pytrnsys

Oct 27, 2020

Contents

1 Table of contents **3**
- 1.1 Getting Started . . . 3
- 1.2 Running simulations . . . 14
- 1.3 Processing data . . . 17
- 1.4 Example systems . . . 25
- 1.5 Ddck repository . . . 28
- 1.6 Developer's guide . . . 35
- 1.7 pytrnsys . . . 37

2 Indices and tables **45**
- 2.1 Developers . . . 45
- 2.2 Acknowledgments . . . 45

Python Module Index **47**

Index **49**

The pytrnsys package provides a complete python-based framework to run, process, plot, and report TRNSYS simulations. It is designed to give researchers a fast, fully automatized, and reproducible way to execute and share TRNSYS simulations by the use of a single short configuration file. In addition, a large variety of commands is accessible to post-process simulation results in one shot. This functionality extends beyond processing TRNSYS generated data and can also be used for generic data.

The package was developed at the SPF - Institute for Solar Technology at the OST - Eastern Switzerland University of Applied Sciences.

CHAPTER 1

Table of contents

1.1 Getting Started

1.1.1 Installation

Up to now, only TRNSYS17 is fully supported in all the example projects. In order to use pytrnsys, you need the following prerequisites on your machine:

- TRNSYS17 is installed.
- A working LaTeX distribution. We recommend MiKTex due to its included package management system.

TRNSYS17 should be installed under:

```
C:/Trnsys17
```

If the path to your TRNSYS17 installation is different from that you will need to manually copy dll-files (see below) and adjust the TRNSYS-path in the config-file.

Additional optional prerequisites are:

- Automated plotting is done by matplotlib. GLE is needed to create Q-vs-T plots using the commands *plotHourlyQvsT* or *plotTimestepQvsT*
- GLE is also supported by the configuration file keyword *setPrintDataForGle* which exports a .gle file which can be used for further plotting in GLE.
- Inkscape can be used to save the plots in the enhanced meta file format by using the *plotEmf keyword* in the processing.

The pytrnsys package is available for python>3.5 through pip:

```
pip install pytrnsys
```

Along the main pytrnsys-package two additional packages will be installed. The package pytrnsys_examples contains different example projects that can be used out of the box to investigate different parametrizations of the represented

systems. Up to now, the example projects contain a solar thermal system for domestic hot water preparation and a pv battery system. The other package pytrnsys_ddck contains the ddck-repository.

Before you can run the example, you need to copy dll-files to your TRNSYS17-installation. This is done by executing:

```
pytrnsys-dll
```

If the path of your TRNSYS17-installation is different from the one specified above you need to manually copy all dll-files from:

```
ptrnsys_ddck/dlls
```

to:

```
Trnsys17/UserLib/ReleaseDLLs
```

You can now test the setup by executing one of the example projects. The default solar domestic hot water system can be run by executing:

```
pytrnsys-run
```

in the python environment in which pytrnsys was installed. The command will launch a parametric study of different solar collector areas that aims to determine the solar fraction of the domestic hot water system. It will run on multiple cores in parallel using the total amount of cores of your machine minus 4. The simulations will be executed in a new folder called solar_dhw_simulaions that will be created in the current working directory.

Once the simulations are finished the simulation results can be processed using the following command inside the newly created simulation folder solar_dhw:

```
pytrnsys-process
```

The most important files created by the processing are a results pdf-file in each subfolder of the parametric runs as well as comparison plots in the main folder.

1.1.2 The philosophy of pytrnsys

Pytrnsys provides a python framework for TRNSYS along with a working methodology. This means that the workflow of using TRNSYS with pytrnsys is different compared to using a standard methods such as Studio. The main purpose of pytrnsys is to facilitate the life of the user allowing to use most of the funcionality of the python package without having to know python in detail. For that we use config files with some scripting funcionality that is described along this documentation. The proposed methodology works at three different level:

pytrnsys methodology

- **Build a TRNSYS deck**
  - The idea is to use a modular approach stacking files with an extension *.ddck together to form a single dck TRNSYS file.
  - The ddck files are structured in a way that can be reused/modified easily to adpat to new cases. These files should be uploaded to GIT repositories if sharing/reusing is foreseen.
  - Our core idea to build a TRNSYS deck is to use a flow solver and an hydraulic ddck file which is custom to each case. A TRNSYS flow solver is an own-developed TYPE that gives the mass flow of all pipes and elements given the mass flows of pumps and positon of 3-way controlled valves for each time step.

  - This hydraulic file also includes all TYPEs for the hydraulic elements such as pipes and tee-pieces. Thus, when connecting to all elements such as solar collectors the mass flow and temperature of the pipe that enters the collector which has a specific format name can be used directly. That is, connection between elements is very easy and can be done in a fully automatic way.
  - At SPF we have a TRNSYS Graphical User Interface (GUI) under development. One of the functionalities of the GUI is to export the hydraulic set-up such that can be used directly with the flow solver. For examples the hydraulic files you will find in the examples are exported from our GUI. However, the GUI is not publicly available at the moment.
  - Although it is theoretically possible to use the TYPE flow solver without a GUI it is very tedious to do so. This means that without the GUI the flow solver might not help you.
  - If you are interested in the TRNSYS GUI you can contact danil.carbonell@pf.ch. Currently this GUI is being shared with institutes in the framework of research projects. Until we can offer this GUI in a more "professional" version, a collaboration within a project might be the easiest way to get access to it. We are not a software development company and we don't get the GUI developments paid at all, thus we need to improve it and extend it within research projects.
  - If you don't have the GUI you can still work with the pytrnsys without any problem. However, you will need to know/connect the inputs (mass flow and temperature) for each component like in normal TRNSYS. Our GUI and flow solver makes this almost fully automatic. This is the only limitation of the open source version. All the rest can be used 100%.

- **Run a TRNSYS deck**
  - Once a TRNSYS deck has been generated with the method described above, by your own method or by Studio you can execute this deck with a lot of nice functionalities. For example you can easily run parametric studies in parallel and modifiy the deck file using a configuration file.
- **Process a TRNSYS simulation**
  - Once the simulations are done you can easily process all results including several results from parametric studies using a config file where the main processing calculations can be done.
  - Some automatic processing is always done. For example the energy demand and the energy balance of the system is calculated automatically provided a proper syntaxis is used in the TRNSYS deck.
  - The custom-made processing can be easily added. To fully use our processing functionality you need a working latex environment.
  - The processing functionality includes monthly and hourly calculations, files with results, and different types of automatic plots.
  - Basically all functionality we see is of use in general we add it into the config file. Other more project specific processing we do at python level. You will see how to do this at the developer's guide section.
  - Our method of processing TRNSYS simulations is based on our method to build a TRNSYS deck, so to fully use all functionalities you will need to change your own deck to have a similar structure as the one we have. For example the results are always stored in a temp subfolder and to do the automatic energy balance you need to provide the data with specific namings convention. However, still many functionality can be theoretically done if you don't follow our method and style, but we never checked this, so you might find issues there.

This package is not intended to substitute your skills in TRNSYS, but if you have them it will make your life easier. For those that don't know TRNSYS yet it will make the introduction easier, or at least this is our hope.

1.1.3 Load the example projects and the TRNSYS files

The core idea of pytrnsys is to use modular parts of a TRNSYS dck file called ddck files. Due to the modularization these ddck files can be stacked together in order to build a complete system simulation. The main repository of ddck files is installed together with pytrnsys in a separate package called pytrnsys_ddck. Pytrnsys has different example projects included that are installed in a separate package called pytrnsys_examples. You can copy the ddck files as well as the example projects to a local directory of your machine be executing the following command in the chosen folder:

```
pytrnsys-load
```

This will create two folders pytrnsys_examples and pytrnsys_ddck. In pytrnsys_ddck you find the whole default ddck repository of pytrnsys. In pytrnsys_examples the example projects are located. In each example project subfolder, a run configuration file and a process configuration file as well as some ddcks that are custom to the project can be found.

You can test the new local setup by again executing the solar domestic hot water system. The local set up is executed by running `pytrnsys_run` with a configuration file as the first and single argument:

```
pytrnsys-run path_to_your_pytrnsys_examples/solar_dhw/run_solar_dhw.config
```

Similarly, after the simulation is finished, you can change to the newly created folder solar_dhw and execute:

```
pytrnsys-process path_to_your_pytrnsys_examples/solar_dhw/process_solar_dhw.config
```

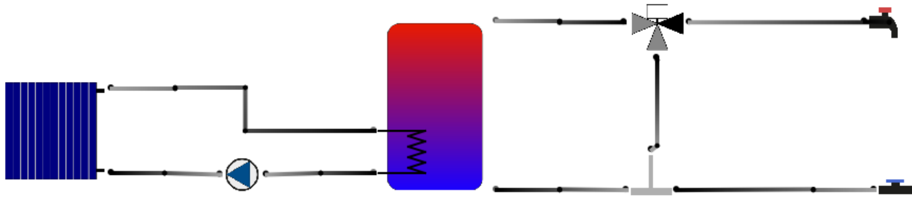
Congratulations! You now have your own pytrnsys installation with a local version of the example projects and ddck files that you can change as you wish. In the following section, you will learn about the opportunities pytrnsys offers for customizing the simulations.

1.1.4 Setting up your simulation

Pytrnsys offers a large number of possibilities that allow the user to customize the system without having to change the dck files by hand. To learn how to use pytrnsys you can go through the tutorial:

Tutorial

If you correctly installed pytrnsys and were able to use the two commands **pytrnsys-run** and **pytrnsys-process** you most likely already simulated and processed your first system with pytrnsys. But for sure you would like to adapt the example system to your weather data and to your demand profile, excecute other parametric studies, use different system sizing, analyse other simulation results etc.. In this tutorial, you learn how to do all this and much more. We will work with the solar domestic hot water example system that looks like this:



Create your own local system

You can set up a local copy of the pytrnsys example systems and the pytrnsys ddck-repository by executing **pytrnsys-load** in the target folder of your choice. This process is explained in the *getting started section*. Once you have done this, you should see a sub-folder pytrnsys_examples that contains the following files:

```
pytrnsys_examples
+-- solar_dhw
    +--latexNames.json
    +--process_solar_dhw.config
    +--run_solar_dhw.config
    +--solar_dhw_control.ddck
    +--solar_dhw_control_plotter.ddck
    +--solar_dhw_hydraulic.ddck
    +--solar_dhw_storage1.ddck
```

Besides the two configuration files, the folder contains some ddck files that are custom to the project. The file latex-Names.json contains a dictionary that is used to translate TRNSYS variable names into the latex format used in the results pdf file.

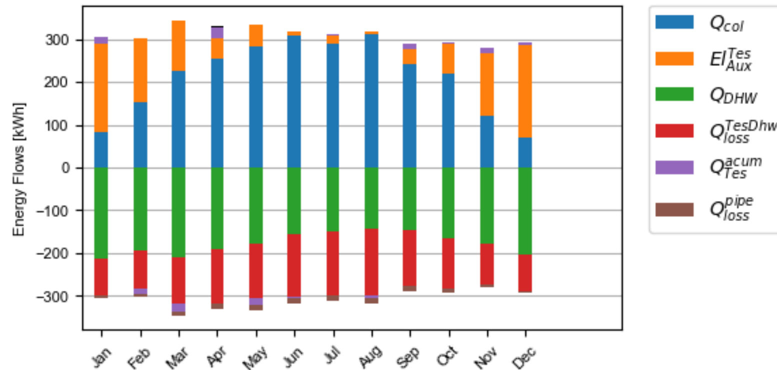
If this system is executed with **pytrnsys-run**, a new simulation folder with the name solar_dhw is created that has the following structure:

```
solar_dhw (simulation base folder)
+-- debugParralelRun.dat
+-- runLogFile.config
+-- solar_dhw.dck
+-- UnitsType.info
+-- (location of comparison plots)
+-- solar_dhw-Ac2-VTes75 (simulation sub folder)
    +--solar_dhw-Ac2-VTes75.dck
    +--solar_dhw-Ac2-VTes75.log
    +--solar_dhw-Ac2-VTes75.lst
    +-- (location of single simulation results and plots)
    +--temp
       +--*.Prt
```

As you can see, pytrnsys creates a base dck file solar_dhw.dck that is then copied to the subfolders where the actual dck file containing possible changes that were defined in the configuration file are included. So in a parametric study, each single simulation will have its own subfolder. In the temp folder of each subfolder, the simulation results defined in the TRNSYS printers will be stored.

The finished simulation results can be processed by running **pytrnsys-process** inside the base simulation folder solar_dhw. The processing will add various files to the folder structure. Single simulation plots and results will be placed in the subfolders while comparison plots and comparison data will be added to the simulation base folder.

As in all pytrnsys projects, a heat balance will be produced for the solar domestic hot water example system. The heat balance plot of the unchanged example system looks like



To have an overview of the by default created plots and result files please go through the different simulation folders of the example system.

Work with the run-configuration file

The easiest way to work with pytrnsys is to use a pre-defined system and to modify it with the configuration files. The pytrnsys configuration files offer a large amount of functionalities that are described in detail in in the *config file page*. In the following sections, some of the most important functions are explained in a step-by-step guide.

Change TRNSYS variables

Constants and Equation of the TRNSYS dck-file can be changed by the following line in the config file:

```
deck trnsysVariableName value
```

In the runconfiguration file run_solar_dhw.config, we can see that there are already three such lines that change the dck-file:

```
deck START 0    # 0 is midnight new year
deck STOP  8760 #
deck sizeAux 3
```

It is recommended to always have the START and STOP variable exposed in the configuration file since they define the simulated timespan and are of high importance. In addition, the variable sizeAux is changed to a value of 3. This variable defines the power in kW of the auxiliary heater inside the thermal storage. We can choose now any other variable in one of the used ddck files that we would like to change. Let us say we would like to change the slope of the thermal collector. In order to identify the relevant parameter we have to open the ddck of the used solar collector model /solar_collector/type1/type1.ddck. In there we see that the collector surface tilt definition **C_tilt** is a dependency of another ddck, in particular the variable **slopeSurfUser_1**. Looking at the solar_dhw.dck file we see that the definition of **slopeSurfUser_1** is done in the file /weather/weather_data_base.ddck. Therefore, in the configfile we can add the following line to simulate a facade collector with slope 90:

```
deck slopeSurfUser_1 90
```

If we would like to add the collector slope to the parametric study, we can use the `variation` keyword:

```
variation slopeSurfUser_1 30 45 60 75 90
```

When this line is added without removing any of the other variation lines the total amount of simulations will increase to 6x2x5=60 which will take a while. Feel free to reduce the number of values per variation to save time.

Change the used ddck-files

In the solar domestic hot water system, the following ddck files are used by default:

```
string PYTRNSYS$ "..\..\pytrnsys_ddck\"
string LOCAL$ ".\"

PYTRNSYS$ generic\head
PYTRNSYS$ demands\dhw\dhw_sfh_task44
PYTRNSYS$ weather\weather_data_base
PYTRNSYS$ weather\SIA\normal\CitySMA_dryN
PYTRNSYS$ solar_collector\type1\database\type1_constants_CobraAK2_8V
PYTRNSYS$ solar_collector\type1\type1
LOCAL$ solar_dhw_control
LOCAL$ solar_dhw_storage1
LOCAL$ solar_dhw_hydraulic
LOCAL$ solar_dhw_control_plotter
PYTRNSYS$ generic\end
```

In pytrnsys, it is possible to replace some of the ddck files depending on the structure of the project. In this case, it is possible to replace the domestic hot water with another demand as well as to chose another weather data location. The current city which is Zurich (SMA) can be replaced with Locarno in the south of Switzerland. In Locarno, there are more hours of sunlight in the year which will help us to have a better performance for the solar domestic hot water system. In the default database pytrnsys-ddck there are many different Swiss cities. Locarno can be chosen by:

```
PYTRNSYS$ weather\SIA\normal\CityOTL_dryN
```

You can go through the weatherSIA folder in the ddck repository to see all by default available weather data ddck files.

Run the modified configuration file

Now you are almost ready to run your new simulation. In order to not overwrite the default system run you should specify a new folder name. This can be done by changing the *addResultsFolder* parameter:

```
string addResultsFolder "my_new_solar_dhw"
```

Save you configuration file and use it with the **pytrnsys-run** command to start the simulation.

Work with the processing-configuration file

In the default example system processing file, there are already some custom calculations and custom plots given as examples. In this section be will go through the process of adding some more calculations and plot the results of the custom calculations.

Add custom calculations to the processing

In the default processing configuration file of the solar domestic hot water system, the monthly and overall solar fraction of the system is calculated:

```
calcMonthly fSolarMonthly = qSysIn_Collector/qSysOut_DhwDemand
calc fSolar = qSysIn_Collector_Tot/qSysOut_DhwDemand_Tot
```

Another interesting quantity to analyze the performance of a solar system is the Total Solar Efficiency

$$\eta^{coll} = \frac{Q^{collector}}{E_{irradiance}}$$

This can be implemented using the simulation results. In the monthly printer section of the collector ddck file /solar_collector/ype1/type1.ddck we can see that the power gain and the irradiated power per area are integrated, printed and accessible in the processing as **PColl_kWm2** and **IT_Coll_kWm2**. So we have everything to calculate the Solar Efficiency. Again we calculate the monthly values as well as the overal yearly value:

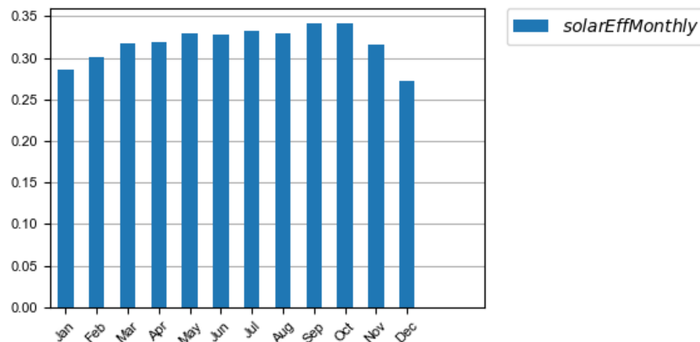
```
calcMonthly solarEffMonthly = PColl_kWm2/IT_Coll_kWm2
calc solarEff = PColl_kWm2_Tot/IT_Coll_kWm2_Tot
```

Add custom plots to the processing

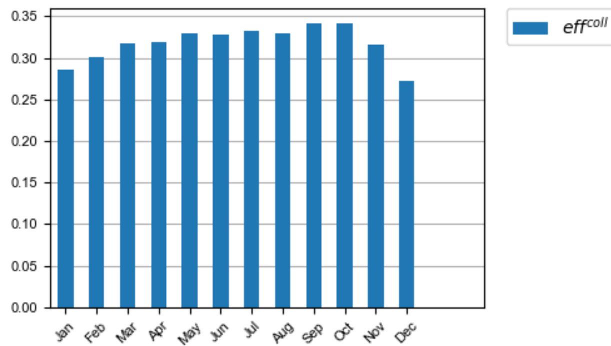
We can plot the new results in different ways. First of all, we can use the monthly values to create a monthly bar plot by including:

```
stringArray monthlyBars "solarEffMonthly"
```

This will result in a plot that looks like this:

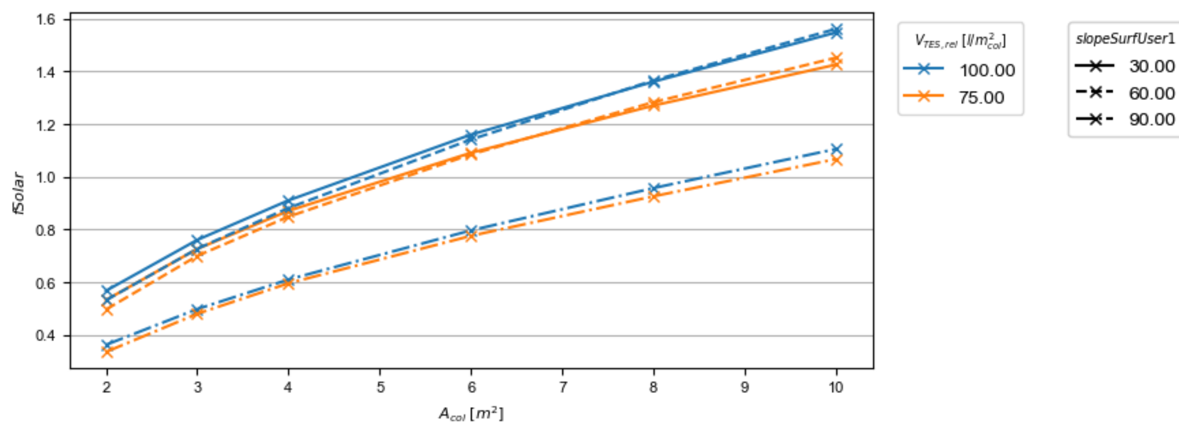


By default, pytrnsys will use the variable name in all the legends. We can change this to a nicer looking LaTeX-formatted string by adding an entry to the dictionary in the projects latexNames.json. Adding a line “effSolarMonthly”: “η^{coll}” in the json file and rerunning the processing will give a plot with a nicer legend:



We can also create a comparison plot of the solar efficiency including the parametric runs on the collector area, the storage size and the collector slope by using the following line:

```
stringArray comparePlot "AcollAp" "solarEff" "volPerM2Col" "slopeSurfUser_1"
```



Do parametric runs with different ddcks

For some tasks, it is not enough to replace a single number in the dck file. Such tasks are for example the replacement of the weather data, the replacement of the demand data or the change of the parametrization of a component i.e. the solar collector. In pytrnsys this is solved by the possibility to define a seperate ddck for each case and to loop though the ddck files during the parametric runs.

Let us say that instead of changing the collector slope in our example system, we would like to use different domestic hot water profiles. In the default ddck repository, pytrnsys offers both a domestic hot water profile for a single family house as well as one for a multi-family building. If we would like to include both in the same parametric run, we can include the following line:

```
changeDDckFile dhw_sfh_task44 dhw_sfh_task44 dhw_mfh
```

The `changeDDckFile` command interprets the first argument as the substring to be replaced in the ddck including line of the configuration file which in this case would be the following:

```
PYTRNSYS$ demands\dhw\dhw_sfh_task44
```

This line will internally be changed to the following arguments of `changeDDckFile`. Since the first argument is repeated, an unchanged variation will be used. The third argument will result in a variation that builds the dck file based on the line changed to:

```
PYTRNSYS$ demands\dhw\dhw_mfh
```

There is no restriction to the substrings used. It is also possible to write:

```
changeDDckFile dhw\dhw_sfh_task44 dhw\dhw_sfh_task44 dhw\dhw_mfh
```

That way ddck files that are located in other folders could be used. The file name of the changed ddck file will be used in the name of the variation's subfolder and will also be saved to the results json-file.

Use the scaling functionality

Now that we changed the demand profiles of the simulation, we will end up with very different solar fractions for the two cases since a solar collector field that is designed for a single family home will be much too small for a multi-family building. In reality, it is a standard procedure to size the collector field relative two the expected demand.

In order to define relative system dimensioning, pytrnsys offers to possibility to read in the results file of an earlier simulation run and to use the values as a scaling parameter. In our case this requires that we pre-run the simulation in order to find the exact domestic hot water demand of the two different profiles. We can do this by running a configuration file that consists of no other variations but the *changeDDckFiles* line defined in the previous chapter:

```
changeDDckFile dhw_sfh_task44 dhw_sfh_task44 dhw_mfh
```

This should result in a simulation folder with the following subfolders:

```
solar_dhw (simulation base folder)
+-- SFH_DHW-dhw_mfh (simulation sub folder)
+-- SFH_DHW-dhw_sfh_task44 (simulation sub folder)
```

Before we are able to process we should make sure that we add the simulation result that we would like to use for scaling to the results file. In our case this is the yearly sum of the monthly integrated and printed values of **P_dhw_kW** that are by default available in the processing as **P_dhw_kW_Tot**. In the results file definition line we add:

```
stringArray results  "Pdhw_kW_Tot" "**" "**"
```

We now have our simulation results ready to be used in the scaling. The scaling can be activated by setting the *scaling* parameter in the run configuration file from **"False"** to **"toDemand"**. We then have to tell pytrnsys where it can find the scaling values. This is done by adding the following line:

```
string scalingReference "absolutePathToYourBaseResltsFile\SFH_DHW-dhw_sfh_task44-
↪results.json"
```

The argument of the parameter *scalingReference* should be the results json-file of the simulation that corresponds to the first argument of the `changeDDckFiles` line. For each ddck-variation defined in `changeDDckFiles` Pytrnsys will take the file names and do the same substring replacement in the path in `scalingReference`. When the folder with the scaling values is onmodified, pytrnsys should be able to find the correct values for each variation.

Finally, we should also tell pytrnsys which value in the results file it should use. We can do this by adding the following line:

```
string scalingVariable "Pdhw_kW_Tot/1000"
```

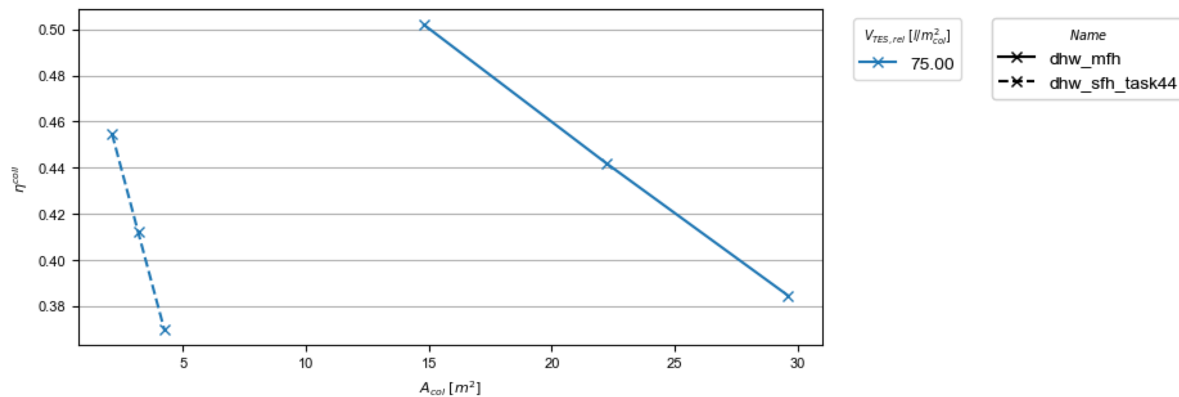
As you can see we can also add arithmetic operations to the value. As an example, here the value is converted from kW to MW.

We are now ready to define our parametric study using relative sizing of parameters. As soon as the scaling is set to **"toDemand"**, pytrnsys will always multiply the values given in the `variation` statement with the scaling variable.

So we can now size our collector area relative to the domestic hot water demand. A realistic sizing would be to have about 1.5 m^2MWh so we add slight variations as:

```
variation Ac AcollAp 1 1.5 2
```

This will finally result in a more comparable results for the solar fraction:



Run pytrnsys with an external deck file

Pytrnsys can also be used if you want to use its functionality on a full external dck file of your TRNSYS project that you have exported from the TRNSYS Studio or have created in your own way. To do this simply use this file as a single entry in the ddck section of the run configuration file:

```
string LOCAL$ "pathToYourDckFile"
LOCAL$ yourDckFile
```

Create your own ddck files

You already learned how to replace a ddck file with another one that is available in the ddck repository. Pytrnsys also allows you to create your own custom ddck files and include them into your project. In this chapter, we will go through the process of creating and including a new domestic hot water profile ddck that we can use in the simulation of the solar domestic hot water system.

If you executed the **pytrnsys-load** command you have your own local pytrnsys ddck repository that you are free to change. It is however recommended to save your own ddck files in a different folder that is under version control by GIT. That way, you can keep track of your work and savely overwrite the pytrnsys_examples when an update of the base repository is released. We also recommend to have your own repository in the same structure as the ddck repository.

In order to do so, in the folder or GIT repository of your choice, create a subfolder called demands that contains another subfolder called dhw. Inside this folder we create the new ddck that contains our custom domestic hot water reader. To have the right ddck structure you can for example copy the file dhw_mfh.ddck from the pytrnsys_ddck repository. Now you can perform any changes that you like, for example exchange the file that is used and adapt the TRNSYS type 9 accordingly.

After you created your new ddck file you can add your custom ddck repository to the ddck paths in the run config file and add replace the the domestic hot water line:

```
CUSTOMREPOSITORY$ demands\dhw\dhw_your_file
```

Get access to the pytrnsys GUI

Pytrnsys is still under development by the SPF Institute for Solar Technology. Therefore, up to this point the pytrnsys GUI is not available for the public. If you would like to use pytrnsys to create you own new system hydraulics please contact dani.carbonell@spf.ch.

Or directly read through all the options of the configuration files and play around by yourself.

Configuration files

Pytrnsys runs and processes TRNSYS simulations based on configuration files. The general idea behind this is to provide a fast and easily accessible way to define, run and analyse both single simulations as well as parametric studies. There are distinct configuration files for running and processing. Both are described in the subsequent sections but follow the same syntax and format.

The configuration file does not require a header. It should contain different keyword commands on single lines. Comments start with `#` characters. End of line comments are possbile.

The config file supports the following basic types:

bool	with True/False as possible value
int	with any interger as possible value
string	with any string as possible value
stringArray	array of strings

Parameters that are used to specifiy the run can be defined by:

```
keyword parameter_name value
```

In the case of an integer this would for example be:

```
int reduceCpu  4
```

Note: The `string` and `stringArray` always have to be specified with parentheses.

1.2 Running simulations

1.2.1 Ddck files

The core of the run configuration file is the ddck section. In this part of the configuration file, the different modular ddck files that should be used in the simulation are specified. Pytrnsys offers its own ddck repository in the separate package pytrnsys_ddck that is installed together with the main package and used in the example projects of pytrnsys_examples. In the ddck section of the config file, the different ddcks that should be merged to the simulation's main dck file are specified according to the following syntax:

```
ROOTPATH1$ pathtoddck1\ddck1
ROOTPATH1$ pathtoddck2\ddck2
ROOTPATH2$ pathtoddck3\ddck3
```

The root of the ddck repositories used has to be defined elsewhere in the configuration file:

```
string ROOTPATH1$ "pathToTheRepository1Root"
string ROOTPATH2$ "pathToTheRepository2Root"
```

An example can be found in the example section below. The path to the repository root can be either absolute or relative. If a relative path is detected, pytrnsys will interpret it as relative to the configuration file location.

1.2.2 Parameter variation

A second core feature of pytrnsys is activated in the run configuration file in a parameter variation section. Pytrnsys allows either to modify TRNSYS simulation parameters in the configuration file statically or with variations that result in parametric runs. A static parameter change that can affect TRNSYS variables that are defined in EQUATIONS or in CONSTANTS blocks of the dck file are initiated by:

```
deck trnsysVariableName value
```

A parametric study is defined by:

```
variation trnsysVariableName value1 value2 value3 ...
```

Both keywords can be used multiple times. If multiple variations are used, they are combined depending on the parameter `combineAllCases`. If this parameter is set to True all variations are combined pairwise. So if n values are given for variation 1 and m values are in variation 2 the total amount of simulations executed will be (m x n). If `combineAllCases` is set to False, the amount of values of all variations has to be equal and they are combined according to their order.

In addition to a single equation or constant line in the dck, pytrnsys offers the possibility to loop through different ddck files during a parametric study. A parametric study on ddck files can be defined by:

```
changeDDckFile originalDdck ddckVariation1 ddckVariation2 ddckVariation3 ...
```

1.2.3 Parameters

There are different additional parameters that define the simulation runs. The ones that have no default values are mandatory.

Generic

`ignoreOnlinePlotter` (bool, default False) If set to True, the TRNSYS online plotters are commented out in all the dck-files. No online plotters are shown during the simulation run. The TRNSYS progress bar window is still displayed.

`removePopUpWindow` (bool, default False) Online plotters as well as the progress bar window are suppressed during the simulations. (TRNSYS hidden mode)

`checkDeck` (bool, default True) If set to True, during merging the ddck-files, the specified and given amount of Equations and Parameters in each block are checked for inconsistencies.

`parseFileCreated` (bool, default True) Saves the parsed dck-file that can be used to locate the line where `checkDeck` found errors.

`runCases` (bool, default True) If set to False, the dck-files are created and saved in the normal structure but not executed.

`reduceCpu` (int, default 0) Number of CPUs that are not used in the parallel simulation runs.

outputLevel (string, default "INFO") Output message level according to the logging package. (Options are "DEBUG", "INFO", "WARNING", "ERROR", and "CRITICAL".)

Automatic Work Bool

doAutoUnitNumbering (bool, default True) If set to True, the units of the merged dck-file are renumbered to avoid duplicates. This parameter should usually be set to the default True.

generateUnitTypesUsed (bool, default True) If set to True, a file called `UnitType.info` containing the TRNSYS-Type numbers used is saved in the main run-folder.

addAutomaticEnergyBalance (bool, default True) If set to True, an automatic energy balance printer is created in the dck file. For more information see *Default plotting for TRNSYS results*.

Paths

trnsysExePath (string, default "environmentalVariable") Path to the `TRNExe.exe` of the TRNSYS installation. If not set, pytrnsys tries to find the path in the system environmental variable "TRNSYS_EXE".

pathBaseSimulations (string) If specified, the location of the simulation run is changed to the given path. It overrules the normal behavior of executing the simulations in the command line working directory.

addResultsFolder (string or False, default False) If specified as a string, a new folder for the simulations is created with this name.

Scaling

scaling (("False","toDemand"), default False) If set to "toDemand" the parameter scaling functionality is activated. Please refer to *scaling tutorial* for more details.

scalingReference (string) Path to the scaling results. Please refer to *scaling tutorial* for more details.

scalingVariable (string) Variable that is taken from the results json file for scaling. Please refer to *scaling tutorial* for more details.

nameRef (string) Base name of the dck-file created. Default base name is "pytrnsysRun".

runType (("runFromConfig", "runFromCases", "runFromFolder"), default "runFromConfig") "runFromCases" and "runFromFolder" offer some advanced option for custom simulation runs.

1.2.4 Example

Here is an example of a run configuration file. It is taken from the example project solar_dhw (`run_solar_dhw.config`):

```
######### Generic ########################
bool ignoreOnlinePlotter  True
int reduceCpu  4
bool parseFileCreated True
bool runCases True
bool checkDeck True

############# AUTOMATIC WORK BOOL##############################

bool doAutoUnitNumbering True
```

(continues on next page)

(continued from previous page)

```
bool generateUnitTypesUsed True
bool addAutomaticEnergyBalance True

############PATHS################################

string trnsysExePath "C:\Trnsys17\Exe\TRNExe.exe"
string addResultsFolder "solar_dhw"
string PYTRNSYS$ "..\..\pytrnsys_ddck\"
string LOCAL$ ".\"

###############SCALING########################

string scaling "False" #"toDemand"
string nameRef "SFH_DHW"
string runType "runFromConfig"

############PARAMETRIC VARIATIONS##################

bool combineAllCases True
variation Ac AcollAp 2 3 4 6 8 10
variation VTes volPerM2Col 75 100

############FIXED CHANGED IN DDCK##################

deck START 0    # 0 is midnight new year
deck STOP  8760 #
deck sizeAux 3

############USED DDCKs##################

PYTRNSYS$ generic\head
PYTRNSYS$ demands\dhw\dhw_sfh_task44
PYTRNSYS$ weather\weather_data_base
PYTRNSYS$ weather\SIA\normal\CitySMA_dryN
PYTRNSYS$ solar_collector\type1\database\type1_constants_CobraAK2_8V
PYTRNSYS$ solar_collector\type1\type1
LOCAL$ solar_dhw_control
LOCAL$ solar_dhw_storage1
LOCAL$ solar_dhw_hydraulic
LOCAL$ solar_dhw_control_plotter
PYTRNSYS$ generic\end
```

1.3 Processing data

Pytrnsys processing modules automatically read in the simulation results of the pytrnsys runs and - by default - calculates energy balances as well as collects some of the most important information of the simulation like iteration problems and system performance factors in a results pdf file. The process configuration file allows to configure the processing. In addition further calculations with the simulation results and additional plots can be defined.

Besides that, the processing functionality can be used on generic data that do not originate from TRNSYS simulations.

1.3.1 Parameters

There are different general parameters in the processing configuration file that allow to change different settings

Paths

`latexNames` (string) Path to the latexNames json-file. Can either be an absolute path or a path relative to the configuration file. If not specified, the default latexName json-File of pytrnsys is used.

`pathBase` (string) Path of the folder to be processed. If not specified, the current working directory is used instead.

`inkscape` (string) Path of the Inkspace executable. Required for using *plotEmf <ref-plotEmf>*.

Generic

`typeOfProcess` (string, default 'completeFolder')

This parameter defines how data is processed. There are various possible arguments:

- 'casesDefined': TBD
- 'citiesFolder': Data sets are defined by subfolders in `pathBase`, which are named according to the `cities` parameter.
- 'completeFolder': Identifies data sets through lst files in `pathBase` and its subfolders.
- 'config': TBD
- 'json': Identifies data sets through json files in `pathBase` and its subfolders. Can also be used on `-results.json` files.

`processParallel` (bool, default True) If set to True, pytrnsys will process the simulation sub-folders in parallel. The amount of parallel processes will be the total amount of CPUs minus `reduceCpu`.

`processQvsT` (bool, default True) Flag to disable the QvsT processing. Since this is computationally very expensive it can be useful to disable the QvsT plots if not needed.

`cleanModeLatex` (bool, default False) If set to True, all plot files will be deleted after they are collected in the results pdf-file. If set to False, they will remain in the simulation subfolder.

`forceProcess` (bool, default True) If set to False, allready processed folders will not be processed again.

`plotStyle` (string, default 'line') If set to 'dot', dots will be used instead of lines for the respective plots.

`setPrintDataForGle` (bool, default True) Print the Data of the plots for further use in GLE plots.

`figureFormat` (string, default 'pdf') Format in which the plots of the processing will be saved. All formats that are supported by *matplotlib.pyplot.savefig <https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.savefig.html>* are supported

`plotEmf` (bool, default False) If set to true, all plots will be exported in the emf format. Requires Inkscape.

Time selection

Pytrnsys is designed to process one full year. If more than a year is simulated, the months that are used for processing have to be specified.

`yearReadedInMonthlyFile` (int, default -1) Year of the simulation that is used for processing. 0 is the first year, 1 the second year and so on. If the value is set to -1 pytrnsys will use the last 12 months of the simulation for processing.

`firstMonth` (["January", "February", "Mach", . . . , "December"], default "January") Month in the chosen year where the 12-month processing period begins. If the value is e.g. "November" November to October will be analysed.

1.3.2 Processing TRNSYS data

During processing pytrnsys reads in the following values automatically:

1. All parameter and equation variables that are statically defined in the dck.file. Pytrnsys recursively detects static variables by checking for any type outputs in the variables involved.
2. All monthly printer values of the simulation. The pytrnsys ddcks save all printer files in the temp folder inside the directory where the simulation is executed. If custom printers are defined, the same location is required.
3. All hourly printer values of the simulation.

All values can be adressed in the config file by their name in the header of the trnsys printer file. It is recommended to duplicate the internal TRNSYS name in the header of the printer.

Note: While TRNSYS is not case sensitive, Python is. So be careful about upper and lower cases during post processing. If the string in the configuration file does not match the header of the printer file or the TRNSYS name of the static parameter in the dck-file, pytrnsys will not be able to find the value and throw a key-error.

By default, pytrnsys also calculates the following values:

4. Total yearly sum of the monthly printed values. The yearly sum of a monthly printed variable with the name `foo` can be accessed for further processing by `foo_Tot`.
5. The maximum hourly value of an hourly printed file. The hourly maximum of an hourly printed variable with the name `bar` can be accessed by `bar_Max`.

1.3.3 Processing generic data

To process generic data, add the following expression to the header of your configuration file:

```
bool isTrnsys False
```

You then need to specify how pytrnsys should access your data. One way is to identify a data set with a json file that includes the parameters of the data set in the format of a python dictionary. When you have such a json in each data set folder, you should use:

```
string typeOfProcess "json"
```

Furthermore, you need to specify the folder (here, e.g.: `dataFolder`) containing your data sets with:

```
string pathBase "..\dataFolder"
```

The program will look for json-files in `dataFolder` and on each subfolder level. It will then load csv-files, which are in the same folders as the json-files it found. At the moment it can load hourly, daily, and monthly data. The names of the respective csv-files need to contain the keywords `_Stunden`, `_Tage`, or `_Monat`.

1.3.4 Calculations

In the processing-configuration file, the user can specify custom calculations based on the TRNSYS results that were read in and the values that are calculated by default. The type of each equation has to be defined by a key word that tells pytrnsys what values should be used. This is necessary since some variables could be both in an hourly as well as a monthly printer. The following calculation keywords are available:

`calc` Calculates a new scalar value out of other scalar values such as static TRNSYS parameters or yearly sums or hourly maxima.

calcMonthly Calculates new monthly values (array with length 12) out of other monthly values or scalar values.

calcDaily Calculates new daily values (array with length 365) out of other hourly values or scalar values.

calcHourly Calculates new hourly values (array with length 8760) out of other hourly values or scalar values.

calcMonthlyFromHourly Calculates new monthly values (array with length 12) out of hourly values or scalar values.

A calculations section could be of the following structure. A full working example can be found in the example below:

```
calc alpha = foo_Tot/bar_Max
calcMonthly = foo/foo_Tot*1000
calcHourly = (bar+100)**2
```

acrossSetsCalc Can execute calculations across data sets with variables from the results json-files. Equations are provided as arguments and indicated by a = and conditions by : and stated as `key:value`. A function call (optional arguments in square brackets) then looks like:

```
stringArray acrossSetsCalc "x-variable" "y-variable" "calculation variable"
↪"equation 1" ["equation 2"] ... ["key 1:value 1"] ["key 2:value 2"] ...
```

Here `calculation variable` is a key of the results json-files and specifies what arguments can go into an equation. An example for an equation looks like:

```
nameOfValueToBeCalculated=(foo+bar)*100
```

where `foo` and `bar` are valid values of the `calculation variable`. The program will take different data sets with the same `x`- and `y`- but different `calculation variable`-values and execute the equation for these. Hence, you need to ensure that these combination exist in your data sets. A csv with the calculated results will be generated.

1.3.5 Results file

For further custom processing of the simulation results, required scalar and monthly values can be saved to a results json-file.

results Determines which variables should be stored in a dedicated json-file for each data set:

```
stringArray results "variable 1" "variable 2" ...
```

pathInfoToJson Scans the paths of the generated `-results.json` files for keywords and adds them as the respective `parameter name` in said json-files, and adds an empty string, if it doesn't find any of the keys in the respective path:

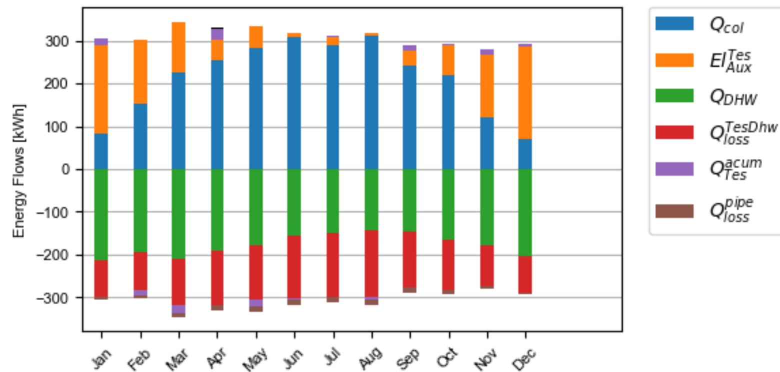
```
stringArray pathInfoToJson "parameter name" "key 1" "key 2" ...
```

1.3.6 Plotting

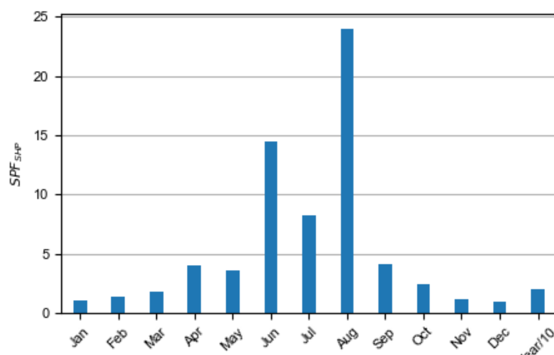
Default plotting for TRNSYS results

By default the processing creates a pdf with the following content:

1. A table displaying the total simulation time and the number of iteration errors.
2. A table with the monthly heat balance. The values are also shown in a plot, in the case of the solar domestic hot water example system this looks like the following:



3. A electricity balance similar to the heat balance.
4. The system seasonal performance factor both in a table and a plot. Again, the SPF plot of the solar domestic hot water system looks like:

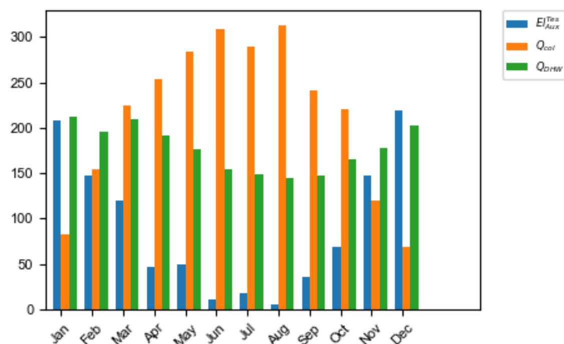


Custom plotting and printing

The user can add additional monthly plots to the processing of a single simulation run by the use of the following parameters. The custom defined plots will automatically be added to the result pdf-file:

Note: If an argument in the code excerpts below is set in square brackets, it is optional.

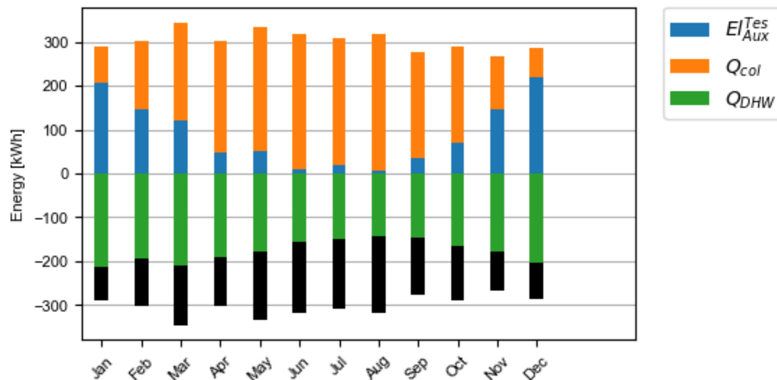
`monthlyBars` Plots a monthly bar plot that shows all variables grouped side by side.



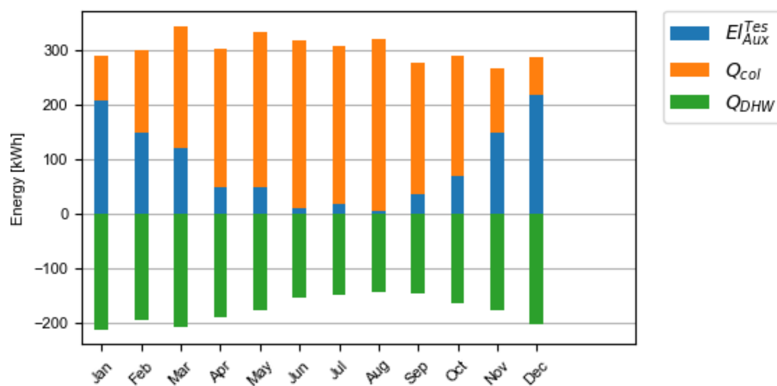
monthlyBalance Custom monthly balance. The sign of the values can be inverted by adding a - in front of the variable name. If positive and negative values don't add up to zero, the imbalance is shown as black bars. When adding the optional `style:relative` the bars will be shown as values relative to the positive sum of the monthly energy values:

```
stringArray monthlyBalance "pdf name" ["style:relative"] "variable 1" "variable 2
↪" ...
```

In the solar domestic hot water example system this can be demonstrated by plotting the two system inputs Q_{col} and El_{Aux}^{Tes} and the usable output of the domestic hot water demand. The imbalance in this case are the overall losses of the system.



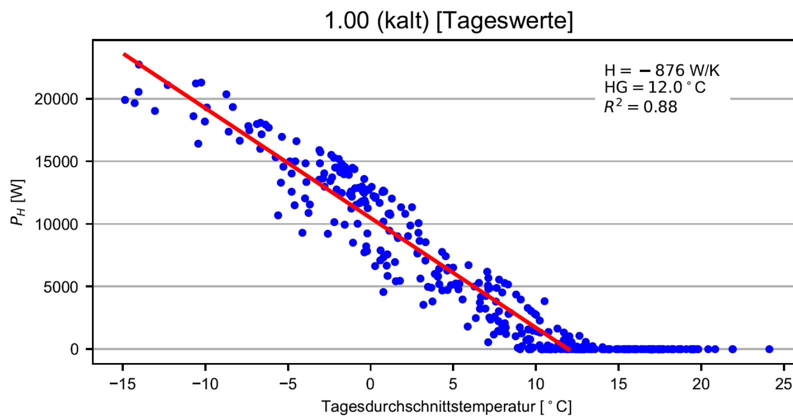
monthlyStackedBar Similar to the `monthlyBalance` but without showing the imbalance.



fitHeatingLimit This function was created to plot and fit heating power values against average daily temperatures. In principle it can plot any daily or hourly data against average daily temperature. The time resolution of the heating power data (or its equivalent) needs to be specified as `daily` or `hourly` (`heatingDataTimeStep`) when calling the function:

```
string fitHeatingLimit "y-variable" "heatingDataTimeStep"
```

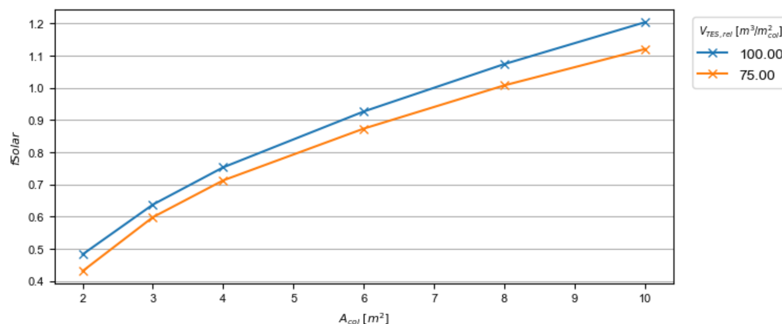
A linear fit is done for `daily`, while the data only are shown for `hourly`.



Note: All variables used in `comparePlot`, `comparePlotConditional`, and `acrossSetsCalculationsPlot` need to be saved in the `-results.json` files.

`comparePlot` When processing parametric runs, scalar results of the simulations can be visualized in comparison plots. The first variable of the string array is shown on the x-axis. The second variable is shown on the y-axis. The third is represented as different lines, and the fourth as different marker styles:

```
stringArray comparePlot "x-variable" "y-variable" ["series 1 variable"] ["series 2 variable"]
```



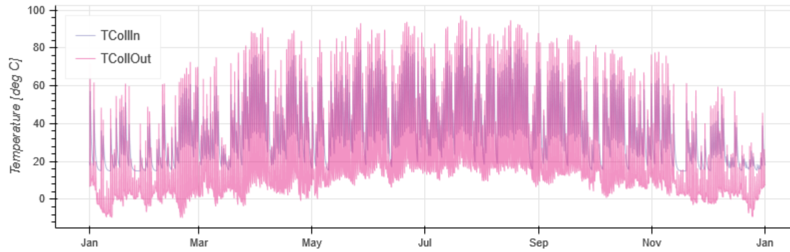
`comparePlotConditional` Same as `comparePlot`, but with the additional feature of imposing conditions on the data that is supposed to be plotted. For a `key` in the results json, a condition is indicated by a `:` and stated as `key:value`:

```
stringArray comparePlotConditional "x-variable" "y-variable" ["series 1 variable"] ["series 2 variable"] ["key 1:value 1"] ["key 2:value 2"] ...
```

`acrossSetsCalculationsPlot` Has the same basic functionality as `acrossSetsCalc`, but can plot the results of equations provided:

```
stringArray plotCalculationsAcrossSets "x-variable" "y-variable" "calculation variable" "equation 1" ["equation 2"] ... ["key 1:value 1"] ["key 2:value 2"] ...
```

`plotHourly` Hourly printed values can be displayed in a interactable html-plot that is created using the bokeh plotting library.



plotHourlyQvsT Adds a cumulative plot that contains a line for each heat temperature pair given in the string array. Used to show at what temperature levels the heat is released or consumed in different system components. Uses hourly printer files.

plotTimestepQvsT Adds a cumulative plot that contains a line for each heat temperature pair given in the string array. Used to show at what temperature levels the heat is released or consumed in different system componenets. Uses timestep printer files.

1.3.7 Example

The following processing-configuration file is part of the solar domestic hot water example system:

```
######### Generic ########################
bool processParallel False
bool processQvsT True
bool cleanModeLatex False
bool forceProcess  True
bool setPrintDataForGle True
bool printData True
bool saveImages True
int reduceCpu 1

######### Time selection ########################
int yearReadedInMonthlyFile -1
int firstMonthUsed 6     # 0=January 1=February 6=July 7=August

############# PATHS ##############################
string latexNames ".\latexNames.json"
string pathBase "C:\Daten\OngoingProject\pytrnsysTest\SolarDHW_newProfile"

############# CALCULATIONS ##############################

calcMonthly fSolarMonthly = Pcoll_kW/Pdhw_kW
calc fSolar = Pcoll_kW_Tot/Pdhw_kW_Tot

calcMonthly solarEffMonthly = PColl_kWm2/IT_Coll_kWm2
calc solarEff = PColl_kWm2_Tot/IT_Coll_kWm2_Tot

############# CUSTOM PLOTS ##############################
stringArray monthlyBars "elSysIn_Q_ElRot"  "qSysIn_Collector" "qSysOut_DhwDemand"
stringArray monthlyBars "solarEffMonthly"
stringArray monthlyBalance "elSysIn_Q_ElRot"  "qSysIn_Collector" "-qSysOut_DhwDemand"
stringArray monthlyStackedBar "elSysIn_Q_ElRot" "qSysIn_Collector" "-qSysOut_DhwDemand
↪"

stringArray plotHourly "Pcoll_kW" "Pdhw_kW" "TCollIn" "TCollOut"  # "effColl" #␣
↪values to be plotted (hourly)
```

(continues on next page)

(continued from previous page)

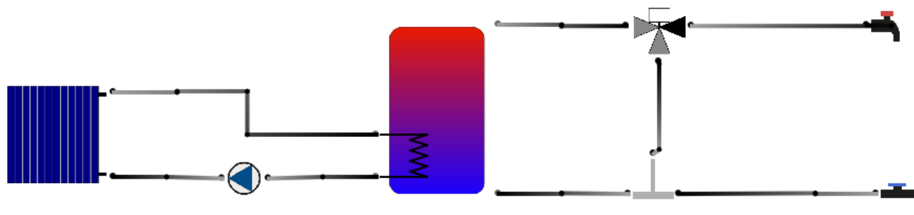
```
stringArray plotHourlyQvsT "Pdhw_kW"  "Tdhw" "Pcoll_kW" "TCollOut"

stringArray comparePlot "AcollAp" "fSolar" "volPerM2Col"
stringArray comparePlot "AcollAp" "fSolar" "volPerM2Col"
stringArray comparePlot "AcollAp" "Pdhw_kW_Tot" "volPerM2Col"


############# RESULTS FILES #############################
stringArray hourlyToCsv "CollectorPower" "IT_Coll_kWm2" "PColl_kWm2"
stringArray results  "AcollAp"  "Vol_Tes1"   "fSolar"  "volPerM2Col"  "Pdhw_kW_Tot" #␣
↪values to be printed to json
```

1.4 Example systems

At the moment, pytrnsys offers two example systems that show the capabilities of the framework. Other example systems will be added in the future. Follow the project on github to stay updated on new releases.

1.4.1 Solar domestic hot water system for single family house

This system is used as the demo system for the tutorial on this webpage. It has the following system hydraulics:

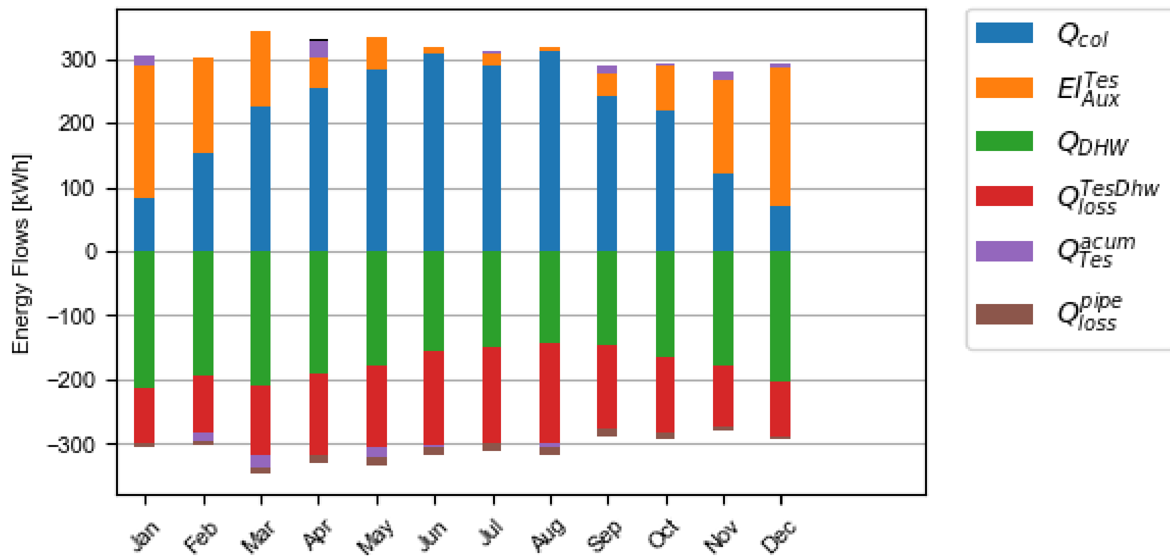


The system is build using the following ddck files:

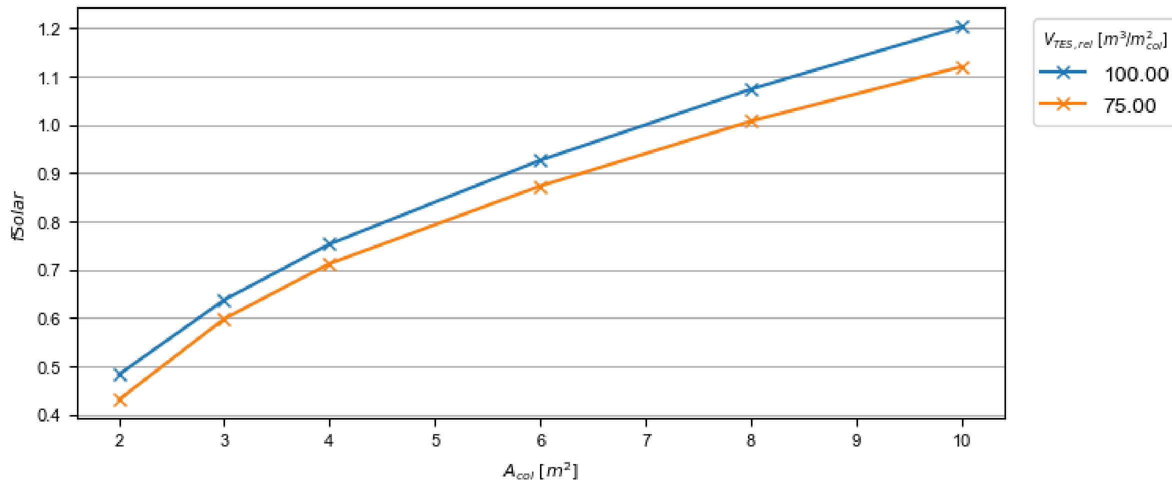
```
string PYTRNSYS$ "..\..\pytrnsys_ddck\"
string LOCAL$ ".\"

PYTRNSYS$ generic\head
PYTRNSYS$ demands\dhw\dhw_sfh_task44
PYTRNSYS$ weather\weather_data_base
PYTRNSYS$ weather\SIA\normal\CitySMA_dryN
PYTRNSYS$ solar_collector\type1\database\type1_constants_CobraAK2_8V
PYTRNSYS$ solar_collector\type1\type1
LOCAL$ solar_dhw_control
LOCAL$ solar_dhw_storage1
LOCAL$ solar_dhw_hydraulic
LOCAL$ solar_dhw_control_plotter
PYTRNSYS$ generic\end
```

It uses a Cobra AK 2.8V solar thermal collector. The collector area is used for a parametric study and varied in a range between 2 m^2 and 10 m^2. The size of the thermal storage is defined relatively to the collector. In the parametric study the two sizings 75 l/m^2 and 100 l/m^2 are used. The domestic hot water demand is taken from the IEA Task 44. The weather data is taken from the swiss standard reference year of Zurich.

The monthly balance of the system with 6 m^2 collector area and a 75 l/m^2 looks like this:



The comparison plot of the parametric study showing the solar fraction looks like:



1.4.2 PV system with battery

This system showcases the capability of pytrnsys to simulate PV systems. Since it does not contain any hydraulic components, no hydraulics file is necessary. The following ddcks are used:

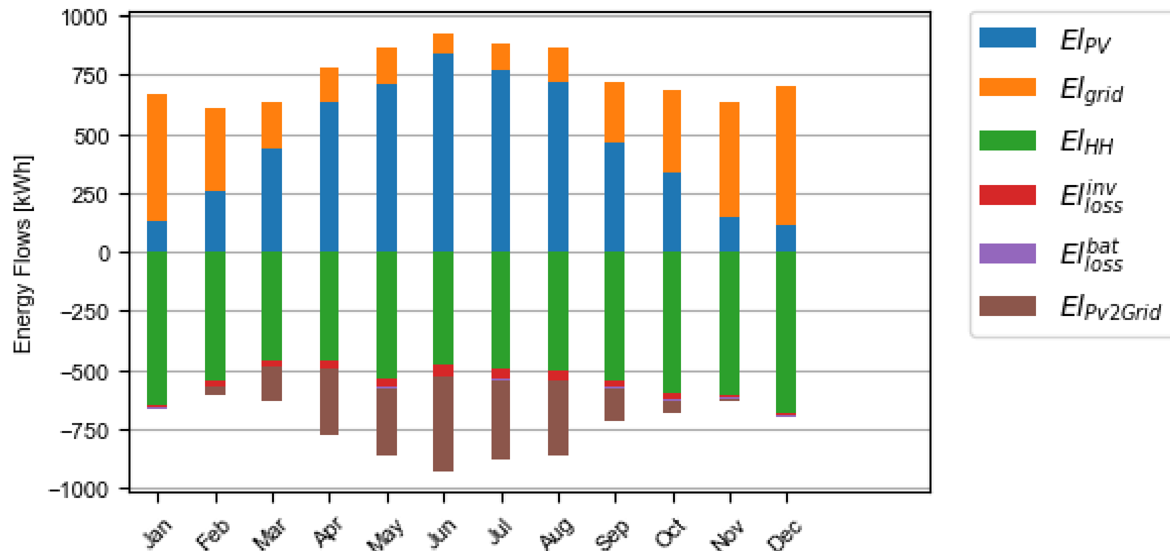
```
string PYTRNSYS$ "..\..\pytrnsys_ddck\"
PYTRNSYS$ generic\head
PYTRNSYS$ demands\electricity\elDemand
PYTRNSYS$ weather\weather_data_base
```

(continues on next page)

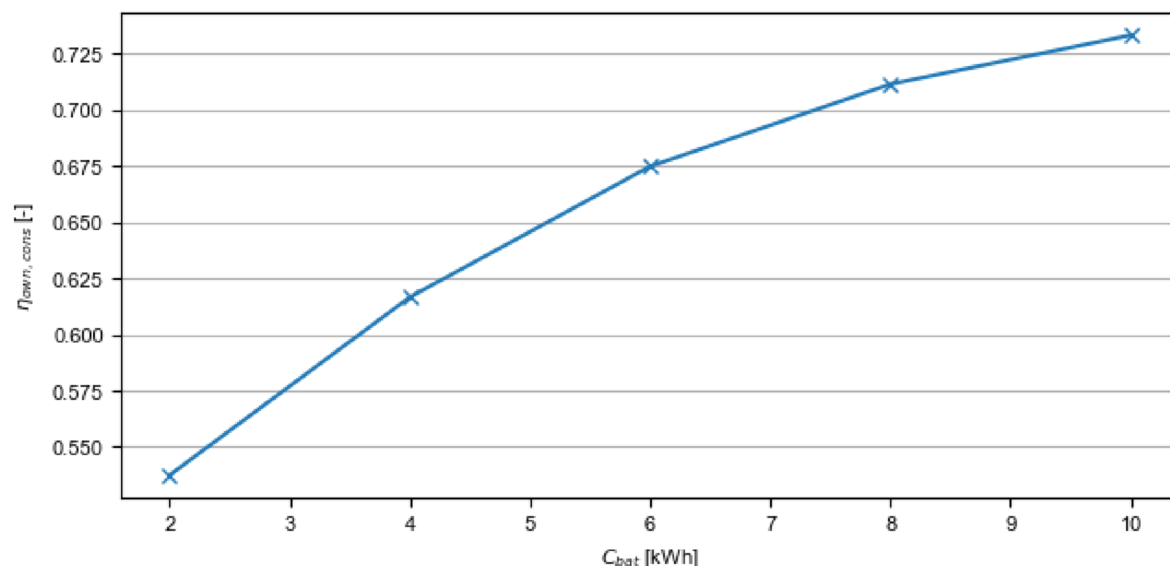
(continued from previous page)

```
PYTRNSYS$ weather\SIA\Normal\CitySMA_dryN
PYTRNSYS$ pv\type194\type194
PYTRNSYS$ pv\type194\database\sunskin_roof_module_eternit
PYTRNSYS$ pv\type194\database\fronius_symo_inverter
PYTRNSYS$ battery\basic_battery
PYTRNSYS$ generic\end
```

The system simulates a 5.7 kWp in roof PV installation with an household electricity demand of 6568 kWh. The automatically generated electrical energy balance for the system including a battery with a usable capacity of 6 kWh is:



A parametric study on the capacity of the used battery is defined for a range of the usable battery capacity of 2 kWh to 10 kWh. The resulting own consumption rate for the different battery capacities is



1.5 Ddck repository

Pytrnsys enhances system simulations by making use of component modularization. This is done by separating the dck file to single components of an energy system simulation, which are defined by ddck files. In a pytrnsys run, these ddck files will then be merged together in order to build the full system dck. Pytrnsys comes with a default repository of ddck files that is installed alongside the main package.

1.5.1 Repository content

The repository contains the following subfolders:

```
pytrnsys_ddck
+-- air_source_heat_pump
+-- battery
+-- boiler
+-- brine_source_heat_pump
+-- brine_source_heat_pump_two_hex
+-- building
+-- control
+-- demands
+-- dlls
+-- generic
+-- ground_heat_exchanger
+-- heat_exchanger
+-- building
+-- ice_storage
+-- pv
+-- solar_collector
+-- weather
+-- __init__.py
+-- example.ddck
+-- generalVariables.ddck
+-- NOMENCLATURE.txt
```

Each subfolder represents one component, that is supported by default in the pytrnsys GUI. A lot of the components - but not all of them - are used in the pytrnsys example systems. Each component subfolder has the following structure. Not all folders and files are present in all components:

```
pytrnsys_ddck
+-- component subfolder
    +-- typeX subfolder1
        +-- typeX.ddck
        +-- database
    +-- typeY subfolder2
        +-- typeY.ddck
        +-- database
    +-- non type related ddcks
    +-- profiles
```

Some components can be represented by different TRNSYS types. Usually a more basic one of the standard library and a advanced custom model that allow for detailed parametrization. Each TRNSYS type is represented by its own subfolder in the component folder. In the type subfolder, the main ddck of the type is stored. This ddck contains both the TRNSYS type parameter and input section, the type output section, as well as some equations for parameter and input scaling or output processing. The parametrization of the model is outsourced in the database folder where a specific version - for example parameters of a specific collector curve - can be specified.

1.5.2 Ddck file structure

The file sctructure of the ddck files is shown in the example.ddck file in the pytrnsys_ddck repository root. Each ddck should contain the following parts. Some parts can be empty if not used.

Header

In the header the name of the ddck as well as a contact persion, the creation date and some information about the last changes are specified. In addition, there is a section for general descriptions about the ddck:

```
*******************************
**BEGIN example.ddck
*******************************

*****************************************
** Contact person : author
** Creation date  : date
** Last changes   : date, name
*****************************************

***************************************************************************
** Description:
** Overall description of the ddck file
** TODO: Any improvements needed
***************************************************************************
```

Inputs from hydraulics solver

The pytrnsys ddcks are designed to be used in combination with the hydraulic solver type 935. In each timestep the hydraulic solver computes all mass flows and component input temperatures starting depending on the component output temperatures, the pump powers and the valve positions. Therefore, the mass flows and component input temperatures are outputs of the hydraulic solver that have to be connected to the component ddcks as inputs:

```
***********************************
** inputs from hydraulic solver
***********************************

** tIn from hydraulic
** mIn from hydraulic
```

Outputs to the hydraulic solver

Similar to the inputs, the output temperatures of the component should be stated here, such that they are easily accessible to be connected to the hydraulics file:

```
***********************************
** outputs to hydraulic solver
***********************************

** which outputs will be used to connect the hydraulic solver
** typically tOutType will be defined here to be used in the hydraulic ddck
```

Outputs to the energy balance

In the processing, pytrnsys automatically computes the systems heat and electricity energy balance. All variables that should be collected for the energy balance have to be specified in this section according to the right nomenclature:

```
****************************************************************************************
** outputs to energy balance in kWh and ABSOLUTE value
** Following this naming standard : qSysIn_name, qSysOut_name, elSysIn_name, elSysOut_
↪name
****************************************************************************************

** Add here those variables that will go into the overall energy balance of the system
** These values will be used to automatically generate the energy balance
```

Dependencies with other ddck files

In order to enhance modularization, dependencies with other ddcks should be kept minimal. Dependencies that cannot be avoided and are neither part of the component-database relation or the general variables should be declared and reassigned to an internally used variable in this part:

```
***********************************
** Dependencies with other ddck
***********************************

** Re-assing here the variables necessary from other types
** variableInternal = variableExternal
** Exception: those from general variables
```

Outputs to other ddck files

Variables that are designated to be used in other ddck files should be added here:

```
***********************************
** outputs to other ddck
***********************************

** Add here the outputs of the TYPE or TYPES that will be used in other types
** Exception: those for printers and so on dont need to be here.
```

Precalculations related to parameter scaling and pre-processing

Usually, in the declaration of a TRNSYS component, many parameters are calculated out of more general system variables. All calculations to determine the right parameters inputs for the type go here:

```
***********************************
** Begin CONSTANTS
***********************************
```

Type section

TRNSYS has its own syntax that calls the type dll files. This core part of the ddck goes here:

```
***********************************
** Begin TYPE
***********************************
```

Component printers

Each component should have a monthly as well as an hourly printer. This helps to simplify the setup and the processing of the simulation. In addition, an online plotter is a nice tool for the debugging of the system:

```
***********************************
** Monthly printer
***********************************

***********************************
** Hourly printer
***********************************

***********************************
** Online plotter
***********************************
```

1.5.3 Hydraulics files

The hydraulics file represents the systems hydraulics layout. Each pytrnsys example system except the pv battery system has its own hydraulic layout file. In order to create your own hydraulic files that represent the hydraulics of your choice you need access to the pytrnsys GUI. The hydraulics file are not part of the ddck repository. The hydraulic files of the example systems are located in the example system folder of **pytrnsys_examples**.

1.5.4 Examples

The following example shows the ddck file of the solar collector type 1 used in the solar domestic hot water system:

```
*******************************
**BEGIN Type1.ddck
*******************************

*****************************************
** Contact person : Dani Carbonell
** Creation date  : 10.01.2010
** Last changes   : 03.2020 Jeremias Schmidli
*****************************************

***************************************************************************
** Description:
** Collector model using efficiency curve efficiency
***************************************************************************

***********************************
** inputs from hydraulic solver
***********************************

EQUATIONS 2
```

(continues on next page)

(continued from previous page)

```
TCollIn = TPiColIn
MfrColl = ABS(MfrPiColIn)

***********************************
** outputs to hydraulic solver
***********************************

EQUATIONS 1
TCollOut = [28,1]

***********************************
** outputs to other ddck
***********************************

*******************************************************************************************
** outputs to energy balance in kWh and ABSOLUTE value
** Following this naming standard : qSysIn_name, qSysOut_name, elSysIn_name, elSysOut_
↪name
*******************************************************************************************

EQUATIONS 1
qSysIn_Collector = PColl_kW

***********************************
** Dependencies with other ddck
***********************************

EQUATIONS 1
pumpColOn = puColOn

CONSTANTS 2
C_tilt = slopeSurfUser_1  ! @dependencyDdck Collector tilt angle / slope [°]
C_azim = aziSurfUSer_1    ! @dependencyDdck Collector azimuth  (0:s, 90:w, 270: e) [°]

EQUATIONS 4
**surface-8
IT_Coll_kJhm2 = IT_surfUser_1  ! Incident total radiation on collector plane, kJ/hm2
IB_Coll_kJhm2 = IB_surfUser_1  ! incident beam radiation on collector plane, kJ/hm2
ID_Coll_kJhm2 = ID_surfUser_1  ! diffuse and ground reflected irradiance on collector␣
↪tilt
AI_Coll = AI_surfUser_1  ! incident angle on collector plane, °

EQUATIONS 5
IT_Coll_kW = IT_Coll_kJhm2/3600     ! Incident total radiation on collector plane, kW/
↪m2
IB_Coll_kW = IB_Coll_kJhm2/3600     ! incident beam radiation on collector plane, kW/
↪m2
ID_Coll_kW = ID_Coll_kJhm2/3600     ! diffuse and ground reflected irradiance on␣
↪collector tilt (kW/m2)
IT_Coll_Wm2 = IT_surfUser_1/3.6
IT_Coll_kWm2 = IT_surfUser_1/3600

***********************************
** Begin CONSTANTS
***********************************

CONSTANTS 3
```

(continues on next page)

(continued from previous page)

```
MfrCPriSpec = 15  ! Coll. Prim. loop spec. mass flow [kg/hm2]
AcollAp=5         ! Collector area
MfrCPriNom = MfrCPriSpec*AcollAp !

***********************************
** Begin TYPE
***********************************

UNIT 28 TYPE 1
PARAMETERS 11
nSeries       ! number in series
AcollAp       ! collector area
cpBri          ! fluid specific heat kj(kgK
efficiencyMode ! efficiency mode
testedMfr      ! tested flow rate kg/(hm2)
Eta0          ! intercept efficiency
a1            ! efficiency slope kJ/hm^2K
a2            ! efficiency curvature kJ/hm^2K^2
2             ! optical mode
FirstOrderIAM  ! 1st order IAM
SecondOrderIAM ! 2nd order IAM
INPUTS 9
TCollIn
MfrColl
Tamb
IT_Coll_kJhm2
IT_H
ID_Coll_kJhm2
0,0
AI_Coll !Flo check ! JS: This was defined wrong before (C_azim, even though it is␣
↪incident angle input). Now it should be correct.
C_tilt !Flo check  ! JS: This should be correct
*** INITIAL INPUT VALUES
20 0 10 0 0 0 GroundReflectance 45 0

EQUATIONS 4
**MfrCout = [700,2]
Pcoll = [28,3] !kJ/h
PColl_kW = Pcoll/3600
PColl_kWm2 = PColl_kW/(AcollAp+1e-30)
PColl_Wm2  = PColl_kWm2*1000


***********************************
** Monthly printer
***********************************

CONSTANTS 1
unitPrintSol = 31

ASSIGN temp\SOLAR_MO.Prt unitPrintSol

UNIT 32 TYPE 46
PARAMETERS 6
unitPrintSol ! 1: Logical unit number, -
-1           ! 2: Logical unit for monthly summaries, -
1            ! 3: Relative or absolute start time. 0: print at time intervals␣
↪relative to the simulation start time. 1: print at absolute time intervals. No␣
↪effect for monthly integrations
```

(continues on next page)

(continued from previous page)

```
-1            ! 4: Printing & integrating interval, h. -1 for monthly integration
1             ! 5: Number of inputs to avoid integration, -
1             ! 6: Output number to avoid integration
INPUTS 4
Time  Pcoll_kW  PColl_kWm2  IT_Coll_kWm2
**
Time  Pcoll_kW  PColl_kWm2  IT_Coll_kWm2

***********************************
** Hourly printer
***********************************

CONSTANTS 1
unitHourlyCol = 33

ASSIGN    temp\SOLAR_HR.Prt    unitHourlyCol

UNIT 34 TYPE 46     ! Printegrator Monthly Values for System
PARAMETERS 7
unitHourlyCol ! 1: Logical unit number, -
-1             ! 2: Logical unit for monthly summaries, -
1              ! 3: Relative or absolute start time. 0: print at time intervals␣
↪relative to the simulation start time. 1: print at absolute time intervals. No␣
↪effect for monthly integrations
1              ! 4: Printing & integrating interval, h. -1 for monthly integration
2              ! 5: Number of inputs to avoid integration, -
4              ! 6: Output number to avoid integration
5              ! 7: Output number to avoid integration
INPUTS 6
Pcoll_kW  PColl_kWm2  IT_Coll_kWm2 TCollOut TCollIn MfrColl
**
Pcoll_kW  PColl_kWm2  IT_Coll_kWm2 TCollOut TCollIn MfrColl
```

A specific parametrization can be added by using a ddck from the database for example the type1_CONSTANTS_cOBRAak2_8v.ddck:

```
******************************
**BEGIN Type1_Constants_CobraAK2_8V.ddck
*******************************

*****************************************
** Solar Thermal Data for covered collector.
** Very well performing collector Cobra AK 2.8V
** Version : v0.0
** Last Changes: Jeremias Schmidli
** Date: 10.03.2020
******************************************

CONSTANTS 11

Eta0= 0.857      ! Eta0 (a0) of collector (zero heat loss efficiency)
a1 = 4.16*3.6     ! linear heat loss coefficient of collector [kJ/hm^2K] ![W/m2K]*3.6
a2 = 0.0089*3.6    ! quadratic heat loss coefficient of collector [kJ/hm^2K^2] ![W/
↪m2K2]*3.6

AbsorberArea = 2.435 !m2
```

(continues on next page)

(continued from previous page)

```
TotArea = 2.768 !m2

nSeries = 1
efficiencyMode = 1
testedMfr = 200/AbsorberArea !l/hm2

GroundReflectance = 0.2

FirstOrderIAM = 0.108
SecondOrderIAM = 0
*******************************
**END Type1_Constants_Test.ddck
*******************************
```

1.6 Developer's guide

1.6.1 Get the source code from github

Pytrnsys is available as open source code under the MIT license. If you want to run pytrnsys from source, it is recommended to use a designated virtual environment. Make sure that in your python environment the following dependencies are installed:

- numpy
- scipy
- pandas
- matplotlib
- seaborn
- bokeh

If you would like to contribute to pytrnsys and the documentation, you also need the following python packages:

- sphinx
- recommonmark
- sphinx_rtd_theme

You can fork the source code at (toBeFilledInWhenPublic) or by the following command:

```
git clone (toBeFilledInWhenPublic)
```

Once you have you local copy of the pytrnsys source code you can add the pytrnsys package subfolder to your Python project source paths. Once you have the pytrnsys package installed from source and you should be able to execute:

```
import pytrnsys
```

in your Python environment.

1.6.2 Run the example systems from source

You can run pytrnsys with the following minimal code example

```
import pytrnsys.rsim.runParallelTrnsys as runTrnsys

pathConfig  = "pathToTheConfigFile"
configFile = "run_solar_dhw.config"
runTool = runTrnsys.RunParallelTrnsys(pathConfig,configFile=configFile)
```

The “pathToTHeConfigFile” should be replaced with the full path to the folder examples/solar_dhw in your local repository. This script replaces the **pytrnsys-run** command and starts a pytrnsys run with the given configuration file. Similarly the processing can be started with the following minimal example

```
from pytrnsys.psim import processParallelTrnsys as pParallelTrnsys

pathConfig = "pathToTheConfigFile"
configFile = "process_solar_dhw.config"
tool = pParallelTrnsys.ProcessParallelTrnsys()
tool.readConfig(pathConfig,configFile)
tool.process()
```

If you would like to continue to modify the config file as described in the *tutorial*, make a local copy of the example folders such that tha changes will not be tracked by GIT.

1.6.3 Create your own processing class

Pytrnsys provides a large number of possibilities to process and plot results with the processing configuration file. But sometimes this is not enough! If you would like to add your own Python code to the processing you can create your own processing class that inherits from the class pytrnsys.psim.processTrnsysDf.

```
import pytrnsys.psim.processTrnsysDf as processTrnsys

class MyProcess(processTrnsys.ProcessTrnsysDf):

    def __init__(self,pathFolder, fileName):
        processTrnsys.ProcessTrnsysDf.__init__(self,pathFolder, fileName)

    #define your own functions
    def myCalculation()
        myValue=foo+bar

    # overwrite this function and fill it with your content
    def customCalculations()
        self.myCalculation
```

This class can then be saved in your preferred location. In order to use the custom processing class the pytrnsys.rsim.runParallelTrnsys class has to be modified such that it instantiates the new class. This can be done by replacing the run script in the following way.

```
from pytrnsys.psim import processParallelTrnsys as pParallelTrnsys
import yourCustomClassFile

class MyProcessParallelTrnsys(pParallelTrnsys.ProcessParallelTrnsys):

    def __init__(self):
        pParallelTrnsys.ProcessParallelTrnsys.__init__(self)

    #The definition of this class is a must
```

(continues on next page)

(continued from previous page)

```
    def getBaseClass(self, classProcessing, pathFolder, fileName):
        return yourCustomClassFile.MyProcess(pathFolder, fileName)

if __name__ == '__main__':
    pathConfig = "pathToTheConfigFile"
    configFile = "process_solar_dhw.config"
    tool = MyProcessParallelTrnsys()
    tool.readConfig(pathConfig,configFile)
    tool.process()
```

1.6.4 General guidelines for developers

Pytrnsys is open source and developers are invited to submit their own contributions. If you would like to develop for pytrnsys, we are interested in who you are. We are happy about a short message by mail. Please discuss new ideas first in the issue board. You are invited to work on the issues and create a pull request when finished. When working on the code, please consider the following style guidelines:

- we use the UpperCamelCase convention for Class names and the lowerCamelCase convention for everything else
- Please use Numpy/Scipy inline code documentation as much as possible
- Please chose meaningful variable names and use in line comments only where really needed.

1.7 pytrnsys

1.7.1 pytrnsys package

Subpackages

pytrnsys.pdata package

Submodules

pytrnsys.pdata.loadBase module

Class to load results file storing the names into self.namesVariables and the results into self.variables. As a difference of the loadBaseNumpy the results are stored into a normal list instead of numpy arrays. Author : Dani Carbonell Date : 14.12.2012

class pytrnsys.pdata.loadBase.**loadBase**(*_name*)

Bases: object

loadFile(*skypChar*, *replaceChar*)

setReadLabels(*_labels*)

setSkypedLines(*_number*)

setSplitArgument(*_split*)

setVerbose(*_verbose*)

pytrnsys.pdata.loadBaseNumpy module

pytrnsys.pdata.loadInputFile module

pytrnsys.pdata.processFiles module

Author : Dani Carbonell Date : 14.12.2012

`pytrnsys.pdata.processFiles.`**`purgueComments`**(*lines*, *commentsChar*)

`pytrnsys.pdata.processFiles.`**`purgueLines`**(*lines*, *skypChar*, *replaceChar*, *skypedLines=0*, *removeBlankLines=False*, *removeBlankSpaces=False*)

pytrnsys.pdata.processHourly module

Module contents

pytrnsys.physprop package

Submodules

pytrnsys.physprop.physProp module

Module contents

pytrnsys.plot package

Submodules

pytrnsys.plot.downSizeForPlotting module

pytrnsys.plot.plotBokeh module

pytrnsys.plot.plotGle module

pytrnsys.plot.plotGround module

pytrnsys.plot.plotMatplotlib module

pytrnsys.plot.plotTrnsysUtils module

Module contents

pytrnsys.psim package

Submodules

pytrnsys.psim.checkItProblemsClass module

pytrnsys.psim.checkSimulationDataClass module

pytrnsys.psim.debugProcess module

This class allows to print data on parallel runs of wich simulations have been finished and which ones are still missing. Author : Jeremias Schmidli, Daniel Carbonell Date : 02.04.2018

class pytrnsys.psim.debugProcess.**DebugProcess**(*_path*, *_fileName*, *cases*)
Bases: object

addCase(*i*)

addLines(*lines*)

finish()

start()

pytrnsys.psim.processMonthlyDataBase module

pytrnsys.psim.processParallel module

Main class to process TRNSYS cases in parallel. Author : Jeremias Schmidli Date : 02-03-2017 ToDo : Make use of runParallel and avoid repetition.

pytrnsys.psim.processParallel.**getCpuHexadecimal**(*cpu*)

pytrnsys.psim.processParallel.**getNumberOfCPU**()
Returns the number of CPUs in the system

pytrnsys.psim.processParallel.**processParallel**(*cmds*, *reduceCpu=0*, *outputFile=False*, *estimedCPUTime=1*)
Exec commands in parallel in multiple process (as much as we have CPU)

pytrnsys.psim.processParallelTrnsys module

pytrnsys.psim.processTrnsysBase module

pytrnsys.psim.processTrnsysDf module

pytrnsys.psim.processTrnsysFiles module

pytrnsys.psim.resultsProcessedFile module

pytrnsys.psim.simulationLoader module

Module contents

pytrnsys.report package

Submodules

pytrnsys.report.latexReport module

Module contents

pytrnsys.rsim package

Submodules

pytrnsys.rsim.executeTrnsys module

pytrnsys.rsim.runParallel module

pytrnsys.rsim.runParallelTrnsys module

Module contents

pytrnsys.trnsys_util package

Submodules

pytrnsys.trnsys_util.LogTrnsys module

pytrnsys.trnsys_util.buildTrnsysDeck module

pytrnsys.trnsys_util.createTrnsysDeck module

Created on Thu Aug 08 07:57:08 2013

@author: dcarbone

class pytrnsys.trnsys_util.createTrnsysDeck.**CreateTrnsysDeck**(*_path*, *_name*, *_variations*)

Bases: object

generateDecks()
This function will generate as many deck cases as the number of variations defined in the confg file. The deck are stored in self.path using the variations and the base name to create the individual names :return:

generateDecksDeprecated
This function will generate as many deck cases as the number of variations defined in the confg file. The deck are stored in self.path using the variations and the base name to create the individual names :return:

getParameters(*i*)

pytrnsys.trnsys_util.deckTrnsys module

pytrnsys.trnsys_util.deckUtils module

pytrnsys.trnsys_util.readConfigTrnsys module

pytrnsys.trnsys_util.readTrnsysFiles module

pytrnsys.trnsys_util.trnsysComponent module

Module contents

pytrnsys.utils package

Submodules

pytrnsys.utils.cleanFiles module

Clean files produced by TRNSYS simulations and by processing the results to store only essential data. Author : Daniel Carbonell Date : 21-01-2014 ToDo :

class `pytrnsys.utils.cleanFiles.`**`CleanFiles`**(*_path*)

Bases: `object`

`addFileToBeCopied`(*name*)

`addFileToBeErased`(*name*)

`addFileToBeErasedByDefault`(*name*)

`addFileToBeIgnored`(*name*)

`addFolderToBeErased`(*name*)

`addFoldersToBeErasedByDefault`()

`copyFilesToNewFolder`(*dstFolder=None*)

`copyTree`(*dst*)

`ig_f`(*dir, files*)

`ignore_notToBeCopied`(*dir, files*)

`removeFiles`()

`removeFolders`()

`removeTempFolder`()

`renameFiles`(*source, end*)

`replicateFolderStructure`(*dst*)

pytrnsys.utils.copyDllFiles module

`pytrnsys.utils.copyDllFiles.`**`dllCopy`**()

pytrnsys.utils.costCalculation module

pytrnsys.utils.costCalculationVar module

pytrnsys.utils.costConfig module

pytrnsys.utils.loadExamplesAndDdcks module

pytrnsys.utils.loadExamplesAndDdcks.**load**()

pytrnsys.utils.log module

pytrnsys.utils.log.**setup_custom_logger**(*name*, *level*)

pytrnsys.utils.unitConverter module

File to do basic conversion units Author : Daniel Carbonell Date : 2014 ToDo :

class pytrnsys.utils.unitConverter.**UnitConverter**

Bases: object

getBarToMmca()

getBarToPa()

getJToMWh()

getJTokWh()

getKJhToW()

getMJTokWh()

getPaToBar()

getPaToMca()

getPaToMmca()

getPaToPsi()

getPaTokgCm2()

getPsiToPa()

getWToKJh()

getkJToMWh()

getkPaToBar()

getkWhToJ()

getkgCm2ToPa()

pytrnsys.utils.unitConverter.**getBarToMmca**()

pytrnsys.utils.unitConverter.**getBarToPa**()

pytrnsys.utils.unitConverter.**getJToMWh**()

pytrnsys.utils.unitConverter.**getJTokWh**()

pytrnsys.utils.unitConverter.**getKJhToW**()

pytrnsys.utils.unitConverter.**getMJTokWh**()

pytrnsys.utils.unitConverter.**getPaToBar**()

pytrnsys.utils.unitConverter.**getPaToMca**()

pytrnsys.utils.unitConverter.**getPaToMmca**()

pytrnsys.utils.unitConverter.**getPaToPsi**()

pytrnsys.utils.unitConverter.**getPaTokgCm2**()

pytrnsys.utils.unitConverter.**getPsiToPa**()

pytrnsys.utils.unitConverter.**getWToKJh**()

pytrnsys.utils.unitConverter.**getkJToMWh**()

pytrnsys.utils.unitConverter.**getkPaToBar**()

pytrnsys.utils.unitConverter.**getkWhToJ**()

pytrnsys.utils.unitConverter.**getkWhToMJ**()

pytrnsys.utils.unitConverter.**getkgCm2ToPa**()

pytrnsys.utils.utilsSpf module

Module contents

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

2.1 Developers

- Daniel Carbonell : SPF Institute for Solar Technology, Rapperswil, Switzerland.
- Mattia Battaglia : SPF Institute for Solar Technology, Rapperswil, Switzerland.
- Jeremias Schmidli : SPF Institute for Solar Technology, Rapperswil, Switzerland.
- Maike Schubert : SPF Institute for Solar Technology, Rapperswil, Switzerland.
- Martin Neugebauer : SPF Institute for Solar Technology, Rapperswil, Switzerland.

2.2 Acknowledgments

A first version of this package was created in 2013 and since then it has evolved considerably. We would like to thank the Swiss Federal Office Of Energy (SFOE) who supported many projects related to simulations of renewable energy systems where this code has been developed. We would also like to thank the European Union's Horizon 2020 research and innovation programme for the funding received in TRI-HP under the Grant Agreement No. 81488. This project allowed to dedicate efforts in sharing the code with the consortium and to make the code usable for the others.

Python Module Index

p

pytrnsys, 43
pytrnsys.pdata, 38
pytrnsys.pdata.loadBase, 37
pytrnsys.pdata.processFiles, 38
pytrnsys.physprop, 38
pytrnsys.plot, 38
pytrnsys.psim, 39
pytrnsys.psim.debugProcess, 39
pytrnsys.psim.processParallel, 39
pytrnsys.report, 40
pytrnsys.rsim, 40
pytrnsys.trnsys_util, 41
pytrnsys.trnsys_util.createTrnsysDeck, 40
pytrnsys.utils, 43
pytrnsys.utils.cleanFiles, 41
pytrnsys.utils.copyDllFiles, 41
pytrnsys.utils.loadExamplesAndDdcks, 42
pytrnsys.utils.log, 42
pytrnsys.utils.unitConverter, 42

Index

A

addCase() (*pytrnsys.psim.debugProcess.DebugProcess method*), 39
addFileToBeCopied() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addFileToBeErased() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addFileToBeErasedByDefault() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addFileToBeIgnored() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addFoldersToBeErasedByDefault() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addFolderToBeErased() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
addLines() (*pytrnsys.psim.debugProcess.DebugProcess method*), 39

C

CleanFiles (*class in pytrnsys.utils.cleanFiles*), 41
copyFilesToNewFolder() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
copyTree() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
CreateTrnsysDeck (*class in pytrnsys.trnsys_util.createTrnsysDeck*), 40

D

DebugProcess (*class in pytrnsys.psim.debugProcess*), 39
dllCopy() (*in module pytrnsys.utils.copyDllFiles*), 41

F

finish() (*pytrnsys.psim.debugProcess.DebugProcess method*), 39

G

generateDecks() (*pytrnsys.trnsys_util.createTrnsysDeck.CreateTrnsysDeck method*), 40
generateDecksDeprecated (*pytrnsys.trnsys_util.createTrnsysDeck.CreateTrnsysDeck attribute*), 40
getBarToMmca() (*in module pytrnsys.utils.unitConverter*), 42
getBarToMmca() (*pytrnsys.utils.unitConverter.UnitConverter method*), 42
getBarToPa() (*in module pytrnsys.utils.unitConverter*), 42
getBarToPa() (*pytrnsys.utils.unitConverter.UnitConverter method*), 42
getCpuHexadecimal() (*in module pytrnsys.psim.processParallel*), 39
getJTokWh() (*in module pytrnsys.utils.unitConverter*), 42
getJTokWh() (*pytrnsys.utils.unitConverter.UnitConverter method*), 42
getJToMWh() (*in module pytrnsys.utils.unitConverter*), 42
getJToMWh() (*pytrnsys.utils.unitConverter.UnitConverter method*), 42
getkgCm2ToPa() (*in module pytrnsys.utils.unitConverter*), 43
getkgCm2ToPa() (*pytrnsys.utils.unitConverter.UnitConverter method*), 42
getKJhToW() (*in module pytrnsys.utils.unitConverter*), 43
getKJhToW() (*pytrnsys.utils.unitConverter.UnitConverter method*),

42
getkJToMWh() (in module pytrnsys.utils.unitConverter), 43
getkJToMWh() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getkPaToBar() (in module pytrnsys.utils.unitConverter), 43
getkPaToBar() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getkWhToJ() (in module pytrnsys.utils.unitConverter), 43
getkWhToJ() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getkWhToMJ() (in module pytrnsys.utils.unitConverter), 43
getMJTokWh() (in module pytrnsys.utils.unitConverter), 43
getMJTokWh() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getNumberOfCPU() (in module pytrnsys.psim.processParallel), 39
getParameters() (pytrnsys.trnsys_util.createTrnsysDeck.CreateTrnsysDeck method), 40
getPaToBar() (in module pytrnsys.utils.unitConverter), 43
getPaToBar() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getPaTokgCm2() (in module pytrnsys.utils.unitConverter), 43
getPaTokgCm2() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getPaToMca() (in module pytrnsys.utils.unitConverter), 43
getPaToMca() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getPaToMmca() (in module pytrnsys.utils.unitConverter), 43
getPaToMmca() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getPaToPsi() (in module pytrnsys.utils.unitConverter), 43
getPaToPsi() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getPsiToPa() (in module pytrnsys.utils.unitConverter), 43
getPsiToPa() (pytrnsys.utils.unitConverter.UnitConverter method), 42
getWToKJh() (in module pytrnsys.utils.unitConverter), 43
getWToKJh() (pytrnsys.utils.unitConverter.UnitConverter method), 42

I

ig_f() (pytrnsys.utils.cleanFiles.CleanFiles method), 41
ignore_notToBeCopied() (pytrnsys.utils.cleanFiles.CleanFiles method), 41

L

load() (in module pytrnsys.utils.loadExamplesAndDdcks), 42
loadBase (class in pytrnsys.pdata.loadBase), 37
loadFile() (pytrnsys.pdata.loadBase.loadBase method), 37

P

processParallel() (in module pytrnsys.psim.processParallel), 39
purgueComments() (in module pytrnsys.pdata.processFiles), 38
purgueLines() (in module pytrnsys.pdata.processFiles), 38
pytrnsys (module), 43
pytrnsys.pdata (module), 38
pytrnsys.pdata.loadBase (module), 37
pytrnsys.pdata.processFiles (module), 38
pytrnsys.physprop (module), 38
pytrnsys.plot (module), 38
pytrnsys.psim (module), 39
pytrnsys.psim.debugProcess (module), 39
pytrnsys.psim.processParallel (module), 39
pytrnsys.report (module), 40
pytrnsys.rsim (module), 40
pytrnsys.trnsys_util (module), 41
pytrnsys.trnsys_util.createTrnsysDeck (module), 40
pytrnsys.utils (module), 43
pytrnsys.utils.cleanFiles (module), 41
pytrnsys.utils.copyDllFiles (module), 41
pytrnsys.utils.loadExamplesAndDdcks (module), 42
pytrnsys.utils.log (module), 42
pytrnsys.utils.unitConverter (module), 42

R

removeFiles() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
removeFolders() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
removeTempFolder() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
renameFiles() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41
replicateFolderStructure() (*pytrnsys.utils.cleanFiles.CleanFiles method*), 41

S

setReadLabels() (*pytrnsys.pdata.loadBase.loadBase method*), 37
setSkypedLines() (*pytrnsys.pdata.loadBase.loadBase method*), 37
setSplitArgument() (*pytrnsys.pdata.loadBase.loadBase method*), 37
setup_custom_logger() (*in module pytrnsys.utils.log*), 42
setVerbose() (*pytrnsys.pdata.loadBase.loadBase method*), 37
start() (*pytrnsys.psim.debugProcess.DebugProcess method*), 39

U

UnitConverter (*class in pytrnsys.utils.unitConverter*), 42