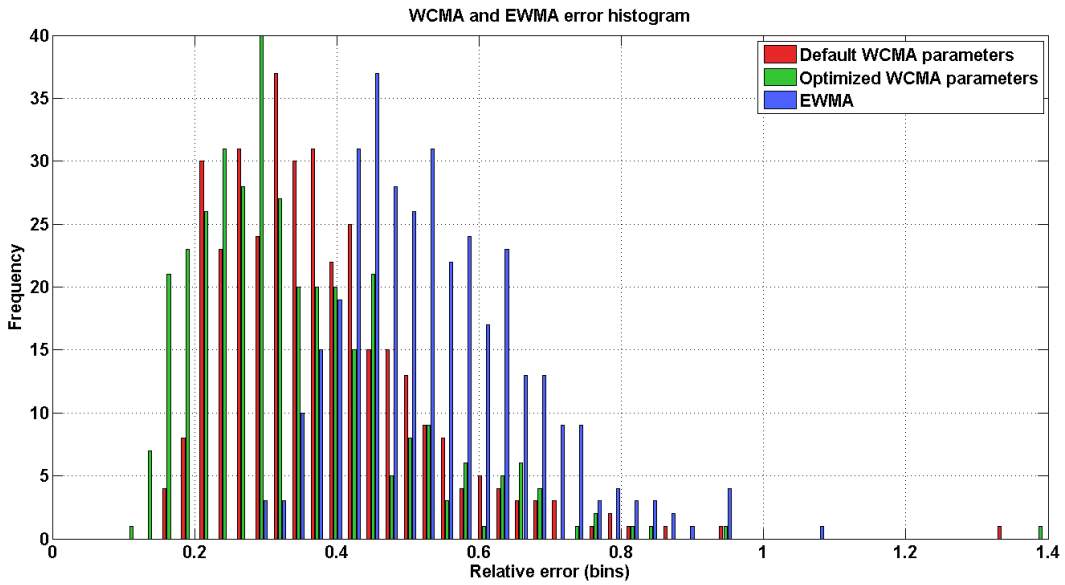


Figure 5.1: Error histogram showing the distribution of relative errors of WCMA (default parameters in red, optimal parameters in green) and EWMA (blue) prediction algorithms.

Professional weather forecast should be implemented and tested on a real system together with energy prediction to determine its impact on power management and long-term stability of the system. Since the main problem of power management with energy

prediction alone is the scope in which the prediction takes place, weather forecast would probably have a larger impact on the system. Because prediction algorithms can predict only one value in advance, the power management can not be made very sophisticated and stable over longer time periods, since the changes in the duty cycle are made either during the rising and setting of the sun when hourly values are used, or when the weather is changing locally, on a timescale of a couple of days, when daily values are used. The power management software will drive the WLAN harder in case the system sees a short excess of energy, but can not predict a possible long period of bad weather, that will drain the battery even with intense power saving. The battery might drain below critical in such a case. On the other hand the power management software can not predict a possible period of intense sunshine in the summer, for example, and will try to conserve power in the cloudy days preceding this period, resulting in the saturation of the battery charge at 100% for a sizable amount of time, losing the surplus harvested energy even if the duty cycle is at 100% constantly.

One solution would be to pre-program an average (or maximum) value of the harvested energy into the system, taking into account several years of statistical measurements. This would give the power management algorithm information about seasonal changes in harvested energy. But the weather can be very unpredictable and vary greatly, so our system would have to be over-sized to compensate for a lack of accuracy of this method. It would also be difficult to produce from statistical data, values to input into the power management algorithm, that are significant for high alpine environments.

Although trying out several approaches of calculating the duty cycle, none of the power management versions could achieve zero power outages, even with minimal power requirements of $P_{base} + 0.1 * P_{WLAN}$. Because the system has a reconnect hysteresis that requires the battery to charge above 60% SoC after it has been discharged below critical, the power outages last even longer. In this time, the station is inoperative. If the reconnect hysteresis were smaller, the system would be able to reboot faster and perhaps run the WLAN at least at minimal duty cycle, providing minimal required functionality. This might force the system into a new critical battery shutdown even sooner than with a larger reconnect hysteresis. The reduced down-time might not even make a difference, because we have not achieved our goal of uninterrupted operation. But since the system is going to fail inevitably, it might make sense to try and reduce the down-time anyway.

There is another mechanism spoiling our simulations. There are two possibilities for what kind of data our prediction algorithms receive during system shutdown: no SoC (and consequently harvested energy) data, or false valued data. Also, after the system is running again, the difference in SoC, from which the harvested energy is calculated, is enormous. This makes the duty cycle too big right after reboot, and the system drains more power than ideal for the harvested energy conditions. This contributes to faster battery draining and reaching a new critical battery shutdown sooner.

Chapter 6

Conclusion

Any power management effort is unsuccessful in providing 100% reliability to our system. The hardware simply does not allow it. Simulations in power management version 2.3 have shown that only an unfeasibly large battery capacity, compared to the one used with real CoreStations, can buffer energy surplus and deficit associated with seasonal variations in solar conditions.

The battery capacity and solar panel power must be determined properly according to the type and strength of the load. Only then can any kind of power management effort improve system reliability and performance. If the power hardware is under-scaled, like in our case, not even aggressive duty cycling will guarantee zero power outages, and the best we can do is to try and minimize the system down-time with more complex and elaborate prediction algorithms, weather forecast and power management schemes.

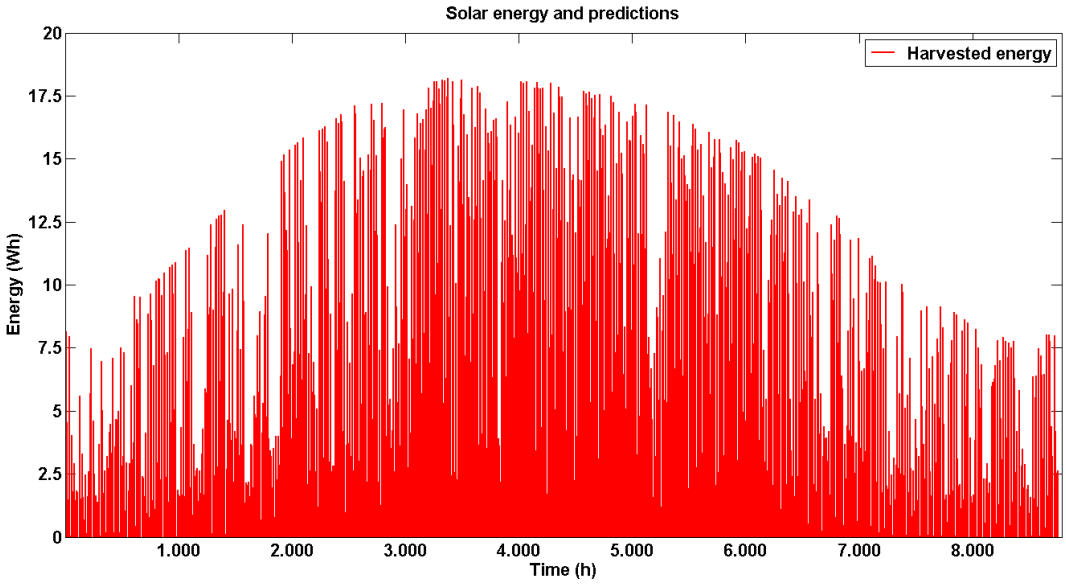


Figure 6.1: Harvested energy values in 1h intervals for 366 days.

Figure 6.1 shows one year of hourly harvested energy values. The mean value of harvested energy allows the system to run the WLAN at an average duty cycle of 20%. Since the moving average of the harvested energy is a sinusoid-like curve, approximately half of it is under the mean value and half of it is above the mean value. This means that for approximately six months, our battery will be drained on average, and for the other six, it will be charged, as can be seen in Figures 4.3, 4.4, 4.5. The weather variations however might shift this to 7 months of draining and 5 of charging, or move these periods one month earlier or later. This can not be predicted. In the end, the overwhelming factor in the reliability of the system is the choice of hardware. Statistical weather data, experience and an "educated guess" can help determine the battery size and solar panel power.

6.1 Outlook

Even though our system can not be 100% reliable, we can still try and improve performance and decrease down-time. The choice of energy prediction algorithms has an impact on the system and if WCMA is chosen, the parameter optimization could be further investigated together with its effects on the overall power management. The optimal parameters might be dependent on the input data or the order in which we optimize the parameters. This can be investigate further.

An important part of this thesis, the weather forecast, could be implemented on an actual system and tested in another thesis. This would require additional changes to BackLog and implementation of code into the core functionality, as well as longer testing periods.

Version 2.3 of the power management algorithm is able to provide system reliability over a period of 12 years with both EWMA anc WCMA prediction algorithm, without prior knowledge of the maximum allowed mean duty cycle value of 20% (in our case). This means, that with a large battery, endeavors in power management can provide zero power outages and increase performance. Perhaps a larger solar panel would not only facilitate a larger overall duty cycle for WLAN, but also decrease the needed battery size, as this would be recharged more quickly, even during winter.

Further investigation could also be done to see if the seasonal variation in solar conditions and consequently the harvested energy is greater than we are able to mitigate with duty cycling, and what kind of proportion would the WLAN power consumption have to have compared to the base power consumption of the CoreStation, to equalize this effect.

Bibliography

[1] Bernhard Buchli, Mustafa Yuecel, Roman Lim, Tonio Gsell, and Jan Beutel. Demo abstract: Feature-rich platform for wsn design space exploration. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 115–116. IEEE, 2011.

[2] Backlog overview. https://people.ee.ethz.ch/~perma/wiki/wiki/BackLog/Overview. [Online; accessed 28th of March 2013].

[3] Student Thesis Guidelines. https://www.tik.ee.ethz.ch/w1/images/c/ca/Student_Thesis_Guidelines_EN.pdf. [Online; accessed 23th of March 2013].

[4] Declaration of originality. http://www.ethz.ch/faculty/exams/plagiarism/confirmation_en.pdf. [Online; accessed 25th of March 2013].

[5] Jan Beutel, Bernhard Buchli, Federico Ferrari, Matthias Keller, Marco Zimmerling, and Lothar Thiele. X-sense: Sensing in extreme environments. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[6] David R Cox. Prediction by exponentially weighted moving averages and related methods. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 414–422, 1961.

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

[8] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B Srivastava. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):32, 2007.

[9] Petar Čisar and Sanja Maravić Čisar. Optimization methods of ewma statistics. *Acta Polytechnica Hungarica*, 8(5):73–87, 2011.

[10] Carlo Bergonzini, Davide Brunelli, and Luca Benini. Algorithms for harvested energy prediction in batteryless wireless sensor networks. In *Advances in sensors and In-*

terfaces, 2009. IWASI 2009. 3rd International Workshop on, pages 144–149. IEEE, 2009.

[11] J Stuart Hunter. The exponentially weighted moving average. *J. QUALITY TECHNOL.*, 18(4):203–210, 1986.

Appendix A

Code

*** The following are snippets of python code used in various parts of this thesis.***

```
#################################################################################################
### Creating battery SoC from harvested energy data and predicting values with EWMA and WCMA.
#################################################################################################
import csv
import numpy

""" Initial battery state of charge = 1 (full)
    wlan current = 250mA
    base current = 150mA
    wlan energy = hourly energy in Wh """

batt_volt = 12.0                # Voltage in V
batt_cap = 34                   # Nominal capacity in Ah
batt_e = batt_volt*batt_cap*0.8 # Available energy in Wh
wlan_p = 0.25*batt_volt         # Power consumption in W
base_p = 0.15*batt_volt         # Power consumption in W

## Loading harvested energy data from file ##
with open('eH_30W_daily_12x8784x1_scaled.csv','r+b') as file1:
    reader = csv.reader(file1, delimiter=',')
    pv_matrix = list(reader)
pv_list = map(float,pv_matrix[0])   #Choose data for 1 of 12 years (0-11)

#################################################
def EWMA(Y, N=4):
    """ Computes an N-period exponential moving average for the time series Y
        Y is the input list, N is the number of previous values used for one step
        Returns a list of the exponential moving average approximation. """

ema0 = sum(Y[:N])/float(N)  #Starting value = average over N samples
    alpha = 2/float(1+N)        #Scaling factor between [0,1]
    ema = list(ema0 for i in range(N))

""" S(t+1) = alpha*S(t) + (1-alpha)*Y(t) """

for i,val in enumerate(Y[N-1:]):
        ema.append(alpha*ema[i] + (1-alpha)*val)
    none = ema.pop() ## Destroys the last value, which goes over len(Y)
    return ema

#################################################
def WCMA(E, alpha=0.7, D=4, K=3):
```

```
""" "E"         is a list of 24 hourly measurements concatenated by day
    "alpha"     is a weighting factor (similar to EWMA)
    "K"         is the number of previous samples used by WCMA
    "D"         is the number of previous days used by WCMA

    WCMA needs (24*D+K) previous samples to make a prediction for sample
    [d,n+1], so n_start >= (24*D+K-1)

    "d" is the label for the current day
    "n" is the label for the current sample ("n+1" = next sample)

    First possible prediction: d=D-1, n=K-1

    FORMULA:
    E[d,n+1]  =  (alpha * E[d,n])  +  (1-alpha) * GAP * M_D[d,n+1]

    E[d,n+1]    is the predicted value
    E[d,n]      is the curent mesured value
    M_D[d,n+1]  is the mean for "D" past days at sample "n+1"
    GAP         is a factor of past solar conditions

    For code reference: S[d,n+1] is the predicted value and E are measurements.
    We are only storing predictions to observe accuracy. """

    S = list(0 for i in range(len(E)+1))
    MD = list(0 for i in range(len(E)+1))
    GAP = list(0 for i in E)
    P = list((i+1.0)/K for i in range(K))

    for n, e in enumerate(E):
        if (D*24) <= n <= (D*24+(K-1)):
            MDsum = 0.0
            for i in range(D):
                MDsum = MDsum + E[n-(i+1)*24]
            MD[n] = (MDsum/D)
        if n >= (D*24+(K-1)):
            V = list((E[n-(K-1)+i]/MD[n-(K-1)+i]) for i in range(K))
            GAP[n] = (numpy.dot(V,P)/sum(P))
            MDsum = 0.0
            for i in range(D):
                MDsum = MDsum + E[(n+1)-(i+1)*24]
            MD[n+1] = (MDsum/D)
            S[n+1] = alpha*e+(1-alpha)*GAP[n]*MD[n+1]
    none = S.pop()
    none = MD.pop()
    return S

################################################
## 1. CREATING BATTERY SoC FROM HARVESTED ENERGY DATA

wlan_off = 0
power_down = 0
batt_SoC = []
energy = batt_e

for index, harv_e in enumerate(pv_list):

    if not power_down:
        if wlan_off:
            energy += (harv_e - base_p)
        elif not wlan_off:
```

```
            energy += (harv_e - wlan_e - base_p)
    elif power_down:
        energy += harv_e

    if (energy>batt_e*0.6) and (wlan_off):
        wlan_off = 0
    elif (energy>batt_e*0.05) and (power_down):
        power_down = 0
    elif (energy<=batt_e*0.05):
        power_down = 1
        wlan_off = 1

    if energy > batt_e:
        energy = batt_e
    elif energy < 0:
        energy = 0

    batt_SoC.append(energy)

##################################################
## 2. WRITING BATTERY SoC, EWMA AND WCMA TO FILE

ewma = EWMA(batt_SoC,N=4) #alpha = 0.6
##wcma = WCMA(batt_SoC, alpha=0.7, D=4, K=3)  ## Default parameters
wcma = WCMA(batt_SoC, alpha=0.5, D=9, K=1) ## Optimized parameters

results = [batt_SoC, ewma, wcma]
with open('SoC_calc_noPM.csv', 'wb') as file5:
    file_writer = csv.writer(file5)
    for i in range(len(results[0])):
        file_writer.writerow([x[i] for x in results])

#################################################################################################################
### OpenWeatherMap.org weather forecast API implementation and WSF calculation.
#################################################################################################################
from datetime import datetime
import csv
import numpy
import requests

Pwlan = 0.250 * 12
Pbase = 0.200 * 12
Ebat_max = 12 * 34 * 0.8
today = datetime.now()
timestamp = float(today.strftime("%s"))

##################################################
## 1. 5-DAY FORECAST

""" Forecast data for every 3 hours for 5 days = 41 values.
    Endpoint: http://api.openweathermap.org/data/2.5/forecast
    Keys:   "city"       : dict (location info)
            "cod"        : HTTP code (200 = OK)
            "cnt"        : len(response["list"])
            "list"       : 24/3*5 + 1 (41 values), each value = dictionary
                       Keys:   clouds  : {"all": cloud_value}
                               dt_txt  : DateTime (string)
                               weather : list(dict("key": "value",...))
                               dt      : linux timestamp (int)
                               main    : temp, pressure, ...
                               wind    : speed, degrees
    Querry parameters: "q": "London,uk" - search by city name
```

```
                      "lat": "47.378" - lattitude (N-S)
                      "lon": "8.553" - longitude (E-W)
                      "id": 524901" - search by city ID """

query_params = {"APPID": "***INSERT OWN API-ID***","lat": "47.378","lon": "8.553"}
endpoint = "http://api.openweathermap.org/data/2.5/forecast"
response = requests.get(endpoint, params=query_params)
weather = response.json()

timestamp5 = []
clouds5 = []
for idx, item in enumerate(weather["list"]):
    timestamp5.append(int(item["dt"]))
    clouds5.append(int(item["clouds"]["all"])/100.0)

time5 = list(((timestamp5[i]-timestamp5[0])/86400.0) for i in range(len(timestamp5)))
sky5 = list(1-clouds5[i] for i in range(len(clouds5)))

##################################################
## 2. 14 DAY FORECAST:

""" Daily forecast data for 14 days = 14 values.
    Endpoint: http://api.openweathermap.org/data/2.5/forecast/daily
    Keys:   "city"       : dict (location info)
            "cod"        : HTTP code (200 = OK)
            "cnt"        : len(response["list"])
            "list"       : all data values (each day = one dict)
                       Keys:   clouds  : percent clouds
                               temp    : min, max, eve, morn, night, day
                               humidity: percent humidity
                               pressure: in mbar
                               weather : main, id, icon, description
                               dt      : linux time
                               speed   : wind speed
                               deg     : wind direction
    Querry parameters: "q": "London,uk" - search by city name
                       "lat": "47.378" - lattitude (N-S)
                       "lon": "8.553" - longitude (E-W)
                       "id": 524901" - search by city ID
                       "mode": "json" or "xml"
                       "units": "metric" or "imperial"
                       "cnt": "10" - days to display (1 to 14, default = 7) """

query_params = {"APPID": "***INSERT OWN API-ID***","lat": "47.378",
                "lon": "8.553","mode": "json","units": "metric","cnt": "14"}
endpoint = "http://api.openweathermap.org/data/2.5/forecast/daily"
response = requests.get(endpoint, params=query_params)
weather = response.json()

timestamp14 = []
clouds14 = []
for idx, item in enumerate(weather["list"]):
    timestamp14.append(int(item["dt"]))
    clouds14.append(int(item["clouds"])/100.0)

time14 = list(((timestamp14[i]-timestamp14[0])/86400.0) for i in range(len(timestamp14)))
sky14 = list(1-clouds14[i] for i in range(len(clouds14)))
time105 = [round(x*(time5[1]-time5[0]),4) for x in range(0,int((len(time14)-1)/(time5[1]-time5[0])+1))]

##################################################
## 3. WEATHER SCALING FACTOR - WSF
```

```
k14 = numpy.polyfit(time14,sky14,1)
yfit = list(numpy.polyval(k14,time105))
##yscale = [x for x in yfit[0:len(sky5)]] #5day average
yscale = [x for x in yfit[0:int(1.0/(time5[1]-time5[0]))]] #1day average
sky5scale = [a*b for a,b in zip(sky5,yscale)]

WSF = float(sum(sky5scale))/len(sky5scale)
if WSF > 1:
    WSF = 1
elif WSF < 0:
    WSF = 0

#########################################################################################################
### WCMA parameter optimization
#########################################################################################################
import csv
import numpy

with open('eH_30W_daily_12x8784x1_scaled.csv','r+b') as file1:
    reader = csv.reader(file1, delimiter=',')
    pv_matrix = list(reader)

pv_list = map(float,pv_matrix[0])
pv_list_plus = list(pv_list[i]+100 for i in range(len(pv_list)))
""" For the purposes of calculating the WCMA, there need to be non-zero values
in our pv_list_plus, otherwise we divide by 0  when calculating vector V. """

##################################################
def EWMA(Y, N=4):
    ...

def WCMA(E, alpha=0.7, D=4, K=3):
    ...
##################################################
## 1. SEARCHING FOR BEST "alpha" AND "D"

""" error_matrix1 contains maximum error values for alpha-D combinations.
    errors1 contains lists of daily errors from which these maximums are taken ."""
error_matrix1 = []
error_average1 = []
errors1 = []
## Starting parameters: ##
dmax = 11
N = 24
K = 3
error_list1 = [] #temp
error_list2 = [] #temp

for a in range(1,11):
    for d in range(1,dmax):
        wcma = WCMA(pv_list_plus, alpha=0.1*a, D=d+1, K=K)
        wcma = list(wcma[i]-100 for i in range(len(wcma)))
        param_pair_error = []
        for i in range(len(pv_list)/N - dmax - 1):
            ##Calculate for 366 days -1 at end -dmax in front
            err = 0
            count = 0
            ## Calculate errors for individual days. ##
            for j in range(N):
                idx = dmax*24 + N*i
                thr1 = 0.1*max(pv_list[idx:idx+24])
                thr2 = 0.1*max(wcma[idx:idx+24])
```

```
                ##Thresholds for current day, depend on "i"
                if pv_list[idx+j] > thr1 and wcma[idx+j] > thr2:
                    err = err + abs( (float(wcma[idx+j])/float(pv_list[idx+j])) -1)
                    count +=  1
            err = err/float(count)
            ##Error for current day, appended to list for chosen "alpha, D"
            param_pair_error.append(err)
        errors1.append(param_pair_error)
        ##One list for "alpha, D" with errors for whole year
        error_list1.append(max(param_pair_error))
        error_list2.append(numpy.mean(param_pair_error))
    error_matrix1.append(error_list1)
    error_average1.append(error_list2)
    error_list1 = []
    error_list2 = []

tmp = []
for i in error_average1:
    tmp.extend(i)
value,index = min((value, index) for (index, value) in enumerate(tmp))
alpha = ((index/(dmax-1))+1)*0.1 ## Optimized alpha (= 0.5)
best_D_avg = index%(dmax-1) + 2

##################################################
## 2. SEARCHING FOR BEST "D" AND "K"

error_matrix2 = []
error_average2 = []
errors2 = []

for k in range(1,11):
    for d in range(1,dmax):
        wcma = WCMA(pv_list_plus, alpha=alpha, D=d+1, K=k)
        wcma = list(wcma[i]-100 for i in range(len(wcma)))
        param_pair_error = []
        for i in range(len(pv_list)/N - dmax - 1):
            ##Calculate for 366 days -1 at end -dmax in front
            err = 0
            count = 0
            ## Calculate errors for individual days. ##
            for j in range(N):
                idx = dmax*24 + N*i
                thr1 = 0.1*max(pv_list[idx:idx+24])
                thr2 = 0.1*max(wcma[idx:idx+24])
                ##Thresholds for current day, depend on "i"
                if pv_list[idx+j] > thr1 and wcma[idx+j] > thr2:
                    err = err + abs( (float(wcma[idx+j])/float(pv_list[idx+j])) -1)
                    count +=  1
            err = err/float(count)
            ##Error for current day, appended to list for chosen "alpha, D"
            param_pair_error.append(err)
        errors2.append(param_pair_error)
        ##One list for "alpha, D" with errors for whole year
        error_list1.append(max(param_pair_error))
        error_list2.append(numpy.mean(param_pair_error))
    error_matrix2.append(error_list1)
    error_average2.append(error_list2)
    error_list1 = []
    error_list2 = []

tmp = []
for i in error_average2:
```

```
    tmp.extend(i)
value,index = min((value, index) for (index, value) in enumerate(tmp))
K = ((index/(dmax-1))+1) ## Optimized K (= 1)
D = index%(dmax-1) + 2 ## Optimized D (= 9)

#################################################
## 3. APPROXIMATION RESULTS FOR DEFAULT AND OPTIMIZED "alpha, K, D"

ewma_e_list = EWMA(pv_list,N=4)
wcma_e_list = WCMA(pv_list_plus, alpha=0.7, D=4, K=3)
wcma_e_list = list(wcma_e_list[i]-100 for i in range(len(wcma_e_list)))
wcma_e_opt = WCMA(pv_list_plus, alpha=alpha, D=D, K=K)
wcma_e_opt = list(wcma_e_opt[i]-100 for i in range(len(wcma_e_opt)))
results = [pv_list,ewma_e_list, wcma_e_list, wcma_e_opt]

#################################################
## 4.1 CALCULATE ERRORS FOR DEFAULT "alpha, K, D"
default_error = []
for i in range(len(pv_list)/N - dmax - 1):
    ##Calculate for 366 days -1 at end -dmax in front
    err = 0
    count = 0
    for j in range(N):
        idx = dmax*24 + N*i
        thr1 = 0.1*max(pv_list[idx:idx+24])
        thr2 = 0.1*max(wcma_e_list[idx:idx+24])
        ##Thresholds for current day, depend on "i"
        if pv_list[idx+j] > thr1 and wcma_e_list[idx+j] > thr2:
            err = err + abs( (float(wcma_e_list[idx+j])/float(pv_list[idx+j])) -1)
            count +=  1
    err = err/float(count)
    ##Error for current day, appended to list for optimal "alpha, D, K"
    default_error.append(err)

## 4.2 CALCULATE ERRORS FOR OPTIMIZED "alpha, K, D"
optimized_error = []
for i in range(len(pv_list)/N - dmax - 1):
    ##Calculate for 366 days -1 at end -dmax in front
    err = 0
    count = 0
    for j in range(N):
        idx = dmax*24 + N*i
        thr1 = 0.1*max(pv_list[idx:idx+24])
        thr2 = 0.1*max(wcma_e_opt[idx:idx+24])
        ##Thresholds for current day, depend on "i"
        if pv_list[idx+j] > thr1 and wcma_e_opt[idx+j] > thr2:
            err = err + abs( (float(wcma_e_opt[idx+j])/float(pv_list[idx+j])) -1)
            count +=  1
    err = err/float(count)
    ##Error for current day, appended to list for default "alpha, D, K"
    optimized_error.append(err)

## 4.3 CALCULATE ERRORS FOR EWMA
ewma_error = []
for i in range(len(pv_list)/N - dmax - 1):
    ##Calculate for 366 days -1 at end -dmax in front
    err = 0
    count = 0
    for j in range(N):
        idx = dmax*24 + N*i
        thr1 = 0.1*max(pv_list[idx:idx+24])
        thr2 = 0.1*max(ewma_e_list[idx:idx+24])
```

```
        ##Thresholds for current day, depend on "i"
        if pv_list[idx+j] > thr1 and ewma_e_list[idx+j] > thr2:
            err = err + abs( (float(ewma_e_list[idx+j])/float(pv_list[idx+j])) -1)
            count +=  1
    err = err/float(count)
    ##Error for current day, appended to list
    ewma_error.append(err)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
################################################
## 5. WRITING RESULTS TO FILE

with open('matlab1_max_alphaD_KD.csv', 'w+') as file2:
    file_writer = csv.writer(file2)
    file_writer.writerows(error_matrix1)
    file_writer.writerow("")
    file_writer.writerows(error_matrix2)
with open('matlab2_avg_alphaD_KD.csv', 'w+') as file3:
    file_writer = csv.writer(file3)
    file_writer.writerows(error_average1)
    file_writer.writerow("")
    file_writer.writerows(error_average2)
with open('ParameterOptimization2.csv', 'wb') as file4:
    file_writer = csv.writer(file4)
    for i in range(len(results[0])):
        file_writer.writerow([x[i] for x in results])
with open('matlab3_all_errors_histogram.csv', 'w+') as file5:
    file_writer = csv.writer(file5)
    file_writer.writerow(default_error)
    file_writer.writerow("")
    file_writer.writerow(optimized_error)
    file_writer.writerow("")
    file_writer.writerow(ewma_error)
##########################################################################################################
*** Matlab script to plot optimization results. ***
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% WCMA parameter optimization - MATLAB script
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc
close all
results = transpose(csvread('ParameterOptimization2.csv'));
eH = results(1,:);
ewma = results(2,:);
wcma_def = results(3,:);
wcma_opt = results(4,:);

t = [1:length(eH)];
figure
plot(t,eH,t,ewma,t,wcma_def,t,wcma_opt)
title('Solar energy and predictions');
xlabel('Day');
ylabel('Energy');
legend('Harvested energy','ewma','wcma default param','wcma optimal param');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tmp = csvread('matlab1_max_alphaD_KD.csv');
error_max_alphaD = tmp(1:10,:);
error_max_KD = tmp(11:end,:);
tmp = csvread('matlab2_avg_alphaD_KD.csv');
error_avg_alphaD = tmp(1:10,:);
error_avg_KD = tmp(11:end,:);
clear tmp

figure
```

```
surf(2:11,0.1:0.1:1,error_max_alphaD)
title('Error max alpha-D, K=3');
xlabel('D');
ylabel('alpha');

figure
surf(2:11,1:10,error_max_KD)
title('Error max K-D, alpha=0.5');
xlabel('D');
ylabel('K');

figure
surf(2:11,0.1:0.1:1,error_avg_alphaD)
title('Error avg alpha-D, K=3');
xlabel('D');
ylabel('alpha');

figure
surf(2:11,1:10,error_avg_KD)
title('Error avg K-D, alpha=0.5');
xlabel('D');
ylabel('K');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
errors = csvread('matlab3_all_errors_histogram.csv');
errors_default = errors(1,:);
errors_optimized = errors(2,:);
errors_ewma = errors(3,:);

figure
plot(errors_default,'b');
hold on
plot(errors_optimized,'r');
hold on
plot(errors_ewma,'g');
title('WCMA and EWMA errors');
xlabel('Day');
ylabel('Relative error');
legend('Default WCMA parameters','Optimized WCMA parameters', 'EWMA');

bins = 50;
[histogram,bin] = hist(transpose(errors),bins);
figure
bar(bin,histogram);
title('WCMA and EWMA error histogram');
colormap([1,0.5,0.5;0.5,1,0.5;0.5,0.5,1])
grid on
xlabel('Relative error (bins)');
ylabel('Frequency');
legend('Default WCMA parameters','Optimized WCMA parameters', 'EWMA');
% Expected values of the Poisson-distributed errors for EWMA and WCMA predictions
avgdef = mean(errors_default)
avgopt = mean(errors_optimized)
avgewma = mean(errors_ewma)

#########################################################################################################
### Power management code
#########################################################################################################

""" Version 1.0:
I don't like the duty cycle (DC) calculation formula: it uses harvested energy (Eg) to
calculate the DC, and we can't get Eg in a real system, unless we calculate it from
battery SoC! Eg is only accessible to us for simulation purposes.
```

```
    Problems:
    - Our battery is too small for the job (problems in the winter)
    - In the summer, we don't drive the wlan as hard as we could!
      Surplus energy from the battery is wasted... """

import csv
import numpy

""" Initial battery state of charge = Ebat_max (full, 100%)
    wlan current = 250mA
    base current = 200mA """
Vbat = 12.0                 # Voltage in V
Cbat = 34                   # Nominal capacity in Ah
Ebat_max = Vbat*Cbat*0.8    # Available energy in Wh
Pwlan = 0.25*Vbat           # Power consumption in W
Pbase = 0.20*Vbat           # Power consumption in W
Tint = 1.0                  # Time interval between samples in h - IMPORTANT!!!
Twlan_min = 0.1             # Minimum WLAN on-time (or percentage)

with open('eH_30W_daily_12x8784x1_scaled.csv','r+b') as file1:
    reader = csv.reader(file1, delimiter=',')
    pv_matrix = list(reader)
pv_list = map(float,pv_matrix[0])   #Choose data for 1 of 12 years (0-11)

def EWMA(Y, N=4):
...

def EWMAstep(Y, N=4):
    """ Same as EWMA(), but returnes one value (prediction). """
    if not 0<N<len(Y):
        raise Exception("Must have 0<N<len(Y)")

    ema0 = sum(Y[:N])/float(N)  #Starting value = average over N samples
    alpha = 1- 2/float(1+N)     #Scaling factor between [0,1]
    ema = list(ema0 for i in range(N))

    """ S(t+1) = alpha*S(t) + (1-alpha)*Y(t) """

    for i,val in enumerate(Y[N-1:]):
        ema.append(alpha*ema[i+N-1] + (1-alpha)*val)
    pred = ema.pop()
    return pred

def WCMA(E, alpha=0.7, D=4, K=3):
...

def WCMAstep(E, alpha=0.7, D=4, K=3):
    """ Does the same as WCMA(), but returnes one value. """
    if not 0<(D*24+K-1)<=len(E):
        raise Exception("Must have 0<(D*24+K-1)<=len(E)")

    S = list(0 for i in range(len(E)+1))
    MD = list(0 for i in range(len(E)+1))
    GAP = list(0 for i in E)
    P = list((i+1.0)/K for i in range(K))

    for n, e in enumerate(E):
        if (D*24) <= n <= (D*24+(K-1)):
            MDsum = 0.0
            for i in range(D):
                MDsum = MDsum + E[n-(i+1)*24]
            MD[n] = (MDsum/D)
```

```
        if n >= (D*24+(K-1)):
            V = list((E[n-(K-1)+i]/MD[n-(K-1)+i]) for i in range(K))
            GAP[n] = (numpy.dot(V,P)/sum(P))
            MDsum = 0.0
            for i in range(D):
                MDsum = MDsum + E[(n+1)-(i+1)*24]
            MD[n+1] = (MDsum/D)
            S[n+1] = alpha*e+(1-alpha)*GAP[n]*MD[n+1]
    none = MD.pop()
    pred = S.pop()
    return pred

## 1.1 CREATING BATTERY SoC - NO WLAN! Control run
Toff = [0,0,0,0,0]

power_down = 0
noWLAN_SoC = []
energy = Ebat_max

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    noWLAN_SoC.append(energy)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        power_down = 0
    elif (energy<=Ebat_max*0.05) and not power_down:
        power_down = 1
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - Pbase)
    elif power_down:
        energy += Eg
        Toff[0] += 1
    if energy > Ebat_max:
        energy = Ebat_max
    elif energy < 0:
        energy = 0

## 1.2 CREATING BATTERY SoC - duty_cycle = 100%
power_down = 0
noPM_SoC = []
energy = Ebat_max

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    noPM_SoC.append(energy)
    ... *** Code same as "no wlan" ***
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - Pwlan - Pbase)
    elif power_down:
        energy += Eg
        Toff[1] += 1
...

## 1.3 CREATING BATTERY SoC - simple duty cycling
""" First values are all maximal (duty cycle, battery state of charge).
    The energy calculations are as follows:
    "pv-list" is the harvested energy from the sun. In a real system,
    this would be measured, here it is simulated in advance.
    "DC_SoC" is the battery state of charge at the beginning of the
```

```
    time interval, taking into account energy consumption/surplus in
    the previous time interval.
    "duty_cycle" is calculated for the duration of the current time
    interval, provided we have a current and a previous battery state
    of charge 'measurement'.
    "energy" is modified with respect to power consumption/surplus
    during the current time interval. It will be appended to SoC... """
power_down = 0
duty_cycle = 1.0
DC_SoC = []
energy = Ebat_max
DChist = []

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # DC calculation for this time interval
    # dSoC is a calculation for this interval based on history #
    if len(DC_SoC) > 1:
        dSoC = DC_SoC[index] - DC_SoC[index-1]
        duty_cycle = ( ( ((Eg-dSoC)/Tint) - Pbase) / Pwlan )
        if duty_cycle < Twlan_min/float(Tint):
            duty_cycle = Twlan_min/float(Tint)
        if duty_cycle > 1:
            duty_cycle = 1
    # Emergency power-down
    ...
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - Pwlan*duty_cycle - Pbase)
    elif power_down:
        duty_cycle = 0
        energy += Eg
        Toff[2] += 1
if energy > Ebat_max:
...

    DChist.append(duty_cycle)

## 1.4 CREATING BATTERY SoC - EWMA PM
power_down = 0
duty_cycle = 1.0
ewma_SoC = []
energy = Ebat_max
ewmaDChist = []
N_ewma = 4

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)

    # DC calculation for this time interval
    # dSoC is a forecast for this interval based on EWMA #
    if len(ewma_SoC) > N_ewma:
        if len(ewma_SoC)<(20*N_ewma):
            dSoC = EWMAstep(ewma_SoC,N=N_ewma) - ewma_SoC[index]
        # Restricting EWMA calculation to smaller number of samples
        else:
            dSoC = EWMAstep(ewma_SoC[index-20*N_ewma+1:index],N=N_ewma) - ewma_SoC[index]
        duty_cycle = ( ( ((Eg-dSoC)/Tint) - Pbase) / Pwlan )
...
```

```
    elif power_down:
        duty_cycle = 0
        energy += Eg
        Toff[3] += 1
    if energy > Ebat_max:
...

    ewmaDChist.append(duty_cycle)

## 1.5 CREATING BATTERY SoC - WCMA PM
power_down = 0
duty_cycle = 1.0
wcma_SoC = []
energy = Ebat_max
wcmaDChist = []
A_wcma = 0.7
D_wcma = 4
K_wcma = 3
N_wcma = 24*D_wcma+K_wcma-1

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # DC calculation for this time interval
    # dSoC is a forecast for this interval based on WCMA
    # Also: restricting WCMA calculation to necessary number of samples
    if len(wcma_SoC) > N_wcma+1:
        dSoC = WCMAstep(wcma_SoC[index-N_wcma-1:index],alpha=A_wcma,D=D_wcma,K=K_wcma) - wcma_SoC[index-1]
        duty_cycle = ( ( ((Eg-dSoC)/Tint) - Pbase) / Pwlan )
        if duty_cycle < Twlan_min/float(Tint):
            ...
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - Pwlan*duty_cycle - Pbase)
    elif power_down:
        duty_cycle = 0
        energy += Eg
        Toff[4] += 1
    if energy > Ebat_max:
        ...

    wcmaDChist.append(duty_cycle)

## 2. PRINTING POWER MANAGEMENT TO FILE
results = [noWLAN_SoC,noPM_SoC,DC_SoC,DChist,ewma_SoC,ewmaDChist,wcma_SoC,wcmaDChist]
item_length = len(results[0])
with open('SoC_calc_PM1.0.csv', 'wb') as file6:
    file_writer = csv.writer(file6)
    for i in range(item_length):
        file_writer.writerow([x[i] for x in results])
print "Finished SoC_calc_PM v1.0, Results stored in .csv files."
print "Hours offline:"
print "No WLAN: "+str(Toff[0])+" = %.4f"%(Toff[0]/8784.0*100)+"%."
print "No PM: "+str(Toff[1])+" = %.4f"%(Toff[1]/8784.0*100)+"%."
print "DC PM: "+str(Toff[2])+" = %.4f"%(Toff[2]/8784.0*100)+"%."
print "EWMA PM: "+str(Toff[3])+" = %.4f"%(Toff[3]/8784.0*100)+"%."
print "WCMA PM: "+str(Toff[4])+" = %.4f"%(Toff[4]/8784.0*100)+"%."
print ""
print "During ON-time, average duty cycle is:"
print "DC PM: %.4f"%(sum(DChist)/(8784-Toff[2])*100)+"%."
print "EWMA PM: %.4f"%(sum(ewmaDChist)/(8784-Toff[3])*100)+"%."
```

```
print "WCMA PM: %.4f"%(sum(wcmaDChist)/(8784-Toff[4])*100)+"%."

################################################
""" Version 2.0:
    Different duty cycle calculation - using SoC to calculate Eg """
...
## 1.3 CREATING BATTERY SoC - simple duty cycling
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # DC calculation for this time interval
    # dSoC is a calculation for this interval based on history #
    if len(DC_SoC) > 1:
        dSoC = DC_SoC[index] - DC_SoC[index-1]
        duty_cycle += dSoC/float(Pwlan)

if duty_cycle < Twlan_min/float(Tint):
            ...

## 1.4 CREATING BATTERY SoC - EWMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        dSoC = ewma_SoC[index] - ewma_SoC[index-1]
        ewmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    if len(ewma_SoC) > N_ewma+1:
        if len(ewma_SoC)<(20*N_ewma):
            Eg_pred = EWMAstep(ewmaEg,N=N_ewma)
        # Restricting EWMA calculation to smaller number of samples
        else:
            Eg_pred = EWMAstep(ewmaEg[index-20*N_ewma+1:index],N=N_ewma)
        duty_cycle = ((Eg_pred/Tint)-Pbase)/Pwlan
        if duty_cycle < Twlan_min/float(Tint):
            ...

## 1.5 CREATING BATTERY SoC - WCMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        dSoC = wcma_SoC[index] - wcma_SoC[index-1]
wcmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    # Eg is a forecast for this interval based on WCMA
    # Also: restricting WCMA calculation to necessary number of samples
    if len(wcma_SoC) > N_wcma+1:
        Eg_pred = WCMAstep(wcmaEg[index-N_wcma-1:index],alpha=A_wcma,D=D_wcma,K=K_wcma)
        Eg_pred -= 100
        duty_cycle = ((Eg_pred/Tint)-Pbase)/Pwlan
        if duty_cycle < Twlan_min/float(Tint):
            ...

################################################
""" Version 3.0:
    Duty cycle calculation same as 2.0, also scale with %SoC """
```

```
...
## 1.3 CREATING BATTERY SoC - simple duty cycling
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # DC calculation for this time interval
    # dSoC is a calculation for this interval based on history #
    if len(DC_SoC) > 1:
        dSoC = DC_SoC[index] - DC_SoC[index-1]
        duty_cycle += dSoC/float(Pwlan)
        if duty_cycle > 1:
            duty_cycle = 1
        duty_cycle = duty_cycle*energy/float(Ebat_max)
        if duty_cycle < Twlan_min/float(Tint):
            duty_cycle = Twlan_min/float(Tint)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        ...

## 1.4 CREATING BATTERY SoC - EWMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        dSoC = ewma_SoC[index] - ewma_SoC[index-1]
        ewmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    if len(ewma_SoC) > N_ewma+1:
        if len(ewma_SoC)<(20*N_ewma):
            Eg_pred = EWMAstep(ewmaEg,N=N_ewma)
        # Restricting EWMA calculation to smaller number of samples
        else:
            Eg_pred = EWMAstep(ewmaEg[index-20*N_ewma+1:index],N=N_ewma)
        duty_cycle = ((Eg_pred/Tint)-Pbase)/Pwlan
        if duty_cycle > 1:
            duty_cycle = 1
        duty_cycle = duty_cycle*energy/float(Ebat_max)
        if duty_cycle < Twlan_min/float(Tint):
            duty_cycle = Twlan_min/float(Tint)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        ...

## 1.5 CREATING BATTERY SoC - WCMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        dSoC = wcma_SoC[index] - wcma_SoC[index-1]
        wcmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    if len(wcma_SoC) > N_wcma+1:
        Eg_pred = WCMAstep(wcmaEg[index-N_wcma-1:index],alpha=A_wcma,D=D_wcma,K=K_wcma)
        Eg_pred -= 100
        duty_cycle = ((Eg_pred/Tint)-Pbase)/Pwlan
        if duty_cycle > 1:
            duty_cycle = 1
```

```
        duty_cycle = duty_cycle*energy/float(Ebat_max)
        if duty_cycle < Twlan_min/float(Tint):
            duty_cycle = Twlan_min/float(Tint)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        ...

################################################
""" Version 4.0:
    Duty cycle calculation same as 2.0
    Turn station off during night (~12h), and also
below critical battery """

## 1.1 CREATING BATTERY SoC - NO WLAN!!! Control run
...
for index, Eg in enumerate(pv_list):
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        power_down = 0
    elif (energy<=Ebat_max*0.05) and not power_down:
        power_down = 1
    # Power off during night:
    if Eg == 0.0:
        power_down = 1
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        ...

## 1.2 CREATING BATTERY SoC - NO duty_cycle
...
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        power_down = 0
    elif (energy<=Ebat_max*0.05) and not power_down:
        power_down = 1
    # Power off during night:
    if Eg == 0.0:
        power_down = 1
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        ...

## 1.3 CREATING BATTERY SoC - duty_cycle PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # Power off during night:
    if Eg == 0.0:
        power_down = 1
    # DC calculation for this time interval
    if len(DC_SoC) > 1:
        dSoC = DC_SoC[index] - DC_SoC[index-1]
        duty_cycle += dSoC/float(Pwlan)
        if duty_cycle > 1:
            duty_cycle = 1
        if duty_cycle < Twlan_min/float(Tint):
            duty_cycle = Twlan_min/float(Tint)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down and (not Eg == 0.0):
```

```
        ...

## 1.4 CREATING BATTERY SoC - EWMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Power off during night:
    if Eg == 0.0:
        power_down = 1
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        dSoC = ewma_SoC[index] - ewma_SoC[index-1]
        ewmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    if len(ewma_SoC) > N_ewma+1:
        ...

## 1.5 CREATING BATTERY SoC - WCMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # Power off during night:
    if Eg == 0.0:
        power_down = 1
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        dSoC = wcma_SoC[index] - wcma_SoC[index-1]
        wcmaEg.append(dSoC + Pbase + Pwlan*duty_cycle)
    # DC calculation for this time interval
    if len(wcma_SoC) > N_wcma+1:
        ...

## 2. PRINTING POWER MANAGEMENT TO FILE
results = [noWLAN_SoC,noPM_SoC,DC_SoC,DChist,ewma_SoC,ewmaDChist,wcma_SoC,wcmaDChist]
item_length = len(results[0])
with open('SoC_calc_PM.csv', 'wb') as file6:
    file_writer = csv.writer(file6)
    for i in range(item_length):
        file_writer.writerow([x[i] for x in results])
print "Finished SoC_calc_PM v4.0, Results stored in .csv files."
print "There are 4104 'controlled shutdown hours' in the year"
print "Of those: hours offline, critical offline"
print "No WLAN: "+str(Toff[0])+", "+str(Toff[0]-4104)+"."
print "No PM: "+str(Toff[1])+", "+str(Toff[1]-4104)+"."
print "DC PM: "+str(Toff[2])+", "+str(Toff[2]-4104)+"."
print "EWMA PM: "+str(Toff[3])+", "+str(Toff[3]-4104)+"."
print "WCMA PM: "+str(Toff[4])+", "+str(Toff[4]-4104)+"."
print ""
print "During ON-time, average duty cycle is:"
print "DC PM: %.4f"%(sum(DChist)/(8784-Toff[2])*100)+"%."
print "EWMA PM: %.4f"%(sum(ewmaDChist)/(8784-Toff[3])*100)+"%."
print "WCMA PM: %.4f"%(sum(wcmaDChist)/(8784-Toff[4])*100)+"%."

##################################################
""" Version 2.1:
    Similar to 2.0, slight changes to duty cycle calculation: only changing
the duty cycle once per day (at midnight). This should eliminate
erratic duty cycle changes and stabilize the system power consumption. """
...
## 1.3 CREATING BATTERY SoC - simple duty cycling
```

```
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # DC calculation once per day at 00:00
    # dSoC is a calculation for entire day (24h) #
    if len(DC_SoC) > 1:
        if not index%24:
            dSoC = DC_SoC[index] - DC_SoC[index-24]
            duty_cycle += dSoC/float(Pwlan*24)

    if duty_cycle < Twlan_min/float(Tint):
        ...

## 1.4 CREATING BATTERY SoC - EWMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        if not index%24:
            dSoC = ewma_SoC[index] - ewma_SoC[index-24]
            ewmaEg.append(dSoC + Pbase*24 + Pwlan*duty_cycle*24)
    # DC calculation for this time interval
    if len(ewmaEg) > N_ewma+1:
        if len(ewmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(ewmaEg,N=N_ewma)
        # Restricting EWMA calculation to smaller number of samples
        else:
            Eg_pred = EWMAstep(ewmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan
    if duty_cycle < Twlan_min/float(Tint):
        ...

## 1.5 CREATING BATTERY SoC - WCMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        if not index%24:
            dSoC = wcma_SoC[index] - wccma_SoC[index-24]
            wcmaEg.append(dSoC + Pbase*24 + Pwlan*duty_cycle*24)
    # DC calculation for entire day
    if len(wcmaEg) > N_wcma+1:
        Eg_pred = WCMAstep(wcmaEg[index/24-N_wcma-1:index/24],alpha=A_wcma,D=D_wcma,K=K_wcma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan
    # This EWMA part helps the system work for the number of days it
    # takes the WCMA to initialize with the ~100 samples (3.5 months!) :/
    elif len(wcmaEg) > N_ewma+1:
        if len(wcmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(wcmaEg,N=N_ewma)
        else:
            Eg_pred = EWMAstep(wcmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan

    if duty_cycle < Twlan_min/float(Tint):
```

```
        ...

################################################
""" Version 2.2:
    Same as 2.1, slight changes to duty cycle calculation.
    Only changing the duty cycle once per day (at midnight), but also
    changing it in no more than 0.25 increments! (even slower dc change).

    Terrible idea!
    Does almost nothing, but make the stats a little worse than 2.1 does. """
...
Vbat = 12.0                 # Voltage in V
max_increment = 0.25        # Maximum increment of change for duty cycle
...
## 1.3 CREATING BATTERY SoC - simple duty cycling
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # DC calculation once per day at 00:00
    if len(DC_SoC) > 1:
        if not index%24:
            dSoC = DC_SoC[index] - DC_SoC[index-24]
            increment = dSoC/float(Pwlan*24)
            if increment > max_increment: increment = max_increment
            if increment < -max_increment: increment = -max_increment
            duty_cycle += increment

    if duty_cycle < Twlan_min/float(Tint):
        ...

## 1.4 CREATING BATTERY SoC - EWMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        if not index%24:
            dSoC = ewma_SoC[index] - ewma_SoC[index-24]
            ewmaEg.append(dSoC + Pbase*24 + Pwlan*duty_cycle*24)
    # DC calculation for this time interval
    if len(ewmaEg) > N_ewma+1:
        if len(ewmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(ewmaEg,N=N_ewma)
        # Restricting EWMA calculation to smaller number of samples
        else:
            Eg_pred = EWMAstep(ewmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            increment = ((Eg_pred/24)-Pbase)/Pwlan - ewmaDChist[index-1]
            if increment > max_increment: increment = max_increment
            if increment < -max_increment: increment = -max_increment
            duty_cycle += increment

    if duty_cycle < Twlan_min/float(Tint):
        ...

## 1.5 CREATING BATTERY SoC - WCMA PM
...
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
```

```
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        if not index%24:
            dSoC = wcma_SoC[index] - wccma_SoC[index-24]
            wcmaEg.append(dSoC + Pbase*24 + Pwlan*duty_cycle*24)
    # DC calculation for entire day
    if len(wcmaEg) > N_wcma+1:
        Eg_pred = WCMAstep(wcmaEg[index/24-N_wcma-1:index/24],alpha=A_wcma,D=D_wcma,K=K_wcma)
        if not index%24:
            increment = ((Eg_pred/24)-Pbase)/Pwlan - wcmaDChist[index-1]
            if increment > max_increment: increment = max_increment
            if increment < -max_increment: increment = -max_increment
            duty_cycle += increment
    # This EWMA part helps the system work for the number of days it
    # takes the WCMA to initialize with the ~100 samples (3.5 months!) :(
    elif len(wcmaEg) > N_ewma+1:
        if len(wcmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(wcmaEg,N=N_ewma)
        else:
            Eg_pred = EWMAstep(wcmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            increment = ((Eg_pred/24)-Pbase)/Pwlan - wcmaDChist[index-1]
            if increment > max_increment: increment = max_increment
            if increment < -max_increment: increment = -max_increment
            duty_cycle += increment

    if duty_cycle < Twlan_min/float(Tint):
        ...

#########################################################################################################
### Power management on augmented system
#########################################################################################################
""" Version 2.3:
    Similar to 2.1, but having oversized battery, constant DC, 12 years
of PV data... Used to prove a point (panel/battery too small). """
...
Vbat = 12.0                 # Voltage in V
Cbat = 34                   # Nominal capacity in Ah
Ebat_max = 7500 # To prove that Pbase+Pwlan are too big for PV-module
Pwlan = 0.25*Vbat           # Power consumption in W
Pbase = 0.20*Vbat           # Power consumption in W
Tint = 1.0                  # Time interval between samples in h
Twlan_min = 0.1             # Minimum WLAN on-time

with open('eH_30W_daily_12x8784x1_scaled.csv','r+b') as file1:
    reader = csv.reader(file1, delimiter=',')
    pv_matrix = list(reader)

pv_list = []
for item in pv_matrix:
tmp = map(float,item)
pv_list.extend(tmp)
pv_list_plus = list(pv_list[i]+100 for i in range(len(pv_list)))
...

## 1.3. CONSTANT DUTY CYCLE = 20%
Toff = [0,0,0,0,0]
power_down = 0
""" The system can handle numpy.mean(pv_list), which is 3.0W of
    total consumption, that gives me 20% duty cycle (4h48min per day MAX)."""
duty_cycle = 0.20
DC_SoC = []
```

```
energy = Ebat_max
for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    DC_SoC.append(energy)
    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        power_down = 0
    elif (energy<=Ebat_max*0.05) and not power_down:
        power_down = 1
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - duty_cycle*Pwlan - Pbase)
    elif power_down:
        energy += Eg
        Toff[2] += 1
    if energy > Ebat_max:
        energy = Ebat_max
    elif energy < 0:
        energy = 0

## 1.4 CREATING BATTERY SoC - EWMA PM

power_down = 0
duty_cycle = 1.0
ewma_SoC = []
energy = Ebat_max
ewmaDChist = []
ewmaEg = []
N_ewma = 4

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    ewma_SoC.append(energy)
    # Making energy prediction:
    if len(ewma_SoC) > 1:
        if not index%24:
            ewmaEg.append(sum(pv_list[index-24-1:index-1]))
    # DC calculation for this time interval
    # Eg is a forecast for this interval based on EWMA #
    if len(ewmaEg) > N_ewma+1:
        if len(ewmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(ewmaEg,N=N_ewma)
        # Restricting EWMA calculation to smaller number of samples
        else:
            Eg_pred = EWMAstep(ewmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan

    if duty_cycle < Twlan_min/float(Tint):
        duty_cycle = Twlan_min/float(Tint)

if index > 720:
pvmean = numpy.mean(pv_list[index-720:index-1])/max(pv_list[index-720:index-1])
if duty_cycle > pvmean:
duty_cycle = pvmean
elif index > 2:
pvmean = numpy.mean(pv_list[:index-1])/max(pv_list[:index-1])
if duty_cycle > pvmean:
duty_cycle = pvmean

    # Emergency power-down
```

```
    if (energy>Ebat_max*0.6) and power_down:
        power_down = 0
    elif (energy<=Ebat_max*0.05) and not power_down:
        power_down = 1
        duty_cycle = 0
    # Energy consumption throughout current time interval
    # and until beginning of next time interval (next iteration)
    if not power_down:
        energy += (Eg - Pwlan*duty_cycle - Pbase)
    elif power_down:
        duty_cycle = 0
        energy += Eg
        Toff[3] += 1
    if energy > Ebat_max:
        energy = Ebat_max
    elif energy < 0:
        energy = 0

    ewmaDChist.append(duty_cycle)

## 1.5 CREATING BATTERY SoC - WCMA PM

power_down = 0
duty_cycle = 1.0
wcma_SoC = []
energy = Ebat_max
wcmaDChist = []
wcmaEg = []
A_wcma = 0.7
D_wcma = 4
K_wcma = 3
N_wcma = 24*D_wcma+K_wcma-1

for index, Eg in enumerate(pv_list):
    # Energy at beginning of time interval
    wcma_SoC.append(energy)
    # Making energy prediction:
    if len(wcma_SoC) > 1:
        if not index%24:
            wcmaEg.append(sum(pv_list[index-24-1:index-1]))
    # DC calculation for entire day
    # Also: restricting WCMA calculation to necessary number of samples
    if len(wcmaEg) > N_wcma+1:
        Eg_pred = WCMAstep(wcmaEg[index/24-N_wcma-1:index/24],alpha=A_wcma,D=D_wcma,K=K_wcma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan
    # This EWMA part helps the system work for the number of days it
    # takes the WCMA to initialize with the ~100 samples (3.5 months!) :(
    elif len(wcmaEg) > N_ewma+1:
        if len(wcmaEg)<(20*N_ewma):
            Eg_pred = EWMAstep(wcmaEg,N=N_ewma)
        else:
            Eg_pred = EWMAstep(wcmaEg[index/24-20*N_ewma+1:index/24],N=N_ewma)
        if not index%24:
            duty_cycle = ((Eg_pred/24)-Pbase)/Pwlan

if duty_cycle < Twlan_min/float(Tint):
        duty_cycle = Twlan_min/float(Tint)

if index > 720:
pvmean = numpy.mean(pv_list[index-720:index-1])/max(pv_list[index-720:index-1])
if duty_cycle > pvmean:
```

```
duty_cycle = pvmean
elif index > 2:
pvmean = numpy.mean(pv_list[:index-1])/max(pv_list[:index-1])
if duty_cycle > pvmean:
duty_cycle = pvmean

    # Emergency power-down
    if (energy>Ebat_max*0.6) and power_down:
        ...
```