

---

# SolarTherm Documentation

*Release 0.1*

**SolarTherm**

**Jan 11, 2022**



---

## Contents

---

<b>1</b>	<b>User Guide</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Installation . . . . .	2
1.3	Usage . . . . .	6
1.4	Tutorial . . . . .	11
<b>2</b>	<b>Development</b>	<b>23</b>
2.1	Coding Style . . . . .	23
2.2	OpenModelica . . . . .	24
2.3	Libraries . . . . .	27
2.4	Modelling . . . . .	30
2.5	Models . . . . .	30



### 1.1 Introduction

SolarTherm is a Modelica-based simulator for concentrating solar thermal (CST) systems. It aims to provide an easy way to simulate and optimise CST systems, including but not limited to those systems that produce electricity, solar fuels, or heat for industrial processes. The project is focussed on the research community, who want to be able to quickly assess novel combinations of different receivers, storage, fluids, control strategies, power cycles, or/and heat integration. The simulation is at a level of detail required to accurately assess whole system performance and economic viability. More details were presented by [Scott et al \(2017\)](#).

SolarTherm contains two main parts: (1) a Modelica model library for common CST components and system configurations, and (2) a collection of tools for organising parameters, running simulations, training surrogate models, and analysing results. Details can be seen in this documentation.

A list of publication that employed SolarTherm includes:

- [Turchi et al, 2021](#). CSP Gen3: Liquid-Phase Pathway to SunShot, NREL Technical Report NREL/TP-5700-79323
- [Gunawan et al, 2021](#). System modelling and optimisation of a particle-based CSP system, Technical Report, Solar Thermal Group, Australian National University
- [Rahbari et al, 2021](#). Methanol fuel production from solar-assisted supercritical water gasification of algae: A techno-economic annual optimisation, Sustainable Energy & Fuels
- [Fontalvo et al, 2020](#). System-level simulation of molten salt small-scale CSP, AIP Conference Proceedings
- [Rahbari et al, 2020](#). The impact of low-cost H<sub>2</sub> on the solar fuel process design: A case study in solar gasified Fischer–Tropsch fuels, AIP Conference Proceedings
- [Kee et al, 2020](#). System level analysis of a sodium boiler receiver and PCM storage CSP plant using SolarTherm, AIP Conference Proceedings
- [Calle et al, 2020](#). Techno-economic assessment of a high-efficiency, low-cost solar-thermal power system with sodium receiver, phase-change material storage, and supercritical CO<sub>2</sub> recompression Brayton cycle, Solar Energy

- [Guccione, 2020](#). Design and Optimization of a Sodium-Molten Salt Heat Exchanger for Concentrating Solar Power applications, master thesis, KTH, School of Industrial Engineering and Management (ITM).
- [Shirazi et al, 2019a](#). System-level simulation of a solar-driven liquid fuel production plant via gasification-Fischer-Tropsch route, AIP Conference Proceedings
- [Shirazi et al, 2019b](#). A solar fuel plant via supercritical water gasification integrated with Fischer-Tropsch synthesis: System-level dynamic simulation and optimisation, Energy Conversion and Management
- [Rahbari et al, 2019](#). A solar fuel plant via supercritical water gasification integrated with Fischer-Tropsch synthesis: Steady-state modelling and techno-economic assessment, Energy Conversion and Management
- [Calle et al, 2018a](#). System-level simulation of a novel solar power tower plant based on a sodium receiver, PCM storage and sCO<sub>2</sub> power block, AIP Conference Proceedings
- [Rahbari et al, 2018](#). Dynamic Performance of Fischer-Tropsch Liquid Fuel Production from Solar-Assisted Supercritical Water Gasification of Algae, AIChE Annual Meeting
- [Calle et al, 2018b](#). SolarTherm: A New Modelica Library and Simulation Platform for Concentrating Solar Thermal Power Systems, Simul. Notes Eur.
- [Kee et al, 2018](#). A dynamic model of a sodium/salt PCM energy storage system, MATHMOD.
- [Scott et al, 2017](#). A flexible Modelica-based simulator for CSP systems, AIP Conference Proceedings

## 1.2 Installation

This section goes through the installation process for SolarTherm. The required dependencies and software packages are reviewed at the beginning, followed by more detailed installation instructions for two Linux distributions and Windows. The scripting and tools in SolarTherm have currently been tested on Ubuntu (20.04 and 18.04) and Windows-MSYS2. Although they should also run on Mac, we have not tested it yet.

### 1.2.1 Overview

The SolarTherm package contains a series of CST related Modelica models to perform dynamic performance analysis. It is also integrated with several software packages to expand its simulation capability, for example, Solstice for Monte-Carlo ray tracing simulations, GLPK for linear programming optimisation, Tensorflow, SAM SSC (SAM simulation core) for surrogate modelling, etc. The Modelica models can be run by either OMEdit or a terminal command. [OMEdit](#) is a graphical user interface that enables the users to easily create models, edit connections, run simulations and plot results.

The SolarTherm package also contains a series of additional functions that perform financial calculations (e.g. LCOE, LCOF), parametric sweep, system optimisation and sensitivity analysis. It also integrates with excel spreadsheets and Dakota software package for expanding data sampling and optimisation capabilities. These functions are handled externally from Modelica, and must be run from a terminal command. It is recommended that using OMEdit for Modelica model development and inspection, and using terminal commands to perform techno-economic analysis, system optimisation and sensitivity analysis.

The required dependencies and software packages are listed below:

#### OpenModelica

A working version of OpenModelica is required. Instructions for installing OpenModelica (including OMEdit) on each platform:

- [Windows](#)

- [Mac](#)
- [Linux](#)
- [Source](#)

We found that the OpenModelica version v1.14.2 is stable and works better for all of our SolarTherm models, especially for the particle system model that involves several surrogate models (e.g. ANN, Kriging, SAM SSC).

An instruction of the OpenModelica installation is provided below in this documentation.

## Dependencies

Prerequisite Python packages:

- [scons](#)
- [distro](#)
- [wheel](#)
- [numpy](#)
- [scipy](#)
- [DyMat](#)
- [solsticepy](#) (Python wrapper for Solstice ray-tracing simulations)
- [openpyxl](#) (for loading/exporting parameter from/to an excel spreadsheet)
- [colorama](#) (for highlighting keywords in output)s
- [pytest](#) (for running tests)

Optional:

- [matplotlib](#) (for plotting figures)
- [pyswarm](#) (for optimisation using pyswarm method)
- [cma](#) (for optimisation using cma method)
- [tensorflow](#) (for training surrogate models)
- [pyDOE](#) (for training surrogate models)
- [bayesian-optimization](#) (for optimisation of ANN models)
- [pandas](#) (handling data for surrogate model training)
- [CoolProp](#) (for calculating fluid properties)

## Optional external software packages/libraries

SolarTherm integrates with a number of software packages to expand its applications:

- [Solstice](#) (for Monte-Carlo ray tracing simulations)
- [Dakota](#) (for optimisation and uncertainty quantification)
- [glpk](#) (for linear programming)
- [gsl](#) GNU Scientific Library (for Kriging surrogate models)
- [SSC](#) SAM simulation core (for surrogate models)

- TensorFlow C API (for deploying surrogate models inside SolarTherm)

### 1.2.2 Installation Instruction (Ubuntu 18.04 and 20.04)

The terminal commands for building the essential packages on Ubuntu (18.04 and 20.04) are listed below.

#### 1. OpenModelica

Installation for the latest version

```
$ for deb in deb deb-src; do echo "$deb http://build.openmodelica.org/apt `lsb_
↪release -cs` release"; done | sudo tee /etc/apt/sources.list.d/openmodelica.list
$ wget -q http://build.openmodelica.org/apt/openmodelica.asc -O- | sudo apt-key add -
$ sudo apt update
$ sudo apt install build-essential openmodelica omlib-modelica-3.2.3 libglpk-dev
```

Installation for v 1.14.2 (this version of omc works better with our models when we tested it, especially for the particle systems.)

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 3A59B53664970947
$ echo "deb https://build.openmodelica.org/omc/builds/linux/releases/1.14.2/ bionic_
↪release"| sudo tee /etc/apt/sources.list.d/openmodelica.list
$ wget -q http://build.openmodelica.org/apt/openmodelica.asc -O- | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install build-essential openmodelica omlib-modelica-3.2.2 libglpk-dev_
↪glpk-utils libgsl-dev
```

#### 2. SolarTherm dependencies

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade setuptools wheel
$ python3 -m pip install scons scipy matplotlib DyMat pyswarm cma pytest solsticepy_
↪openpyxl distro colorama
```

#### 3. Solstice

```
$ sudo apt install libpolyclipping-dev libtbb-dev libyaml-dev libgomp1
$ export UBER='lsb_release -cs'
$ export SOLSTICEURL="https://cloudstor.aarnet.edu.au/plus/s/Tao06XnrGRiwoiC/download?
↪path=%2F&files=solstice-0.9-x86_64-$UBER.tar.gz"
$ sudo tar xzv --strip-components=3 -C /usr/local < <(wget "$SOLSTICEURL" -q -O-)
$ export PATH=$PATH:/usr/local/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
$ solstice --version
```

#### 4. Dakota

Prerequisite for Ubuntu 20.04:



```
$ sudo apt install openmpi-bin libltdl7 liblapack3 libhwloc15 \
libsglcbblas0 libquadmath0 libboost-regex1.71.0 libgsl23 \
libevent-2.1-7 libgfortran5 libboost-filesystem1.71.0 libopenmpi3 \
libc6 libblas3 libstdc++6 libevent-pthreads-2.1-7 \
libboost-serialization1.71.0
$ OS=ubuntu-20.04
$ mpirun --version
```

Prerequisite for Ubuntu 18.04:

```
$ sudo apt install libc6 libboost-serialization1.65.1 libstdc++6 \
libboost-filesystem1.65.1 libgcc1 libquadmath0 liblapack3 \
libboost-regex1.65.1 libboost-system1.65.1 libblas3 libc6 \
libgfortran4 openmpi-bin libopenmpi-dev
$ OS=ubuntu-18.04
$ mpirun --version
```

Install Dakota:

```
$ DAKOTA_VERSION=6.14.0
$ export PKGN=dakota-${DAKOTA_VERSION}-${OS}-x86_64-jp
$ export DAKURL="https://cloudstor.aarnet.edu.au/plus/s/Tao06XnrGRiwoiC/download?path=
↪%2F&files=${PKGN}.tar.gz"
$ sudo tar xzv --strip-components=3 -C /usr/local < <(wget "$DAKURL" -q -O-)
$ export PATH=$PATH:/usr/local/bin # needed for Ubuntu 18.04
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib # needed for 18.04
$ dakota --version
$ export PYTHONPATH=$PYTHONPATH:/usr/local/share/dakota/Python
$ python3 -c "import dakota.interfacing;print(dakota.interfacing.__file__)"
```

## 5. Build and install SolarTherm

Clone the SolarTherm source code, change to the SolarTherm source directory and compile the package using `scons`:

```
$ git clone https://github.com/SolarTherm/SolarTherm.git SolarTherm
$ cd SolarTherm
$ scon
$ scon install
```

The default installation prefix is `~/ .local`. A user defined prefix can be given to the ‘PREFIX’ variable to change the installation prefix, for example:

```
$ scon PREFIX=/the/user/defined/directory
$ scon install PREFIX=/the/user/defined/directory
```

The default SolarTherm modelica library prefix is `~/ .openmodelica/libraries` on Linux and `~/ .local/lib/omlibrary` on Windows (MSYS2). If OpenModelica is installed at a different prefix, then the full path to the library directory should be given to the `INSTALL_OMLIBRARY` variable. For example:

```
$ scon PREFIX=/the/user/defined/directory INSTALL_OMLIBRARY=/the/directory/where/
↪Openmodelica/installed
$ scon install PREFIX=/the/user/defined/directory INSTALL_OMLIBRARY=/the/directory/
↪where/Openmodelica/installed
```

The last step is to set up the correct environment variables for the command line to find SolarTherm. A tool (`st`) has been created by the `scons` to automatically set the correct environment for the current terminal. By default, `st` is

located in `~/ .local/bin/` and can be called directly. The Solartherm environment can be activated by:

```
$ st env
```

The command `exit` deactivates the environment.

Once the environment is correctly set up, tests can be run from the tests directory with the command:

```
$ python -m pytest
```

### 1.2.3 Installation Instruction (Windows)

On Windows platforms, the SolarTherm terminal commands are run from MSYS2. The installation includes MSYS2 system and OMEdit.

The full instruction is available on SolarTherm Wiki [here](#).

### 1.2.4 Build omc from Source

This section will be added to show how to build openmodelica (omc) from source, e.g. for supercomputer applications.

## 1.3 Usage

The basic process for using SolarTherm is to construct a Modelica model of the system, drawing on components from the provided Modelica libraries, and then simulate it using one of the provided scripts. If required components are missing from the provided libraries, an external model could be integrated into the SolarTherm library.

Here we list the different components of the project along with a basic description. A full tutorial on the process is provided in a later section.

Note: SolarTherm is still being developed. The instructions provided here will change over time, providing the user with more capabilities.

### 1.3.1 Components

**SolarTherm** A Modelica library containing typical system components required to model a full solar thermal system. Models with varying degrees of complexity are included so that users can make a trade off between simulation speed and accuracy.

**solartherm** A Python library that provides common functions and classes for the other tools (e.g. `st_simulate`, `st_optimise` etc) to use.

**st\_simulate** A Python script for simulating a system model. It compiles the model, calls the simulation, and post processes the results. It can also be used for performing parameter sweeps or just changing individual parameters without having to recompile the model each time.

**st\_optimise** A Python script for model optimisation. See more details in the next section.

**st\_sensitivity** A Python script that takes uncertain parameters in cost and performance, conducting Latin hypercube sampling (LHS) for sensitivity analysis on the techno-economic performance of a designed system. See more details in the next section.

**st\_contingency** A Python scripts that predicts contingency budget based on a re-optimal system design when cost and performance parameters are changed due to uncertain factors. See more details in the next section.

**st\_plotmat** A Python script for easily plotting the results from a simulation. Behind the scenes it uses the common `matplotlib` library.

**st\_inspect** A Python script for converting simulation results into CSV data. Results are output to the console (standard output) but can also be directed into a file (e.g. `st_inspect myfile.res > output.csv`)

**st\_wea\_to\_mo** A Python script for converting TMY3 weather data into a format that is suitable for import into a simulation table.

**st\_get\_aemo\_prices** A Python script that downloads spot market prices from [AEMO](#) for a particular year and region, and then converts them into a simulation-table-ready format.

### 1.3.2 The optimisation tool

Our optimisation tool (i.e. the `st_optimise` script) integrates with a range of optimisation algorithm packages, which is capable of handling a wide range of optimisation problems in concentrating solar thermal applications. An example of how to use this command-line tool is shown below:

```
st optimise $mofn --method dakota_soga --objective lcoe --excel $excel_file --wd
↪$working_dir --np 8
```

- `$mofn` is the directory of the SolarTherm model
- **--method flag: this flag simply defines which optimisation method is going to be used. Options are:**
  - `dakota_soga` : single objective genetic algorithm (SOGA) from [DAKOTA](#)
  - `dakota_moga` : multi-objective genetic algorithm (MOGA) from [DAKOTA](#)
  - `pso` : particle swarm optimisation (PSO) provided by [pyswarm](#) package
  - `cma` : covariance matrix adaptation evolution strategy (CMA-ES) for non-linear numerical optimisation in Python [cma](#) package
  - `ga1` : genetic algorithm (GA) provided by [pyevolve](#) package
  - `ga2` : genetic algorithm (GA) provided by [deap](#) package
  - `nsga2` : non dominated sorting genetic algorithm (NSGA-II) provided by [deap](#) package
  - `COBYLA`, `SLSQP`, `TNC` or `L-BFGS-B` provided by [scipy](#) package

One can follow the following format to use this flag after the `st_optimise` command, for example:

```
--method dakota_soga
```

- **--objective flag** This flag is used to specify the objective function(s) that are going to be optimised. At the moment, these objectives can be the levelised cost of electricity/fuel (i.e. `lcoe` or `lcof`), the capacity factor (i.e. `capf`), the annual electricity/fuel produced per year (`epy` or `fpy`) and/or the market spot revenue (`srev`). It is evident that the nature of some of these functions is for minimisation, while the others must be maximised in order to make logical sense. The default for the optimisation algorithm is minimising. A '-' sign needs to be given if the objective function is to be maximised. See an example shown below:

```
--objective lcoe,-epy
```

Note that it is invalid to put the '-' sign directly after `--objective`. At the moment, maximisation is only available in multi-objective optimisation. This will be refined in future work.

- **--excel flag** This flag specifies the directory of an excel spreadsheet that contains simulation parameters. There is an example spreadsheet provided in the `examples` folder in the SolarTherm package, which can be used as a template to fit into the `st` simulations.
- **--par flag** Parameters can also be given by this flag, which takes the parameters with bounds and optional starting value in the form of `NAME=LOW, HIGH [, START]`
- **--wd flag** This flag specifies the working directory to save the simulation outputs
- **--np flag** This flag specifies the number of processors to run the simulations. For example `--np 8` for parallel multi-processing simulation with 8 processors.
- **--restart flag** This flag is given when you want to restart (continue) an optimisation at the point it was terminated (e.g. the optimisation has already been started but yet finished). This option is available only for the methods of `dakota_soga` and `dakota_moga`.
- **--max\_eval flag** This flag specifies the maximal number of evaluation when using `dakota_soga` or `dakota_moga` method
- **--pop\_size flag** This flag specifies the population size when using `dakota_soga` or `dakota_moga` method
- **--num\_generations flag** This flag specifies the number of generations when using `dakota_soga` or `dakota_moga` method
- **--seed flag** This flag specifies the seed of data sampling when using `dakota_soga` or `dakota_moga` method
- **--fuel flag** This flag is for those systems producing fuel rather than electricity. The `-- fuel` flag must be mentioned after the `st_optimise` command.
- **--maxiter flag** This flag defines the maximum number of iterations (not necessarily number of simulations). It should be noted that this flag is only for `pso`, `cma` and `SciPy` optimisation methods.
- **--dm flag** This flag is used to specify the decision-making methods for `nsga2`. The options are `linmap` or `topsis`. In LINMAP method, a solution on the Pareto front curve with the minimum spacial distance from an ideal point (the point at which each single objective has its optimum value regardless of satisfaction of other objectives) is selected as the best optimum design point. In TOPSIS method, in addition to the ideal point, a non-ideal point (the point at which each objective has its worst value) is defined. In fact, the basic principle of this method is that the selected final optimal point must have the shortest distance from the ideal point and the farthest distance from the non-ideal point. Below shows the format this flag is used:

```
--dm linmap
```

- **--outfile\_p and -- outfig flags** These flags are used to save the optimal solutions and Pareto front graph from the `nsga2` optimisation into files. The arguments given to these flags are the paths at which these files are going to be saved. For instance:

```
--outfile_p ../examples/result.txt
```

A full list of flags and instructions can be seen in the help manual by running the following command:

```
st optimise --help
```

### 1.3.3 The sensitivity and contingency analysis tool

Our sensitivity and contingency analysis tools (i.e. the `st_sensitivity` and `st_contingency` script) coordinate uncertainties in system performance and cost parameters, provide information on likelihood of the techno-economic performance of the simulated system model, and estimate contingency budget to mitigate risks.

### st\_sensitivity

The `st_sensitivity` tool focuses on the effect of uncertain parameters on a fixed design. An example of how to use this command-line tool is shown below:

```
st sensitivity $mofn --sample lhs --excel $excel_file --ns 1000 --wd $working_dir --
↳ np 8 --runsolstice
```

- `$mofn` is the directory of the SolarTherm model
- **--sample flag** This flag is followed by the data sampling method. Options are `lhs` for Latin Hypercube sampling, and `random` for Monte-Carlo random sampling
- **--excel flag** This flag is followed by the directory of an excel spreadsheet that contains simulation parameters. There is an example spreadsheet provided in the `examples` folder in the SolarTherm package, which can be used as a template to fit into the `st` simulations.
- **--ns flag** This flag followed by the number of data samples
- **--wd flag** This flag is followed by the working directory to save the simulation outputs
- **--np flag** This flag is followed by the number of processors to run the simulations. The default (without giving this flag) is the maximal available processors on the machine. Set it to 1, the program will be run in a single processor mode.
- **--runsolstice flag** This flag needs to be given if optical simulations (using `Solstice`) are involved in the system model. The program will be indicated to create different folders (i.e. `optic_case_1`) to save the optical results for each data sample.

A full list of flags and instructions can be seen in the help manual by running the following command:

```
st sensitivity --help
```

### st\_contingency

Uncertainties are commonly involved in a system design process. The optimal configuration of a system would be changed if some cost and performance parameters are deviated. The `st_contingency` tool re-optimises system design to cooperate with changes in performance parameters, assists sensitivity analysis and provides guidance on allocating contingency budgets to mitigate the risks. An example of how to use this command-line tool is shown below:

```
st contingency $mofn --excel $excel_file --ns 1000 --wd $working_dir --np 8 --
↳ matfile_default_design $matfn --target_likelihood 0.9 --runsolstice --plot
```

- `$mofn` is the directory of the SolarTherm model
- **--excel flag** This flag is followed by the directory of an excel spreadsheet that contains simulation parameters. It is a compulsory input for the `st_contingency` tool, where all the design, performance and cost parameters are specified. There is an example spreadsheet provided in the `examples` folder in the SolarTherm package, which can be used as a template to fit into the `st` simulations.
- **--ns flag** This flag followed by the number of data samples. The recommended number is greater than 1000. The data interpolation will be failed if too less number of data samples are created.
- **--wd flag** This flag is followed by the working directory to save the simulation outputs
- **--np flag** This flag is followed by the number of processors to run the simulations. The default (without giving this flag) is the maximal available processors on the machine. Set it to 1, the program will be run in a single processor mode.

- **--matfile\_default\_design flag** This flag is followed by the directory of the .mat result file of the default case (e.g. the optimal design at the default design condition). The default case is used as the base case to analyse risks and allocate contingency budget
- **target\_likelihood flag** This flag is followed by the likelihood of LCOE that is of interests, e.g. if 0.9 is given then the P90 LCOE will be presented after the simulation. An alternative flag is `target_lcoe`, which specifies the LCOE of interests and the program will return the likelihood to achieve this LCOE value and how much contingency budget is required. These two flags cannot be given at the same time.
- **--runsolstice flag** This flag needs to be given if optical simulations (using `Solstice`) are involved in the system model. The program will be indicated to create different folders (i.e. `optic_case_1`) to save the optical results for each data sample.
- **--plot flag** Plots of the distribution of the data samples, statistics of LCOE, energy production and capital costs will be created and shown if this flag is given.

The `st_contingency` tool can also take data sample files directly as an input, and post process the data to provide contingency and statistic analysis. Just putting an additional flag to the command introduced above:

```
--samplefile $samplefn
```

A full list of flags and instructions can be seen in the help manual by running the following command:

```
st_contingency --help
```

## Common flags for all tools

<b>-h, --help</b>	show this help message and exit
<b>-v</b>	verbose
<b>--start START</b>	simulation start time: <number>[.y,d,m,s]
<b>--stop STOP</b>	simulation stop time: <number>[.y,d,m,s]
<b>--step STEP</b>	simulation time step: <number>[.y,d,m,s]
<b>--tolerance TOLERANCE</b>	simulation tolerance: <number>
<b>--initStep INITSTEP</b>	simulation initial step size used by <code>dassl</code> and <code>ida</code> methods: <number>[.y,d,m,s]
<b>--maxStep MAXSTEP</b>	simulation maximum absolute step size used by <code>dassl</code> and <code>ida</code> methods: <number>[.y,d,m,s]
<b>--integOrder INTEGORDER</b>	simulation maximum integration order used by <code>dassl</code> and <code>ida</code> methods: <number>
<b>--solver SOLVER</b>	solver choice for OpenModelica
<b>--nls NLS</b>	non-linear solver choice for OpenModelica e.g. <code>newton</code> , <code>hybrid</code> , <code>kin-sol</code> , <code>mixed</code> , and <code>homotopy</code>
<b>--lv LV</b>	a comma-separated String list specifying which logging levels to enable, e.g. <code>LOG_DASSL</code> , <code>LOG_SOLVER</code> etc
<b>--fuel</b>	run post-processing calculations for levelised cost of fuel
<b>--np NP</b>	number of processes (set to 0 or 1 for serial mode)
<b>--peaker</b>	peaker configuration

## 1.4 Tutorial

In this section we will step through the process of constructing two simple solar thermal power plant models. These models are loosely based around a central tower receiver design with sodium heat-transfer fluid and storage.

The SimpleSystem model only considers energy transfers. It has a very flat structure, only using a few components from libraries. This makes it easy for the reader to understand exactly what is being modelled. Discrete states are used for the control system.

The FluidSystem model is a bit more complicated as it models the heat-transfer fluid. This model makes extensive use of models in the SolarTherm library. It shows the object-orientated approach to modelling where different generic components can be customised and connected together.

### 1.4.1 SimpleSystem

This model is provided as an example along with the source code: `examples/SimpleSystem.mo`. The different parts of this model will be described in this section.

The time-dependent variables are:

- $I_{dni}$ : DNI of sun (input)
- $\dot{Q}_{sch}$ : Scheduled discharge of tank (input)
- $\dot{Q}_{rec}$ : Heat flow into receiver from concentrator
- $\dot{Q}_{chg}$ : Heat flow charging tank
- $\dot{Q}_{dis}$ : Heat flow discharging tank
- $P_{elec}$ : Output electric power
- $E$ : Energy stored in tank
- $S_{con} \in \{1, 2, 3\}$ : Concentrator state (off, starting, on)
- $S_{blk} \in \{1, 2, 3\}$ : Power block state (off, starting, on)

The parameters are:

- $A_{rec}$ : Area of receiver aperture
- $C$ : Concentration ratio of concentrator/receiver combination
- $\eta_{rec}$ : Receiver efficiency
- $\eta_{blk}$ : Power block efficiency
- $P_{rate}$ : Power block electrical rating
- $E_{max}$ : Maximum amount of stored energy

The equations describing the system are:

$$\begin{aligned}\dot{Q}_{rec} &= \begin{cases} CA_{rec}I_{dni} & \text{if } S_{con} = 3 \\ 0 & \text{otherwise} \end{cases} \\ \dot{Q}_{chg} &= \eta_{rec}\dot{Q}_{chg} \\ \frac{dE}{dt} &= \dot{Q}_{chg} - \dot{Q}_{dis} \\ \dot{Q}_{dis} &= \begin{cases} \dot{Q}_{sch} & \text{if } S_{blk} \geq 2 \\ 0 & \text{otherwise} \end{cases} \\ P_{elec} &= \begin{cases} \eta_{blk}\dot{Q}_{dis} & \text{if } S_{blk} = 3 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

The state diagram for the concentrator is:

The state diagram for the power block is:

## Model

We start off a new model with the name SimpleSystem:

```
model SimpleSystem
  // Imports

  // Parameters

  // Variables/Models

initial equation

algorithm

equation

end SimpleSystem;
```

We have given the model four sections: public, initial equation, algorithm and equation. The public section is implicit, and we have further split it up into sections for imports, parameters and variables/models.

In the import section we import some packages which include standard types, constants and functions for doing conversions.

```
import SI = Modelica.SIunits;
import CN = Modelica.Constants;
import CV = Modelica.SIunits.Conversions;
```

## Parameters

The first two parameters provide the file names for tabulated weather data and spot market prices. We place these in a subfolder `resources` relative to where the simulation is run. These files have a format that can be read by table models from the Modelica standard library. The weather file can also be a standard TMY3 file, which will



automatically get converted to a suitable format with the `st_py_wea_to_mo` script. The `st_get_aemo_prices` script can be used manually to download and convert prices from AEMO.

```
parameter String weaFile = "resources/Mildura_Real2010_Created20130430.motab";
parameter String priFile = "resources/aemo_vic_2014.motab";
```

The next group of parameters describe physical values of the system. Of particular interest are `t_storage` and `P_name` which we will modify for different experiments.

```
parameter SI.Area A_con = 700 "Area of concentrator";
parameter SI.Area A_rec = 1 "Area of receiver aperture";
parameter Real C = 0.65*A_con/A_rec "Concentration ratio";
parameter SI.Efficiency eff_rec = 0.9 "Receiver efficiency";
parameter SI.Efficiency eff_blk = 0.48 "Power block efficiency";
parameter SI.Power P_name = 100000 "Power block nominal power";
parameter Real t_storage(unit="hour") = 5 "Hours of storage";
parameter SI.Energy E_max = P_name*t_storage*3600/eff_blk "Max stored energy";
```

This group of parameters define some trips for the control.

```
parameter SI.Energy E_up_u = 0.95*E_max "Upper energy limit";
parameter SI.Energy E_up_l = 0.93*E_max "Upper energy limit";
parameter SI.Energy E_low_u = 0.07*E_max "Lower energy limit";
parameter SI.Energy E_low_l = 0.05*E_max "Lower energy limit";
parameter SI.Irradiance dni_stop = 100 "DNI at which concentrator stops";
parameter SI.Irradiance dni_start = 200 "DNI at which concentrator starts";

parameter SI.Time t_con_on_delay = 20*60 "Delay until concentrator starts";
parameter SI.Time t_blk_on_delay = 15*60 "Delay until power block starts";
```

These parameters provide a way for defining a schedule for operating the power block. The current values are equivalent to always running the power block at full power (subject to there being enough stored energy).

```
parameter Integer n_sched_states = 1 "Number of schedule states";
parameter Integer sch_state_start(min=1, max=n_sched_states) = 1 "Starting schedule_
↪state";
parameter SI.Time t_sch_next_start = 0 "Time to next schedule change";
parameter SI.HeatFlowRate Q_flow_sched_val[n_sched_states] = {
    P_name/eff_blk
} "Heat flow at schedule states";
parameter SI.Time t_delta[n_sched_states] = {
    24*3600
} "Time differences between schedule states";
```

The last group of parameters calculate the capital and maintenance costs of the system, which are required to calculate the LCOE for the plant.

```
parameter SolarTherm.Utilities.Finances.Money C_cap =
    120*A_con // field cost
    + 135*C*A_rec // receiver cost
    + (30/(1e3*3600))*E_max // storage cost
    + (1440/1e3)*P_name // power block cost
    "Capital costs";
parameter SolarTherm.Utilities.Finances.MoneyPerYear C_main =
    10*A_con // field cleaning/maintenance
    "Maintenance costs for each year";
parameter Real r_disc = 0.05 "Discount rate";
parameter Integer t_life(unit="year") = 20 "Lifetime of plant";
```

### Variables/Models

First we define the models that load in the weather and price data, and provide access to them through variables.

```
SolarTherm.Utilities.Weather.WeatherSource wea(weaFile=weaFile);  
SolarTherm.Utilities.Finances.SpotPriceTable pri(fileName=priFile);
```

The physical variables for the plant are defined.

```
SI.HeatFlowRate Q_flow_rec "Heat flow into receiver";  
SI.HeatFlowRate Q_flow_chg "Heat flow into tank";  
SI.HeatFlowRate Q_flow_dis "Heat flow out of tank";  
SI.Power P_elec "Output power of power block";  
  
SI.Energy E(min=0, max=E_max) "Stored energy";  
  
SI.HeatFlowRate Q_flow_sched "Discharge schedule";
```

The control states and discrete variables for triggering time events are defined.

```
Integer con_state(min=1, max=3) "Concentrator state";  
Integer blk_state(min=1, max=3) "Power block state";  
Integer sch_state(min=1, max=n_sched_states) "Schedule state";  
  
Real t_con_next "Time of next concentrator event";  
Real t_blk_next "Time of next power block event";  
Real t_sch_next "Time of next schedule change";
```

Last of all we define variables to integrate up the amount of revenue from the spot market and the total amount of electricity generated. We also fix their starting value to 0, which could alternatively be done in the initial equation section instead.

```
SolarTherm.Utilities.Finances.Money R_spot(start=0, fixed=true)  
    "Spot market revenue";  
SI.Energy E_elec(start=0, fixed=true) "Generate electricity";
```

### Initial Equation

The initial equation section allows us to specify starting values (or equations) for variables. We only do it for the state variables, as the initial values for the rest of the variables can be calculated from these.

```
E = E_low_1;  
Q_flow_sched = Q_flow_sched_val[sch_state_start];  
con_state = 1;  
blk_state = 1;  
sch_state = sch_state_start;  
t_con_next = 0;  
t_blk_next = 0;  
t_sch_next = t_sch_next_start;
```

### Algorithm

The algorithm section is used to directly set the value of a variable on the left hand of the `:=` symbol. In an algorithm section a variable can be set more than once this way. This is different from an equation section where the number of equations must remain the same for all possible active branches.

According to the Modelica language spec, most (if not all) the relations in this section could actually go in the equation section. However, the OpenModelica specification hasn't fully implemented the spec with regards to systems of equations with discrete variables. So we instead have to put them in the algorithm section. It is not clear exactly what does and doesn't work, so in practice some trial and error and reading of compilation errors may be needed.

The first part of the algorithms section defines all the state transitions.

```
when con_state >= 2 and (wea.wbus.dni <= dni_stop or E >= E_up_u) then
    con_state := 1; // off sun
elsewhen con_state == 1 and wea.wbus.dni >= dni_start and E <= E_up_l then
    con_state := 2; // start onsteering
elsewhen con_state == 2 and time >= t_con_next then
    con_state := 3; // on sun
end when;

when blk_state >= 2 and (Q_flow_sched <= 0 or E <= E_low_l) then
    blk_state := 1; // off
elsewhen blk_state == 1 and Q_flow_sched > 0 and E >= E_low_u then
    blk_state := 2; // starting
elsewhen blk_state == 2 and time >= t_blk_next then
    blk_state := 3; // on
end when;

when time >= t_sch_next then
    sch_state := mod(pre(sch_state), n_sched_states) + 1;
end when;
```

The second part deals with setting the counters when a state changes and setting the flow schedule.

```
when con_state == 2 then
    t_con_next := time + t_con_on_delay;
end when;

when blk_state == 2 then
    t_blk_next := time + t_blk_on_delay;
end when;

for i in 1:n_sched_states loop
    when sch_state == i then
        Q_flow_sched := Q_flow_sched_val[i];
        t_sch_next := time + t_delta[i];
    end when;
end for;
```

## Equation

The equation section includes the differential and algebraic equations. These equations change as the states change, but the total number of equations always remains the same.

```
Q_flow_chg = eff_rec*Q_flow_rec;

der(E) = Q_flow_chg - Q_flow_dis;

Q_flow_rec = if con_state <= 2 then 0 else C*wea.wbus.dni*A_rec;

Q_flow_dis = if blk_state <= 1 then 0 else Q_flow_sched;
```

(continues on next page)

(continued from previous page)

```
P_elec = if blk_state <= 2 then 0 else eff_blk*Q_flow_dis;
der(E_elec) = P_elec;
der(R_spot) = P_elec*pri.price;
```

## 1.4.2 FluidSystem

This model is provided as an example along with the source code: `examples/FluidSystem.mo`. The different parts of this model will be described in this section.

By pushing components of the plant into their own models, for example, receiver variables and equations into a receiver model, we can reuse them in different plants and greatly simplify the system model. The top level system model then just needs to set the parameters and connect together components.

The FluidSystem model is considerably more complicated than the SimpleSystem model, but most of the extra complexity is hidden in the component models. Some modelling of the control is still done at the system level, but this could also be put into a model of its own.

The diagram below gives a rough outline of the different components in the model and how they connect together.

We focus on sections of interest in FluidSystem instead of walking through the whole model.

The Modelica standard library makes it easy to define different fluids and use them interchangeably in models. A fluid is defined in a Media package, which gets passed on to all models that used the fluid. Here we load in the Sodium fluid package and define some common system parameters which will be used by all models contained with the system model.

```
replaceable package MedRec = SolarTherm.Media.SodiumConst;

inner Modelica.Fluid.System system(
  energyDynamics=Modelica.Fluid.Types.Dynamics.FixedInitial,
  allowFlowReversal=false);
```

After defining all plant parameters, the component models are created (some of the constructor arguments have been omitted for brevity) and some control variables.

```
SolarTherm.Utilities.Weather.WeatherSource wea(weaFile=weaFile);
SolarTherm.Utilities.Finances.SpotPriceTable pri(fileName=priFile);

SolarTherm.Optics.IdealInc con(A_con=A_con, A_foc=A_rec);

SolarTherm.Receivers.Plate rec(...);

SolarTherm.Pumps.IdealPump pmp_rec(...);
SolarTherm.Pumps.IdealPump pmp_ext(...);

SolarTherm.Storage.FluidTank ctnk(...);
SolarTherm.Storage.FluidTank htnk(...);

SolarTherm.HeatExchangers.Extractor ext(...);

SolarTherm.PowerBlocks.HeatGen pblk(...);
```

(continues on next page)

(continued from previous page)

```
SolarTherm.Control.Trigger hf_trig(...);
SolarTherm.Control.Trigger cf_trig(...);

Boolean radiance_good "Adequate radiant power on receiver";
Boolean fill_htnk "Hot tank can be filled";
Boolean fill_ctnk "Cold tank can be filled";
```

The connectors of these components are then connected together in the equation section.

```
connect (wea.wbus, con.wbus);
connect (wea.wbus, rec.wbus);
connect (wea.wbus, pblk.wbus);
connect (con.R_foc, rec.R);
connect (ctnk.port_b, pmp_rec.port_a);
connect (pmp_rec.port_b, rec.port_a);
connect (rec.port_b, htnk.port_a);

connect (htnk.port_b, pmp_exc.port_a);
connect (pmp_exc.port_b, ext.port_a);
connect (ext.port_b, ctnk.port_a);

connect (ext.Q_flow, pblk.Q_flow);
connect (ext.T, pblk.T);

connect (hf_trig.x, htnk.m);
connect (cf_trig.x, ctnk.m);
```

Finally the equations for controlling the whole system are included.

```
radiance_good = rec.R >= R_go;

fill_htnk = not hf_trig.y;
fill_ctnk = not cf_trig.y;

rec.door_open = radiance_good and fill_htnk;
pmp_rec.m_flow_set = if radiance_good and fill_htnk then m_flow_fac*rec.R/(A_
↳ con*1000) else 0;
pmp_exc.m_flow_set = if fill_ctnk then m_flow_pblk else 0;

con.track = true;
```

### 1.4.3 Simulating

Copy the model to be simulated (here SimpleSystem) to a clean working directory. Copy the resources folder examples/resources/ to the same directory so that the simulation can find the weather data and market prices.

The command `st_simulate SimpleSystem.mo` will compile the model, compile the simulation and then run the simulation in one go. Let's do this step-by-step instead so that we can catch any errors along the way. First compile the model to check for errors in our Modelica code (the `--nosc` flag turns off simulation compilation and `--nosim` turns off running the simulation):

```
st_simulate --nosc --nosim SimpleSystem.mo
```

Fix any errors that are produced by the OpenModelica compiler. The next step is to compile the simulation using the compiled model (the `--nomc` flag turns off compiling the model, which we already just did):

```
st_simulate --nomc --nosim SimpleSystem.mo
```

Once this is done, we can run the simulation with its defaults (`--noc` turns off all compilation):

```
st_simulate --noc SimpleSystem.mo
```

At the end of the simulation some performance metrics are produced for the system, including the LCOE.

It is easy to produce a time-series plot of variables in the model, here DNI, output electricity, storage charging and stored energy:

```
st_plotmat SimpleSystem_res_0.mat wea.wbus.dni:P_elec,Q_flow_chg E
```

One day of operation in this time series is shown below.

The use of colons, commas and spaces in the above command highlights the simple notation used to indicate where to place each curve. The names of the variables that may be plotted can be identified by either examining the original model, or alternatively listing them:

```
st_plotmat -n SimpleSystem_res_0.mat
```

## 1.4.4 Results

### Control Switching

The plots below show the operation of the plants for 10 consecutive days out of the year. The top two plots are for SimpleSystem and the bottom two for FluidSystem. SimpleSystem is displayed in terms of stored energy and power charging and discharging the storage tank. FluidSystem represents these as the hot tank mass and mass flows instead. The charging and discharging powers and mass flows are directly controlled in response to the fluid levels in the tanks and the current solar DNI.

Both models have approximately equivalent values for storage hours and power block ratings. The collector, receiver and heat transfer components of SimpleSystem are more efficient than for FluidSystem. This results in more energy being captured and the tank filling up faster. It also results in more switching events to prevent the tank from overflowing. The power blocks in both models run at full power when there is available energy in the tank. This will prevent as much energy being spilt as possible, however a more financially rewarding control strategy would also take into consideration prices on the spot market.

The next plot focuses in on the control state of SimpleSystem for a single day. It shows transitions between the different control states and how it relates to the transfers of energy. The concentrator switches from the off state (`con_state = 1`) to the start tracking state (`con_state = 2`) in the morning once the solar DNI gets strong enough. It takes some time for the concentrator to fully focus and for the receiver to heat up, which is approximated by a delay before power actually starts to come out of the receiver (`con_state = 3`). At one point in the day the tank reaches near its upper limit, and the concentrator momentarily switches off to avoid overflowing the tank. The power block state follows a similar behaviour, but switches on and off in response to the tank energy being above/below a certain threshold.

The commands to produce and plot the above results are:

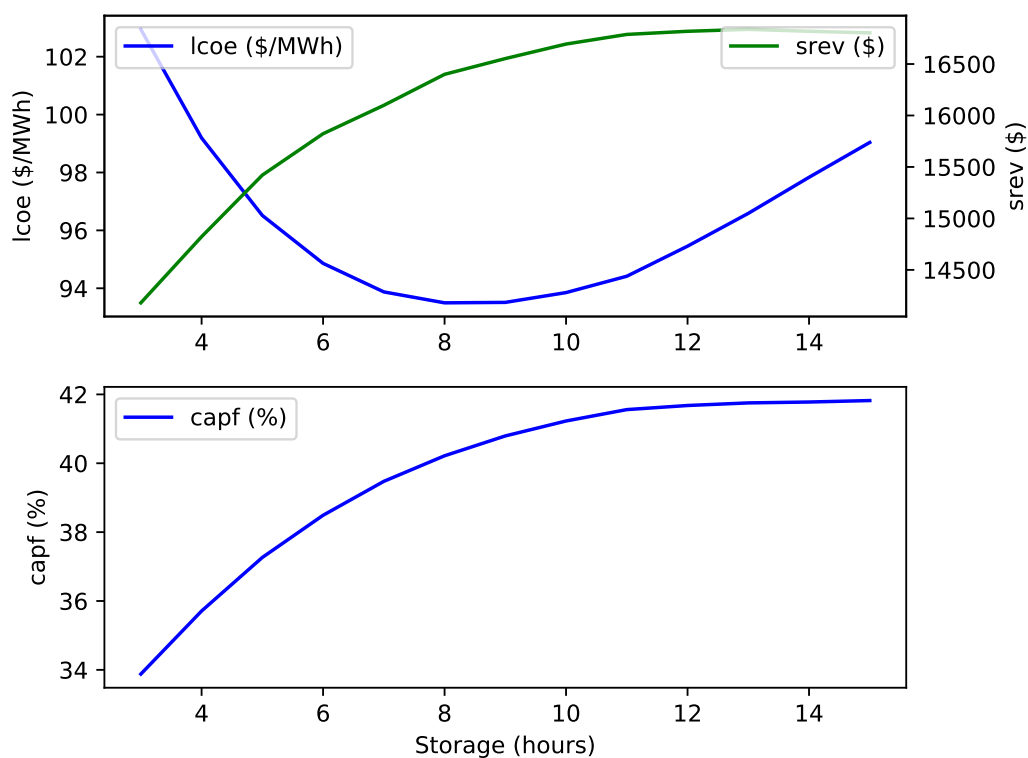
```

st_simulate --stop 1y --step 5m SimpleSystem.mo t_storage=6 P_name=75000
st_plotmat --xlim 2.25e6 3.1e6 SimpleSystem_res_0.mat wea.wbus.dni:P_elec E:Q_flow_
↪chg,Q_flow_dis
st_simulate --stop 1y --step 5m FluidSystem.mo t_storage=6 P_name=75000
st_plotmat --xlim 2.25e6 3.1e6 FluidSystem_res_0.mat wea.wbus.dni:P_elec htnk.m:pmp_
↪rec.m_flow_set,pmp_ext.m_flow_set
st_plotmat --xlim 2.25e6 2.34e6 SimpleSystem_res_0.mat wea.wbus.dni:con_state,blk_
↪state E:Q_flow_chg,Q_flow_dis

```

## Parameter Sweeps

The following plot shows how the LCOE, spot market revenue and capacity factor vary with storage size for the SimpleSystem model.



The next plot shows the same thing but for the FluidSystem model.

This plot shows how the LCOE changes for different power block ratings and storage hours for the SimpleSystem.

The same thing for the FluidSystem.

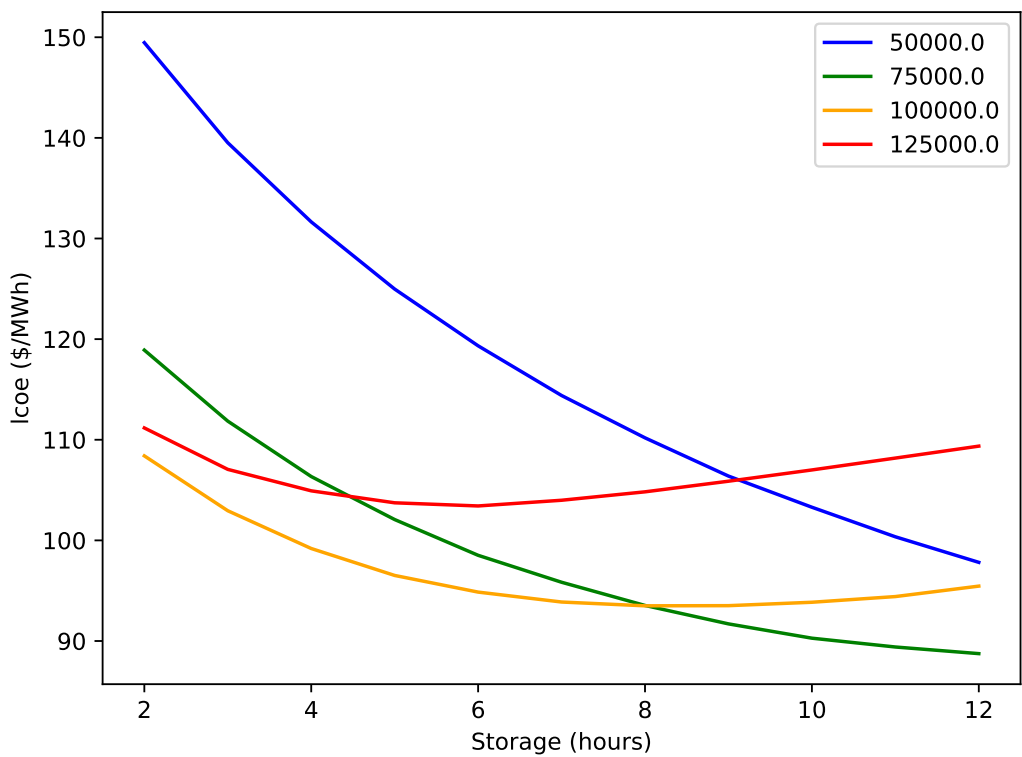
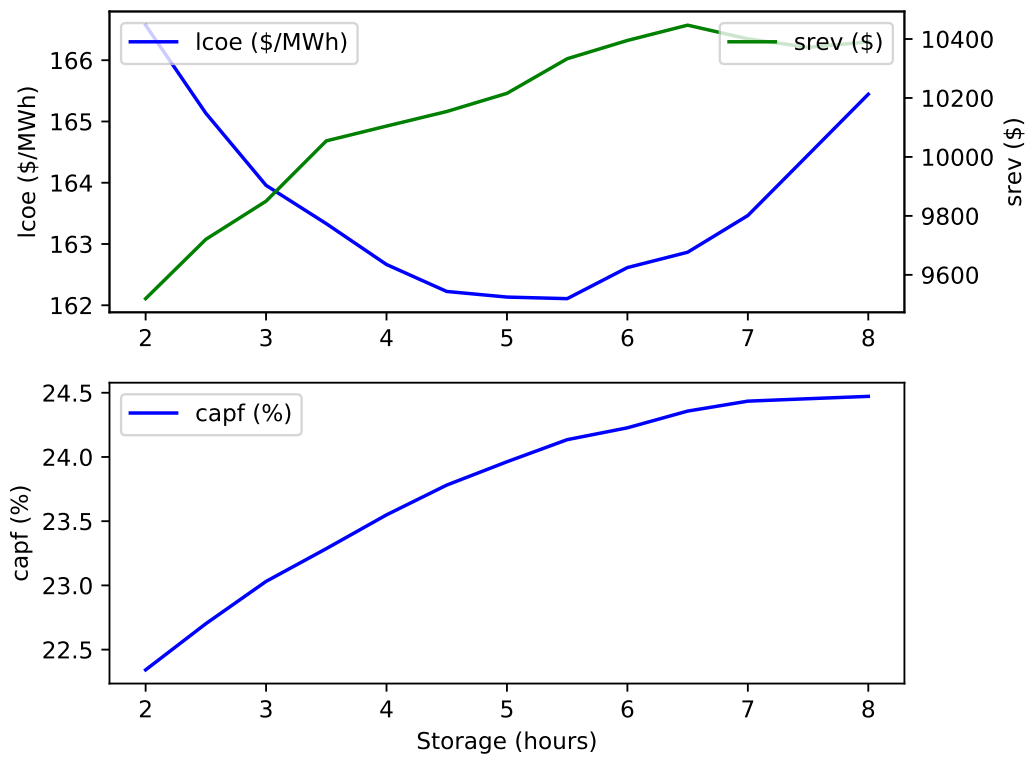
The commands to produce the results for the above plots are respectively:

```

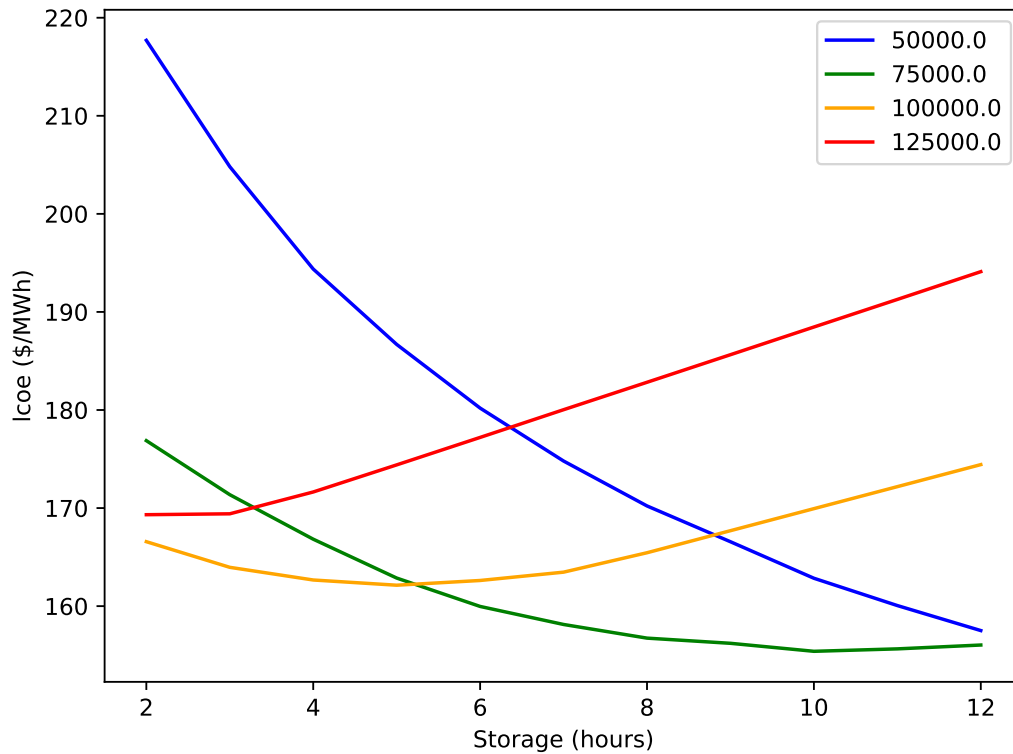
st_simulate --stop 1y --step 5m SimpleSystem.mo t_storage=3,4,6,7,8,9,10,11,12,13,14,
↪15
st_simulate --stop 1y --step 5m FluidSystem.mo t_storage=2,2.5,3,3.5,4,4.5,5,5.5,6,6.
↪5,7,7.5,8
st_simulate --stop 1y --step 5m SimpleSystem.mo P_name=50000,75000,100000,125000 t_
↪storage=2,3,4,5,6,7,8,9,10,11,12
st_simulate --stop 1y --step 5m FluidSystem.mo P_name=50000,75000,100000,125000 t_
↪storage=2,3,4,5,6,7,8,9,10,11,12

```

(continues on next page)







(continued from previous page)

## Optimisation

We can apply simple search techniques to optimise model parameters. For example, the following call will search (using the default method) for a value that minimises the LCOE for SimpleSystem over the power block rate and storage parameters:

```
st_optimise --maxiter 20 --stop 1y --step 5m -v SimpleSystem.mo P_name=50000,150000,
↪100000 t_storage=3,15,5
```

After 20 iterations the best solution it has found is for a power block rate of 69kW and storage time of 14hrs. The same approach for the FluidSystem produces a solution of 68kW and 8hrs. Note that both system models don't have any thermal losses from the storage vessels, and the power block costs are much higher than the storage costs, this tends to favour a very large storage system so that the power block size can be reduced.



## 2.1 Coding Style

### 2.1.1 Modelica

Modelica code should use the following conventions:

- `ModelNames`
- `function_names`
- `v_desc`, for example: `Q_flow_in`, `P_elec`, `p_max`, `epsilon_ext`
- `INNER_OUTER_CLASSES`

For common components, the following “acronyms” can be used in model names, model instances and packages:

- **CL:** Collector
- **RC:** Receiver
- **ST:** Storage
- **PB:** PowerBlock
- **HE:** HeatExchanger
- **CN:** Constants
- **CV:** Conversions
- **SI:** SIUnits
- **FI:** Finances

## 2.2 OpenModelica

### 2.2.1 Solvers

The different integration methods for the C runtime are listed [here](#). The default solver used is DASSL. The C++ runtime has support for the IDA solver and other sundials packages.

#### DASSL

This is a BDF method that selects its order (1-5) and step size at each step. It uses Newton's method to solve the resulting system of equations. It provides Newton's method with an initial guess by evaluating a polynomial that interpolates the last few time steps. These points are then used to calculate initial guesses for the derivatives. It has additional features like root finding functionality, and an initial condition solver for when not all initial values are given explicitly.

#### Runge-Kutta

This is the classical 4-th order Runge-Kutta method based on a fixed step.

#### Euler

Standard Euler method with fixed step.

### 2.2.2 Compiler

The OpenModelica Compiler (OMC) is used to flatten Modelica models, transform them and output them into a format suitable for simulation. The compiler source code is mainly written in MetaModelica and is contained in the OMCompiler/Compiler directory.

### 2.2.3 Model Flattening and Debugging

Return flat Modelica for last class in file `file.mo` using MSL and SolarTherm libraries:

```
omc file.mo Modelica SolarTherm
```

Create graph dot model for DAE of class (silencing flat model output):

```
omc -q -d=daedumpgraphv file.mo
```

Use `omc --help=debug` to get more debug options.

### 2.2.4 Simulation

The OMC SimCode module outputs the model into the target language. By default this is C source code, but other formats can be selected with the `--simCodeTarget` flag (see also OMCompiler/Compiler/SimCode/SimCodeMain.mo).

The simulation runtime source code is contained under OMCompiler/SimulationRuntime. This contains the C and C++ runtimes and several others.

## 2.2.5 C Simulation Runtime

The C code target is selected by default and the resulting code is generated by:

```
omc -q -s file.mo Modelica SolarTherm
```

The generated source is then compiled and linked to the C simulation runtime library into a final executable. When the last class in the input file is `Class`, the executable is built with:

```
make -f Class.makefile
```

It is then simulated with:

```
./Class
```

Experiment annotations in `Class` don't seem to be used. The `Class_init.xml` file contains model information and simulation parameters. The C simulation runtime reads this file, so things like time intervals can be changed without recompiling the simulation executable.

The simulation executable can accept a number of command line (call with `-help` and `-help=<flag>` for more information). For example to output events:

```
./Class -lv=LOG_EVENTS
```

By default the simulation will output the results into a file named `Class_res.mat`, which is a MAT-file version 4. Another optional output is a CSV file, but this dramatically slows down the simulation.

Listed below are the source files generated in this process and an initial attempt to explain their purpose.

**.makefile** Makefile for building the simulation executable. It could be manually edited if trouble with linking libraries in compilation.

**\_init.xml** Is an extension of the FMI standard. It has 3 main parts. The first is a high level description of the mode including the model name and number of variables. The second is the experiment parameters and the third is a list of the model variables along with their types, reference values and some parameters.

**\_info.json** Appears to contain similar information as that in the variables section of `Class_init.xml`.

**.c** This contains the main function for the simulation, which puts together the simulation data (the `DATA` struct defined in `SimulationRuntime/c/simulation_data.h`) from the other source files and then passes it off to the `_main_SimulationRuntime` function (defined in `SimulationRuntime/c/simulation/simulation_runtime.{h,cpp}`). This data contains information about the model (e.g., number of states) and callbacks for evaluating different parts of the model. The file defines functions for evaluating equations and algorithms in the model. It then builds the DAE and ODE systems from these functions which are used as callbacks.

**\_functions.{h,c}** These files appear to contain struct definitions for any records that are defined and wrapper functions for any external C code used. Included by most other files here.

**\_model.h** A whole heap of macros that are used for conveniently accessing things like the current value of a variable in the simulation. The macros are used in the callback functions and operate on the simulation data passed to the callback function. Included by most other files here.

**\_literals.h** I think just some simple string literals defined. Included by most other files here.

**\_includes.h** Empty in the examples examined.

**\_records.c** Contains some additional code for records, for example names for the record and record fields.

**\_01exo.c** Constructors and destructors for external objects.

**\_02nls.c** Contains nonlinear systems (haven't observed it populated yet).

**\_03lsy.c** Linear systems (haven't observed it populated yet).

**\_04set.c** Initial state set (haven't observed it populated yet).

**\_05evt.c** Raw time, zero crossing, relation and discrete events. For zero crossings it contains a callback function that calls relevant functions defined in `.c` in order to update variables, and another callback function for checking transitions.

**\_06inz.c** Contains equation and algorithm functions for initialising variables. These equations are named with indices smaller than the functions in `.c`. Also contains function to initialise mixed systems (haven't observed it populated yet).

**\_07dly.c** Delay. Don't know what this does and haven't seen it populated.

**\_08bnd.c** Updates bound parameters and variable attributes start, nominal, min and max. Not sure exactly when this would be used. It contains equation and algorithm functions for this purpose with indices greater than those in `.c`.

**\_09alg.c** Collects together the functions from `.c` for the algebraic system.

**\_10asr.c** Contains functions checking and throwing asserts. These functions have indices greater than those in `_08bnd.c`.

**\_11mix.{h,c}** Contains mixed systems (haven't observed it populated yet).

**\_12jac.{h,c}** Contains functions for calculating and initialising the Jacobian. It seems to always contain sparsity information. The compiler flag `--generateSymbolicJacobian` needs to be set to produce equations for the symbolic calculation of the jacobian. Additionally a compiler call with `-g=Optimica` flag on an `optimization` class will produce stuff.

**\_13opt.{h,c}** Contains functions needed in optimisation like the objective and Lagrangian and for grabbing bounds from variables. Need to call compiler with `-g=Optimica` flag and have an `optimization` class to get something interesting here.

**\_14lnz.c** Linearisation. Not sure what for and haven't seen it populated.

The C simulation runtime is located under `OMCompiler/SimulationRuntime/c`. Interesting files include:

**simulation\_data.h** Contains the `DATA` struct and others.

**simulation/simulation\_runtime.{h,cpp}** Contains `_main_SimulationRuntime` function which gets called to run the simulation with the model data from the generated model code.

**simulation/solver/solver\_main.c** The `solver_main_step` function gets called to step the simulation and then passes the stepping off to the appropriate backend solver.

**simulation/solver/perform\_simulation.c** Contains the main simulation while loop in the function `prefixedName_performSimulation`.

**simulation/solver/dassl.c** Contains the DASSL related functions including the `dassl_step` function. According to a comment in the file the integrated zero crossing method is disabled and zero crossings are instead handled outside DASSL (not same thing as internal root finding, which by default is turned on). It has functions for the symbolic or numeric calculations of Jacobians. The method can be selected with the `-dasslJacobian` flag, where `coloredNumerical` is the default. See the `_12jac.{h,c}` files for where the symbolic jacobian comes from (has to be enabled at during model compilation).

## 2.2.6 C++ Simulation Runtime

An example of C++ code generation:

```
omc -q -s --simCodeTarget=C++ file.mo Modelica SolarTherm
```

The resulting files are slow to compile and sometimes fail for more complicated models. The simulation parameters are passed to the executable on the command line instead of being read from an XML file. See the `Class.sh` script for an example of the command line switches. It is able to use IDA instead of DASSL, which worked on at least a very simple example.

## 2.2.7 Language

Here are some notes on different aspects of the Modelica language. Some of the quirks presented here may just be specific to OpenModelica, whereas others are part of the language specification.

### Expandable Connectors

Expandable connectors can have some defined components:

```
expandable connector Bus
  Real a;
end Bus;
```

Those that are undefined must be connected to a defined component, and the connect operator must be used:

```
model Model
  input Real x;
  input Real y;
  Bus bus1;
  Bus bus2;
equation
  connect(x, bus1.a);
  connect(y, bus1.b);

  connect(bus2.b, bus1.b); // Fail: two undefined components connected
  bus1.b = y; // Fail: connect must be used to construct b
end Model;
```

Expandable connectors cannot have flow components, but may contain non-expandable flow components.

The direction (input/output) of an expandable connector component will be determined using the normal rules with respect to the direction of the component connected to.

### Discrete Equations

OpenModelica doesn't yet appear to fully support having systems of equations. This error might be triggered by other issues. If in doubt put all the `when` clauses in an algorithm section.

## 2.3 Libraries

Here we list a number of existing libraries and within them the components that might be useful. Many of these libraries provide components that can be used for the detailed simulation of solar thermal power plants. However, it is yet to be seen if they are suitable for performing year long simulations and if they have enough flexibility to enable rapid development and prototyping of new configurations.

### 2.3.1 Modelica

A number of standard and third party Modelica libraries are listed [here](#) with more details found [here](#).

#### Modelica

The [Modelica standard library](#).

**Machines** Models for synchronous, asynchronous and DC machines.

**Fluid** 1-dimensional thermo-fluid flow in pipes, vessels, fluid machines, valves and fittings. Any of the media from the Media library can be used. The components in this library need a System component at the top level to describe the surrounding environment, assumptions, initialisation and default parameters. Some of the assumption options are: reversible flows, dynamic vs steady-state, ignore storage of mass, momentum and energy.

**Connectors** Ideal mixing is used to handle joins of more than two connectors. It makes use of [stream](#) connections to handle enthalpy and other fluid properties. For a single substance medium, mass flow rate, absolute pressure and specific enthalpy are used. For multi-substance media mass fractions are used for both significant components and trace components (fluid properties are ignored but mass is balanced). The connectors don't model any friction or velocity changes at the point of connection, so explicit components should be modelled if this or non-ideal mixing is needed (for example fittings).

**Vessels** Closed volume and open tank vessels as well as base classes to build your own. They are ideally mixed and have the option for heat transfer through a thermal port. Pressure losses for the fluid ports can be turned on to model different types of port connections to the vessel.

**Pipes** Models for static (no mass or energy storage) and dynamic pipes. A dynamic pipe can be used to produce the same results as the static pipe by changing a setting. There are lots of parameter settings, so it might be best to refer to the examples to get an idea of how to model different types of pipes. A finite volume method and staggered grid scheme is used to approximate the PDEs. The pipe is split into a user selectable number of nodes. I has heat ports which can be used for thermal losses or other heat exchange purposes.

**Machines** Pistons and pumps. The centrifugal pumps can either be ideally controlled or externally operated by a shaft or electronic signal.

**Media** Media thermodynamic parameters are defined with functions, polynomial equations or tables. Fluids include ideal gases, water, air, table-based incompressible fluids and compressible liquids. Instructions are provided on how to [define](#) and [use](#) media. Custom fluids can be provided and will work with other components as long as they extend the media interface.

**Thermal** This is split into two libraries, one on incompressible fluid heat flows and one on heat transfer with lumped elements. It is not obvious how the thermal fluid flows is different from what is provided in the Fluid library. One possibility is that the media is simplified here because the focus is on thermal transfer?

**Blocks** Blocks contains input/output blocks that can do a whole range of things including PID control, filtering, transfer functions, mathematical operations, logic, sampling, limiting, delays, signal generation and table interpolation (for tabled data provided in hard-coded C).

**Utilities** This library provides functionality for file operations, reading/writing to files/streams and operations on strings.

#### ThermoSysPro

[ThermoSysPro](#) is an EDF library that provides common components for the static and dynamic modelling of nuclear, thermal and solar power plants, and energy conversion systems.



It has models of combustion chambers, boilers, heat exchangers and machines. It also has models for [molten salt](#) and a [solar collector](#). There is a paper from SolarPACES 2013 on using this library for modelling three types of solar plant by [Baligh El Hefni 2013](#).

It is open source, but the only access I have found to the library is from checking out the OpenModelica fork [here](#).

## ThermoCycle

[ThermoCycle](#) is unique in its ability to handle non-conventional working fluids (refridgerants, ammonia, etc). It has a focus on organic rankine cycles and heat pumps. The connectors used are compatible with the Modelica standard library. It uses the external library [CoolProp](#) in order to obtain the properties of these complex fluids, which is connected through the [ExternalMedia](#) interface. The library documentation has some potentially useful tips on how to handle numerical problems.

It contains models for a couple of solar collectors and some tutorials on their website. They have a number of [papers](#), including one on dynamic simulation of a micro-CSP plant with storage [Ireland et al. 2014](#). The publications page also contains links to a couple of repositories which might contain additional models resulting from these publications.

I have only seen mention of Dymola on the website, so it is yet to be seen whether this will work under OpenModelica. The github link to the library is [here](#).

## ThermoPower

[ThermoPower](#) is for the dynamic modelling of thermal power plants with the purpose of studying control system strategies and architectures. It contains model a model of a gas turbine and a steam turbine.

It lists quite a number of papers including one on a solar supercritical CO2 brayton cycle by [Casella and Colonna 2011](#).

It was developed primarily for Dymola. It has some reporting on the test failures under OpenModelica. The github repository is [here](#).

## Buildings

[Buildings](#) is used to model the thermal performance of buildings.

**ReaderTMY3** Reads TMY3 weather data, including those supplied with EnergyPlus. It is a block component type that provides the data on ports to which other components can connect. An example is provided [here](#). This example failed on an initial test with OpenModelica.

## 2.3.2 Other

### ExternalMedia

[ExternalMedia](#) provides a framework for interfacing external fluid property code. It works with [FluidProp](#) and [CoolProp](#), but has only been tested for OpenModelica on Windows (Dymola on Linux).

### CoolProp

[CoolProp](#) is a C++ library for calculating fluid properties.

## FluidProp

FluidProp is proprietary but a free version is available with a subset of the features.

## 2.4 Modelling

### 2.4.1 DAEs

DAEs are a system of differential equations in one independent variable. In DAEs, the derivatives of some dependent variables might not explicitly appear.

The system of DAEs can be converted to a system of ODEs (where the derivatives are explicit) by differentiating the equations with respect to the independent variable. The number of times they need to be differentiated to convert to a system of ODEs provides the index. To solve a system of DAEs they are typically first reduced to an index of 1 (often more efficient than fully converting to ODE and solving), although techniques for solving high index DAEs also exist. For more information see [wolfram](#) or [wikipedia](#).

The primary solver used by OpenModelica for solving index 1 or less DAEs is DASSL. Some information on the solver is provided [here](#). Another popular DAE solver is IDA (which is derived from DASPK).

### 2.4.2 PDEs

OpenModelica can only solve DAEs or ODEs, so one of a number of different approximation methods will be required for any PDEs:

- Finite difference method
- Finite element method
- Finite volume method
- Weak form

### 2.4.3 Discrete Events

Have to make sure the number of equations is equal to the number of variables at all times.

### 2.4.4 Considerations

**From Modelica.Fluid documentation:** “The resulting simulation performance however often strongly depends on the model structure and modeling assumptions made. In particular the direct connection of fluid volumes generally results in high-index DAEs for the pressures. The direct connection of flow models generally results in systems of implicit nonlinear algebraic equations.”

## 2.5 Models