|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| SoC | State of Charge |
| DoD | Depth of Discharge |
| DST | Development Support Tool |
| LoL | Loss of Load |
| MSSE | Micro and Small Scale Enterprises |
| PV | Photovoltaic |

**Abstract**

Objective

Material and Methods

Results

Conclusions

150-200 words

# Introduction

A general introduction goes here.

## Background

* Problem Formulation
  + Appeal of system.
* Literature Survey
  + External research
    - Homer, ret screen….
  + NTNU
* What remains to be done.

## Objectives

The main objectives of this Master’s project are:

1. To rewrite, scrutinize, and test the DST
2. To develop new features for the DST
3. Open source.

## Limitations

## Approach

Listing the main parts of the methodology, stating why each step was important. Check the disposition note in evernote

### Rewriting and reorganizing existing code

### Creating a user interface

### Simulating a biomass power system

### Documentation

## Structure of the Report

# Methodology

Part 1, present; part 2 explain how. Results

## The Development Support Tool

The original functionality of the tool, excluded any implementation details, will be explained here. The development support tool as initially developed and described by Stephano Mandelli. There are three stages of calculations, namely the simulations, the economic analysis and the search for optimal solutions.

### Simulations

Different combinations of PV and battery sizes are “tested” in the simulation. Given a range of PV and battery sizes, estimations or measurements of a load profile, irradiation data and ambience temperature data, the interaction between these data sets are simulated. There are also scalar parameters affecting the physical performance of the components, and for the conditions under which the system will operate. This means that every combination of PV and battery sizes will each have a full timeline simulation of absorbing energy from the sun, serving the load, charging and discharging the battery. This results in a computation complexity of, given a

The system works as follows: for every PV size, an array of absorbed power is calculated, if the absorbed power (with losses) are sufficient to cover the load demand, the surplus power will charge the battery (for current battery size). If the absorbed power is insufficient, the battery will discharge. If there is insufficient power available, or if the load demand fluctuate too much, there will be a loss of load.

For every discharge cycle, the DoD is used to calculate how large fraction of the battery lifetime is consumed. This calculation stems from a function called cycles\_to\_failure. This is a fitted function to the battery “cycles to failure” vs. “depth of discharge” characteristics, proposed by the battery manufacturer.

### Economic Analysis

The economic analysis will take each combination of PV and battery, which now have timelines of simulation results associated with them, and produce economical calculations. The outputs are mainly based on calculations on the input parameters, and not simulation outputs. The exception is the battery replacements needed and LCoE (the cost for each kWh supplied).

The sum of partial cycles used is not an hourly data set, it’s a fraction of the battery lifetime consumed during simulation time. Replacing batteries are very costly, and therefore the cycles\_to\_failure function has a major impact on the economic results.

### Finding Optimal Solutions

The last step is finding some simulations that display the desired dimensions for performance and the optimal cost amongst these. The user defines a range of LLP and an accepted deviation from the given LLP values. For each LLP in this range, if a match is found in the simulation outputs it will be listed as an optimal solution. Usually, there are several matches. The simulation with the lowest NPC will be chosen as the optimal solution among these.

## The Logplot.m script

The inherited code is displayed in Appendix B. The code of the tool is located within one file. This implies that the user will have to run the entire file, regardless of the desired functionality. The lack of modularity comes from the intertwined functionality, but also a practice of overwriting variables during iterations.

### A Use-Case for Logplot.m

1. Parameter Input

The user needs to find the parameters that affect the economic and physical conditions for his planned purchase. Since both technology and economic conditions change as time goes by, any result produced by the DST will be useless if the parameters are not up-to-date.

The parameters important to change for the user are: batt\_ratio, coeff\_cost\_BoSeI. coeff\_T\_pow, costBatt\_coef\_b, costBatt\_coeff\_a, costINV, costOeM\_spec, costPV, eff\_BoS, eff\_char, eff\_disch, eff\_inv, irr, irr\_nom, Load, LT, max\_batt, max\_PV, max\_y\_repl, min\_batt, min\_PV, n\_batt, n\_PV, r\_int, SoC\_min, SoC\_start, step\_batt, step\_PV, T\_amb, T\_nom, T\_ref, x\_llp and the function cycles\_to\_failure.

These parameters are hardcoded somewhere in the script. It depends on the users understanding of the different parameters, based on the comments that might accompany them and the parameter name, whether they should be researched and changed or not. There is no clear distinction between calculation variables and constant parameters.

1. Simulation Scope Input

The size of the components in the battery is defined by the variables min\_batt, max\_batt, step\_batt, min\_PV, max\_PV and step\_PV. This defines the simulation space. There will be n\_batt, n\_PV simulations based on these variables. If the user cannot find sizes in the scale of the load demand with appropriate LLP or NPC, an expansion of this range is necessary. If the user is interested in smaller variations between alternatives, for more detailed search, a smaller range and smaller step is necessary.

1. Run Simulation

The simulation will run through all the PV and battery combinations. Calculating a loss of load matrix LL. The calculation complexity is. The stages for simulation, economic analysis and optimum search are all executed within this step.

* 1. Plotting and Accessing Calculation Data

When you know what combination of PV and battery you wish to inspect, it must be hard-coded into the simulations part in order to access the data and make the necessary plots. In order to know what results occur during a simulation, a breakpoint must be inserted in order to view the data-set points before they are overwritten.

1. Solution Space Inspection/Refinement

The calculations that are not used later in the code is overwritten in every iteration, the user is unable to see exactly what has happened during the simulations. After the simulation you will be able to access the optimal values in the MA\_opt\_norm\_bhut\_jun\_10\_16 output matrix. These will tell you which PV/battery combinations in the predefined LLP range were found as optimal. The measure of quality is given by LLP, NPC and LCoE. At this stage the user knows better what range to examine closer based on these. Return to step 2-4 to inspect values with break points, and hardcode what solutions to plot. Repeat until the solution is satisfactory based on the LLP, NPC and LCoE values.

## Code Review and Redesign

Logplot.m was difficult to use and develop from a number of reasons. The formatting made the code hard to read, there were no naming convention to help developers to understand the script functionality and the script had no modularity to ensure safe development, testability and encapsulation of variables.

I observed that collaborators did not understand most of the functionality of the implementation, neither in a macro or micro perspective, this lead to downstream and upstream errors as the script was developed. The state of logplot.m also made development time-demanding, as each collaborator had to understand the entire script in order to confidently and safely make changes.

The restructuring of logplot.m to the new DST follows a union of several code quality standards, those of Richard Johnson (Johnson 2002) and (Butler, et al. 2009) set the main guidelines for naming convention, and the quality standards of the former and (Martin 2009) for macro architecture.

### Naming Conventions, Commenting and Formatting

The rewritten DST naming convention follows the quality standards from (Butler, et al. 2009). These metrics were found to be correlated with error frequency in in 8 established open source Java applications libraries. The different measures of quality is found in Table 1: Code Quality Metrics.

Some metrics can benefit from an explanation. The ‘Capitalization Anomaly’ states that regardless of the abbreviation, the capitalization should only be used as a substitute for white-space between words. The ‘Identifier Encoding’ metric state that the type, such as integer, double or string, should not be used as a Hungarian-style prefix, whereas the *kind* can be indicated this way. The reason is explained in (Spolsky 2005).

Table 1: Code Quality Metrics (Butler, et al. 2009)

|  |  |  |
| --- | --- | --- |
| Name | Description | Example of flawed identifier(s) |
| Capitalization Anomaly | Identifiers should be appropriately capitalized | HTMLEditorKit, pagecounter |
| Consecutive Underscores | Consecutive underscores should not be used in identifier names | foo\_\_bar |
| Dictionary Words | Identifier names should be composed of words found in the dictionary and abbreviations, and acronyms that are more commonly used than the unabbreviated form. | strlen |
| Excessive Words | Identifier names should be composed of no more than four words or abbreviations | floatToRawIntBits() |
| Enumeration Identifier Declaration Order | Unless there are compelling and obvious reasons otherwise, enumeration constants should be declared in alphabetical order | enum Card {ACE, EIGHT, FIVE, FOUR, JACK, KING ...} |
| External Underscores | Identifiers should not have either leading or trailing underscores. | \_foo\_ |
| Identifier Encoding | Type information should not be encoded in identifier names using Hungarian notation or similar | int iCount; |
| Long Identifier Name | Long identifier names should be avoided where possible | getPolicyQualifiersRejected |
| Naming Convention Anomaly | Identifiers should not consist of non-standard mixes of upper and lower case characters. | FOO\_bar |
| Number of Words | Identifiers should be composed of between two and four words. | ArrayOutOfBoundsException, name |
| Numeric Identifier Name | Identifiers should not be composed entirely of numeric words or numbers | FORTY\_TWO |
| Short Identifier Name | Identifiers should not consist of fewer than eight characters, with the exception of: c, d, e, g, i, in, inOut, j, k, m, n, o, out, t, x, y, z | name |

The quality measures from Table 1: Code Quality Metrics are sometimes conflicting, and some trade-off has to be accepted. The priority of the metrics in the DST rewrite are determined by the experience that was made when discussing the script with collaborators, and from the personal experience of working with the tool.

There is sometimes a conflict between the metric pairs of ‘Dictionary Words’ and ‘Naming Convention Anomaly’ opposed to ‘Number of Words’ and ‘Long Identifier Name’. Meaning that some dictionary words or common abbreviations are not available for simplifying the variable name without diminishing the self-explaining property of the longer name. In these cases, ‘Dictionary Words’ are prioritized because they maintain the naming convention that every line should be as self-explaining as possible.

My argument is that the DST does not usually make very complex calculations, such as numeric estimates, discretization or recursions. The consequence is a simple program flow, where the program-counter moves predictably. When the DST makes calculations, it is usually the theoretical equations that make the code complex. This results in single line calculations that can be understood alone, and is not depending on a larger logical architecture. This relieves the need of a compactly expressed code, and makes the use of ‘Dictionary Words’ a higher priority. This has a self-documenting effect, the code has a narrative that explain its functionality. There is however, few exceptions where the variable name is longer than 4 words in the rewrite (excluded class prefix).

The following conversion of names were applied to the DST. All variables, except those used temporarily in functions, are associated with a class. The full class names are prefixing the variables in the table below.

Table 2: the Complete List of Variable Name Changes in Rewritten DST

|  |  |
| --- | --- |
| Names | |
| Logplot.m | After Rewrite |
| batt\_balance | SimulationOutputs.neededBattOutputKw\*\* |
| batt\_balance | SimulationOutputs.neededBattOutputKw\*\*\* |
| batt\_i | jBatt |
| batt\_ratio | BatteryParameters.powerEnergyRatio |
| budget | EconomicParameters.budget |
| coeff\_cost\_BoSeI | EconomicParameters.installBalanceOfSystemCost |
| coeff\_T\_pow | PvParameters.powerDearteDueTemperature |
| costBatt\_coef\_b | EconomicParameters.battCostFixed |
| costBatt\_coeff\_a | EconomicParameters.battCostKwh |
| costBatt\_tot | EconomicAnalysisOutputs.battCostTot\*\* |
| costBoSeI\_tot | EconomicAnalysisOutputs.installBalanceOfSystemTotCost |
| costINV | EconomicParameters.inverterCostKw |
| costINV\_tot | EconomicAnalysisOutputs.inverterCostTot |
| costOeM | operationMaintenanceCost |
| costOeM\_spec | EconomicParameters.operationMaintenanceCostKw |
| costPV | EconomicParameters.pvCostKw |
| CRF | EconomicAnalysisOutputs.capitalRecoveryFactor |
| cycles\_failure | SimulationOutputs.sumPartialCyclesUsed\*\* |
| Den\_rainflow | nMaxPartialCycles |
| DoD | depthOfDischarge |
| eff\_BoS | EconomicParameters.balanceOfSystem |
| eff\_cell | cellEfficiency |
| eff\_char | BatteryParameters.chargingEfficiency |
| eff\_disch | BatteryParameters.dischargingEfficiency |
| eff\_inv | InverterParameters.efficiency |
| ELPV | SimulationOutputs.pvPowerAbsorbedUnused |
| EPV | (deprecated) |
| filename | (deprecated) |
| flow\_from\_batt | SimulationOutputs.battOutputKw\*\* |
| IC | EconomicAnalysisOutput.investmentCost |
| irr | SimulationInputData.irradiation |
| irr\_nom | PvParameters.nominalIrradiation |
| LCoE | EconomicAnalysisOutputs.levelizedCostOfEnergy |
| LL | SimulationOutputs.lossOfLoad |
| LLP | SimulationOutputs.lossOfLoadProbability |
| Load | SimulationInputData.load |
| LT | EconomicParameters.plantLifetimeYears |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(1) | OptimalSolutions.lossOfLoadProbabilities |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(2) | OptimalSolutions..netPresentCosts |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(3) | OptimalSolutions.pvKw |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(4) | OptimalSolutions.battKwh |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(5) | OptimalSolutions.levelizedCostsOfEnergy |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(6) | OptimalSolutions.investmentCosts |
| max\_batt | SimulationParameters.battStopKwh |
| max\_PV | SimulationParameters.pvStopKw |
| max\_y\_repl | BatteryParameters.maxOperationalYears |
| min\_batt | SimulationParameters.battStartKwh |
| min\_PV | SimulationParameters.pvStartKw |
| n\_batt | SimulationParameters.nBattSteps |
| n\_PV | SimulationParameters.nPvSteps |
| NPC | EconomicAnalysisOutputs.netPresentCost |
| num\_batt | EconomicAnalysisOutput.nBattEmployed |
| P\_pv | SimulationOutputs.pvPowerAbsorbed\*\*\*\* |
| peak | loadPeakKw |
| Pow\_max | battMaxPowerFlow |
| PV\_i | iPv |
| Pvpower\_i | iPvKw |
| r\_int | EconomicParameters.interestRate |
| SoC | SimulationOutputs.stateOfCharge\*\* |
| SoC\_min | BatteryParameters.minStateOfCharge |
| SoC\_start | BatteryParameters.initialStateOfCharge |
| step\_batt | SimulationParameters.battStepKwh |
| step\_PV | SimulationParameters.pvStepKw |
| T\_amb | SimulationInputData.temperatureC |
| T\_cell | pvTemperatureC |
| T\_nom | PvParameters.nominalCellTemperatureC |
| T\_ref | PvParameters.nominalAmbientTemperatureC |
| total\_loss\_load | SimulationOutputs.lossOfLoadTot\*\* |
| x\_llp | SimulationParameters.llpSearchTargets |
| YC | EconomicAnalysisOutput.operationMaintenanceReplacementCost\*\* |
| years\_to\_go\_batt | battOperationalYears |
| Stored for after-simulation inspections in the rewritten DST | |
| \* Hourly data points are now stored in this variable | |
| \*\* PV and battery combination data points are now stored in this variable | |
| \*\*\* Both time and PV/battery combinations are now stored in this variable. | |
| \*\*\*\* Time and PV iterations are now stored in this variable | |

The unit of variables is included in some names, in order to easily detect errors when writing the code. When the unit of the variable is obvious, for example when discussing power or load, the unit is implicit and can be excluded.

The variables iPv and jBatt are important to notice. They represent the current iteration in the simulation space, meaning the i’th PV iteration and j’th battery iteration. Outside the simulation execution, these represent a reference to the iterations, and are the index of simulations in the different matrixes. Meaning that if we have a pvStartKw = 100 and battStartKw = 100, the loss of load time series can be referred to in SimOut.lossOfLoad(:,1,1), here iPv = 1, and jBatt = 1. This way of referring to simulations is recurring in the rewritten DST.

#### Classes in the rewritten DST

Classes are the way the rewritten DST store variables in between functions and encapsulations, this has some great advantages. Classes allow for generation of variables automatically at initiation, hiding this repeated functionality from the user. The class SimulationInputData will for example only need the names of the files, and the folder name containing them to initiate the class. The result is a class containing all the data sets, variables for the length of the time series, and it will also generate warnings when the data-sets are of different lengths, and need to be preprocessed. This is all independent of which OS or computer you are currently using. The SimulationInputData constructor is displayed in Figure 2:1.

Figure 2:1 The SimulationInputData constructor function

A class implementation will make prevent assignments and accesses of variables that are not predefined for the class. For example if you attempt SimData.x Matlab will return: “The class SimulationInputData has no property or method named 'x'”. The class will also make passing of parameters to functions much less time demanding. If you need simulation parameters for a function, you can simply pass the simulation parameters all in one class. This allows for higher development agility and effective encapsulation of variables. As seen in

Lastly, the class will make the origin of the variables explicit, meaning that there is no question of what is meant by BatteryParameters.chargingEfficiency. This will assist the comprehension of the code.

A negative consequence is that the classes make variable names longer. These are however shortened in the different functions as they are passed. For example BatteryParameters.chargingEfficiency will typically be written BattParam.chargingEfficiency. This is ok because the number of classes are comprehensibly few and it contains its own full name as the type name.

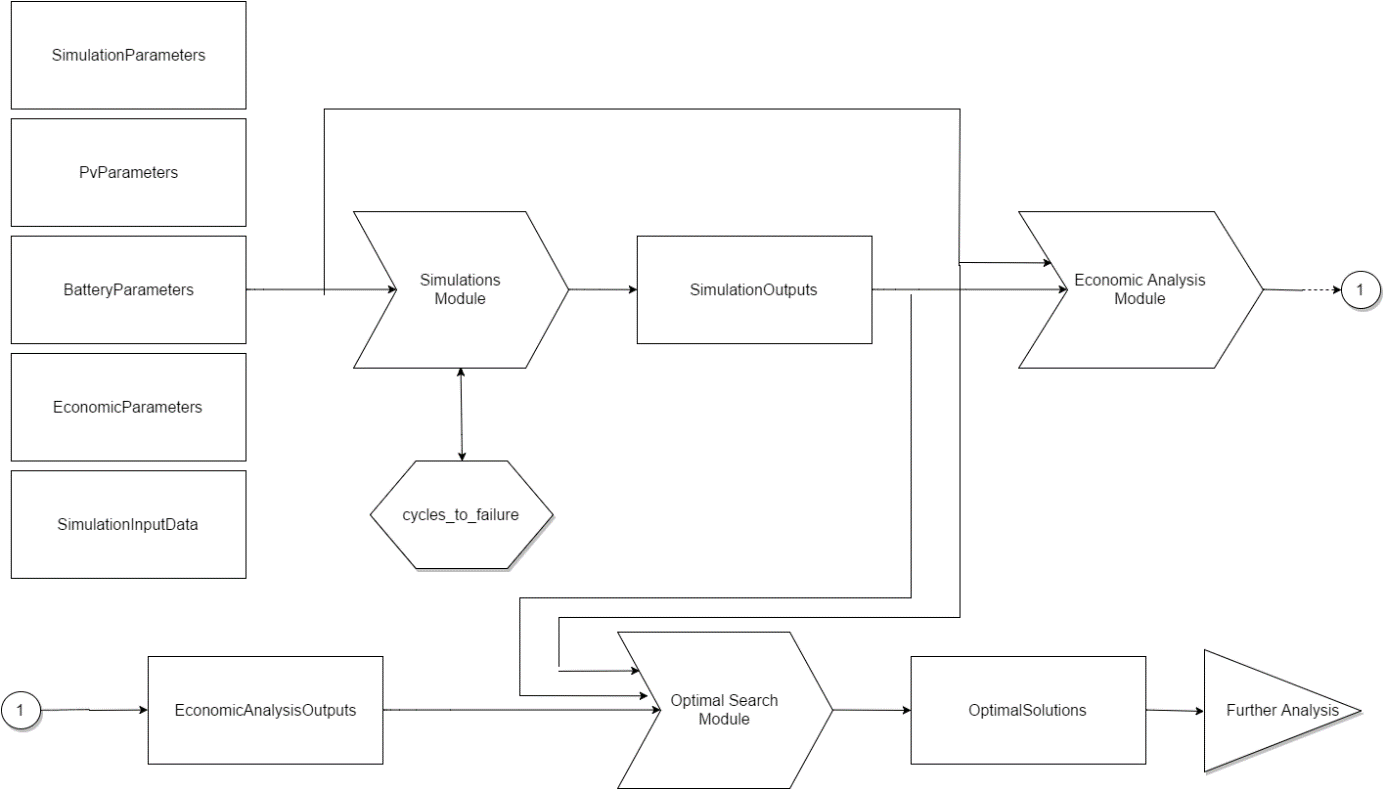


Figure 2:2: Class and Module Diagram

The classes enter the workspace when displayed in a rectangle.

### Modularization and Encapsulation

Describe the concept

Describe the implementation

Example the SoC and power balance average implementation

Encapsulation basically means that variables are restricted to their scope. This will protect variables from unwanted access or interference, which results in a safer implementation. In the rewritten DST, encapsulation is mainly used to keep the workspace clean. When entering a function in Matlab, the variables that are passed to the function will be copied. The only calculations that comes out of a function is therefore the output variable of the function. This way, all variables that are irrelevant outside the function, are deleted.

Each module has an associated output class in the rewritten DST. The result is a workspace with only input and output classes (See Figure 2:2: Class and Module Diagram). This defines a clear intersection between modules. This also means that modules can be used in any order desired, or even used inside each other. A single simulation can be called simply by restricting the simulation parameters, or the plotting function can be repeated multiple times with different solutions.

This configuration allows for agile development and testing, as modules can be tested or replaced quickly without changing the rest of the code. If you wish to plot a solution, it will be enough to pass one or two classes as arguments to the plotting function. If you wish to modify an existing module, it can simply be copied and modified as desired without losing the previous functionality.

The modularization in the rewritten DST is based on the principle that one thing should be done at a time. It’s been said about quality code that it “does one thing well” (Martin 2009), this cannot be followed too strictly in this Matlab implementation. The reason being that the data-sets are too large to continuously pass between functions. Function calls in Matlab are ‘call by value’, meaning that a variable passed to a function is copied to the function scope. The potential memory consumption and computation overhead while doing this, could affect the development and testing flow. Hence, further modularization is reserved for a python implementation, where variables are pointers and ‘call by reference’ is default behavior.

### Formatting

Due to increased length in variable names, a consistent column style of arithmetic. The equations are aligned by the operators as seen in Figure 2:3. The idea is that the user will get used to seeing the equations in this form, so the predictability of the format will remove ambivalence and doubts while speeding up equation comprehension. This is introduced because of the wide formatting in Matlab and the increased length of variable names in the rewritten DST. This way the user can read the script without scrolling. This formatting might not be necessary for readability in other languages.



Figure 2:3Rewritten DST formatting example

### Evaluation of Finished Rewrite

In order to make a quantitative comparison of the rewrite and the original code, the bugs from the original DST was replicated in the rewritten DST. This way the changes in architecture is isolated and will produce the exact same data, if the implementation is correct. The relevant module is only Economic Analysis, the replicated module is named bugged\_economic\_analysis and prints a warning when run.

#### Correctness

Comparison of different outputs is used to prove maintained correctness. The function isequal(A,B) will compare every element in two matrices, and returns true (1) if the matrices are identical, and false (0) if there are one or more element with any kind of difference.

Given an input with large span and high resolution, there will be enough data points to safely state that the new implementation conserve the functionality from the old. We assume that the probabilities, the probability that a data point is equal despite an erroneous implementation, are independent. We also assume that these probabilities are independent of each other, and if they should be independent, the implementation would be correct. The probability of two matrices instead of data-points being equal, can then be described as follows.

One last assumption is that the average is not larger 50%. We wish to be at least 99.999% sure that our implementation is correct. We will use this assumption to state that consecutive correct answers despite of a wrong implementation is, and consequently the minimum required number of data points are:

For one time series in the DST we have 8760 data points in hours, which alone is sufficient to ensure correctness, but only for the modules that utilize the entire time series. When we evaluate the modules with lesser complexity, the data points decrease, therefore we need more complexity than one time series for testing the entire DST functionality.

Table 3: Complexity and output source of compared simulation pair outputs

|  |  |
| --- | --- |
| Variable Names | |
| Before Rewrite | After Rewrite | Module | n Data Points |
| SoC | stateOfCharge | sapv\_plant\_simulation |  |
| LL | lossOfLoad | sapv\_plant\_simulation |  |
| NPC | netPresentCost | economic\_analysis |  |
| LCoE | levelizedCostOfEnergy | economic\_analysis |  |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,1) | lossOfLoadProbabilities | llp\_constrained\_optimum |  |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,3:4) | [pvKw, battKwh] | llp\_constrained\_optimum |  |

In the rewritten DST, the calculations are executed in modules that pass parameters between them. These are placed in three classes that describe the key outputs of each module. Logplot.m is rewritten so that it saves more variables to workspace, we can now compare with the rewritten DST.

Given the validity of the coarse calculations above, we assume more than 99.99% certainty that correct calculations are a result of a correctly rewritten DST. Given that these assumptions are incorrect, some redundancy is introduced to compensate. Two simulation pairs are executed. The first simulation has PV and battery combinations, and 15 optimal solutions. The second simulation pair span PV and battery combinations.

The two comparisons are displayed below.

% Paramaters for Simulation Solution Space

SimParameters = SimulationParameters;

SimParameters.pvStartKw = 100;

SimParameters.pvStopKw = 200;

SimParameters.pvStepKw = 5;

SimParameters.battStartKwh = 1200;

SimParameters.battStopKwh = 1300;

SimParameters.battStepKwh = 5;

SimParameters.llpSearchAcceptance = 0.005;

SimParameters.llpSearchTargets = 0.01:0.005:0.30;

Figure 2:4: Parameters for simulations that produce comparison outputs for simulation pair 1.

Table 4: Results of comparisons in simulation pair 1

|  |  |  |  |
| --- | --- | --- | --- |
| Output A | Output B | n Data Points | Isequal(A,B) |
| SoC | stateOfCharge | 3863160 | True |
| LL | lossOfLoad | 3863160 | True |
| NPC | netPresentCost | 441 | True |
| LCoE | levelizedCostOfEnergy | 441 | True |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,1) | lossOfLoadProbabilities | 15 | True |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,3:4) | pvKw, battKwh | 15 | True |

% Paramaters for Simulation Solution Space

SimParameters = SimulationParameters;

SimParameters.pvStartKw = 150;

SimParameters.pvStopKw = 170;

SimParameters.pvStepKw = 5;

SimParameters.battStartKwh = 1150;

SimParameters.battStopKwh = 1270;

SimParameters.battStepKwh = 5;

SimParameters.llpSearchAcceptance = 0.005;

SimParameters.llpSearchTargets = 0.10:0.005:0.80;

Figure 2:5: Parameters for simulations that produce comparison outputs for simulation pair 2.

Table 5: Results of comparisons in simulation pair 2

|  |  |  |  |
| --- | --- | --- | --- |
| Input A | Input B | n Data Points | Isequal(A,B) |
| SoC | stateOfCharge | 183960 | True |
| LL | lossOfLoad | 183960 | True |
| NPC | netPresentCost | 625 | True |
| LCoE | levelizedCostOfEnergy | 625 | True |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,1) | lossOfLoadProbabilities | 53 | True |
| MA\_opt\_norm\_bhut\_jun15\_20\_10(:,3:4) | pvKw, battKwh | 53 | True |

As seen in Table 4: Results of comparisons in simulation pair 1, and Table 5: Results of comparisons in simulation pair 2, the output matrices are equal. This proves that the rewritten DST preserves the functionality of Logplot.m

#### Computation Speed

The computation speed is a specification in (Mandelli, et al. 2014) and should be maintained in the rewritten DST. An unacceptable change in computation time is when the computation complexity is increased. A small proportional gain is in most non real time application not a problem. The DST is still functional at double computation speed. But a large computation time increase could reduce the users’ perceived quality and satisfaction with the system, and should be avoided.

Matlab has a Profiler timing tool which is used to evaluate the rewritten DST. This tool will list every function call made either explicitly or implicitly by Matlab. The tool will display how much time a function spend waiting for other functions, and how long the program counter works inside the function. The latter is called ‘Self Time’, and will benchmark calculations done within the function without calling other functions. When time is spent waiting for other functions, one should inspect the calls in order to determine if they are necessary. One method of shortening function calls can be to define the functions inline in the script, rather than in a separate file. This was tested with the cycles\_to\_failure function. The improvement was less than 0.1%. The function was kept external to maintain modularity.

The profiler tool output is displayed in Table 6 and Table 7. A large bright blue band will indicate that there might be possible improvements in reducing function calls, and dark blue will indicate self-time.

The rewritten DST will call the modules from a main function. The economic analysis modules is called bugged\_economic\_analysis because a bugg from logplot.m is replicated in order to compare the two programs.

Table 6:The Profiler Tool run on the previous DST. Only the 7 top functions are displayed

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Function Name | Calls | Total Time | Self Time\* | Total Time Plot (dark band = self time) | | [logplot](file6.html) | 1 | 4.063 s | 3.986 s |  | | [cycles\_to\_failure](file31.html) | 36500 | 0.046 s | 0.046 s |  | | [finfo](file17.html) | 3 | 0.031 s | 0.015 s |  | | [importdata](file19.html) | 3 | 0.031 s | 0.000 s |  | | [matfinfo>matfinfosub](file25.html) | 3 | 0.016 s | 0.000 s |  | | [mat2str](file26.html) | 3 | 0.016 s | 0.016 s |  | | [matfinfo](file28.html) | 3 | 0.016 s | 0.000 s |  | | […] | […] | […] | […] | […] | | **Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling. | | | | | |

Table 7: The Profiler Tool Run on the rewritten DST. Only the 7 top functions are displayed

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Function Name | Calls | Total Time | Self Time\* | Total Time Plot (dark band = self time) | | [main](file1.html) | 1 | 5.761 s | 0.000 s |  | | [sapv\_plant\_simulation](file22.html) | 1 | 5.698 s | 5.651 s |  | | [cycles\_to\_failure](file28.html) | 36500 | 0.047 s | 0.047 s |  | | [importdata](file9.html) | 3 | 0.032 s | 0.016 s |  | | [...mulationInputData.SimulationInputData](file21.html) | 1 | 0.032 s | 0.000 s |  | | [bugged\_economic\_analysis](file23.html) | 1 | 0.031 s | 0.031 s |  | | [fileparts](file4.html) | 6 | 0.016 s | 0.016 s |  | | […] | […] | […] | […] | […] |   **Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling. |

We can see that the increased computation time is in the Self Time column, the dark blue band, which means that Matlab does not call many additional help functions. The main function in Table 7 has a large bright blue band, because it calls the other module functions. The important take away is that the increased computation time is in ‘Self Time’.

The overhead we experience is not because of calculations, but can be explained as a result of the overhead caused by copying large amounts of variables from the main functions to the modules. Additionally, the rewritten DST has much larger amounts of output data, which will make for memory access and allocation overheads.

The memory access and allocation overhead can be reduced by implementing call-by-reference instead of call-by-value modules. Matlab has handle classes and value classes. In the rewritten DST the value class is currently used. The handle class works like a ‘call to reference’ function in the C language, and will be faster than copying the values. The decision was made to keep the value class implementation. It is likely to be easier to debug, pointer passing tend to produce more cryptic errors, hence complicating development. The pointer functionality can be implemented when the DST is being prepared for shipping.

The overhead in the rewrite is acceptable because of the added modularity and simulation history accessibility in the DST. The only time simulations will be rerun is when the simulation space does not include any desirable solution, whereas logplot.m had to be rerun for a number of reasons such as plotting and investigating a simulation. The user will be able to investigate the simulation space thoroughly and run numerous economic and optimization modules without repeating the simulation step in the rewritten DST.

## Graphical User Interface

For development purposes, a graphical user interface (GUI) is implemented as a part of the DST. The GUI will help developers to understand the needs of the user and works as a prototype for the finished DST. The user interface programing in Matlab involves initiating different uicontrols. These are objects that are constructed by setting their type, appearance, position and defining their call-function.

The call-function is called every time the object is interacted with. In figure Figure 2:6 and Figure 2:7 an example of a uicontrol initiation is displayed, and its call-function.



Figure 2:6 The uicontrol initiation for the 'Run Optimal Solutions' button



Figure 2:7 The call-function when pressing the 'Run Optimal Solutions' button

A use-case of the DST with GUI functionality is sketched in section 2.7.

### dst\_platform.m

The dst\_platform GUI is meant as the start platform for a use-case. The main function of the GUI is to initiate the parameter classes, and pass these classes to different modules depending on the GUI run-settings. This tool vastly simplifies testing of new functionality. One can run a new module by changing which function to be called at button press, make a new run button, or simply use the tool to initiate the classes and pass the parameters manually.

The GUI can save presets, meaning that any combination of inputs and settings to the dst\_platform can be saved and will load when it is chosen from the dropdown menu. This will allow the user to inspect the differences between different input parameters, and retrace old settings at a later time. Saving is done by entering a name for the current preset and pressing ‘Save’, this will overwrite any preset with the same name. Pressing ‘Delete’ will delete the currently chosen preset in the dropdown menu.

|  |
| --- |
| \\webedit.ntnu.no\gardhi\dstReferenceManual\dst_gui.png  Figure 2:8: the DST Platform GUI |

The dst\_platform also has a plotting functionality to get a comprehensive overview of the simulation output space. This plot will appear when pressing the ‘Simulation Overview’ button. A detailed description can be found in section 2.4.3.1 Simulation overview.

The ‘Help’ button will open a browser window with the online documentation of the DST, this is described briefly in section 2.6 Documentation.

The ‘Input Data Files’ panel is where the filename of the different input files are passed from. The ‘Data Set’ field is the folder with the different files belonging to the same data-set.

The ‘Run Solution Explorer’ button will open the solution\_explorer GUI which is designed to compare and evaluate solutions found by the optimum search. The Solution Explorer is described in detail in section 2.4.2 solution\_explorer.m.

#### Running-modes Available from the GUI

Under the ‘Run Calculations’ header, there is a checkbox for ‘Generation Strategy – Biomass’. This run-mode will be described in section 2.5 Biomass System. When checked, the system will simulate with a generic operation of a biomass system, parallel with the PV panel.

The ‘Optimal Solution Output’ radio-button choices below the biomass checkbox decide which optimization module that should run. The ‘LLP Constrained’ optimums is the original optimization from logplot.m, this is now the llp\_constrained\_optimums function. This module will iterate through a set of LLP values. If one or more matches of LLP are found in the simulation output space (with a given acceptance), the simulation with the lowest NPC is chosen as the optimal solution.

The ‘NPC Constrained’ optimum search works the same way, the difference is that instead of searching a range of LLP, it searches a range of NPC, and picks the smallest LLP as the optimal solution when a match is found. The module is implemented in a function called npc\_constrained\_optimums.

The ‘LCoE Constrained’ optimum search will call the lcoe\_optimums module. This function simply chose the lowest LCoE in the simulation output space. It will also generate a warning if the lowest LCoE is found at a boundary, indicating that there might still be lower LCoE values in solutions if the simulation space is expanded further.

### solution\_explorer.m

|  |
| --- |
| \\webedit.ntnu.no\gardhi\dstReferenceManual\solution_explorer.png  Figure 2:9 The Solution Explorer |

### Plots

Plots are important in the DST. Users can visualize the hourly levels in order to understand how the system work, but also when and how much the micro grid fail. The previous DST would not allow any single function to run without running entire simulations. In the case of plotting, one would have to simulate the system in order to find what solution (combination of PV and battery) one wished to examine, then one would have to hard-code the DST to plot this solution as it was iterated over. This continuous hard coding and redundant computation was vastly time consuming. The rewritten DST has one function for each plot.

#### Simulation overview

#### State of Charge

The existing SoC plotting was not modified extensively. It was made a module (function) and renamed, and will also access any given simulation. This is a very useful plot, it gives the user an intuitive impression of how the micro grid performs in simulation.

The green plot line is power from the PV that neither serves load nor charges batteries because the batteries are already fully charged. The red plot line indicate loss of load in kW as there is no sufficient output to serve the load. As the optimization works today, the DST minimize the LoL (the negated red line).

|  |
| --- |
| Figure 2:10: A full year overview from the State Of Charge plot |

|  |
| --- |
| Figure 2:11: A more detailed part of the State of Charge plot |

#### Power Balance

To understand the simulation results, it will be helpful to inspect how the PV and batteries interact when serving the load without considering parameters such as Balance of System, and in efficiency loss between charging, discharging and inverter. This way we can see how power is in effect through the micro grid. We can also potentially see which component that fail to serve the load, whether it is inability to discharge fast enough, or because the batteries are fully discharged, or if the irradiation levels are too low.

The previous DST had an incorrect implementation of this. Upon inspection, it can seem like the cause is a lack of understanding the variables. The implementation with the new naming convention can be seen in Figure 2:10, with the old naming convention in Figure 2:11. The legend that describe the yellow plot line (seen in Figure 2:12) reads ‘Energy from PV’. This line is actually the absorbed power, not the utilized power. This means that some of this power will never serve the load, nor charge the batteries. The variable name P\_pv does not help to clarify. The legend that describe the brown plot line, reads ‘Energy flow from battery’, in reality, this is the needed battery output, demanded by the load. This means that the plot will express a perfectly functioning system, as all the ‘needed power’ is branded wrongly as ‘supplied power’. Like earlier, the variable name bat\_balance\_pos does not help to clarify.

The subplus function is used to extract only the positive values from an array. In the case of negative neededBattOutput values, it means energy that can be used for charging the batteries.

|  |
| --- |
|  |

Figure 2:12: The previous attempt at making a power balance plot in the outdated DST. This code has the new naming conventions.

|  |
| --- |
|  |

Figure 2:13: The same code as in Figure 2:10 with the old naming conventions.

|  |
| --- |
| Figure 2:14: The old attempt at making a power balance plotThe ‘Energy from PV’ and ‘Energy flow from battery’ is in reality the potential energy absorbed by the PV, and the needed battery output to meet the load demands respectively. |

|  |
| --- |
| Figure 2:15: The same error that is explained in Figure 2:12 is replicated in the daily average version. |

|  |
| --- |
| Figure 2:16: The calculation of an average day in the previous DST. Complexity |

Because the concept of a Power Balance plot is useful, a new function was implemented to plot the intended functionality of the previous attempt. The code is displayed in Figure 2:16. The three plot lines are irradiationUtilized, load and netLoadSupply.

irradiationUtilized is simply the pvPowerAbsorbed – pvPowerAbsorbedUnutilized. The latter variable accounts for the power that is wasted when the batteries are fully charged, and when the batteries maximum charging kW is exceeded. Meaning that the batteries can’t charge fast enough to exploit all the power from the PV.

The batteryNetOutput is retrieved from the battOutputKw positive values with subplus, with efficiency loss excluded. Meaning this is the power from the batteries that actually reaches the load. battOutputKw represent the actual flow from battery. Previously, battOutputKw was scalar and got overwritten at each iteration. The variable was modified to have dimensions for each battery/PV combination and time-step. This way, any simulation scenario can be reviewed without additional computation.

In the case of insufficient battery capacity to supply load, the variable would remain as if the load demand was met. This was corrected as seen in Figure 2:15. The tentative stateOfCharge variable represent the percentage of battery that is needed for the load. The difference between stateOfCharge and minStateOfCharge is the demand that exceeds the minimum SoC, this is converted back to kW and removed from the battOutputKw variable (because it’s never output from the batteries).

|  |
| --- |
| Figure 2:17: Correcting the battOutputKw variable to not account for power that is not output when batteries are at min SoC |

The pvNetLoadSupply is the power that is actually supplied to the load. This can occur in two ways. Either the neededBatteryOutput is negative, meaning that the power from the PV is greater than the load. In this case the PV power supply the load directly, and the excess power charges the battery. The values of load is therefore equal to the pvNetLoadSupply at these points. In the second case the power from the PV is insufficient to serve the load, meaning the neededBattOutputKw is positive, and all the pvPowerAbsorbed goes to serve the load directly and is therefore equal to pvNetLoadSupply.

|  |
| --- |
| Figure 2:18: The new power balance plotting implementation. There are two functions, this one makes the average day plot, the other one plot for every hour of the year.. They are identical except for the calculations underneath the %averages comment and the plotting of these averages instead of their original arrays.. |

|  |
| --- |
| Figure 2:19: The get\_daily\_average function calculates the average day of any array with hourly time increments. Complexity . This a faster and generic algorithm that both improves readability and ease of implementation (as seen in Figure 2:16). |

The decision was made to plot the netLoadSupply, instead of the pvNetLoadSupply and battNetLoadSupply separately. When the two are plotted separately it is harder to see whether the summed plots cover the load or not. It will not tell us accurately whether the system fails or succeed. The origin of power is still clear in the plots, despite this. When irradiationUtilized is non-zero, the PV power will always first cover the load, if the irraditationUtilized is zero, the battery will support the load alone.

The integral of irradiationUtilized and netLoadSupply will sum to zero, this way one can see the proportions of energy input and load demand. When there is loss of load, the netLoadSupply will not cover the load plot line. This will help the user understand the magnitude of the lost load. The new colors aim to appeal to intuition. The red line alone is very visual, and signals that there is LoL occurring. If the system is functioning well, the red line is hidden by the blue line. The irradiationUtilized is yellow so that the user understand its role, without reading the legend.

The resulting plots can be seen in Figure 2:18 and Figure 2:19. The functions are easily maintained and modified due to the modularization and strict naming conventions.

|  |  |
| --- | --- |
| Figure 2:20: The new power balance plot. Here the yellow plot lines represent the power that is actually utilized from the PV. The blue line represent the net power to serve the load from PV and batteries. The red line is the load demand in Kw. The blue line will overlap the red line whenever the load demands are met. | |
| Figure 2:21: The Average Day Power Balance.  This is a very powerful plot for understanding the general status of the microgrid, because it explains how the batteries and PV interact. |

## Biomass System

## Documentation

## A Use-Case for the DST

# Summary and Recommendations for Further Work

## Summary and Conclusions

Through renaming every variable to optimize the readability of the DST, the design flaws and inner working of the tool became apparent. It was discovered that certain parts of the hard coded and hard-to-find parameters were responsible for….

## Discussion

## Recommendation for Further Work

### The Rainflow Counting Algorithm

The Rainflow Counting algorithm is an algorithm initially used to account for stress exposure in materials, initially in full cycles, and later in partial cycles. One cycle is one instance of full stress exposure, a partial cycle is an instance of a partial stress exposure. The algorithm has since been developed to account for stress in batteries (You and Rasmussen 2011) and other appliances that go through similar wear.

The DST is intended for micro and small scale enterprises (MSSE). These are likely to rely on small economical margins.

### Original Algorithm: (You and Rasmussen 2011)

1. Initiate a vector in encounter of a stress local minimum.
2. Note increase in stress during rainfall of vector
3. Count occurrences of ranges as one cycle
4. Sum the expended partial cycles for every stress level accounted for

Where n represents the number of bins chosen in the study; Nc(DOD) represents the number of consumed partial cycles at a given DoD level, derived by counting in the corresponding period; No(DOD) represents the maximum number of partial cycles that can be performed before battery failure at that DoD level.

Where ExpL denotes the expected lifetime of the BS, and Tp represents the length of the counting time period.

### Implemented Algorithm:

1. Discover discharge valleys (can currently only occur after 8 consecutive hours of discharging)
2. Count cycles to failure:
3. Accumulate cycles to failure
4. Estimate battery lifespan

A requirement for finding the lifespan in years, is that the spendage of lifetime fractions are summed over one year precicely. This way we get the amount of years that the battery need.

### Improving the Implemented Rainflow Algorithm

Every input related part of the DST should assume a generic form. This implementation assumes the that the rainflowCounter will keep counting for exactly one year, there was therefore added a year counter to the algorithm as seen when calculating ExpL in the previous implemented algorithm. This makes the algorithm more generic, it will still work when the input time range changes.

|  |
| --- |
| Figure 3:1: Plotting DoD(Cycles To Failure) w/initial parameters.  There are not sufficient points of DoD in the intervall 0-20% to represent the Cycles to Failure accurately. |
|  |

The cyclesToFailure is calculated for every instance of DoD valley that occurs, which has an complexity of **O(n).** This resolution in cycle values might not be required considered the massive abstraction level of which we are operating. A future extension of the DST will at some point trigger far more occurences of DoD, before a simulation is considered complete. Users might want to input longer time series or change the 8 hour consecutiveness condition. In earlier versions of the algorithm (Downing and Socie 1982), the method proposed is preemptively generating a table of Cycles To Failure for every DoD percentage.

If the rate of change in DoD is not too close to zero, then the points of DoD will sufficiently describe the Cycles To Failure, as there are many values of DoD for each value of Cycles To Failure. By inspecting Figure 3:1, here the points of DoD percentage per cycles varies from 0.0001 to 0.01. It would not be safe generate an array with 1% resolution, the DoD rate is too low until about 20%. We can not guarantee that the DoD valleys wont occur in the 0%-20% range, even though it is less typical of a discharge cycle. One solution is to increase the resolution of the DoD values to 0,1%, or perhaps smaller, in the precalculated array. The increased resolution will decrease computational gain. The DST overall complexity is , and the algorithm triggers a maximum of 3 times per day. The decision was made to keep the initial algorithm that calculates on each occurrence of DoD minimums. The computation time gain of employing a precalculated array is too small, compared to the extensive testing and assumptions required, to defend this modification.

The 8-hour consecutiveness condition is similarly implemented in (Downing and Socie 1982), here the condition is 3 points. This can be understood as a lowpass filtering of the input. To scrutinize test changes in this condition is unnecessairy considering the following facts.

1. This technique is legitimate in the reference material. Meaning that the physical consequences of “flutter” between the considered cycles is likely to be neglectable.
2. If we inspect our typical SoC as seen in Figure 2:8 and Figure 2:9 the cycles have consistent cycles of 12 hours. i.e the flutter is unlikely to occur in the first place.

### Simulation Borders Analytically

The DST relies initially on producing a very large solution space. The user will usually have to make several simulations, to pinpoint the desired resolution for analysis. For example, if a user is not familiar with the PV sizes and battery sizes usual for the load scale under consideration. The user will have to make a wide simulation initially, and then make a finer solution space after examining the solutions produced the first round.

This narrowing down can be somewhat eliminated.

# Appendences

1. HTML Documentation
2. Old DST: Logplot.m