|  |  |
| --- | --- |
|  |  |
| LCoE | Levelized Cost of Energy |
| NPC | Net Present Cost |
| LLP | Loss Of Load Probability |
| GUI | Graphical User Interface |
| UI | User Interface |
| DoD | Depth of Discharge |
| DST | Development Support Tool |
| LoL | Loss of Load |
| MSSE | Micro and Small Scale Enterprises |
| PV | Photovoltaic |
| SoC | State of Charge |
| OeMeR | Operations, maintenance and replacement |
| OeM | Operation and maintenance |
|  |  |
|  |  |
|  |  |
|  |  |

**Abstract**

Objective

Material and Methods

Results

Conclusions

150-200 words

# Introduction

## Background

The following chapter contains the problem formulation, literature survey, a presentation of the program under development and of what possible development this implies.

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

PV Power Generation

Biomass Power Generation

### PV Power Generation

### Biomass Power Generation

### The Development Support Tool

The original functionality of the tool, excluded any implementation details, is 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 LoL 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.

## Problem Formulation

## 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

Reference to literature in code quality

### Creating a user interface

### Simulating a biomass power system

Why Biomass?

Alternatives?

### Documentation

## Structure of the Report

The rest of this thesis mainly consist of the methodology and corresponding discussion of what has been done. During the methodology part each successful addition to the DST is explained. The explanation starts with presenting the methods and necessities.

# Methodology

## 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.

It was observed that collaborators did not understand most of the functionality in logplot.m, not 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 detailed 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. During simulation we have for loops for PV iPv = pvStartKw : pvStepKw : pvStopKw, for battery jBatt = battStartKwh : battStepKwh : battStopKwh, and time t = 1 : nHours. When accessing data points later we use the same variable name as a place holder. Meaning that if we have a pvStartKw = 100 and battStartKw = 100, the simulation outputs for LoL can be accessed as 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.

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 2:1 The SimulationInputData constructor function

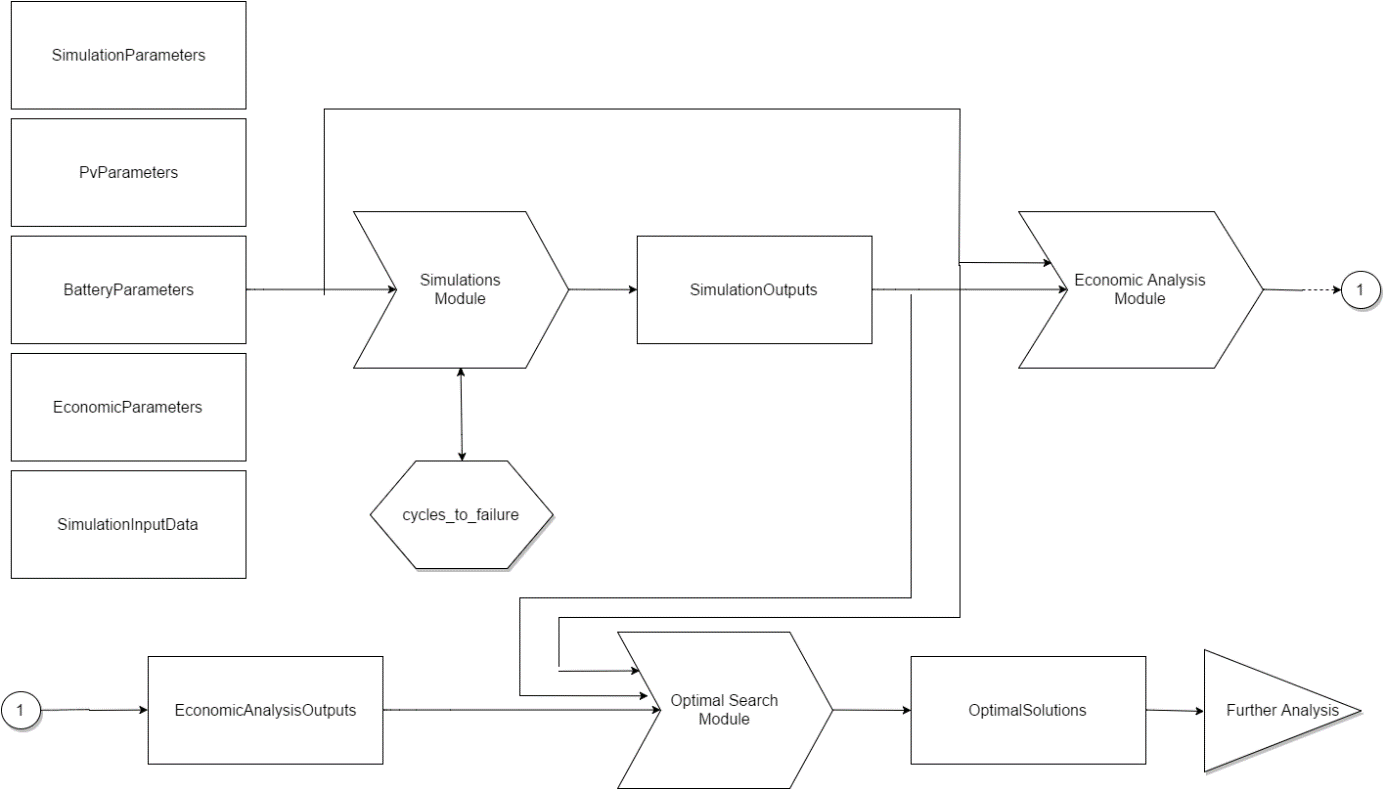


Figure 2:2: Class and Module Diagram

The classes enter the workspace when displayed in a rectangle.

### Modularization and Encapsulation

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.

## Biomass System

A generic system for simulating the use of a biomass system is implemented in the module function pvbiomass\_plant\_simulation. This module is identical to the sapv\_plant\_simulation in the operation of battery, but also has the contribution from a biomass system for power production. The contribution is added to the needed\_batt\_output variable. This variable is pvPowerAbsorbedKw – load when no biomass generator is running. The system is meant to simulate the use of a generator parallel to the stand-alone PV plant to compensate for low irradiation.

The behavior is not one particular system, it is supposed to be any system choice for generating power with biomass. The system will also be able to simulate whether it is manually operated or fully automated.

### Preemptive Run-mode

The PV usually can have a very high output capacity in terms of kW. To replace this potential capacity can require a very large generator. A large generator is expensive and hence a primary solution rather than a secondary solution. Also, a biomass fueled power generation greater than 30 kW is not typically available off-the-shelf. When running with batteries, we have the opportunity to generate power over time, and hence meeting the demands with a lower output capacity, hence a considerably cheaper solution.

The system assumes that a day with irradiation levels lower than a certain threshold can be predicted with a weather forecast. The power generation can then be started before the battery is unable to meet load demand. For simplicity, we term the low irradiation days as cloudy days and high irradiation days as sunny days.

To simulate a forecasts, the algorithm calculates the pvPowerAbsorbed vector, and finds the peak value. The pvPowerAbsorbedKw values are smoothed because of the nominal irradiation and temperature factors of the PV operation. This calculation is displayed in Figure 2:6 The calculation of the pvPowerAbsorbed vector.

Using the pvPoweAbsorbedKw instead of the irradiation directly, will dampen any irregularly high peaks and give a more intuitive peak value. A threshold is given by the user as a fraction named peakPvPowerAbsorbedTreshold. This threshold will determine if a day is forecast as cloudy or sunny, depending on its relative size compared to the global peak. This comparison is displayed in Figure 2:7 The comparison of peakPvPowerAbsorbedKw values.

At the start of a day, the peak pvPowerAbsorbedKw of the following day is compared to the peak pvPowerAbsorbedKw value of the entire simulation time series. If the peak of the current day is lower than a fraction of the global maximum, the weather is forecast as cloudy, and vice versa.



Figure 2:6 The calculation of the pvPowerAbsorbed vector



Figure 2:7 The comparison of peakPvPowerAbsorbedKw values

When a day is forecast as cloudy, the system will run in preemptive mode until the battery is full, a sunny day is forecast or the biomass/fuel has run out. The entire state machine can be seen in section 2.4.5.

### Startup Delay

The most important generic simulation choice is the startup\_delay parameter, which become relevant during LoL occurrences.

The time needed to start the biomass power generation will vary between solutions. The alternatives range between a very fast response time, for example a gas generator with automated startup at LoL, and a very slow response time, for example a furnace driven steam turbine or an incomplete combustion process, where the response is initiated when an operator discovers the LoL. The very worst case can span hours, as the operator might have to spend time on getting to the plant to operate it, and perhaps additional delays.

The implementation alternative of biomass simulation will involve choosing a generation strategy, such as furnace powered steam generator, and conversion rates of biomass kg to kWh, and respond delays. This implementation would be much like mapping numbers to text, and would do the user a disfavor in hiding the implications of his choices. The preprogrammed time estimates might also not apply because of tweaks or new innovation to the system. A better solution is the one proposed, and complement the solution with information about startup delays and other implications available to the user in online documentation.

### Waiting to Retry

When the biomass system produce power, it will check whether the grid is offline despite the additional contribution. If this is the case, the consumed resources are wasted as the grid is still not functioning. The simulation has a parameter for how long the user prefers to wait if this happens. If the waiting time is zero, the system will keep running, if not the system will start running when the waiting time has passed.

### Biomass kg to kWh Conversion

The conversion from biomass kg to kWh will vary between biomass plant solutions, and the conversion rate is therefore left to the user, the biomass supply parameter unit is already in kWh. This is because of the two factors that will be specific for the user’s situation, the plant efficiency and the biomass potential energy density.

The plant efficiency depends on the kind of system implemented, how it’s operated and the system’s quality of design. The biomass potential energy depends on the biomass kind and how the biomass is utilized or refined to power the system. The kind of biomass can also be subject of geographical variations. Because of the amounts of permutations here, it is left to the user to consult the biomass system manufacturer, people with experience from a similar plant and with the same materials and potentially make some assumptions in order to find the supply in kWh.

An implementation that does the conversion job for the user carries a large risk of making imprecise or outdated assumptions, which can be cascaded and therefore amplified, when combined with other imprecisions. The conversion is simple math with many parameters, which will be simple for anyone with the resources to implement a power plant. This way the user maintains control of the assumptions. The monitoring of produced kW should be simpler than weighing, as biomass also change weight and potential energy as the moisture changes.

### The Biomass-System State Machine

The implementation of the system biomass simulation was done with a state machine. A state machine has the benefit of only allowing the system to be in one state at any time. This will simplify check of conditions as a state has a limited number of allowed transitions. The state machine implementation will make the code more readable and the program flow comprehensive. It also provides a sketch of the desired functionality. This is useful when implementing as a graphical specification.

The transition diagram can be seen in Figure 2:8. The diagram illustrates the transitions between states without any implementation details.

\\webedit.ntnu.no\gardhi\dstReferenceManual\biomassSystemStateMachine.png

Figure 2:8 Biomass System State-Machine Transition Diagram

### pvbiomass\_plant\_simulation.m

The system state is kept in the variable biomassSystemState. This can be in the following states ‘Idle’, ’Starting Up’, ’Running, ’Running Preemptively’, ‘Waiting to Retry’ and ‘Waiting for Biomass’. The states can perform two actions, they handle transitions to new states, and they set the runBiomassGeneratorHours variable. This variable decides how long the biomass generator should run the current hour. The generator effect code block is executed after the state machine. If the runBiomassGenerator is positive, for example 0.5, it means that the generator should run for 30 minutes. When power contribution of running time is processed, the runBiomassGenerator variable is set back to zero for the next iteration.

When the biomass system generates power, the contribution is subtracted from the neededBattOutputKw variable, the same way power contribution from the PV is subtracted from the load (with losses) to calculate the neededBattOutputKw variable. The biomass contribution is not subject to inverter loss, as mechanical generators output AC power.

The system can be in two states during one hour iteration only if the system changes to the waiting state ‘Starting Up’. This is necessary as waiting time can be less than one hour, and require the system to respond before iterating to the next hour. When the timer is less than 1 hour in a waiting state, the remaining time is calculated. The generator output and biomass consumption is scaled to fit the amount of time of running, found in the variable runBiomassGeneratorHours. The neededBattOutputKw is also scaled accordingly, when comparing to see whether the output is insufficient.

#### Runtime

The code displayed in section 2.4.6.4 Biomass System Source-Code, is executed every hourly iteration of the simulations. There are no further loops, computational complexity is therefore not increased and runtimes will stay within a predictable scale. The overhead of the state machine is a constant that increase the run-time about 10 times. This is justified as the user is expected to narrow the number of simulations, before speculating in biomass implementations. The extra run-time is strictly self-time as discussed in section 2.3.4.2 Computation Speed, meaning that there are no external functions slowing down runtime.

#### Additional Outputs

Biomass system output kW for each hour is added to the SimulationOutputs class. The BiomassOccurrences struct is also passed to the output class, this struct counts the occurrences in the state machine in terms of hours and events, to enable performance monitoring.

|  |
| --- |
| \\webedit.ntnu.no\gardhi\dstReferenceManual\StateMachineFlowchart.png  Figure 2:9 The Biomass System Implementation Flow-Chart |

#### Biomass System Test Results

Goes here

#### Biomass System Source-Code

% biomass system state machine initialization

runBiomassGeneratorHours = 0;

biomassSystemState = 'IDLE';

previousState = 'RUNNING';

% remove rest of biomass from previous simulation

availableBiomassKw = BiomParam.biomassDeliveredKw;

[…]

% delivering biomass

if mod(t, BiomParam.biomassDeliveryIntervalDays\*24) == 0

availableBiomassKw = availableBiomassKw ...

+ BiomParam.biomassDeliveredKw;

end

% A new day starts and the system is preemptive

if mod(t,24) == 0 && BiomParam.isPreemptive

% the predicted weather is cloudy

if strcmp(weatherPredictions{t/24 + 1}, 'cloudy')

isSunny = false;

if strcmp(biomassSystemState, 'WAITING TO RETRY')

previousState = 'RUNNING PREEMPTIVELY';

else

biomassSystemState = 'RUNNING PREEMPTIVELY';

end

% the predicted weather is sunny

else

isSunny = true;

end

end

% STATE MACHINE -----------------------------------------------

% RUNNING PREEMPTIVELY-----------------------------------------

if strcmp(biomassSystemState, 'RUNNING PREEMPTIVELY')

% the system should stop running preemptively because of

% sunny weather or full battery

if isSunny||(stateOfCharge(t,iPv,jBatt) == 1)

biomassSystemState = 'IDLE';

% we stay in preemptive mode

else

runBiomassGeneratorHours = 1;

end

% RUNNING------------------------------------------------------

elseif strcmp(biomassSystemState, 'RUNNING')

% PV can handle the supply alone

if neededBattOutputKw(t,iPv, jBatt) < 0

biomassSystemState = 'IDLE';

% we stay in running mode

else

runBiomassGeneratorHours = 1;

end

% IDLE---------------------------------------------------------

elseif strcmp(biomassSystemState, 'IDLE')

% The battery is at minimum and power is insufficient

if (stateOfCharge(t, iPv, jBatt) == BattParam.minStateOfCharge)...

&& (neededBattOutputKw(t,iPv, jBatt) > 0);

biomassSystemState = 'STARTING UP';

startupDelayTimer = t;

end

% INSUFFICIENT BIOMASS-----------------------------------------

elseif strcmp(biomassSystemState, 'INSUFFICIENT BIOMASS')

% Count hours spent waiting for biomass

BiomOccurrences.waitingForBiomassTime(iPv,jBatt) ...

= BiomOccurrences.waitingForBiomassTime(iPv,jBatt)...

+ 1;

% more biomass is available

if availableBiomassKw > BiomParam.generatorOutputKw

% prepare for normal operation

biomassSystemState = 'IDLE';

end

end

% STARTING UP--------------------------------------------------

if strcmp(biomassSystemState, 'STARTING UP')

% The time left

residualTime = BiomParam.startupDelayHours...

- (t - startupDelayTimer);

% The generator is not needed

if neededBattOutputKw(t,iPv, jBatt) < 0

biomassSystemState = 'IDLE';

% The timer is done

elseif residualTime <= 1

% run the generator

runBiomassGeneratorHours = residualTime;

biomassSystemState = 'RUNNING';

end

% WAITING TO RETRY---------------------------------------------

elseif strcmp(biomassSystemState, 'WAITING TO RETRY')

% The time left

residualTime = BiomParam.retryDelayHours ...

- (t - waitToRetryTimer);

% Account for time spent waiting to retry

BiomOccurrences.waitingToRetryTime ...

= BiomOccurrences.waitingToRetryTime + residualTime;

% The timer is done

if residualTime <= 1

% The timer finish this hour

if residualTime >= 0

% run the generator

runBiomassGeneratorHours = residualTime;

% The timer finished last hour

elseif residualTime < 0

runBiomassGeneratorHours = 1;

end

% return to previous state

biomassSystemState = previousState;

end

end

% The biomass generator should run for runBiomassGeneratorHours

if runBiomassGeneratorHours

generatorOutputKw = 1 - runBiomassGeneratorHours...

\* BiomParam.generatorOutputKw;

% There is sufficient biomass

if availableBiomassKw > generatorOutputKw

% The needed output is larger than generator kw size

if (neededBattOutputKw(t,iPv, jBatt)...

\* runBiomassGeneratorHours) > generatorOutputKw

biomassSystemState = 'WAITING TO RETRY';

waitToRetryTimer = t;

previousState = biomassSystemState;

BiomOccurrences.waitingToRetry(iPv,jBatt)...

= BiomOccurrences.waitingToRetry(iPv,jBatt) + 1;

% The generator will still run the given time

% before the insufficient output is discovered.

end

neededBattOutputKw(t,iPv, jBatt) ...

= neededBattOutputKw(t,iPv, jBatt)...

- generatorOutputKw;

biomassGeneratorOutputKw(t,iPv,jBatt) ...

= generatorOutputKw;

availableBiomassKw = availableBiomassKw...

- generatorOutputKw;

else

biomassSystemState = 'INSUFFICIENT BIOMASS';

end

% will only run what the state machine outputs.

runBiomassGeneratorHours = 0;

end

### economic\_analysis\_biomass.m

There were some additions to the original economic analysis to include the biomass system. When calculating a system with biomass, the economic\_analysis\_biomass.m module can be called. The additions to the outputs in EconomicAnalysisOutputs are the variables bioSysNetPresentCost, biomassPresentCost and bioSysLevelizedCostOfEnergy. These are the NPC of the entire biomass system installation, the money spent on biomass in present cost and the present cost of each kW produced by the system. It is assumed that the lifetime of a biomass system is approximately the same as the PV-plant and that there will be no significant salvage when the period has ended.

The LCoE is calculated by splitting the NPC into equal annual payments, including interests during the down-payment period. This annual payment is divided by the kWh successfully delivered to the load during simulation, which gives us the present cost of each kW.

The additions to the economic analysis can be seen in Figure 2:10.



Figure 2:10 Additions to the Economic Analysis module

## 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 2:11 and Figure 2:12 an example of a uicontrol initiation is displayed, and its call-function.



Figure 2:11 The uicontrol initiation for the 'Run Optimal Solutions' button



Figure 2:12 The call-function when pressing the 'Run Optimal Solutions' button

### dst\_platform.m

The dst\_platform GUI is meant as the start platform for the user. 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 2:13: 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 2.5.3.1 Simulations 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 2.5.2 solution\_explorer.m.

#### Running-Modes Available to the DST platform

Under the ‘Run Calculations’ header, there is a checkbox for ‘Generation Strategy – Biomass’. This run-mode will be described in section 2.4 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.

A detailed explanation of all the functions and modules, are available in the online documentation at <http://folk.ntnu.no/gardhi/dstReferenceManual/referenceManual.html>. The functions are not explained here as they are only utilized by the DST platform as external modules.

### solution\_explorer.m

The solution explorer is implemented to assist the user in understanding the results from the DST. The solution explorer allows the user to choose different solutions and compare them to other solutions. There are a number of outputs that are seen as essential to understand the implications of a choice. These belong to ‘General Info’, ‘Averages’, ‘Worst Case’ and ‘Biomass’.

The Solution Explorer is a way of analyzing the simulations chosen from the optimal search modules. These are stored in the Optimal Solutions class. The tool lists all the solutions stored in the variable ‘OptSol’ in the workspace, as seen beneath the text ‘Pick a Solution to Examine’ in Figure 2:15. Here are some key values to characterize the overall functionality of the simulation, these are NPC, LCoE and LLP. Once a solution is highlighted, the different buttons will produce outputs for examining the current solution.

Each solution has indexes in the different output matrices. These are normally referred to as iPv and jBatt in the code context, in the output windows this translates to ‘PV iteration’ and ‘Battery iteration’.

The radio toggle buttons under the Data Analysis Output windows decide which window the output should appear on. If the ‘Output In New Window’ option is chosen, outputs will appear in a new floating windows. This choice will allow for more than two windows for comparison. This is displayed in figure Figure 2:14 A floating Worst-Case Ouput Window. Plots will always appear in new floating windows (plot default behavior).

|  |
| --- |
| Figure 2:14 A floating Worst-Case Ouput Window |

|  |
| --- |
| \\webedit.ntnu.no\gardhi\dstReferenceManual\solution_explorer.png  Figure 2:15 The Solution Explorer |

#### Available Outputs to the Solution Explorer

The different buttons will output different kinds of information. A LoL value alone is not enough to understand how a system operates. For example, these outputs gives the user an idea of whether the LoL stems from some incidences of lost load, or from a regular daily small LoL. The latter, frequent lost load occurrences, is a worse alternative. A few occurrences of high magnitude of lost load is less critical.

‘General Info’ serves as an overview of recognizable characteristics for the user. The output includes which simulation the solution is pointing to in terms of kW/kWh and indexes, several totals and counters. The counters are for battery replacements during plant lifetime, which are very expensive and also connected to some risk, and occurrences of lost load.

‘Averages’ are for understanding of the general performance of the system. As mentioned earlier we want to know if the system is regularly offline or if there has been some large occurrences, this is visible in terms of time when comparing ‘Average Downtime Length’ and ‘Daily Average Downtime’, and in terms of kW when comparing ‘Average Lost Load During Downtime’ to ‘Daily Average Lost Load’. The outputs from ‘Worst-Case’ are also relevant here.

‘Worst-Case’ will tell you about the different worst-case scenarios for our system. Here we have specific intervals within the time series, and so the time when worst cases occur is also displayed. This way the user knows whether the worst cases are connected. The worst cases also gives an understanding of potential difficulties of running the system. By comparing worst case to averages, the user will understand better what kind of conditions that might occur, and how probable they are. If the worst case is close to the average, the user might have to consider expanding margins in case these conditions might occur more frequently.

‘Biomass-System’ delivers an overview picture of how the biomass system worked and the costs of running and implementing it. The LCoE of the biomass system will enable a comparison of the final cost per kWh of the system. The running times and transitions communicates how well the system is scaled, and how it works compared to its purpose. For example, much time spent in ‘Waiting To Retry’ indicates that the generators kW output is too small to handle the LoL magnitudes.

The unused generator output indicates the extent of failed attempts of running the generator. These attempts fail when the output kW is too small to cover the load, and the generator is shut down until the waitToRetryDelay has finished.

### 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.

#### Simulations Overview

It is vital to understand the simulation space when trying new parameters to the system, or when searching for a feasible solution. The simulations Overview plot is implemented in order to see the results in a larger context, to understand how solutions appear in the simulation space.

The plot is immediately useful when limiting the simulation space and when changing solution resolution, as the first step of using the tool and changing the SimulationsParameters input.

The plots have the same layout as the matrices of which they represent. The matrices plotted in Figure 2:16 are LLP, LCoE, number of batteries employed and NPC. The units in these are assumed understood by the user, the units of currency depend on what currency the user choose as parameter standard.

The tool plotter (the beige square in the upper left figure) can pick single solutions to retrieve iPv, jBatt, the indices of the solution, and the corresponding z-value. The left hand bar display the color legend in the figures. The 3D pan can be used if desired to see depths in the plots, these are by default displayed from above to avoid ambiguities.

The simulation overview will allow the user to spot trends in the solution space. This is important when navigating design choices. There is however a risk that the decisions are based on uncertain results. There is a varying degree of certainty associated with a simulation space, depending on the precision of the parameters. A decisions based on a higher precision than what is achieved, carry the risk of being a poor choice. To make finely tuned decisions the user will have to know the uncertainties of the parameters and how they are used together.

|  |
| --- |
| Figure 2:16 The simulation Overview Plot of SAPV simulations |



Figure 2:17 The Simulation Overview Plot module source-code

#### State of Charge

The SoC plot is inherited from logplot.m, it is now a single module with some minor layout changes. It will access any simulation given by its indices iPv and jBatt and plot the full time series of the simulation. This is a very useful plot as it makes a comprehensive interpretation of the lossOfLoad, batteryOutputKw and pvAbsourbedUnusedKw vectors, and their relationship. All kW values are normalized with respect to the battery capacity.

The green plot line is power from the PV that neither serves load nor charges batteries because of fully charged batteries, this has the offset of 1. The red plot line indicate loss of load in kW, the values are negated and with offset value equal to minimum SoC. This is nonzero when there insufficient output to serve the load from neither PV nor battery, or if discharge demands are too large.

|  |
| --- |
| Figure 2:18: A Full Year of the State Of Charge Plot |

|  |
| --- |
| Figure 2:19: 10 Days Range of the State of Charge Full-Year Plot |

When seeing the full year span of the SoC plot, you can see the frequency and depth of the loss of load in the red lines. This gives a good understanding of how the system performs, it would however be useful to see how this affects the average day.

When plotting the average day of SoC it lose most of the original expressive power. Because the plot lines are normalized and rely on the continuity of the curve to communicate the battery dynamics, when the hourly average of 24 hours is plotted, the continuity disappear and the magnitudes start making less sense, this is seen in Figure 2:20 An Average-Day State Of Charge Plot.

|  |
| --- |
| Figure 2:20 An Average-Day State Of Charge Plot |

#### Power Balance

The initial idea of the power balance plot was to enable comparison between irradiation input and load profile. An average day plot to indicate how each hour of a day normally operates is desired, and require a plot that is not normalized. This will help to understand when the system normally fails, and how much the system deviates from its desired operation.

Logplot.m contains an implementation of this, but the plot had several errors. The legend does not describe the plotted values correctly. The variable names in logplot.m and a lack of testing, complicated by the lack of naming convention and modularity respectfully, are probably to blame. These things are improved in the rewritten DST.

The yellow plot line seen in Figure 2:21 is the value from the variable named P\_pv in logplot.m. The legend says ‘Energy from PV’, but in reality this is the absorbed power, not the utilized power. Consequently, some of this power never serves the load, nor charges the batteries.

The brown plot line is the values from the variable bat\_balance\_pos in logplot.m. The legend says ‘Energy flow from battery’, but in reality, this is the needed battery output, demanded by the load. The plot will therefore express a perfectly functioning system every time, the ‘needed power’ is described wrongly as ‘supplied power’.

The difference between the blue load and the brown plot line in the part where only battery supplies the load, is only from efficiency losses which proportionally increase the needed battery input.



Figure 2:21: The power balance in logplot.m.

|  |
| --- |
| Figure 2:22: The average-day version of the Power Balance plot from logplot.m |

The code for plotting the power balance in logplot.m is displayed in Figure 2:24 and Figure 2:23. The latter is the same snippet of code with the naming convention from the rewritten DST. It is now easy to see that the legend and names have conflicting meanings. It is likely that these errors would not have been made with the new naming convention.

|  |
| --- |
|  |

Figure 2:23: The code for plotting of the Power Balance in logplot.m

|  |
| --- |
|  |

Figure 2:24: The same code as in

Figure 2:23, with the new naming convention

The Power Balance plot is rewritten as a module with the intended functionality from logplot.m. Like the other plots, the module is implemented as a function. The only input that has to change when calling the plotting functions is the indexes of the desired simulation in the simulation space, iPv and jBatt. This makes it easy to call the function from the current workspace, and to plot simulations without much effort despite lacking a UI, like the dst\_platform.

There is one function for plotting the average day, and one for plotting the full year of power balance. These are identical except for the part of calculating daily averages. The choice of not locating this functionality within one function, and not determining the average-day mode with a passed input is to reduce the knowledge needed to utilize the function. This is a key principle of modularity.

The utility of the new Power Balance plot is to compare the different power outputs in scale with each other. When the system supplies the load perfectly, the system is in an equilibrium that is expressed graphically. This is done by plotting all the power to the load after losses. These values are labeled ‘net’ in the plot code to express that every kW goes to the load.

The batteryNetOutput is the battery discharge that reach the load without losses. The values of battOutputKw are positive when the battery discharges. The discharged variable is a vector with the positive values of battOutputKw that is returned by the subplus function. This vector is scaled down with discharging efficiency and the inverter efficiency, this gives us the battNetLoadSupply vector.

The pvNetLoadSupply is the power that is directly supplied to the load from the PV. These values are not output as a vector from the simulations module explicitly. Too many output vectors will take time to compute and will increase the memory-use of the program, this is not justified by a single plotting functionality.

The pvNetLoadSupply is found in two steps. Either the neededBatteryOutput is negative, meaning that the battery is charging. In this case the power from the PV is greater than the load, hence every value in the load variable can account for supplied PV power. In the second case neededBattOutputKw is positive. This means that all the pvPowerAbsorbed goes to serve the load directly and is therefore equal to pvNetLoadSupply.

irradiationUtilized is the pvPowerAbsorbed – pvPowerAbsorbedUnused. The latter accounts for power wasted when the batteries are fully charged, or when the batteries maximum charging rate is exceeded. The utilized irradiation goes to serve the load without any losses, or to the battery with losses. The latter makes the irradiation utilized slightly off-scale from the load, meaning that the summed integral of irradiationUtilized subtracted with the summed integral of netLoadSupply is nonzero. This slight offset does however not disrupt the comparison of the three curves.

The code for the plot\_average\_power\_balance function is displayed in Figure 2:25. The calculation of averages is supported by the generic help-function get\_average\_day. The help function is used different places in the DST and helps reducing development effort and increase readability.

get\_average\_day finds the average value of all 24 hour segments a vector. The input vector must have hour resolution. The function is displayed in Figure 2:26.



Figure 2:25 The plot\_average\_power\_balance function.



Figure 2:26 The get\_average\_day help-function

Instead of plotting pvNetLoadSupply and battNetLoadSupply separately, they are summed together. This will result in a perfect tracking of the load curve when the system functions perfectly, and a deviation from the load curve during loss of load. When this occurs, the blue netLoadSupply plot-line will sink and reveal the red load line. This will make it clear when the power fails, this can be seen in Figure 2:27.

The origin of power can also be followed in the plots. 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. Hence the integral of the difference between netLoadSupply and irradiationUtilized is the battery output, without the loss while charging/discharging and in the inverter. In the case of 90% efficiency in each transition, the integral difference is 72.9% of the actual battery output.

The integral of irradiationUtilized and netLoadSupply will not sum to zero, but one can still see the proportions of energy input and load demand. It will help the user understand the magnitude of the lost load in terms of irradiation and battery dynamics.

The new colors aim to appeal to intuition. 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 necessarily reading the legend.

The resulting plots can be seen in Figure 2:27 and Figure 2:28. The functions are easily maintained and modified because of modularity.

|  |  |
| --- | --- |
| Figure 2:27: The new Power Balance plot, 4 days of the full-year plot. | |
| Figure 2:28: The average-day Power Balance plot. |

## Documentation

To maintain a continuous documentation that is platform-independent, easily maintainable and easily accessible. The new DST has online documentation written in HTML. The code is in rudimentary HTML which can be modified and understood by anyone. HTML can additionally be written by practically any device. This enables continuous development of the documentation as the tool grows and is modified, it will also allow documentation to be read from a phone or any platform with an internet browser installed.

The idea is to have a minimalistic documentation that is easily navigated by a browsers search-functionality. If the user finds a function, parameter or variable that needs explanation, the name of the component can be input with ctrl+f and the explanation will appear without having to click any hyperlinks as everything is in one page. Additionally there is a table of contents and a link to the top on the page, this can be handy if one is investigating several objects. See Figure 2:29.

The documentation is organized in 6 main parts. ‘Notes to the User’ is reserved for the most vital information needed to work with the DST. ‘Modules’ explain the main modules of the DST, their inputs, outputs and functionality. ‘Classes and Properties’ explain the different classes that is used to pass variables between functions. These contain explanations for every large-scope variable in the DST. ‘Plots’ will explain how to use and interpret the plot functions, and lastly the ‘Help Functions’ is a summary of the globally accessible functions, used different places in the DST.

The documentation is several pages long and can be found at <http://folk.ntnu.no/gardhi/dstReferenceManual/referenceManual.html> or <http://gardhi.github.io/dstReference/referenceManual.html>. When someone wants to inherit this code it is sufficient to download the script from the webpage. The Git-hub repository will also be open for collaborators.

|  |
| --- |
| Figure 2:29 The DST Online Reference Manual |

# 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

### Biomass Power Generation

Usefulness

Inconsistency

### Applicability of DST

Simulation space battery replacement steps

## Recommendation for Further Work

### The Rainflow Counting Algorithm

It is highly important to consider the calculations of the battery replacement interval. Replacing batteries are costly and amounts to a majority of the NPC when dealing with SAPV systems. Additional importance can be considered since the DST is intended for micro and small scale enterprises (MSSE). MSSEs are likely to rely on smaller economical margins, and larger personal risks.

Batteries are replaced when they have gone through a certain amount of charging / discharging cycles, the method used in the DST is the Rainflow Counting Algorithm. The Rainflow Counting algorithm was initially to account for stress exposure in materials. One cycle is one instance of full stress exposure, a partial cycle is an instance of a partial stress exposure. The algorithm can be used to account for stress in batteries (You and Rasmussen 2011) and other appliances that go through similar wear.

#### 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. Calculate cycles to failure for each occurring DoD:

This is the result of fitting a typical lead-acid battery cycles to failure vs. DoD characteristics. This equation has to be replaced or modified by the user.

1. Accumulate cycles to failure during simulation time.
2. Find replacement interval of batteries:

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. The expression from point 2 has to be replaced or modified if the user desire precision in the economic analysis.

#### Considerations

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. A more generic calculation is implemented in the rewritten DST as seen below.

|  |
| --- |
| 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. Improvement of omputation-time when employing a precalculated array is too small, compared to the reduced precision of the implementation.

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. The physical consequences of potential “flutter” between cycles is neglectable. Additionally, charging cycles have daily periods, as seen in Figure 2:18 and Figure 2:19. The charging cycle follows the irradiation cycle and will not drop below 24 hours.

#### Suggestion for Future Versions

The equation used in the DST to calculate cycles to failure of a DoD is a fitting a typical Cycles to failure Vs DoD of lead-acid batteries. The equation is defined as: on page 61.

Considering the technologic advances, this characteristic certainly changes over time, and also somewhat between battery manufacturers. I is necessary for the user to obtain this characteristic when using the DST. This can become a great obstacle for some. I propose the following alternatives for future versions:

1. Data-base integration. Whenever a characteristic equation is found, the tool uploads this to an online repository. This does not happen automatically but when users have reliable results to share. The upload must contain date, battery type and comments. If there exist a database like this, implement a configuration to integrate the data for the DST.
2. Cycles to failure vs DoD characteristic estimation tool. A program that takes some parameters to calculate an estimate of the equation. The tool should be time dependent or easily adjustable to improved technology. To some extent, the program should be easy to use.

### Finding Borders Analytically

The DST relies initially on producing a very large solution space. The user will usually have to create the simulation space more than once, to pinpoint the desired resolution for analysis. A user might not familiar with the economic proportions of the PV and battery sizes necessary for the load in question. If this is the case, the user has to start with a wide solution space and then iterate to the desired scope and resolution for analysis.

The outputs used from the simulation module for the economic analysis is the yearsBattOperational value, stating how long a battery is functional, and the lossOfLoadTot for the time series. The years a battery is operational depends on the wear and tear of the system. This value does have a worst case outcome, if every cycle is a full discharge to the minStateOfCharge level. The following analytical expression will calculate this worst case life span.

YO stands for years operational, the years a battery is operational before being replaced. Using this in the economic analysis we achieve a degree of freedom to find sizes for PV and battery expressed by B kWh and PV kW in the following equation:

OeMeR is the net present cost of operation, maintenance and replacement of batteries. B cost/kWh is the cost of batteries per kWh. I cost is the inverter cost. PL is the plant lifetime, YR is the years remaining of battery life after plant lifetime has expired, and OeM cost/kW is the yearly operation and maintenance cost per kW of PV capacity.

If we assume that IC and OeMeR are known, the equations have 2 unknown variables and the system has a solution. A problem is that the ratio between OeMeR and IC will affect the solution in different ways. This means that the user must know what ratio between the two is to be expected. This is problematic. If we have that in systems in general, we can exploit that:

This will enable the user to know only the NPC to get a minimum performance size for this budget. A histogram from the test data shows that the ratio is varying more than 10% as seen in Figure 3:2. This might be too much to make any useful estimates.

|  |
| --- |
|  |

Figure 3:2 Distribution of IC/OeMeR ratios

#### Suggestion for Future Work

A possible connection between the IC/OeMeR ratios and LLP is displayed in Figure 3:3. A general analytical expression may be found from several test-data sets empirically. The results would be useful if precision is consistently higher than 10% deviations. Using these results, a component minimum size can be returned to the user without any exploration of the simulation space.

|  |
| --- |
|  |

Figure 3:3 Dependency between LLP and IC/OeMeR ratio

# Appendences

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