ANDES Documentation

Release 0.6.8

Hantao Cui

Feb 12, 2020

ANDES Tutorial

ANDES is a Python-based power system simulation tool for research. It supports power flow calculation, time domain simulation, and full eigenvalue analysis for transmission networks. ANDES is currently under active development.

CHAPTER 1

Copyright and License

CHAPTER 2

Introduction

ANDES is a free software package for power system simulation and analysis based on the Python programming language. Its main goal is to provide a convenient and efficient tool for researchers to verify new models and implement new algorithms by using the elegent Python language and the extensive packages. ANDES supports power flow calculation, time domain simulation and full eigenvalue analysis for transmission networks.

CHAPTER 3

Installation instructions

3.1 User Mode Installation

3.1.1 Conda Installation for Windows, macOS and Linux

ANDES can be installed in Python 3.7+. We recommend the Miniconda distribution that includes the conda package manager and Python. Downloaded and install the latest Miniconda (x64, with Python 3) from https://conda.io/miniconda.html.

Open the Anaconda Prompt and create an environment for ANDES (optional)

```
conda create --name andes python=3.7
activate andes
```

Add the conda-forge channel and set it as default

```
conda config --add channels conda-forge
conda config --set channel_priority flexible
```

Install ANDES from conda-forge

```
conda install andes
```

3.1.2 Existing Python Environment Installation

If you prefer to use an existing Python installation, you can install ANDES with *pip*

```
python3 -m pip install andes
```

Pip will take care of the minimal dependency for ANDES. The following package installation are also recommended

```
python3 -m pip install matplotlib pandas sympy cvxpy flask requests
```

If you see a *Permission denied* error, you may need to install the packages locally with *–user*

3.2 Development Mode

Alternative to installing as a distribution package, you may install ANDES in development mode so that your changes will take effect immediately.

We still recommend you install ANDES with conda first. Then, remove the ANDES package while preserving the dependent package with

```
conda remove andes --force
```

Next, clone the ANDES source code from https://github.com/cuihantao/andes (or download and unzip it to the desired path). Change the working directory to the root folder of andes and run

```
python3 -m pip install -e .
```

Pip should take care of the rest.

Optional: Install *cvxoptklu* to use KLU for speed up. *cvxoptklu* is a standalone KLU direct solver for linear equations. KLU is generally ~20% faster than UMFPACK. cvxoptklu requires a C compiler, and the *openblas* and *SuiteSparse* libraries.

```
python3 -m install cvxoptklu
```

3.3 Trouble-shooting

There is a known issue of CVXOPT with versions earlier than 1.2.2 in Windows. If the time-domain simulation crashes for the *cases/ieee14/ieee14_syn.dm*, please check and install the latest CVXOPT (=>1.2.2).

CHAPTER 4

Getting Started

This chapter describes the basic usage of a) the command-line interface, b) the Python functions for Notebook or IPython interactive environment.

4.1 Command Line Usage

4.1.1 Command Line Options

Andes is invoked from the command line using the command `andes`. It prints out a splash screen with version and environment information and exits with a "no input" error.

```
ANDES 0.6.8 (Build 9da49e2b, Python 3.7.3 on Darwin)
Session: hcui7, 10/23/2019 10:23:44 AM

error: no input file. Try 'andes -h' for help.
```

Help for the command line options can be retrieved using the `-h` or `--help` command, namely, `andes -h`.

4.1.2 Frequently Used Options

`casename`

The positional argument, path to the case file, is required for andes to populate the power system object and perform studies. The case file, however, is omitted if any utility options are specified in the optional arguments.

To perform a power flow study on the file named `ieee14.dm` in the current directory, run

```
andes ieee14.dm
```

Andes also takes the full path to the case file, for example,

```
andes /home/hcui7/andes/cases/ieee14.dm
```

Output files will be saved to the current directory where andes is called.

Andes also takes multiple files or wildcard. Multiprocessing will be triggered if more than one valid input files are found. For example, to run power flow for files with a prefix of `ieee1` and a suffix (file extension) of `.dm`, run

```
andes ieee1*.dm
```

Andes will run the case files that match the pattern such as `ieee14.dm` and `ieee118.dm`.

`-r, -routine {pflow, tds, eig}`

Option for specifying the analysis routine. *pflow* for power flow, *tds* for time domain simulation, and *eig* for eigenvalue analysis. *pflow* as the default is this option is not given.

For example, to run time domain simulation for `ieee14.dm` in the current directory, run

```
andes ieee14.dm -r tds
```

`-q, --quick-help`

Print out a quick help of parameter definitions of a single given model. For example, `andes -q Bus` prints out the parameter definition of the model `Bus`.

Parameters with an asterisk * are mandatory. Parameters with a number sign # are per unit values in the element base.

`-d, --dump-raw`

Export the given case files to the DOME format supported by andes.

`-v VERBOSE, --verbose VERBOSE`

Program verbosity level. 10 - DEBUG, 20 - INFO, 30 - WARNING, 40 - ERROR, 50 - CRITICAL. For the most debugging output, use `-v 10`.

`-C, --clean`

Option to remove any generated files. Removes files with any of the following suffix: `_out.txt` (power flow report), `_out.dat` (time domain data), `_out.lst` (time domain variable list), and `_eig.txt` (eigenvalue report).

`-g, --group`

Option to print out all the models in a group. To print out all the groups and models they contain, run `andes -g all`.

`--help-config`

Print out a table of help for the specified configurations. Available options are `all`, `system` or any routine name such as `pflow` and `eig`.

For example, `andes --help-config all` prints out all the config helps.

`--save-config`

Saves all configs to a file. By default, save to `~/.andes/andes.conf` file.

This file contains all the runtime configs for the system and routines.

`--load-config`

Load an Andes config file that occurs first in the following search path: a) the specified path, b) current directory, c) home directory

`-v, --verbose` Verbosity level in (10, 20, 30, 40, 50) for (DEBUG, INFO, WARNING, ERROR, CRITICAL). The default is 20 (INFO). Set to 10 for debugging.

4.1.3 Plotting Tool

Andes comes with a command-line plotting tool, *tdsplot* for time-domain simulation output data.

usage: tdsplot [-h] [–xmin LEFT] [–xmax RIGHT] [–ymax YMAX] [–ymin YMIN] [–checkinit] [-x XLABEL] [-y YLABEL] [-s] [-g] [-d] [-n] [–ytimes YTIMES] [–dpi DPI] datfile x [y [y . . .]]

positional arguments:

Argument	Description
datfile	dat file name.
x	x axis variable index
y	y axis variable index

optional arguments:

Argument	Description
-h, –help	show this help message and exit
–xmin LEFT	x axis minimum value
–xmax RIGHT	x axis maximum value
–ymax YMAX	y axis maximum value
–ymin YMIN	y axis minimum value
–checkinit	check initialization value
-x XLABEL, –xlabel XLABEL	manual x-axis text label
-y YLABEL, –ylabel YLABEL	y-axis text label
-s, –save	save to file
-g, –grid	grid on
-d, –no_latex	disable LaTex formatting
-n, –no_show	do not show the plot window
–ytimes YTIMES	y times
–dpi DPI	image resolution in dot per inch (DPI)

4.1.4 Examples

4.1.5 Power Flow Calculation

The example test cases are in the `cases` folder of the package.

Run power flow for `ieee14_syn.dm` using the command

```
ANDES 0.6.8 (Build 9da49e2b, Python 3.7.3 on Darwin)
Session: hcui7, 10/23/2019 11:18:32 AM

Parsing input file <ieee14_syn.dm>
-> Power flow study: NR method, non-flat start
Iter 1.  max mismatch = 2.1699877
Iter 2.  max mismatch = 0.2403104
Iter 3.  max mismatch = 0.0009915
Iter 4.  max mismatch = 0.0000001
Solution converged in 0.0027 second in 4 iterations
Report saved to <ieee14_syn_out.txt> in 0.0016 second.
-> Single process finished in 0.2191 second.
```

The printed message shows that the power flow uses the Newton Raphson (NR) method with non-flat start. The solution process converges in four iterations in 0.002 seconds. The report is written to the file <ieee14_syn_out.txt>.

The power flow report contains four sections: a) system statistics, b) ac bus and dc node data, c) ac line data, and d) the initialized values of other algebraic variables and state variables.

4.1.6 Time Domain Simulation

The other most used routine of andes is the time domain simulation (TDS).

4.1.7 Change Run Config

You can change the configuration of the power flow run by saving the config and editing it.

Run `andes --save-config` to save the config file to the default location. Then, run `andes --edit-config` to edit it. On Microsoft Windows, it will open up a notepad. On Linux, it will use the `$EDITOR` environment variable or use `gedit` by default. On macOS, the default is vim.

To change the power flow solution method, for example, from NR to Fast Decoupled Power Flow (FDPF), find `method = NR `` in the ``[Pflow]` section and modified it to

    method = FDPF

Note that FDPF is an available method. To view the available options, in a command line window, run `andes --help-config pflow`.

4.1.8 Time Domain Simulation

To run the time domain simulation (TDS) for `ieee14_syn.dm`, run

```
$ andes ieee14.dm -r tds
ANDES 0.5.5 (Build g651fdac, Python 3.5.2 on Linux)
Session: 09/06/2018 11:18:55 AM

Parsing input file <ieee14_syn.dm>
-> Power flow study: NR method, non-flat start
Iter 1.  max mismatch = 2.1699877
Iter 2.  max mismatch = 0.2403104
Iter 3.  max mismatch = 0.0009915
Iter 4.  max mismatch = 0.0000001
Solution converged in 0.0054 second in 4 iterations
report written to <ieee14_syn_out.txt> in 0.0019 second.
-> Time Domain Simulation: trapezoidal method, t=20 s
<Fault> Applying fault on Bus <4.0> at t=2.0.
<Fault> Clearing fault on Bus <4.0> at t=2.05.
Time domain simulation finished in 1.2613 seconds.
-> Single process finished in 1.3878 seconds.
```

This execution first solves the power flow as a starting point. Next, the numerical integration is run to simulate 20 seconds during which a predefined fault on Bus 4 happens at 2 seconds.

TDS produces two output files by default: a data file `ieee14_syn_out.dat` and a variable name list file `ieee14_syn_out.lst`. The list file contains three columns: variable indices, variabla name in plain text, and variable name in LaTeX format. The variable indices are needed to plot the needed variable.

4.1.9 Plottting the TDS Results

For example, to plot the generator speed variable of synchronous generator 1 `omega Syn 1` versus time, read the indices of the variable (44) and time (0), run

```
andesplot ieee14_syn_out.dat 0 44
```

In this command, `andesplot` is a plotting tool for TDS output files. `ieee14_syn_out.dat` is data file name. `0` is the index of `Time` for the x-axis. `44` is the index of `omega Syn 1`.

The y-axis variabla indices can also be specified in the Python range fashion . For example, `andesplot ieee14_syn_out.dat 0 44:69:6` will plot the variables at indices 44, 50, 56, 62, and 68.

`andesplot` will attempt to render the image with LaTeX if `dvipng` program is in the search path. In case LaTeX is available but fails (happens on Windows), the option `-d` can be used to disable LaTeX rendering.

A complete list of options for `andesplot` is available using `andesplot -h`.

4.2 Interactive Usage

4.2.1 Running Studies

The Andes Python APIs are loaded into an interactive Python environment (Python, IPython or Jupyter Notebook) using `import`. To start, import the whole package and set up the global logger using

```
>>> import andes
>>> andes.main.config_logger(log_file=None)
```

Create an instance of Power System from the case file, for example, at ‘‘ ieee14_syn.dm‘‘ whole package and set up the global logger using

```
>>> import andes
>>> andes.main.config_logger(log_file=None)
```

Create an instance of Power System from the case file, for example, at ‘‘ ieee14_syn.dm‘‘ whole package and set up the global logger using

```
>>> import andes
>>> andes.main.config_logger(logfile=None)
```

Create an instance of Power System from the case file, for example, at ‘‘ ieee14_syn.dm‘‘

```
>>> ps = andes.system.PowerSystem('ieee14_syn.dm')
```

Next, guess the input file format and parse the data into the system

```
>>> andes.filters.guess(ps)
'dome'
>>> andes.filters.parse(ps)
Parsing input file <ieee14_syn.dm>
True
```

Next, set up the system structure using the parsed input data

```
>>> ps.setup()
<andes.system.PowerSystem at 0x7fd5ea96d4e0>
```

To continue, run the power flow study using

```
>>> ps.pflow.run()
-> Power flow study: NR method, non-flat start
Iter 1.  max mismatch = 2.1699877
Iter 2.  max mismatch = 0.2403104
Iter 3.  max mismatch = 0.0009915
Iter 4.  max mismatch = 0.0000001
Solution converged in 0.0038 second in 4 iterations
Out[8]: (True, 4)
```

To change the run config, change the attributes in `ps.pflow.config`. The config options can be printed out with `print(ps.pflow.config.doc())`.

Before running the TDS or eigenvalue analysis, the dynamic components needs to be initialized with

>> ps.tds.init()

Run the next analysis routine, for example, TDS, with

```
>>> ps.tds.run()
-> Time Domain Simulation: trapezoidal method, t=20 s
<Fault> Applying fault on Bus <4.0> at t=2.0.              |ETA:  0:00:00]
<Fault> Clearing fault on Bus <4.0> at t=2.05.
[100%|##########################################################|Time: 0:00:01]
Time domain simulation finished in 1.2599 seconds.
True
```

Save the results to list and data files with

```
>>> ps.tds.dump_results()
Simulation data sumped in 0.0978 seconds.
```

4.2.2 Plotting Results

The `andes.plot` package can be used interactively for plotting time-domain simulation results. Import functions from the package using

```
>>> from andes.plot import read_dat, read_label, do_plot
```

Specify the files and the indices to plot using

```
>>> dat_file = 'ieee14_syn_out.dat'
>>> lst_file = 'ieee14_syn_out.lst'
>>> x_idx = [0]
>>> y_idx = [44, 50, 56]
```

Call functions *read_dat* and *read_label* to read out the values and names based on the variable indices.

```
>>> x_dat, y_dat = read_dat(dat_file, x_idx, y_idx)
>>> x_name, y_name = read_label(lst_file, x_idx, y_idx)
```

Call function *do_plot* to plot the curves

```
>>> fig, ax = do_plot(xdata=x_dat, ydata=y_dat,
                      xname=x_name, yname=y_name,
                      ylabel='Generator Speed [pu]', grid=True)
```

CHAPTER 5

Input and output formats

CHAPTER 6

power system test cases

CHAPTER 7

Device Modeling

CHAPTER 8

Release History

8.1 v0.5.7 (2018-09-04)

CHAPTER 9

andes

9.1 andes package

9.1.1 Subpackages

andes.config package

Submodules

andes.config.base module

class andes.config.base.**ConfigBase**(*conf=None*, ***kwargs*)
Bases: object

Base class for routine configurations. This class provides functions for storing, accessing, exporting configurations for Andes routines.

check(*self*)
Check for consistency

Returns

bool True for pass, False for fail

config_descr

doc(*self*, *export='plain'*)
Dump help document for setting classes

Parameters

export [{'plain'}, optional] The format of the exported config help docs.

Returns

str A string containing the formattted config help docs

dump_conf(*self*, *conf=None*)
Dump routine configuration to an rc config file

Parameters

conf [configparser.ConfigParser, optional] A config parser object to which this class will be exported. A new object will be created if *conf* is *None*.

Returns

configparser.ConfigParser A config parser object with the values appended

get_alt(*self*, *option*)
Return the available alternative values for an option

Parameters

option: str Name of the option to query

Returns

str A comma-separated string containing all the acceptable alternative values. If *option* is not valid or no alternative value is on file, return an empty string.

get_value(*self*, *option*)
Return the value of the given option

Parameters

option: str option name to retrieve value

Returns

str the current value for the option

load_config(*self*, *conf*)
Load configurations from an rc file

Parameters

rc: str path to the rc file

Returns

None

andes.config.cpf module

class andes.config.cpf.**CPF**(***kwargs*)
Bases: *andes.config.base.ConfigBase*

config_descr

andes.config.eig module

class andes.config.eig.**Eig**(***kwargs*)
Bases: *andes.config.base.ConfigBase*

config_descr

andes.config.pflow module

class andes.config.pflow.**Pflow**(***kwargs*)
Bases: *andes.config.base.ConfigBase*

check(*self*)
Check for consistency

Returns

bool True for pass, False for fail

config_descr

andes.config.system module

class andes.config.system.**System**(***kwargs*)
Bases: *andes.config.base.ConfigBase*

check(*self*)
Check config data consistency

config_descr

andes.config.tds module

class andes.config.tds.**Tds**(***kwargs*)
Bases: *andes.config.base.ConfigBase*

config_descr

Module contents

andes.filters package

Submodules

andes.filters.card module

andes.filters.card.**add_quotes**(*string*)

andes.filters.card.**de_blank**(*val*)
Remove blank elements in *val* and return *ret*

andes.filters.card.**read**(*file*, *system*)
Parse an ANDES card file into internal variables

andes.filters.card.**run**(*system*, *outfile=''*, *name=''*, *doc_string=''*, *group=''*, *data={}*, *descr={}*, *units={}*, *params=[]*, *fnamex=[]*, *fnamey=[]*, *mandatory=[]*, *zeros=[]*, *powers=[]*, *currents=[]*, *voltages=[]*, *z=[]*, *y=[]*, *dccurrents=[]*, *dcvoltages=[]*, *r=[]*, *g=[]*, *times=[]*, *ac={}*, *dc={}*, *ctrl={}*, *consts=[]*, *algebs=[]*, *interfaces=[]*, *states=[]*, *init1_eq={}*, *service_eq={}*, *algeb_eq=[]*, *windup={}*, *hard_limit={}*, *diff_eq=[]*, *anti_windup={}*, *copy_algebs=[]*, *copy_states=[]*, *service_keys=[]*, ***kwargs*)
TODO: Fix the mutable default argument

Parameters

system

outfile

name

doc_string

group

data

descr

units

params

fnamex

fnamey

mandatory

zeros

powers

currents

voltages

z

y

dccurrents

dcvoltages

r

g

times

ac

dc

ctrl

consts

algebs

interfaces

states

init1_eq

service_eq

algeb_eq

windup

hard_limit

diff_eq

anti_windup

copy_algebs

copy_states

service_keys

kwargs

andes.filters.card.**stringfy**(*expr*, *sym_const=None*, *sym_states=None*, *sym_algebs=None*)
Convert the right-hand-side of an equation into CVXOPT matrix operations

andes.filters.card.**testlines**(*fid*)

andes.filters.dome module

Parser for DOME RAW format 0.1 From Book "Power System Modeling and Scripting" by Dr. Federico Milano

andes.filters.dome.**alter**(*data*, *system*)
Alter data in dm format devices

andes.filters.dome.**read**(*file*, *system*, *header=True*)
Read a dm format file and elem_add to system

andes.filters.dome.**testlines**(*fid*)

andes.filters.dome.**write**(*file*, *system*)
Write data in system to a dm file

andes.filters.matpower module

Simple MATPOWER format parser

andes.filters.matpower.**read**(*file*, *system*)
Read a MATPOWER data file into mpc and build andes device elements

andes.filters.matpower.**testlines**(*fid*)

andes.filters.psse module

PSS/E file parser

andes.filters.psse.**add_dyn**(*system*, *model*, *data*)
helper function to elem_add a device element to system

andes.filters.psse.**get_idx**(*system*, *group*, *param*, *fkey*)

andes.filters.psse.**get_nol**(*b*, *mdata*)
Return the number of lines based on data

andes.filters.psse.**get_param**(*system*, *group*, *param*, *fkey*)

andes.filters.psse.**read**(*file*, *system*)
read PSS/E RAW file v32 format

andes.filters.psse.**readadd**(*file*, *system*)
read DYR file

andes.filters.psse.**testlines**(*fid*)
Check the raw file for frequency base

Module contents

andes.filters.**dump_raw**(*system*)

andes.filters.**guess**(*system*)
input format guess function. First guess by extension, then test by lines

andes.filters.**parse**(*system*)
Parse input file with the given format in system.files.input_format

andes.formats package

Submodules

andes.formats.csv module

andes.formats.csv.**dump_data**(*text*, *header*, *rowname*, *data*, *file*)

andes.formats.csv.**format_item**(*item*, *val*)

andes.formats.csv.**format_newline**()

andes.formats.csv.**format_table**(*header*, *data*, *title=None*)

andes.formats.csv.**format_title**(*item*)

andes.formats.latex module

andes.formats.txt module

andes.formats.txt.**dump_data**(*text*, *header*, *rowname*, *data*, *file*, *width=14*, *precision=5*)

andes.formats.txt.**format_item**(*item*, *val*)

andes.formats.txt.**format_newline**()

andes.formats.txt.**format_table**(*header*, *data*, *title=None*)

andes.formats.txt.**format_title**(*item*)

Module contents

andes.models package

Submodules

andes.models.agc module

class andes.models.agc.**AGC**(*system*, *name*)
Bases: *andes.models.agc.AGCSyn*

Alias for class <AGCSyn>

class andes.models.agc.**AGCBase**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Base AGC class. The allocation of Pagc will be based on inverse droop (iR)

fcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.agc.**AGCMPC**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

MPC based AGC using TG and VSC

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.agc.**AGCSyn**(*system*, *name*)
Bases: *andes.models.agc.AGCBase*

AGC for synchronous generators. This class changes the setpoints by modifying the generator pm.

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

class andes.models.agc.**AGCSynVSC**(*system*, *name*)
Bases: *andes.models.agc.AGCSyn*, *andes.models.agc.AGCVSCBase*

AGC class that modifies Synchronous pm and VSC pref

fcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.agc.**AGCTG**(*system*, *name*)
Bases: *andes.models.agc.AGCBase*

AGC class that modifies the turbine governor power reference. Links to TG1 only.

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

class andes.models.agc.**AGCTGVSC**(*system*, *name*)
Bases: *andes.models.agc.AGCTG*, *andes.models.agc.AGCVSCBase*

AGC class that modifies the turbine governor and VSC pref

fcall (*self*, *dae*)

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

class andes.models.agc.**AGCVSCBase** (*system*, *name*)
Bases: object

Base class for AGC using VSC. Modifies the ref1 for PV or PQ-controlled VSCs. This class must be inherited with subclasses of AGCBase

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init1 (*self*, *dae*)

class andes.models.agc.**BArea** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Balancing area class. This class defines power balancing area on top of the *Area* class for calculating center of inertia frequency, total inertia, expected power and area control error.

gcall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

class andes.models.agc.**eAGC** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

gcall (*self*, *dae*)

init1 (*self*, *dae*)

switch (*self*)
Switch if time for eAgc has come

andes.models.avr module

class andes.models.avr.**AVR1** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Automatic Voltage Regulator Type I, DC exciter simplified from IEEE DC1

Se

dSe

fcall (*self*, *dae*)

fxcall (*self*, *dae*)

gcall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

servcall (*self*, *dae*)

class andes.models.avr.**AVR2**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Automatic Voltage Regulator Type II with fast response and high gain

Se

dSe

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.avr.**AVR3**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Automatic Voltage Regulator Type III

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

andes.models.base module

Base class for building ANDES models

class andes.models.base.**ModelBase**(*system*, *name*)
Bases: object

base class for power system device models

build_service(*self*)
Prototype function for building service variables. This function is to be called during *init0* or *init1*. Must be overloaded by derived classes.

check_limit(*self*, *varname*, *vmin=None*, *vmax=None*)
Check if the variable values are within the limits.

Return False if fails.

copy_data_ext(*self*, *model*, *field*, *dest=None*, *idx=None*, *astype=None*)
Retrieve the field of another model and store it as a field.

Parameters

- **model** (*str*) – name of the source model being a model name or a group name
- **field** (*str*) – name of the field to retrieve
- **dest** (*str*) – name of the destination field in `self`

- **idx** (*list, matrix*) – idx of elements to access
- **astype** (*None, list, matrix*) – type cast

Returns None

data_from_dict(*self*, *data*)
Populate model parameters from a dictionary of parameters

Parameters

data [dict] List of parameter dictionaries

Returns

None

data_from_list(*self*, *data*)
Populate model parameters from a list of parameter dictionaries

Parameters

data [list] List of parameter dictionaries

Returns

None

data_keys
Returns a list of all parameters plus `name` and `idx`

Returns

list of parameters

data_to_df(*self*, *sysbase=False*)
Return a pandas.DataFrame of device parameters. :param sysbase: save per unit values in system base

data_to_dict(*self*, *sysbase=False*)
Return the loaded model parameters as one dictionary.

Each key of the dictionary is a parameter name, and the value is a list of all the parameter values.

Parameters **sysbase** (*bool*) – use system base quantities

data_to_elem_base(*self*)
Convert parameter data to element base

Returns

None

data_to_list(*self*, *sysbase=False*)
Return the loaded model data as a list of dictionaries.

Each dictionary contains the full parameters of an element. :param sysbase: use system base quantities :type sysbase: bool

data_to_sys_base(*self*)
Converts parameters to system base. Stores a copy in `self._store`. Sets the flag `self.flag['sysbase']` to True.

Returns None

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

doc(*self*, *export='plain'*)
Build help document into a table

Parameters

- **'latex') export** (*('plain',)*) – export format
- **save** – save to file `help_model.extension` or not
- **writemode** – file write mode

Returns None

elem_add(*self*, *idx=None*, *name=None*, ***kwargs*)
Add an element of this model

Parameters

idx [str, int or float, optional] The unique external reference to this element.

name [str, optional] The name of this element. Automatically assigned if not provided.

kwargs A dictionary containing the (field, value) pairs of the element parameters

Returns

str or int The allocated *idx* of the added element

elem_find(*self*, *field*, *value*)
Return the indices of elements whose field first satisfies the given values

`value` should be unique in self.field. This function does not check the uniqueness.

Parameters

- **field** – name of the supplied field
- **value** – value of field of the elemtn to find

Returns idx of the elements

Return type list, int, float, str

elem_remove(*self*, *idx=None*)
Remove elements labeled by idx from this model instance.

Parameters **idx** (*list,matrix*) – indices of elements to be removed

Returns None

eq_add(*self*, *expr*, *var*, *intf=False*)
Add an equation to this model.

An equation is associated with the addresses of a variable. The number of equations must equal that of variables.

Stored to `self._equations` is a tuple of `(expr, var, intf, ty)` where `ty` is in ('f', 'g')

Parameters

- **expr** (*str*) – equation expression
- **var** (*str*) – variable name to be associated with
- **intf** (*bool*) – if this equation is added to an interface equation, namely, an equation outside this model

Returns None

fcall (*self*, *dae*)

fxcall (*self*, *dae*)

fxcall_idx (*self*)
Precompute the row and column indices of the non-zero elements in the df/dx matrix

Returns

(row indices, col indices)

gcall (*self*, *dae*)

get_element_data (*self*, *idx*)
Get all data associated with an element whose index is *idx* in a dict.

Returns

dict [param, value pairs]

get_field (*self*, *field*, *idx=None*, *astype=None*)
Return *self.field* for the elements labeled by *idx*

Parameters

- **astype** – type cast of the return value
- **field** – field name of this model
- **idx** – element indices, will be the whole list if not specified

Returns field values

get_idx (*self*, *uid*)
Return the `idx` of the elements whose internal indices are `uid`

Parameters **uid** – uid of elements to query

Returns idx of the elements

get_uid (*self*, *idx*)
Return the *uid* of the elements with the given *idx*

Parameters **matrix idx** (*list,*) – external indices

Returns a matrix of uid

gycall (*self*, *dae*)

init1 (*self*, *dae*)

init_limit (*self*, *key*, *lower=None*, *upper=None*, *limit=False*)
check if data is within limits. reset if violates

link_bus (*self*, *bus_idx*)
Return the indices of elements linking the given buses

Parameters **bus_idx** –

Returns

link_from (*self*)

link_to (*self*, *model*, *idx*, *self_idx*)
Register (self.name, self.idx) in *model._from*

log (*self*, *msg*, *level=20*)
Record a line of log in logger

Parameters

- **msg** (*str*) – content of the messag
- **level** – logging level

Returns None

n

Return the count of elements

Returns

int: count of elements

on_bus(*self*, *bus_idx*)

Return the indices of elements on the given buses for shunt-connected elements

Parameters **bus_idx** – idx of the buses to which the elements are connected

Returns idx of elements connected to bus_idx

param_alter(*self*, *param*, *default=None*, *unit=None*, *descr=None*, *tomatrix=None*, *nonzero=None*, *mandatory=None*, *power=None*, *voltage=None*, *current=None*, *z=None*, *y=None*, *r=None*, *g=None*, *dccurrent=None*, *dcvoltage=None*, *time=None*, ***kwargs*)

Set attribute of an existing parameter. To be used to alter an attribute inherited from parent models.

See .. self.param_define for argument descriptions.

param_define(*self*, *param*, *default*, *unit=''*, *descr=''*, *tomatrix=True*, *nonzero=False*, *mandatory=False*, *power=False*, *voltage=False*, *current=False*, *z=False*, *y=False*, *r=False*, *g=False*, *dccurrent=False*, *dcvoltage=False*, *time=False*, *event_time=False*, ***kwargs*)

Define a parameter in the model

Parameters

- **tomatrix** (*bool*) – convert this parameter list to matrix
- **param** (*str*) – parameter name
- **default** (*str, float*) – parameter default value
- **unit** (*str*) – parameter unit
- **descr** (*str*) – description
- **nonzero** (*bool*) – is non-zero
- **mandatory** (*bool*) – is mandatory
- **power** (*bool*) – is a power value in the *self.Sn* base
- **voltage** (*bool*) – is a voltage value in the *self.Vn* base
- **current** (*bool*) – is a current value in the device base
- **z** (*bool*) – is an impedance value in the device base
- **y** (*bool*) – is an admittance value in the device base
- **r** (*bool*) – is a dc resistance value in the device base
- **g** (*bool*) – is a dc conductance value in the device base
- **dccurrent** (*bool*) – is a dc current value in the device base
- **dcvoltage** (*bool*) – is a dc votlage value in the device base

- **time** (*bool*) – is a time value in the device base
- **event_time** (*bool*) – is a variable for timed event

param_remove (*self*, *param: 'str'*) → None
Remove a param from this model

Parameters **param** (*str*) – name of the parameter to be removed

read_data_ext (*self*, *model: str*, *field: str*, *idx=None*, *astype=None*)
Return a field of a model or group at the given indices

Parameters

- **model** (*str*) – name of the group or model to retrieve
- **field** (*str*) – name of the field
- **int, float str idx** (*list,*) – idx of elements to access
- **astype** (*type*) – type cast

Returns

reload_new_param (*self*)
Reload the new params modified in *self._store* and then convert to sysbase and store in class.

service_define (*self*, *service*, *ty*)
Add a service variable of type `ty` to this model

Parameters

- **service** (*str*) – variable name
- **ty** (*type*) – variable type

Returns None

set_field (*self*, *field*, *idx*, *value*, *sysbase=False*)
Set the field of an element to the given value

Parameters

field

idx

value

sysbase

setup (*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

var_define (*self*, *variable*, *ty*, *fname*, *descr=''*, *uname=''*)
Define a variable in the model

Parameters

- **fname** – LaTex formatted variable name string
- **uname** – unformatted variable name string, *variable* as default
- **variable** (*str*) – variable name
- **ty** (*str*) – type code in `('x', 'y')`

- **descr** (*str*) – variable description

Returns

var_to_df(*self*)
Return the current var_to_df of variables

Returns pandas.DataFrame

andes.models.breaker module

class andes.models.breaker.**Breaker**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Simple line breaker model

apply(*self*, *actual_time*)

get_times(*self*)
Return all the action times and times-1e-6 in a list

insert(*self*, *idx=None*, *name=None*, ***kwargs*)

is_time(*self*, *t*)

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

andes.models.bus module

class andes.models.bus.**Bus**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

ac bus model for defining topology definition

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

gisland(*self*, *dae*)
Reset g(x) for islanded buses and areas

gyisland(*self*, *dae*)
Reset gy(x) for islanded buses and areas

init0(*self*, *dae*)
Set bus Va and Vm initial values

class andes.models.bus.**BusOld**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

AC bus model

gisland(*self*, *dae*)
Reset g(x) for islanded buses and areas

gyisland(*self*, *dae*)
Reset gy(x) for islanded buses and areas

init0 (*self*, *dae*)
Set bus Va and Vm initial values

andes.models.coi module

class andes.models.coi.**COI** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

fcall (*self*, *dae*)

gcall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

andes.models.dcbase module

class andes.models.dcbase.**C** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

Pure capacitive line

fcall (*self*, *dae*)

gcall (*self*, *dae*)

init0 (*self*, *dae*)

jac0 (*self*, *dae*)

servcall (*self*, *dae*)

class andes.models.dcbase.**DCBase** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Two-terminal DC device base

v12

class andes.models.dcbase.**DCgen** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

A simple DC generator to impose constantactive power injection

disable_gen (*self*, *idx*)

gcall (*self*, *dae*)

class andes.models.dcbase.**DCload** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

A simple DC load to impose constant active power consumption

disable (*self*, *idx*)

gcall (*self*, *dae*)

class andes.models.dcbase.**Ground** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

DC Ground node

gcall (*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.dcbase.**L**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

Pure inductive line

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.dcbase.**Node**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

DC node class

elem_add(*self*, *idx=None*, *name=None*, ***kwargs*)
Add an element of this model

Parameters

idx [str, int or float, optional] The unique external reference to this element.

name [str, optional] The name of this element. Automatically assigned if not provided.

kwargs A dictionary containing the (field, value) pairs of the element parameters

Returns

str or int The allocated *idx* of the added element

init0(*self*, *dae*)

jac0(*self*, *dae*)

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

class andes.models.dcbase.**R**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

DC Resistence line class

gcall(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.dcbase.**RCp**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

RC parallel line

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.dcbase.**RCs**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

RC series line

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.dcbase.**RLCp**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

RLC parallel line

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.dcbase.**RLCs**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

RLC series

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.dcbase.**RLs**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

DC Resistive and Inductive line

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init0(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

andes.models.event module

Timed event base class

class andes.models.event.**EventBase**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Base class for timed events

apply(*self*, *sim_time*)
Apply the event and :param sim_time: :return:

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

get_times(*self*)
Return a list of occurrance times of the events

Returns list of times

is_time(*self*, *t*)
Return the truth value of whether `t` is a time in `self.get_times()`

Parameters **t** (*float*) – time to check for

Return bool truth value

class andes.models.event.**GenTrip**(*system*, *name*)
Bases: *andes.models.event.EventBase*

Timed generator trip and reconnect

apply(*self*, *sim_time*)
Apply the event and :param sim_time: :return:

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

class andes.models.event.**LoadScale**(*system*, *name*)
Bases: *andes.models.event.EventBase*

Timed load scaling

apply(*self*, *sim_time*)
Apply the event and :param sim_time: :return:

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

class andes.models.event.**LoadShed**(*system*, *name*)
Bases: *andes.models.event.EventBase*

Timed load shedding and reconnect

apply(*self*, *sim_time*)
Apply the event and :param sim_time: :return:

define(*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

andes.models.fault module

class andes.models.fault.**Fault**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

3-phase to ground fault class

apply(*self*, *actual_time*)
Check time and apply faults

gcall(*self*, *dae*)

get_times(*self*)

gycall(*self*, *dae*)

insert(*self*, *idx=None*, *name=None*, ***kwargs*)

is_time(*self*, *t*)

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

andes.models.governor module

class andes.models.governor.**GovernorBase**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Turbine governor base class

data_to_elem_base(*self*)
Custom system base unconversion function

data_to_sys_base(*self*)
Custom system base conversion function

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.governor.**TG1**(*system*, *name*)
Bases: *andes.models.governor.GovernorBase*

Turbine governor model

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.governor.**TG2**(*system*, *name*)
Bases: *andes.models.governor.GovernorBase*

Simplified governor model

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

andes.models.jit module

class andes.models.jit.**JIT** (*system*, *model*, *device*, *name*)
Bases: object

Dummy Just-in-Time initialization class

doc (*self*, ***kwargs*)

elem_add (*self*, *idx=None*, *name=None*, ***kwargs*)
overloading elem_add function of a JIT class

jit_load (*self*)
Import and instantiate this JIT object

andes.models.line module

class andes.models.line.**Line** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

AC transmission line lumped model

build_b (*self*)
build Bp and Bpp for fast decoupled method

build_gy (*self*, *dae*)
Build line Jacobian matrix

build_name_from_bus (*self*)
Rebuild line names from bus names

build_y (*self*)
Build transmission line admittance matrix into self.Y

connectivity (*self*, *bus*)
check connectivity of network using Goderya's algorithm

gcall (*self*, *dae*)

get_flow_by_idx (*self*, *idx*, *bus*)
Return seriesflow based on the external idx on the *bus* side

gycall (*self*, *dae*)

incidence (*self*)
Build incidence matrix into self.C

init0 (*self*, *dae*)

leaf_bus (*self*, *df=False*)
Return leaf bus idx, line idx, and the line foreign key

Returns

(list, list, list) or DataFrame

seriesflow(*self*, *dae*)
Compute the flow through the line after solving PF.

Compute terminal injections, line losses

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

switch(*self*, *idx*, *u*)
switch the status of Line idx

v1
Return voltage phasors at the "from buses" (bus1)

v2
Return voltage phasors at the "to buses" (bus2)

andes.models.measurement module

class andes.models.measurement.**BusFreq**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Bus frequency estimation based on angle derivative

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.measurement.**PMU**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Phasor measurement unit described by low-pass filters

fcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

andes.models.pq module

class andes.models.pq.**PQ**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Static PQ load class

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init0(*self*, *dae*)
Set initial p and q for power flow

init1(*self*, *dae*)
Set initial voltage for time domain simulation

andes.models.pss module

class andes.models.pss.**PSS1**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Stabilizer ST2CUT with dual input signals

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

set_flag(*self*, *value*, *flag*, *reset_val=False*)
Set a flag to 0 if the corresponding value is 0

update_ctrl(*self*)

class andes.models.pss.**PSS2**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Stabilizer IEEEST

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

update_ctrl(*self*)

andes.models.pv module

class andes.models.pv.**PV**(*system*, *name*)
Bases: *andes.models.pv.Stagen*

Static PV generator for power flow

disable_gen(*self*, *idx*)
Disable a PV element for TDS

Parameters

idx

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init0(*self*, *dae*)
Set initial voltage and reactive power for PQ. Overwrites Bus.voltage values

jac0(*self*, *dae*)

class andes.models.pv.**Slack**(*system*, *name*)
Bases: *andes.models.pv.PV*

Static slack generator

gcall (*self*, *dae*)

init0 (*self*, *dae*)
Set initial voltage and reactive power for PQ. Overwrites Bus.voltage values

jac0 (*self*, *dae*)

class andes.models.pv.**Stagen** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Static generator base class

andes.models.recorder module

class andes.models.recorder.**Recorder** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Recorder class that outputs simulation data of the given model, variable and elements

define (*self*)
Hook function where derived models define parameters, variables, service constants, and equations

Returns

elem_add (*self*, *idx=None*, *name=None*, ***kwargs*)
Add an element of this model

Parameters

idx [str, int or float, optional] The unique external reference to this element.

name [str, optional] The name of this element. Automatically assigned if not provided.

kwargs A dictionary containing the (field, value) pairs of the element parameters

Returns

str or int The allocated *idx* of the added element

init1 (*self*, *dae*)

andes.models.series module

class andes.models.series.**SeriesBase** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Base class for AC series devices

andes.models.shunt module

class andes.models.shunt.**Shunt** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Static shunt device

full_y (*self*, *Y*)
Add self(shunt) into full Jacobian Y

gcall (*self*, *dae*)

gycall (*self*, *dae*)

andes.models.synchronous module

Synchronous generator classes

class andes.models.synchronous.**Flux0**
Bases: object

The simplified flux model as an appendix to generator models. 0 = ra*id + psiq + vd 0 = ra*iq - psid + vq

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**Flux2**
Bases: object

Full electromagnetic transient of flux linkage d(psid) = wb * (ra * id + omega * psiq + vd) d(psiq) = wb * (ra * iq - omega * psid + vq)

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**Ord2**(*system*, *name*)
Bases: *andes.models.synchronous.SynBase*

2nd order classical model

build_service(*self*)
Build service variables

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**Ord6a**(*system*, *name*)
Bases: *andes.models.synchronous.SynBase*

6th order the Marconato's Model

build_service(*self*)
Build service variables

fcall(*self*, *dae*)

gcall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**Syn2**(*system*, *name*)
Bases: *andes.models.synchronous.Ord2*, *andes.models.synchronous.Flux0*

2nd-order generator model. Inherited from (Ord2, Flux0)

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**Syn6a**(*system*, *name*)
Bases: *andes.models.synchronous.Ord6a*, *andes.models.synchronous.Flux0*

GENROU, 6th-order generator model with xd2 = xq2.

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.synchronous.**SynBase**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Base class for synchronous generators

build_service(*self*)
Build service variables

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

saturation(*self*, *e1q*)
Saturation characteristic function

set_vf0(*self*, *vf*)
set value for self.vf0 and dae.y[self.vf]

andes.models.vsc module

class andes.models.vsc.**Current1**(*system*, *name*)
Bases: object

Inner current controllers with two PIs

current_fcall(*self*, *dae*)

current_fxcall(*self*, *dae*)

current_gcall(*self*, *dae*)

current_gycall(*self*, *dae*)

current_init1(*self*, *dae*)

current_jac0(*self*, *dae*)

current_servcall(*self*, *dae*)

class andes.models.vsc.**PLL1**(*system*, *name*)
Bases: object

Ideal tracking PLL

pll_fcall(*self*, *dae*)

pll_fxcall(*self*, *dae*)

pll_gcall(*self*, *dae*)

pll_gycall(*self*, *dae*)

pll_init1(*self*, *dae*)

pll_jac0(*self*, *dae*)

class andes.models.vsc.**Power0**(*system*, *name*)
Bases: object

Empty class for inertia emulation control - place holder

power_fcall(*self*, *dae*)

power_fxcall(*self*, *dae*)

power_gcall(*self*, *dae*)

power_gycall(*self*, *dae*)

power_init1(*self*, *dae*)

power_jac0(*self*, *dae*)

class andes.models.vsc.**Power1**(*system*, *name*)
Bases: object

Inertia emulation control device that modifies Pref based on the derivative of frequency deviation

power_fcall(*self*, *dae*)

power_fxcall(*self*, *dae*)

power_gcall(*self*, *dae*)

power_gycall(*self*, *dae*)

power_init1(*self*, *dae*)

power_jac0(*self*, *dae*)

class andes.models.vsc.**Power2**(*system*, *name*)
Bases: object

Inertia emulation control device that modifies Pref based on the derivative of frequency deviation

power_fcall (*self*, *dae*)

power_fxcall (*self*, *dae*)

power_gcall (*self*, *dae*)

power_gycall (*self*, *dae*)

power_init1 (*self*, *dae*)

power_jac0 (*self*, *dae*)

class andes.models.vsc.**VSC** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

VSC model for power flow study

disable (*self*, *idx*)
Disable an element and reset the outputs

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init0 (*self*, *dae*)

jac0 (*self*, *dae*)

setup (*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

switch (*self*, *idx*, *control*)
Switch a single control of <idx>

class andes.models.vsc.**VSC1** (*system*, *name*)
Bases: *andes.models.vsc.VSC1_Common*, *andes.models.vsc.VSC1_Outer1*, *andes.models.vsc.Current1*, *andes.models.vsc.PLL1*, *andes.models.vsc.Power0*

class andes.models.vsc.**VSC1_Common** (*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

Common equations for VSC1

fcall (*self*, *dae*)

fxcall (*self*, *dae*)

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

servcall (*self*, *dae*)

class andes.models.vsc.**VSC1_IE** (*system*, *name*)
Bases: *andes.models.vsc.VSC1_Common*, *andes.models.vsc.VSC1_Outer1*, *andes.models.vsc.Current1*, *andes.models.vsc.PLL1*, *andes.models.vsc.Power1*

VSC1 with Inertia Emulation with frequency sensor at local bus

class andes.models.vsc.**VSC1_IE2** (*system*, *name*)
Bases: *andes.models.vsc.VSC1_Common*, *andes.models.vsc.VSC1_Outer1*, *andes.models.vsc.Current1*, *andes.models.vsc.PLL1*, *andes.models.vsc.Power2*

VSC1 with Inertia Emulation with frequency sensor at remote bus

class andes.models.vsc.**VSC1_Outer1**(*system*, *name*)
Bases: object

Outer power control loop for VSC1

outer_fcall(*self*, *dae*)

outer_fxcall(*self*, *dae*)

outer_gcall(*self*, *dae*)

outer_gycall(*self*, *dae*)

outer_init1(*self*, *dae*)

outer_jac0(*self*, *dae*)

outer_servcall(*self*, *dae*)

class andes.models.vsc.**VSC2A**(*system*, *name*)
Bases: *andes.models.vsc.VSC2_Common*, *andes.models.vsc.Current1*, *andes.models.vsc.VSC2_Speed1*, *andes.models.vsc.VSC2_Voltage1*

Voltage-source controlled VSC with active power droop

class andes.models.vsc.**VSC2B**(*system*, *name*)
Bases: *andes.models.vsc.VSC2_Common*, *andes.models.vsc.Current1*, *andes.models.vsc.VSC2_Speed2*, *andes.models.vsc.VSC2_Voltage1*

Voltage-source controlled VSC with inertia emulation

class andes.models.vsc.**VSC2_Common**(*system*, *name*)
Bases: *andes.models.dcbase.DCBase*

Common equations for voltage-source controlled VSC

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

servcall(*self*, *dae*)

class andes.models.vsc.**VSC2_Speed1**(*system*, *name*)
Bases: object

Active power droop speed control

speed_fcall(*self*, *dae*)

speed_fxcall(*self*, *dae*)

speed_gcall(*self*, *dae*)

speed_gycall(*self*, *dae*)

speed_init1(*self*, *dae*)

speed_jac0(*self*, *dae*)

class andes.models.vsc.**VSC2_Speed2**(*system*, *name*)
Bases: object

Inertia emulation speed control

speed_fcall(*self*, *dae*)

speed_fxcall(*self*, *dae*)

speed_gcall(*self*, *dae*)

speed_gycall(*self*, *dae*)

speed_init1(*self*, *dae*)

speed_jac0(*self*, *dae*)

speed_servcall(*self*, *dae*)

class andes.models.vsc.**VSC2_Voltage1**(*system*, *name*)
Bases: object

Outer voltage controller for voltage-source controlled VSC using two PIs

voltage_fcall(*self*, *dae*)

voltage_fxcall(*self*, *dae*)

voltage_gcall(*self*, *dae*)

voltage_gycall(*self*, *dae*)

voltage_init1(*self*, *dae*)

voltage_jac0(*self*, *dae*)

andes.models.wind module

Wind power classes

class andes.models.wind.**ConstWind**(*system*, *name*)
Bases: *andes.models.wind.WindBase*

Constant wind power class

generate(*self*, *dae*)
Generate the wind speed time and data points

class andes.models.wind.**Weibull**(*system*, *name*)
Bases: *andes.models.wind.WindBase*

Weibull distribution wind speed class

generate(*self*, *dae*)
Generate the wind speed time and data points

class andes.models.wind.**WindBase**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Base class for wind time series

fcall(*self*, *dae*)

gcall(*self*, *dae*)

generate (*self*, *dae*)
Generate the wind speed time and data points

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

servcall (*self*, *dae*)

setup (*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

windspeed (*self*, *t*)
Return the wind speed list at time *t*

andes.models.wind.**uniform**(*low=0.0*, *high=1.0*, *size=None*)
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Note: New code should use the `uniform` method of a `default_rng()` instance instead; see *random-quick-start*.

Parameters

low [float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

high [float or array_like of floats] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

size [int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns

out [ndarray or scalar] Drawn samples from the parameterized uniform distribution.

See also:

randint Discrete uniform distribution, yielding integers.

random_integers Discrete uniform distribution over the closed interval `[low, high]`.

random_sample Floats uniformly distributed over `[0, 1)`.

random Alias for *random_sample*.

rand Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

Generator.uniform which should be used for new code.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

andes.models.windturbine module

class andes.models.windturbine.**MPPT**(*system*, *name*)
Bases: object

MPPT control algorithm

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1(*self*, *dae*)

jac0(*self*, *dae*)

class andes.models.windturbine.**Turbine**(*system*, *name*)
Bases: object

Generic wind turbine model

fcall(*self*, *dae*)

fxcall(*self*, *dae*)

gcall(*self*, *dae*)

gycall(*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

phi

servcall (*self*, *dae*)

windpower (*self*, *ngen*, *rho*, *vw*, *Ar*, *R*, *omega*, *theta*, *derivative=False*)

class andes.models.windturbine.**WTG3** (*system*, *name*)
Bases: *andes.models.base.ModelBase*

Wind turbine type III

fcall (*self*, *dae*)

fxcall (*self*, *dae*)

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init1 (*self*, *dae*)
New initialization function

jac0 (*self*, *dae*)

phi

servcall (*self*, *dae*)

windpower (*self*, *ngen*, *rho*, *vw*, *Ar*, *R*, *omega*, *theta*, *derivative=False*)

class andes.models.windturbine.**WTG4DC** (*system*, *name*)
Bases: *andes.models.base.ModelBase*, *andes.models.windturbine.Turbine*, *andes.models.windturbine.MPPT*

Wind turbine type IV DC output

fcall (*self*, *dae*)

fxcall (*self*, *dae*)

gcall (*self*, *dae*)

gycall (*self*, *dae*)

init1 (*self*, *dae*)

jac0 (*self*, *dae*)

servcall (*self*, *dae*)

v12

andes.models.zone module

class andes.models.zone.**Area** (*system*, *name*)
Bases: *andes.models.zone.Zone*

Area class

setup (*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

class andes.models.zone.**Region**(*system*, *name*)
Bases: *andes.models.zone.Zone*

Region class

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

class andes.models.zone.**Zone**(*system*, *name*)
Bases: *andes.models.base.ModelBase*

Zone class

interchange_varout(*self*)

seriesflow(*self*, *dae*)

setup(*self*)
Set up empty class structure and allocate empty memory for variable addresses.

Returns None

Module contents

andes.routines package

Submodules

andes.routines.base module

class andes.routines.base.**RoutineBase**
Bases: object

Base class for Routines

post(*self*)
Post processing for routine

Returns

bool Success flag

pre(*self*)
Pre-check for routine

Returns

bool Success flag

report(*self*)
Format report from the results

Returns

str Output string

reset(*self*)
Reset internal states of the routine

Returns

None

run(*self*, ***kwargs*)
Entry function for power flow routine

Returns

bool Success flag

andes.routines.eig module

class andes.routines.eig.**EIG**(*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Eigenvalue analysis routine

calc_eigvals(*self*)
Solve eigenvalues of the state matrix `self.As`

Returns

None

calc_part_factor(*self*)
Compute participation factor of states in eigenvalues

calc_state_matrix(*self*)
Return state matrix and store to `self.As`

Returns

matrix state matrix

dump_results(*self*)
Save eigenvalue analysis reports

Returns

None

plot_results(*self*)

run(*self*, ***kwargs*)
Entry function for power flow routine

Returns

bool Success flag

andes.routines.pflow module

class andes.routines.pflow.**PFLOW**(*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Power flow calculation routine

calc_inc(*self*)
Calculate the Newton incrementals for each step

Returns

matrix The solution to `x = -A\b`

dcpf (*self*)
Calculate linearized power flow

Returns

bool success flag, number of iterations

fdpf (*self*)
Fast Decoupled Power Flow

Returns

bool Success flag

newton (*self*)
Newton power flow routine

Returns

bool success flag

newton_call (*self*)
Function calls for Newton power flow

Returns

None

post (*self*)
Post processing for solved systems.

Store load, generation data on buses. Store reactive power generation on PVs and slack generators. Calculate series flows and area flows.

Returns

None

pre (*self*)
Initialize system for power flow study

Returns

None

reset (*self*)
Reset all internal storage to initial status

Returns

None

run (*self*, ***kwargs*)
call the power flow solution routine

Returns

bool True for success, False for fail

andes.routines.tds module

class andes.routines.tds.**TDS** (*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Time domain simulation (TDS) routine

angle_diff(*self*)
Compute the maximum angle difference between generators

Returns

float maximum angular difference

calc_time_step(*self*)
Set the time step during time domain simulations

Parameters

convergence: bool truth value of the convergence of the last step

niter: int current iteration count

t: float current simulation time

Returns

float computed time step size

check_fixed_times(*self*)
Check for fixed times and store in `self.fixed_times`.

Returns

None

compute_flows(*self*)
If enabled, compute the line flows after each step

Returns

None

dump_results(*self*, *success*)
Dump simulation results to `dat` and `lst` files

Returns

None

event_actions(*self*)
Take actions for timed events

Returns

None

implicit_step(*self*)
Integrate one step using trapezoidal method. Sets convergence and niter flags.

Returns

None

init(*self*)
Initialize time domain simulation

Returns

None

load_pert(*self*)
Load perturbation files to `self.callpert`

Returns

None

reset (*self*)
Reset internal states of the routine

Returns

None

restore_values (*self*)
Restore x, y, and f values if not converged

Returns

None

run (*self*, ***kwargs*)
Run time domain simulation

Returns

bool Success flag

run_step0 (*self*)
For the 0th step, store the data and stream data

Returns

None

streaming_init (*self*)
Send out initialization variables and process init from modules

Returns

None

streaming_step (*self*)
Sync, handle and streaming for each integration step

Returns

None

Module contents

class andes.routines.**PFLOW** (*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Power flow calculation routine

calc_inc (*self*)
Calculate the Newton incrementals for each step

Returns

matrix The solution to `x = -A\b`

dcpf (*self*)
Calculate linearized power flow

Returns

bool success flag, number of iterations

fdpf(*self*)
Fast Decoupled Power Flow

Returns

bool Success flag

newton(*self*)
Newton power flow routine

Returns

bool success flag

newton_call(*self*)
Function calls for Newton power flow

Returns

None

post(*self*)
Post processing for solved systems.

Store load, generation data on buses. Store reactive power generation on PVs and slack generators. Calculate series flows and area flows.

Returns

None

pre(*self*)
Initialize system for power flow study

Returns

None

reset(*self*)
Reset all internal storage to initial status

Returns

None

run(*self*, ***kwargs*)
call the power flow solution routine

Returns

bool True for success, False for fail

class andes.routines.**TDS**(*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Time domain simulation (TDS) routine

angle_diff(*self*)
Compute the maximum angle difference between generators

Returns

float maximum angular difference

calc_time_step(*self*)
Set the time step during time domain simulations

Parameters

convergence: bool truth value of the convergence of the last step

niter: int current iteration count

t: float current simulation time

Returns

float computed time step size

check_fixed_times(*self*)
Check for fixed times and store in `self.fixed_times`.

Returns

None

compute_flows(*self*)
If enabled, compute the line flows after each step

Returns

None

dump_results(*self*, *success*)
Dump simulation results to `dat` and `lst` files

Returns

None

event_actions(*self*)
Take actions for timed events

Returns

None

implicit_step(*self*)
Integrate one step using trapezoidal method. Sets convergence and niter flags.

Returns

None

init(*self*)
Initialize time domain simulation

Returns

None

load_pert(*self*)
Load perturbation files to `self.callpert`

Returns

None

reset(*self*)
Reset internal states of the routine

Returns

None

restore_values(*self*)
Restore x, y, and f values if not converged

Returns

None

run(*self*, ***kwargs*)
Run time domain simulation

Returns

bool Success flag

run_step0(*self*)
For the 0th step, store the data and stream data

Returns

None

streaming_init(*self*)
Send out initialization variables and process init from modules

Returns

None

streaming_step(*self*)
Sync, handle and streaming for each integration step

Returns

None

class andes.routines.**EIG**(*system*, *rc=None*)
Bases: *andes.routines.base.RoutineBase*

Eigenvalue analysis routine

calc_eigvals(*self*)
Solve eigenvalues of the state matrix `self.As`

Returns

None

calc_part_factor(*self*)
Compute participation factor of states in eigenvalues

calc_state_matrix(*self*)
Return state matrix and store to `self.As`

Returns

matrix state matrix

dump_results(*self*)
Save eigenvalue analysis reports

Returns

None

plot_results(*self*)

run(*self*, ***kwargs*)
Entry function for power flow routine

Returns

bool Success flag

andes.utils package

Submodules

andes.utils.altmath module

Create a wrapper for numpy array and scipy sparse to provide CVXOPT matrix interfaces

andes.utils.cached module

class andes.utils.cached.**cached**(*func*, *name=None*, *doc=None*)
Bases: object

A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work. See for details: http://stackoverflow.com/questions/17486104/python-lazy-loading-of-class-attributes

andes.utils.math module

andes.utils.math.**aandb**(*a*, *b*)
Return a matrix of logic comparison of A or B

andes.utils.math.**aeqb**(*a*, *b*)
Return a matrix of logic comparison of A == B

andes.utils.math.**ageb**(*a*, *b*)
Return a matrix of logic comparision of A>=B

andes.utils.math.**agtb**(*a*, *b*)
Return a matrix of logic comparision of A>B

andes.utils.math.**aleb**(*a*, *b*)
Return a matrix of logic comparison of A<=B

andes.utils.math.**altb**(*a*, *b*)
Return a matrix of logic comparison of A<B

andes.utils.math.**aneb**(*a*, *b*)
Return a matrix of logic comparison of A != B

andes.utils.math.**aorb**(*a*, *b*)
Return a matrix of logic comparison of A or B

andes.utils.math.**conj**(*a*)
return the conjugate of a

andes.utils.math.**index**(*m*, *val*)
Return the indices of all the `val` in `m`

andes.utils.math.**mfloor**(*a*)
Return the element-wise floor value of a

andes.utils.math.**mmax**(*a*, *b*)
Return a matrix of maximum values in a and b element-wise

andes.utils.math.**mmin**(*a*, *b*)
Return a matrix of minimum values in a and b element-wise

andes.utils.math.**mround**(*a*)
Return the element-wise round value of a

andes.utils.math.**neg**(*u*)
Return the negative of binary states u

andes.utils.math.**not0**(*a*)
Return u if u!= 0, return 1 if u == 0

andes.utils.math.**nota**(*a*)
Return a matrix of logic negative of A

andes.utils.math.**ones**(*m*, *n*)
Return a m-by-n one-value matrix

andes.utils.math.**polar**(*m*, *a*)
Return complex number from polar form m*exp(1j*a)

andes.utils.math.**sdiv**(*a*, *b*)
Safe division: if a == b == 0, sdiv(a, b) == 1

andes.utils.math.**sign**(*a*)
Return the sign of a in (1, -1, 0)

andes.utils.math.**sort**(*m*, *reverse=False*)
Return sorted m (default: ascending order)

andes.utils.math.**sort_idx**(*m*, *reverse=False*)
Return the indices of m in sorted order (default: ascending order)

andes.utils.math.**to_number**(*s*)
Convert a string to a number. If not successful, return the string without blanks

andes.utils.math.**zeros**(*m*, *n*)
Return a m-by-n zero-value matrix

andes.utils.misc module

andes.utils.misc.**get_config_load_path**(*conf_path=None*)
Return config file load path

Priority:

1. conf_path
2. current directory
3. home directory

Parameters

conf_path

andes.utils.misc.**get_log_dir**()
Get a directory for logging

On Linux or macOS, '/tmp/andes' is the default. On Windows, '%APPDATA%/andes' is the default.

Returns

str Path to the logging directory

andes.utils.solver module

class andes.utils.solver.**Solver**(*sparselib='umfpack'*)
Bases: object

Sparse matrix solver class. Wraps UMFPACK and KLU with a unified interface.

linsolve(*self*, *A*, *b*)
Solve linear equation set `Ax = b` and store the solutions in `b`.

Parameters

A Sparse matrix

b RHS of the equation

Returns

None

numeric(*self*, *A*, *F*)
Return the numeric factorization of sparse matrix `A` using symbolic factorization `F`

Parameters

A Sparse matrix

F Symbolic factorization

Returns

N Numeric factorization of `A`

solve(*self*, *A*, *F*, *N*, *b*)
Solve linear system `Ax = b` using numeric factorization `N` and symbolic factorization `F`. Store the solution in `b`.

Parameters

A Sparse matrix

F Symbolic factorization

N Numeric factorization

b RHS of the equation

Returns

None

symbolic(*self*, *A*)
Return the symbolic factorization of sparse matrix `A`

Parameters

sparselib Library name in `umfpack` and `klu`

A Sparse matrix

Returns

symbolic factorization

——-

andes.utils.stock_case module

Utility functions for loading andes stock test cases

`andes.utils.stock_case.`**`get_stock_case`**(*rpath*)
Return the path to the stock cases

`andes.utils.stock_case.`**`stock_case_root`**()
Return the root path to the stock cases

andes.utils.tab module

class `andes.utils.tab.`**`Tab`**(*title=None*, *header=None*, *descr=None*, *data=None*, *export='plain'*)
Bases: `andes.utils.texttable.Texttable`

Use package texttable to create fine-formatted tables for setting helps and device helps. Avoid using this class for static report formatting as it may be slow.

`add_left_space`(*self*, *nspace=1*)
elem_add n cols of spaces before the first col. (for texttable 0.8.3)

`auto_style`(*self*)
automatic styling according to _row_size 76 characters in a row

`draw`(*self*)
generate texttable formatted string

`guess_header`(*self*)

`header`(*self*, *array*)
Specify the header of the table

`set_title`(*self*, *val*)

class `andes.utils.tab.`**`simpletab`**(*header=None*, *title=None*, *data=None*, *side=None*)
Bases: `object`

A simple and faster table class for static report output

`draw`(*self*)

`guess_width`(*self*)
auto fit column width

andes.utils.time module

`andes.utils.time.`**`elapsed`**(*t0=0.0*)
get elapsed time from the give time

Returns: now: the absolute time now dt_str: elapsed time in string

Module contents

class andes.utils.**cached**(*func*, *name=None*, *doc=None*)
Bases: object

A decorator that converts a function into a lazy property. The function wrapped is called the first time to retrieve the result and then that calculated result is used the next time you access the value:

```
class Foo(object):

    @cached_property
    def foo(self):
        # calculate something important here
        return 42
```

The class has to have a *__dict__* in order for this property to work. See for details: http://stackoverflow.com/questions/17486104/python-lazy-loading-of-class-attributes

andes.utils.**get_config_load_path**(*conf_path=None*)
Return config file load path

Priority:

1. conf_path
2. current directory
3. home directory

Parameters

conf_path

andes.variables package

Submodules

andes.variables.call module

class andes.variables.call.**Call**(*system*)
Bases: object

Equation call mamager class for andes routines

build_vec(*self*)
build call validity vector for each device

bus_injection(*self*)

fdpf(*self*)

gisland(*self*)

gyisland(*self*)

int(*self*)

int_fg(*self*)

int_fxgy(*self*)

pfgen(*self*)

pfload(*self*)

seriesflow(*self*)

setup(*self*)
setup the call list after case file is parsed and jit models are loaded

andes.variables.dae module

class andes.variables.dae.**DAE**(*system*)
Bases: object

Class for numerical Differential Algebraic Equations (dae)

add_jac(*self*, *m*, *val*, *row*, *col*)
Add tuples (val, row, col) to the Jacobian matrix `m`

Implemented in numpy.arrays for temporary storage.

anti_windup(*self*, *xidx*, *xmin*, *xmax*)
Anti-windup limiter for state variables.

Resets the limited variables and differential equations.

Parameters

- **xidx** (*matrix, list*) – state variable indices
- **xmin** (*matrix, float, int, list*) – lower limit
- **xmax** (*matrix, float, int, list*) – upper limit

apply_set(*self*, *ty*)
Apply Jacobian set values to matrices

Parameters **ty** – Jacobian type in `('jac0', 'jac')`

Returns

check_diag(*self*, *jac*, *name*)
Check matrix `jac` for diagonal elements that equals 0

find_val(*self*, *eq*, *val*)
Return the name of the equation having the given value

get_size(*self*, *m*)
Return the 2-D size of a Jacobian matrix in tuple

hard_limit(*self*, *yidx*, *ymin*, *ymax*, *min_set=None*, *max_set=None*)
Set hard limits for algebraic variables and reset the equation mismatches

Parameters

- **yidx** (*list, matrix*) – algebraic variable indices
- **ymin** (*matrix, int, float, list*) – lower limit to check for
- **ymax** (*matrix, int, float, list*) – upper limit to check for
- **min_set** (*matrix*) – optional lower limit to set (`ymin` as default)
- **max_set** (*matrix*) – optional upper limit to set (`ymax` as default)

Returns None

hard_limit_remote (*self*, *yidx*, *ridx*, *rtype='y'*, *rmin=None*, *rmax=None*, *min_yset=0*, *max_yset=0*)
Limit the output of yidx if the remote y is not within the limits

TODO: This function needs to be refactored for clearliness.

init1 (*self*)

init_Fx0 (*self*)

init_Gy0 (*self*)

init_f (*self*, *resetz=True*)

init_fg (*self*, *resetz=False*)

init_g (*self*, *resetz=True*)

init_jac0 (*self*)

init_x (*self*)

init_xy (*self*)

init_y (*self*)

reset_Ac (*self*)
Reset `dae.Ac` sparse matrix for disabled equations due to hard_limit and anti_windup limiters.

Returns None

reset_small (*self*, *eq*)
Reset numbers smaller than 1e-12 in f and g equations

reset_small_f (*self*)

reset_small_g (*self*)

resize (*self*)
Resize dae and and extend for init1 variables

set_jac (*self*, *m*, *val*, *row*, *col*)
Set the values at (row, col) to val in Jacobian m

Parameters

- **m** – Jacobian name
- **val** – values to set
- **row** – row indices
- **col** – col indices

Returns None

setup (*self*)

setup_Fx (*self*)

setup_FxGy (*self*)

setup_Gy (*self*)

show (*self*, *eq*, *value=None*)
Show equation or variable array along with the names

temp_to_spmatrix (*self*, *ty*)
Convert Jacobian tuples to matrices

Parameters **ty** – name of the matrices to convert in `('jac0','jac')`

Returns None

andes.variables.devman module

class andes.variables.devman.**DevMan**(*system=None*)
Bases: object

Device Manager class. Maintains the loaded model list, groups and categories

get_param(*self*, *group*, *param*, *fkey*)

register_device(*self*, *dev_name*)
register a device to the device list

register_element(*self*, *dev_name*, *idx=None*)
Register a device element to the group list

Parameters

dev_name [str] model name

idx [str] element idx

Returns

str assigned idx

sort_device(*self*)
Sort device to follow the order of initialization

Returns None

andes.variables.fileman module

class andes.variables.fileman.**FileMan**(*case*, *input_format=None*, *input_path=None*, *addfile=None*, *config=None*, *no_output=False*, *dynfile=None*, *dump_raw=None*, *output_format=None*, *output_path=''*, *output=None*, *pert=None*, ***kwargs*)
Bases: object

Define a File Manager class for PowerSystem

get_fullpath(*self*, *fullname=None*, *relative_to=None*)
Return the original full path if full path is specified, otherwise search in the case file path

andes.variables.fileman.**add_suffix**(*fullname*, *suffix*)
Add suffix to a full file name

andes.variables.report module

class andes.variables.report.**Report**(*system*)
Bases: object

Report class to store system static analysis reports

info

update (*self*, *content=None*)
Update values based on the requested content

Parameters

content

write (*self*, *content=None*)
Write report to file.

Parameters

content: str 'summary', 'extended', 'powerflow'

andes.variables.varname module

class andes.variables.varname.**VarName** (*system*)
Bases: object

Variable name manager class

append (*self*, *listname*, *xy_idx*, *var_name*, *element_name*)
Append variable names to the name lists

bus_line_names (*self*)
Append bus injection and line flow names to *varname*

fname
Return the full formatted variable name list following the order of state vars, algeb vars and line flow vars

Returns list of formatted names

get_xy_name (*self*, *yidx*, *xidx=0*)
Return variable names for the given indices

Parameters

- **yidx** –
- **xidx** –

Returns

resize (*self*)
Resize (extend) the list for variable names

resize_for_flows (*self*)
Extend *unamey* and *fnamey* for bus injections and line flows

uname
Return the full unformatted variable name list following the order of state vars, algeb vars and line flow vars

Returns list of unformatted names

andes.variables.varout module

class andes.variables.varout.**VarOut** (*system*)
Bases: object

Output variable value recorder

TODO: merge in tds.py

concat_t_vars(*self*)
Concatenate `self.t` with `self.vars` and output a single matrix for data dump

Return matrix concatenated matrix with `self.t` as the 0-th column

concat_t_vars_np(*self*, *vars_idx=None*)
Concatenate *self.np_t* with *self.np_vars* and return a single matrix. The first column corresponds to time, and the rest of the matrix is the variables.

Returns

np.array [concatenated matrix]

dump(*self*)
Dump the TDS results to the output *dat* file

Returns succeed flag

dump_np_vars(*self*, *store_format='csv'*, *delimiter=','*)
Dump the TDS simulation data to files by calling subroutines *write_lst* and *write_np_dat*.

Parameters

store_format [str] dump format in *('csv', 'txt', 'hdf5')*

delimiter [str] delimiter for the *csv* and *txt* format

Returns

bool: success flag

get_latest_data(*self*)
Get the latest data and simulation time

Returns

dict [a dictionary containing *t* and *vars*]

get_xy(*self*, *yidx*, *xidx=0*)
Return stored data for the given indices for plot

Parameters

- **yidx** – the indices of the y-axis variables(1-indexing)
- **xidx** – the index of the x-axis variables

Returns None

show(*self*)
The representation of an Varout object

Returns the full result matrix (for use with PyCharm viewer)

Return type np.array

store(*self*, *t*, *step*)
Record the state/algeb values at time t to self.vars

vars_to_array(*self*)
Convert *self.vars* to a numpy array

Returns

numpy.array

write_dat(*self*)
Write `system.Varout.vars` to a `.dat` file :return:

write_lst(*self*)
Dump the variable name lst file

Returns succeed flag

write_np_dat(*self*, *store_format='csv'*, *delimiter=','*, *fmt='%.18e'*)
Write TDS data stored in *self.np_vars* to the output file

Parameters

store_format [str] dump format in ('csv', 'txt', 'hdf5')

delimiter [str] delimiter for the *csv* and *txt* format

fmt [str] output formatting template

Returns

bool [success flag]

Module contents

9.1.2 Submodules

9.1.3 andes.consts module

useful constants

9.1.4 andes.debug module

A dummy debug entry point for use with Pycharm

`andes.debug.`**debug**()

9.1.5 andes.main module

Andes main entry points

`andes.main.`**andeshelp**(*group=None*, *category=None*, *model_list=None*, *model_format=None*, *model_var=None*, *quick_help=None*, *help_option=None*, *help_config=None*, *export='plain'*, *data_example=None*, ***kwargs*)
Print the requested help and documentation to stdout.

Parameters

group [None or str] Name of a group whose model names will be printed

category [None or str] Name of a category whose models will be printed

model_list [bool] If `True`, print the full model list.

model_format [None or str] Names of models whose parameter definitions will be printed. Model names are separated by comma without space.

model_var [None or str] A pair of model name and parameter name separated by dot. For example, `Bus.voltage` stands for the `voltage` parameter of `Bus`.

quick_help [None or str] Name of a model to print a quick help of parameter definitions.

help_option [None or str] A pair of config name and option name separated by dot. For example, `System.sparselib` stands for the `sparselib` option of the `System` config.

help_config [None or str] Config names whose option definitions will be printed. Configs are separated by comma without space. For example, `System,Pflow`. In the naming convention, the first letter is captialized.

export [None or {'plain', 'latex'}] Formatting style available in plain text or LaTex format. This option has not been implemented.

data_example [str, optional] Print example data for a specified model. The data can be used to construct a dm data file.

kwargs [None or dict] Other keyword arguments

Returns

bool True if any help function is executed.

Notes

The outputs can be written to a text file using shell command, for example,

```
andes -q Bus > bus_help.txt
```

andes.main.**cli_new**(*parser*)

Do parse with the provided CLI parser

Parameters

parser [ArgumentParser] An ArgumentParser instance for parsing the command line arguments

Returns

Namespace A namespace containing the parsed arguments

andes.main.**cli_parser**()

Construct a CLI argument parser and return the parsed arguments.

Returns

ArgumentParser An argument parser for parsing command-line arguments

andes.main.**config_logger**(*logger=None*, *name='andes'*, *log_file='andes.log'*, *log_path=''*, *stream=True*, *file=False*, *stream_level=20*, *file_level=10*)

Configure a logger for the andes package with options for a *FileHandler* and a *StreamHandler*. This function is called at the beginning of `andes.main.main()`.

Parameters

name [str, optional] Base logger name, `andes` by default. Changing this parameter will affect the loggers in modules and cause unexpected behaviours.

log_file [str, optional] Logg file name for *FileHandler*, `'andes.log'` by default. If `None`, the *FileHandler* will not be created.

log_path [str, optional] Path to store the log file. By default, the path is generated by get_log_dir() in utils.misc.

stream [bool, optional] Create a *StreamHandler* for *stdout* if `True`. If `False`, the handler will not be created.

stream_level [{10, 20, 30, 40, 50}, optional] *StreamHandler* verbosity level.

Returns

None

andes.main.**edit_conf**(*edit_config=''*, *load_config=None*, ***kwargs*)
Edit the Andes config file which occurs first in the search path.

Parameters

edit_config [bool] If `True`, try to open up an editor and edit the config file. Otherwise returns.

load_config [None or str, optional] Path to the config file, which will be placed to the first in the search order.

kwargs [dict] Other keyword arguments.

Returns

bool `True` is a config file is found and an editor is opened. `False` if `edit_config` is False.

andes.main.**main**(*args=None*)
The main function of the Andes command-line tool.

This function executes the following workflow:

- Parse the command line inputs
- Show the tool preamble
- Output the requested helps, edit/save configs or remove outputs. Exit the main program if any of the above is executed
- Process the input files and call `main.run()` using single- or multi-processing
- Show the execution time and exit

Returns

None

andes.main.**preamble**()
Log the Andes command-line preamble at the *logging.INFO* level

Returns

None

andes.main.**remove_output**(*clean=False*, ***kwargs*)
Remove the outputs generated by Andes, including power flow reports `_out.txt`, time-domain list `_out.lst` and data `_out.dat`, eigenvalue analysis report `_eig.txt`.

Parameters

clean [bool] If `True`, execute the function body. Returns otherwise.

kwargs [dict] Other keyword arguments

Returns

bool `True` is the function body executes with success. `False` otherwise.

andes.main.**run**(*case*, *routine=None*, *profile=False*, *dump_raw=False*, *pid=-1*, *show_data=None*, *exit_now=False*, ***kwargs*)
Entry function to run a single case study. This function executes the following workflow:

- Turn on cProfile if requested

- Populate a `PowerSystem` object
- Parse the input files using filters
- Dump the case file is requested
- Set up the system
- Run the specified routine(s)

Parameters

case [str] Path to the case file

profile [bool, optional] Enable cProfile if `True`.

dump_raw [`False` or str, optional] Path to the file to dump the system parameters in the dm format

routine [Iterable, optional] A list of routines to run in sequence (`('pflow')` as default)

pid [idx, optional] Process idx of the current run

kwargs [dict, optional] Other keyword arguments

Returns

PowerSystem Andes PowerSystem object

`andes.main.`**`run_stock`**(*rpath*, *routine=None*, *profile=False*, *dump_raw=False*, *pid=-1*, *show_data=None*, *exit_now=False*, ***kwargs*)

Run a stock case distributed with andes

`andes.main.`**`save_config`**(*save_config=''*, ***kwargs*)

Save the Andes config to a file at the path specified by `save_config`. The save action will not run if *save_config = ''*.

Parameters

save_config [None or str, optional, ('' by default)] Path to the file to save the config file. If the path is an emtpy string, the save action will not run. Save to *~/.andes/andes.conf* if `None`.

kwargs [dict, optional] Other keyword arguments

Returns

bool `True` is the save action is run. `False` otherwise.

`andes.main.`**`search`**(***kwargs*)

Search for models whose names matches the given pattern. Print the results to stdout.

Deprecated since version 1.0.0: *search* will be moved to `andeshelp` in future versions.

Parameters

search [str] Partial or full name of the model to search for

kwargs [dict] Other keyword arguments.

Returns

list The list of model names that match the given pattern.

9.1.6 andes.plot module

Andes plotting tool

class andes.plot.**TDSData**(*file_name_full*, *path=None*)
Bases: object

A time-domain simulation data container for loading, extracing and plotting data

data_to_df(*self*)
Convert to pandas.DataFrame

export_csv(*self*, *path*, *idx=None*, *header=None*, *formatted=False*, *sort_idx=True*, *fmt='%.18e'*)
Export to a csv file

Parameters

path [str] path of the csv file to save

idx [None or array-like, optional] the indices of the variables to export. Export all by default

header [None or array-like, optional] customized header if not *None*. Use the names from the lst file by default

formatted [bool, optional] Use LaTeX-formatted header. Does not apply when using customized header

sort_idx [bool, optional] Sort by idx or not, # TODO: implement sort

fmt [str] cell formatter

find_var_idx(*self*, *query*, *exclude=None*, *formatted=False*)
Return variable names and indices matching *query*

Parameters

query [str] The string for querying variables

exclude [str, optional] A string pattern to be excluded

formatted [bool, optional] True to return formatted names, False otherwise

Returns

(list, list) (List of found indices, list of found names)

get_header(*self*, *idx*, *formatted=False*)
Return a list of the variable names at the given indices

Parameters

idx [list or int] The indices of the variables to retrieve

formatted [bool] True to retrieve latex-formatted names, False for unformatted names

Returns

list A list of variable names (headers)

get_values(*self*, *idx*)
Return the variable values at the given indices

Parameters

idx [list] The indicex of the variables to retrieve. *idx=0* is for Time. Variable indices start at 1.

Returns

np.ndarray Variable data

guess_event_time(*self*)
Guess the event starting time from the input data by checking when the values start to change

load_dat(*self*, *delimiter=', '*)
Load the dat file into internal data structures *self._data*

Parameters

delimiter [str, optional] The delimiter for the case file. Default to comma.

Returns

None

load_lst(*self*)
Load the lst file into internal data structures *_idx*, *_fname*, *_uname*, and counts the number of variables to *nvars*

Returns

None

plot(*self*, *yidx*, *xidx=(0,)*, *y_calc=None*, *left=None*, *right=None*, *ymin=None*, *ymax=None*, *ytimes=None*, *xlabel=None*, *ylabel=None*, *legend=True*, *grid=False*, *latex=True*, *dpi=200*, *save=None*, *show=True*, ***kwargs*)
Entery function for plot scripting. This function retrieves the x and y values based on the *xidx* and *yidx* inputs and then calls *plot_data()* to do the actual plotting.

Note that *ytimes* and *y_calc* are applied sequentially if apply.

Refer to *plot_data()* for the definition of arguments.

Parameters

xidx [list or int] The index for the x-axis variable

yidx [list or int] The indices for the y-axis variables

Returns

(fig, ax) Figure and axis handles

plot_data(*self*, *xdata*, *ydata*, *xheader=None*, *yheader=None*, *xlabel=None*, *ylabel=None*, *left=None*, *right=None*, *ymin=None*, *ymax=None*, *legend=True*, *grid=False*, *fig=None*, *ax=None*, *latex=True*, *dpi=150*, *greyscale=False*, *save=None*, *show=True*, ***kwargs*)
Plot lines for the supplied data and options. This functions takes *xdata* and *ydata* values. If you provide variable indices instead of values, use *plot()*.

Parameters

xdata [array-like] An array-like object containing the values for the x-axis variable

ydata [array] An array containing the values of each variables for the y-axis variable. The row of *ydata* must match the row of *xdata*. Each column correspondings to a variable.

xheader [list] A list containing the variable names for the x-axis variable

yheader [list] A list containing the variable names for the y-axis variable

xlabel [str] A label for the x axis

ylabel [str] A label for the y axis

left [float] The starting value of the x axis

right [float] The ending value of the x axis

ymin [float] The minimum value of the y axis

ymax [float] The maximum value of the y axis

legend [bool] True to show legend and False otherwise

grid [bool] True to show grid and False otherwise

fig Matplotlib fig object to draw the axis on

ax Matplotlib axis object to draw the lines on

latex [bool] True to enable latex and False to disable

dpi [int] Dots per inch for screen print or save

greyscale [bool] True to use greyscale, False otherwise

save [bool] True to save to png file

show [bool] True to show the image

kwargs Optional kwargs

Returns

(fig, ax) The figure and axis handles

andes.plot.**add_plot**(*x*, *y*, *xl*, *yl*, *fig*, *ax*, *LATEX=False*, *linestyle=None*, ***kwargs*)
Add plots to an existing plot

andes.plot.**check_init**(*yval*, *yl*)
"Check initialization by comparing t=0 and t=end values

andes.plot.**eig_plot**(*name*, *args*)

andes.plot.**isfloat**(*value*)

andes.plot.**isint**(*value*)

andes.plot.**label_texify**(*label*)
Convert a label to latex format by appending surrounding $ and escaping spaces

Parameters

label [str] The label string to be converted to latex expression

Returns

str A string with $ surrounding

andes.plot.**main**(*cli=True*, ***args*)

andes.plot.**parse_y**(*y*, *upper*, *lower=0*)
Parse command-line input for Y indices and return a list of indices

Parameters

y [Union[List, Set, Tuple]]

Input for Y indices. Could be single item (with or without colon), or multiple items

upper [int] Upper limit. In the return list y, y[i] <= uppwer.

lower [int] Lower limit. In the return list y, y[i] >= lower.

andes.plot.**scale_func**(*k*)
Return a lambda function that scales its input by k

Parameters

k [float] The scaling factor of the returned lambda function

Returns

——-

Lambda function

andes.plot.**set_latex**(*with_latex=True*)
Enables latex for matplotlib based on the *with_latex* option and *dvipng* availability

Parameters

with_latex [bool, optional] True for latex on and False for off

Returns

bool True for latex on and False for off

andes.plot.**tdsplot**(*datfile*, *y*, *x=(0,)*, ***kwargs*)
TDS plot main function based on the new TDSData class

Parameters

datfile [str] Path to the andes tds output data file (.dat or .lst file)

x [list or int, optional] The index for the x-axis variable. x=0 by default for time

y [list or int] The indices for the y-axis variable

Returns

TDSData object

andes.plot.**tdsplot_main**()
Entry function for tds plot. Parses command line arguments and calls *tdsplog*

Returns

None

andes.plot.**tdsplot_parse**()
command line input parser for tdsplot

Returns

dict A dict of the command line arguments

9.1.7 andes.system module

Power system class

class andes.system.**Group**(*system*, *name*)
Bases: object

Group class for registering models and elements.

Also handles reading and setting attributes.

get_field(*self*, *field*, *idx*)
Return the field `field` of elements `idx` in the group

Parameters

- **field** – field name
- **idx** – element idx

Returns values of the requested field

register_element(*self*, *model*, *idx*)
Register element with index `idx` to `model`

Parameters

- **model** – model name
- **idx** – element idx

Returns final element idx

register_model(*self*, *model*)
Register `model` to this group

Parameters **model** – model name

Returns None

set_field(*self*, *field*, *idx*, *value*)
Set the field `field` of elements `idx` to `value`.

This function does not if the field is valid for all models.

Parameters

- **field** – field name
- **idx** – element idx
- **value** – value of fields to set

Returns None

class andes.system.**GroupMeta**
Bases: `type`

class andes.system.**PowerSystem**(*case=''*, *pid=-1*, *no_output=False*, *dump_raw=None*, *output=None*, *dynfile=None*, *addfile=None*, *config=None*, *input_path=None*, *input_format=None*, *output_format=None*, *output_path=''*, *gis=None*, *dime=None*, *tf=None*, ***kwargs*)
Bases: `object`

A power system class to hold models, routines, DAE numerical values, file manager, call manager, variable names and valuable values.

check_event(*self*, *sim_time*)
Check for event occurrance for''Event'' group models at `sim_time`

Parameters

sim_time [float] The current simulation time

Returns

list A list of model names who report (an) event(s) at `sim_time`

check_islands(*self*, *show_info=False*)
Check the connectivity for the ac system

Parameters

show_info [bool] Show information when the system has islands. To be used when initializing power flow.

Returns

None

dump_config(*self*, *file_path*)
Dump system and routine configurations to an rc-formatted file.

Parameters

file_path [str] path to the configuration file. The user will be prompted if the file already exists.

Returns

None

freq
System base frequency

get_busdata(*self*, *sort_names=False*)
get ac bus data from solved power flow

get_data_example(*self*, *model*, *n_per_line=5*)
Return a string of example data entry that can be used in a dm input file

Returns

str A string containing the example data

get_event_times(*self*)
Return event times of Fault, Breaker and other timed events

Returns

list A sorted list of event times

get_linedata(*self*, *sort_names=False*)
get line data from solved power flow

get_nodedata(*self*, *sort_names=False*)
get dc node data from solved power flow

group_add(*self*, *name='Ungrouped'*)
Dynamically add a group instance to the system if not exist.

Parameters

name [str, optional ('Ungrouped' as default)] Name of the group

Returns

None

load_config(*self*, *conf_path*, *sys_only=False*, *routine_only=False*)
Load config from an `andes.conf` file.

This function creates a `configparser.ConfigParser` object to read the specified conf file and calls the `load_config` function of the config instances of the system and the routines.

Parameters

conf_path [None or str] Path to the Andes config file. If `None`, the function body will not run.

Returns

None

model_import (*self*)
Import and instantiate the non-JIT models and the JIT models.

Models defined in `jits` and `non_jits` in `models/__init__.py` will be imported and instantiated accordingly.

Returns

None

model_setup (*self*)
Call the `setup` function of the loaded models. This function is to be called after parsing all the data files during the system set up.

Returns

None

mva
System base power in mega voltage-ampere

rmgen (*self*, *idx*)
Remove the static generators if their dynamic models exist

Parameters

idx [list] A list of static generator idx

Returns

——-

None

routine_import (*self*)
Dynamically import routines as defined in `routines/__init__.py`.

The command-line argument `--routine` is defined in `__cli__` in each routine file. A routine instance will be stored in the system instance with the name being all lower case.

For example, a routine for power flow study should be defined in `routines/pflow.py` where `__cli__ = 'pflow'`. The class name for the routine should be `Pflow`. The routine instance will be saved to `PowerSystem.pflow`.

Returns

None

setup (*self*)
Set up the power system object by executing the following workflow:

- Sort the loaded models to meet the initialization sequence
- Create call strings for routines
- Call the `setup` function of the loaded models
- Assign addresses for the loaded models
- Call `dae.setup` to assign memory for the numerical dae structure
- Convert model parameters to the system base

Returns

PowerSystem The instance of the PowerSystem

to_elembase(*self*)
Convert parameters back to element base. This function calls the `data_to_elem_base` function.

Returns

None

to_sysbase(*self*)
Convert model parameters to system base. This function calls the data_to_sys_base function of the loaded models.

Returns

None

wb
System base radial frequency

xy_addr0(*self*)
Assign indicies and variable names for variables used in power flow

For each loaded model with the pflow flag as True, the following functions are called sequentially:

- _addr()
- _intf_network()
- _intf_ctrl()

After resizing the varname instance, variable names from models are stored by calling _varname()

Returns

None

xy_addr1(*self*)
Assign indices and variable names for variables after power flow. This function is for loaded models that do not have the pflow flag as True.

9.1.8 Module contents

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

Acknowledgement

This work was supported in part by the Engineering Research Center Program of the National Science Foundation and the Department of Energy under NSF Award Number EEC-1041877 and the CURENT Industry Partnership Program.

Python Module Index

a

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z