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

# Introduction

A general introduction goes here.

## Background

* Problem Formulation
* Literature Survey
* 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

## Limitations

## Approach

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

## Structure of the Report

# Methodology

Part 1, present; part 2 explain how. Results

## Code Review and Improvement

The DST will be an Open Source collaboration. This implies some demands on the code quality of the project.

### Naming Conventions, Commenting and Formatting

Table 1: the Complete List of Variable Name Changes in Rewritten DST

|  |  |
| --- | --- |
| Names | |
| Before Rewrite | 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.plantLifetime |
| 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.temperature |
| T\_cell | pvTemperature |
| 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 | |

### Modularization and Encapsulation

Describe the concept

Describe the implementation

Example the SoC and power balance average implementation

The rewritten DST has key variables stored for later calculations and plots, where the previous DST would only keep these in memory during simulation. Storing the simulation history will enable the user to only run the simulation once, and use these results until the simulation parameters need to change. I.e. when the current solution become unsatisfactory.

### Testing

The new

Years to go batt example, if tests were ready one would be able to spot a jump in prices.

### Plotting

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.

#### 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 2:1: A full year overview from the State Of Charge plot |

|  |
| --- |
| Figure 2:2: 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 2:3, with the old naming convention in Figure 2:4. The legend that describe the yellow plot line (seen in Figure 2:5) 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 2:3: The previous attempt at making a power balance plot in the outdated DST. This code has the new naming conventions.

|  |
| --- |
|  |

Figure 2:4: The same code as in Figure 2:3 with the old naming conventions.

|  |
| --- |
| Figure 2:5: 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 2:6: The same error that is explained in Figure 2:5 is replicated in the daily average version. |

|  |
| --- |
| Figure 2:7: 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 2:9. 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 2:8. 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 2:8: 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 2:9: 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 2:10: 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 2:9). |

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 2:11 and Figure 2:12. The functions are easily maintained and modified due to the modularization and strict naming conventions.

|  |  |
| --- | --- |
| Figure 2:11: 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 2:12: 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. |

### 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 2:13: 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 2:13, 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 2:1 and Figure 2:2 the cycles have consistent cycles of 12 hours. i.e the flutter is unlikely to occur in the first place.

Later contributors are of course enqouraged to inspect anything that seems reasonable to them.

### 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 was 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 should be enough data points to state that the new implementation conserve the functionality from the old, with a high degree of certainty. We assume that the probabilities are independent, because we assume that the case where the probabilities are dependent is the correct implementation. The probability of two matrices being identical coincidentally can be described as follows.

We make an assumption that the average is no more than 50%. We wish to be at least 99.999% sure that our implementation is correct. We can see that, and consequently the 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. When we evaluate the modules with lesser complexity, the data points decrease, but the chance of a data point being coincidentally wrong is presumed lower, since they are derived from the entirety of the previous data points.

Table 2: 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, and the names have the benefit of a prefix to explain them (ex. SimOutputs.stateOfCharge). The following parameters were set in both the rewritten and original DST.

Given the validity of the coarse calculations above, it can be assumed more than 99.99% certainty for different PV and battery size combinations. In order to thoroughly amend for possible misassumptions, two simulation pairs were executed. The first simulation had PV and battery combinations, and 15 optimal solutions. The second simulation pair output PV and battery combinations.

% 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 2:14: Parameters for simulations that produce comparison outputs for simulation pair 1.

Table 3: 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 2:15: Parameters for simulations that produce comparison outputs for simulation pair 2.

Table 4: 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 |

These results prove that the functionality of the DST has been preserved completely in the rewrite process.

#### 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. Such 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 that was used to evaluate the rewrite. 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 tell you about a function’s complexity. 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 gain was less than 0.1% so the function was kept as an external module, this choice maintains the program architecture.

The profiler tool output is displayed in Table 5 and Table 6. A large bright blue band will indicate that there might be possible improvements in reducing function calls. If Matlab make many implicit calls to help-functions in order to initialize or use classes, it will be displayed here.

Table 5: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 6: 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 6 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 storing and allocation overheads.

The copy overhead can be reduced by implementing handle classes. 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 C/C++, 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. The pointer functionality can be implemented when the DST will be ‘shipped’, since this will be in a different language.

Early in the rewrite, there were get functions in the classes that were called over every iteration. The get functions were a part of a dependent property (class member variable) implementation that supports update of properties that is calculated from the other class properties. The get function update the dependent properties, but they would make calculations for every access to the class variable. The class implementations were changed so that calculating these variables only happens at initiation. In the DST there is no need to update the classes once initiated. We use the results stored in classes, but we do not modify them outside their modules. The run time improved significantly without get functions. The conclusion is: get functions should not be used to return variables that are frequently accessed. (Shure 2012)

Table 7: The run times of the rewritten DST with and without get-functions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | Computation Times [s] | | |
|  |  |  | Before Rewrite | After Rewrite | |
| nPv | nBatt | nHours |  | get functions | w/o get functions |
| 2 | 2 | 8760 | 0.29 | 0.66 | 0.22 |
| 8 | 8 | 8760 | 1.75 | 4.99 | 2.95 |
| 20 | 20 | 8760 | 9.60 | 29.26 | 17.79 |
| 100 | 100 | 8760 | 238.54 | 730.94 | 414.83 |

The overhead is acceptable because of two reasons. First the user will get this time back when using the rewritten DST for other tasks than simulation. Tasks beside simulation will make for the majority time spent designing a microgrid, this includes comprehending the tool’s inner workings. Giving users this insight is also a specification in (Mandelli, et al. 2014). Second, the tool will be rewritten to a lower level language before it is ‘shipped’ to open source. The overhead that Matlab introduce when calling functions and using classes typically less in lower level languages such as C++ or Java.

#### Bug Fixes

The years\_to\_batt bug

The plot\_power\_balance bug

## Data Analysis

## Refining Solution Space

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.

## Biomass Power Generation

# 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