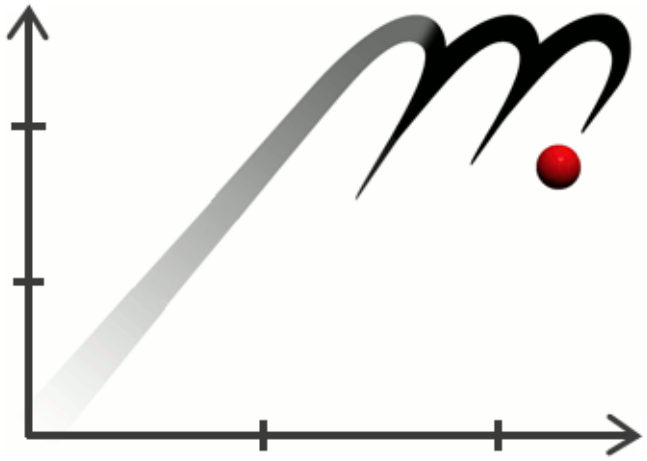


ModelicaRes Documentation

Release 0.10.0

Kevin Davies

April 30, 2014

CONTENTS

1 loadres 4

2 modelicares 6

3 modelicares.simres 7

4 modelicares.linres 20

5 modelicares.multi 24

6 modelicares.exps 30

7 modelicares.texunit 38

8 modelicares.base 41

Python Module Index 59

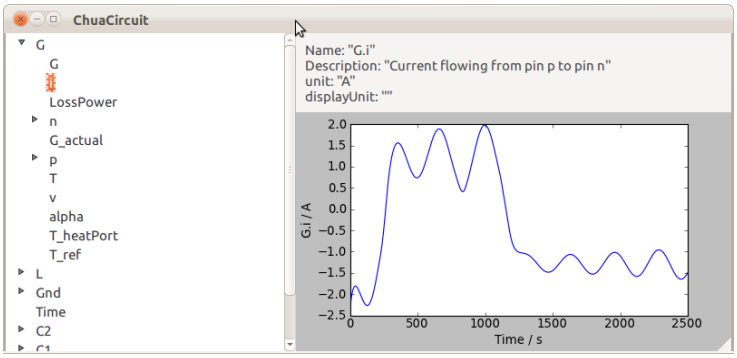
Index 61

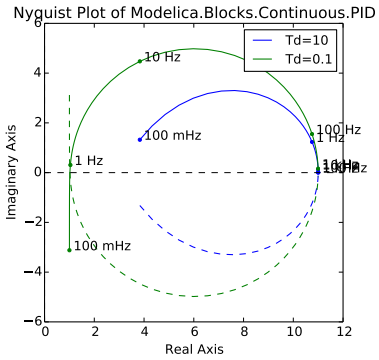
Python utilities to set up and analyze Modelica simulation experiments

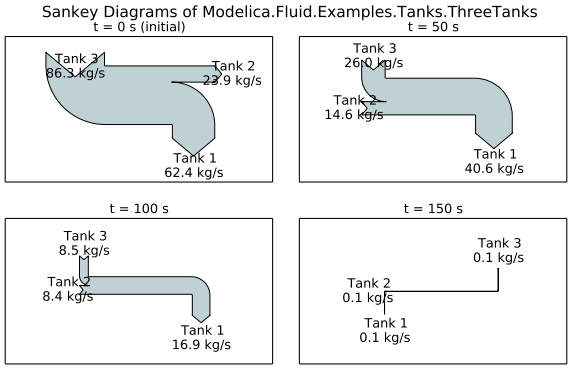
ModelicaRes is a free, open-source tool to manage Modelica simulations, interpret results, and create publishable figures. It is possible to

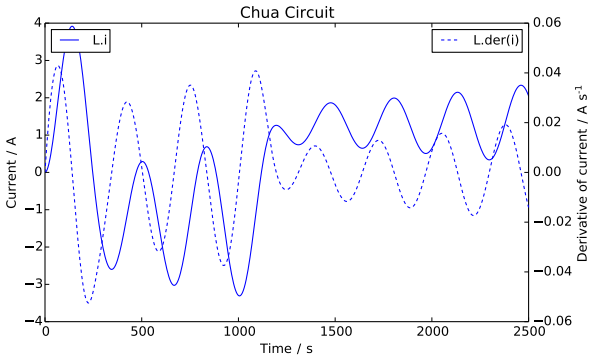
- Auto-generate simulation scripts,
- Browse data,
- Perform custom calculations, and
- Produce various plots and diagrams.

The figures are generated via matplotlib, which offers a rich set of plotting routines. ModelicaRes includes convenient functions to automatically pre-format and label some figures, like xy plots, Bode and Nyquist plots, and Sankey diagrams. ModelicaRes can be scripted or run from a Python interpreter with math and matrix functions from NumPy.









The following chapters describe the components of ModelicaRes. For an introduction, please see `loadres`, which loads data files and provides a Python interpreter to help analyze them.

The top-level module, `modelicares`, provides direct access to the most important classes and functions. Others must be accessed through their submodules. The `modelicares.simres` submodule has classes to load, analyze, and plot simulation results. The `modelicares.linres` submodule has a class to load, analyze, and plot results from linearizing a model. The `modelicares.multi` submodule has functions to load and plot results from multiple data files at once. The `modelicares.exps` submodule has tools to set up and manage simulation experiments. The `modelicares.texunit` submodule has functions to translate Modelica *unit* and *displayUnit* strings into LaTeX-formatted strings. The last submodule, `modelicares.base`, has general supporting functions.

Installation

The easiest way to install this package is to use pip:

```
pip install modelicares
```

On Linux, it may be necessary to have root privileges:

```
sudo pip install modelicares
```

Another way is to download and extract a copy of the package from the main project site, the master branch at GitHub, or the PyPI page. Run the following command from the base folder:

```
python setup.py install
```

Or, on Linux:

```
sudo python setup.py install
```

The *matplotlibrc* file in the base folder has some recommended revisions to matplotlib's defaults. To use it, copy or move it to the working directory or matplotlib's configuration directory. See http://matplotlib.org/users/customizing.html for details.

Credits

The main author is Kevin Davies. Improvements, bug fixes, and suggestions have been provided by Arnout Aertgeerts, Kevin Bandy, Thomas Beutlich, Martin Sjölund, Mike Tiller, and Michael Wetter.

Third-party code has been included from:

- Jason Grout (ArrowLine class),
- Jason Heeris (efficient base-10 logarithm),
- Richard Murray (python-control), and
- Joerg Raedler (method to expand a Modelica variable tree—from DyMat).

License terms and development

ModelicaRes is published under a BSD license (see LICENSE.txt). Please share any modifications you make (preferably on a Github fork from https://github.com/kdavies4/ModelicaRes) in order to help others. If you find a bug, please report it. If you have suggestions, please share them.

See also

The following Python projects are related:

- awesim: helps run simulation experiments and organize results
- BuildingsPy: supports unit testing
- DyMat: exports Modelica simulation data to comma-separated values (CSV), Gnuplot, MATLAB®, and Network Common Data Form (netCDF)
- PyFMI: tools to work with models through the Functional Mock-Up Interface (FMI) standard
- PySimulator: elaborate GUI; supports FMI

CHAPTER

ONE

LOADRES

Load results from Modelica simulation(s) and provide a Python interpreter to analyze the results.

This script can be executed at the command line. It will accept as arguments the names of result files or names of directories with result files. The filenames may contain wildcards. If no arguments are given, the script provides a dialog to choose a file or folder. Finally, it provides working session of IPython with the results preloaded. PyLab is directly imported (`from pylab import *`) to provide many functions of NumPy and matplotlib (e.g., `sin()` and `plot()`). The essential classes and functions of ModelicaRes are directly available as well.

Example:

To begin this example, copy this script (*loadres*) to the current directory along with the *examples* folder.

```
$ loadres examples
Valid: SimRes('.../examples/ChuaCircuit.mat')
Valid: SimRes('.../examples/ThreeTanks.mat')
Valid: LinRes('.../examples/PID.mat')
Simulation results have been loaded into sims[0] through sims[1].
A linearization result has been loaded into lin.
```

where '...' depends on the local system.

You can now explore the simulation results or create plots using the methods in `SimRes`. For example,

```
>>> sims[0].get_FV('L.v')
-0.25352862
>>> sims[0].get_unit('L.v')
'V'
```

If a variable cannot be found, then suggestions are given:

```
>>> sims[0].get_description('L.vv')
L.vv is not a valid variable name.

Did you mean one of the these?
       L.v
       L.p.v
       L.n.v
>>> sims[0].get_description('L.v')
'Voltage drop between the two pins (= p.v - n.v)'
```

To return all values of a variable, use its string as an index:

```
>>> sim['L.v']
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
```

or an argument:

```
>>> sim('L.v')
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
```

To see all the methods, use

```
>>> help(sims[0])
```

or go to SimRes. To search for variable names, use names() with wildcards:

```
>>> sims[0].names('L.p*')
[u'L.p.i', u'L.p.v']
```

Likewise, you can explore the linearization result or create diagrams using the methods in LinRes:

```
>>> print lin
Modelica linearization results from ".../examples/PID.mat"
>>> lin.sys.A
matrix([[   0.,    0.],
        [   0., -100.]])
```

CHAPTER

TWO

MODELICARES

Set up Modelica simulations and load, analyze, and plot the results.

This module provides direct access to the most important functions and classes from its submodules. These are:

- Basic supporting classes and functions (`base` module): `add_arrows()`, `add_hlines()`, `add_vlines()`, `animate()`, `ArrowLine`, `closeall()`, `figure()`, `load_csv()`, `save()`, `saveall()`, and `setup_subplots()`
- To manage simulation experiments (`exps` module): `Experiment`, doe, `gen_experiments()`, `ParamDict`, `read_params()`, `run_models()`, `write_params()`, and `write_script()`
- To handle multiple files at once (`multi` module): `multiload()`, `multiplot()`, `multibode()`, and `multinyquist()`
- For simulation results (`simres` module): `SimRes`
- For linearization results (`linres` module): `LinRes`
- To label numbers and quantities (`texunit` module): `label_number()`, `label_quantity()`, and `unit2tex()`

CHAPTER

THREE

MODELICARES.SIMRES

Classes and functions to Load, analyze, and plot results from Modelica simulations

Classes:

- SimRes - Class to load and analyze results from a Modelica-based simulation
- Info - Aliases for the "get" methods in SimRes

Functions:

- merge_times() - Merge a list of multiple time vectors into one vector

class modelicares.simres.**Info**

Aliases for the "get" methods in SimRes

Example:

```
>>> from modelicares.simres import SimRes, Info

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> Info.FV(sim, 'L.v')
-0.25352862
```

FV(*names*, *f=<function <lambda> at 0x7f9da11eeed8>*)
Alias for SimRes.get_FV()

IV(*names*, *f=<function <lambda> at 0x7f9da11eede8>*)
Alias for SimRes.get_IV()

description(*names*)
Alias for SimRes.get_description()

displayUnit(*names*)
Alias for SimRes.get_displayUnit()

indices_wi_times(*names*, *t_1=None*, *t_2=None*)
Alias for SimRes.get_indices_wi_times()

max(*names*, *f=<function <lambda> at 0x7f9da11ef410>*)
Alias for SimRes.get_max()

mean(*names*, *f=<function <lambda> at 0x7f9da11ef500>*)
Alias for SimRes.get_mean()

min(*names*, *f=<function <lambda> at 0x7f9da11ef5f0>*)
Alias for SimRes.get_min()

times(*names*, *i=slice(0, None, None)*, *f=<function <lambda> at 0x7f9da11ef050>*)
Alias for SimRes.get_times()

tuple(*names*, *i=slice(0, None, None)*)
Alias for SimRes.get_tuple()

unit(*names*)
Alias for SimRes.get_unit()

values(*names*, *i=slice(0, None, None)*, *f=<function <lambda> at 0x7f9da11ef1b8>*)
Alias for SimRes.get_values()

values_at_times(*names*, *times*, *f=<function <lambda> at 0x7f9da11ef2a8>*)
Alias for SimRes.get_values_at_times()

class modelicares.simres.**SimRes**(*fname='dsres.mat'*, *constants_only=False*)
Bases: object

Class to load and analyze results from a Modelica-based simulation

This class contains the following methods:

- browse() - Launch a variable browser
- get_description() - Return the description(s) of variable(s)
- get_displayUnit() - Return the Modelica *displayUnit* attribute(s) of variable(s)
- get_indices_wi_times() - Return the widest index pair(s) for which the time of signal(s) is within given limits
- get_IV() - Return the initial value(s) of variable(s)
- get_FV() - Return the final value(s) of variable(s)
- get_max() - Return the maximum value(s) of variable(s)
- get_mean() - Return the time-averaged value(s) of variable(s)
- get_min() - Return the minimum value(s) of variable(s)
- get_times() - Return vector(s) of the sample times of variable(s)
- get_tuple() - Return tuple(s) of time and value vectors for variable(s)
- get_unit() - Return the *unit* attribute(s) of variable(s)
- get_values() - Return vector(s) of the values of the samples of variable(s)
- get_values_at_times() - Return vector(s) of the values of the samples of variable(s)
- names() - Return a list of variable names that match a pattern
- nametree() - Return a tree of all variable names with respect to the path names
- plot() - Plot data as points and/or curves in 2D Cartesian coordinates
- sankey() - Create a figure with Sankey diagram(s)

On initialization, load Modelica simulation results from a MATLAB® file in Dymola® format.

Arguments:

- *fname*: Name of the file (may include the path)

  The file extension ('.mat') is optional.

- *constants_only*: *True*, if only the variables from the first data table should be loaded

  The first data table typically contains all of the constants, parameters, and variables that don't vary. If only that information is needed, it will save some time and memory to set *constants_only* to *True*.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')
```

__call__(*names*, *action=<function get_values at 0x7f9da11ef230>*, **args*, ***kwargs*)

Upon a call to an instance of SimRes, call a method on variable(s) given their name(s).

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *action*: Method for retrieving information about the variable(s)

  The default is get_values(). *action* may be a list or tuple, in which case the return value is a list or tuple.
- **args*, ***kwargs*: Additional arguments for *action*

Examples:

```
>>> from modelicares.simres import SimRes, Info
>>> sim = SimRes('examples/ChuaCircuit.mat')

# Values of a single variable
>>> sim('L.v')
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
>>> # This is equivalent to:
>>> sim.get_values('L.v')
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)

# Values of a list of variables
>>> sim(['L.L', 'C1.C'], SimRes.get_description)
['Inductance', 'Capacitance']
>>> # This is equivalent to:
>>> sim.get_description(['L.L', 'C1.C'])
['Inductance', 'Capacitance']

# Other attributes
>>> print("The final value of %s is %.3f %s." %
      sim('L.i', (Info.description, Info.FV, Info.unit)))
The final value of Current flowing from pin p to pin n is 2.049 A.
```

__contains__(*name*)

Test if a variable is present in the simulation results.

Arguments:

- *name*: Name of variable

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')

>>> # 'L.v' is a valid variable name:
>>> 'L.v' in sim
True
>>> # but 'x' is not:
>>> 'x' not in sim
True
```

__getitem__(*names*)

Upon accessing a variable name within an instance of SimRes, return its values.

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names

Examples:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')

>>> sim['L.v']
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
>>> # This is equivalent to:
>>> sim.get_values('L.v')
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
```

__len__()

Return the number of variables in the simulation.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')

>>> print("There are %i variables in the %s simulation." %
      (len(sim), sim.fbase))
There are 62 variables in the ChuaCircuit simulation.
```

__repr__()

Return a formal description of the SimRes instance.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim
SimRes('...ChuaCircuit.mat')
```

__str__()

Return an informal description of the SimRes instance.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> print(sim)
Modelica simulation results from "...ChuaCircuit.mat"
```

browse()

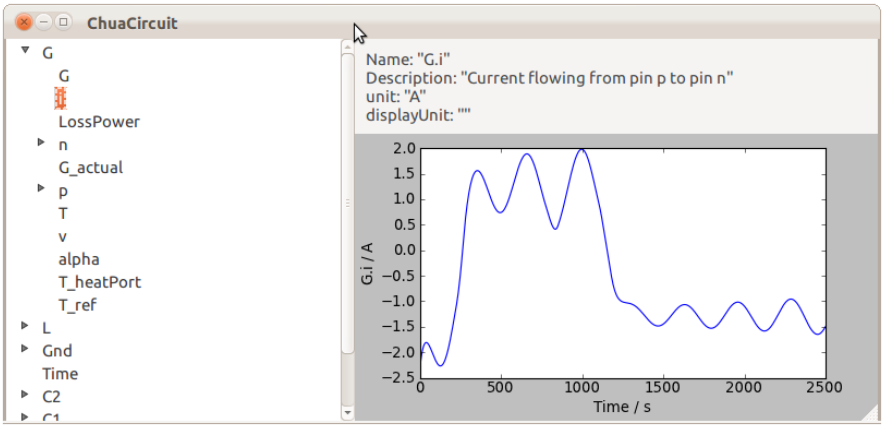
Launch a variable browser.

When a variable is selected, the right panel shows its attributes and a simple plot of the variable over time. Variable names can be dragged and dropped into a text editor.

There are no arguments or return values.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.browse()
```



get_FV(*names*, *f=<function <lambda> at 0x7f9da11eeed8>*)
Return the final value(s) of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of variable names
- *f*: Function that should be applied to the value(s) (default is identity)

If *names* is a string, then the output will be a single value. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of values.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_FV('L.v')
-0.25352862
```

get_IV(*names*, *f=<function <lambda> at 0x7f9da11eede8>*)
Return the initial value(s) of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *f*: Function that should be applied to the value(s) (default is identity)

If *names* is a string, then the output will be a single value. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of values.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_IV('L.v')
0.0
```

get_description(*names*)
Return the description(s) of variable(s).

Arguments:

- *names*: Name(s) of the variable(s) from which to get the description(s)

  This may be a string or (possibly nested) list of strings representing the names of the variables.

If *names* is a string, then the output will be a single description. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of descriptions.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_description('L.v')
'Voltage drop between the two pins (= p.v - n.v)'
```

get_displayUnit(*names*)

Return the Modelica *displayUnit* attribute(s) of variable(s).

Arguments:

- *names*: Name(s) of the variable(s) from which to get the display unit(s)

  This may be a string or (possibly nested) list of strings representing the names of the variables.

If *names* is a string, then the output will be a single display unit. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of display units.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_displayUnit('G.T_heatPort')
'degC'
```

get_indices_wi_times(*names*, *t_1=None*, *t_2=None*)

Return the widest index pair(s) for which the time of signal(s) is within given limits.

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *t_1*: Lower bound of time
- *t_2*: Upper bound of time

If *names* is a string, then the output will be an array of values. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of arrays.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_indices_wi_times('L.v', t_1=500, t_2=2000)
(104, 412)
```

get_max(*names*, *f=<function <lambda> at 0x7f9da11ef410>*)

Return the maximum value(s) of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of variable names
- *f*: Function that should be applied before taking the maximum (default is identity)

If *names* is a string, then the output will be a single value. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of values.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_max('L.v')
0.77344114
```

get_mean(*names*, *f=<function <lambda> at 0x7f9da11ef500>*)
Return the time-averaged value(s) of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of variable names
- *f*: Function that should be applied before taking the mean (default is identity)

If *names* is a string, then the output will be a single value. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of values.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_mean('L.v')
0.014733823
```

get_min(*names*, *f=<function <lambda> at 0x7f9da11ef5f0>*)
Return the minimum value(s) of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of variable names
- *f*: Function that should be applied before taking the minimum (default is identity)

If *names* is a string, then the output will be a single value. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of values.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_min('L.v')
-0.9450165
```

get_times(*names*, *i=slice(0, None, None)*, *f=<function <lambda> at 0x7f9da11ef050>*)
Return vector(s) of the sample times of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *i*: Index (-1 for last), list of indices, or slice of the time(s) to return

  By default, all times are returned.
- *f*: Function that should be applied to all times (default is identity)

If *names* is a string, then the output will be an array of times. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of arrays.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_times('L.v')
array([    0.        , ...  2500.        ], dtype=float32)
```

get_tuple(*names*, *i=slice(0, None, None)*)

Return tuple(s) of time and value vectors for variable(s).

Each tuple contains two vectors: one for times and one for values.

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *i*: Index (-1 for last), list of indices, or slice of the values(s) to return

  By default, all values are returned.

If *names* is a string, then the output will be a tuple. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of tuples.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_tuple('L.v')
(array([    0.        , ... ,  2500.        ], dtype=float32), array([  0.00000000e+00, ... -2.53528625e-
```

Note that this is a tuple of vectors, not a vector of tuples.

get_unit(*names*)

Return the *unit* attribute(s) of variable(s).

Arguments:

- *names*: Name(s) of the variable(s) from which to get the unit(s)

  This may be a string or (possibly nested) list of strings representing the names of the variables.

If *names* is a string, then the output will be a single unit. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of units.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_unit('L.v')
'V'
```

get_values(*names*, *i=slice(0, None, None)*, *f=<function <lambda> at 0x7f9da11ef1b8>*)

Return vector(s) of the values of the samples of variable(s).

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *i*: Index (-1 for last), list of indices, or slice of the values(s) to return

  By default, all values are returned.

- *f*: Function that should be applied to all values (default is identity)

If *names* is a string, then the output will be an array of values. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of arrays.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_values('L.v')
array([  0.00000000e+00, ... -2.53528625e-01], dtype=float32)
```

If the variable cannot be found, then possible matches are listed:

```
>>> sim.get_values(['L.vv'])
L.vv is not a valid variable name.

Did you mean one of these?
       L.v
       L.p.v
       L.n.v
```

The other *get_** methods also give this message when a variable cannot be found.

get_values_at_times(*names*, *times*, *f=<function <lambda> at 0x7f9da11ef2a8>*)

Return vector(s) of the values of the samples of variable(s) at given times.

Arguments:

- *names*: String or (possibly nested) list of strings of the variable names
- *times*: Scalar, numeric list, or a numeric array of the times from which to pull samples
- *f*: Function that should be applied to all values (default is identity)

If *names* is a string, then the output will be an array of values. If *names* is a (optionally nested) list of strings, then the output will be a (nested) list of arrays.

If *times* is not provided, all of the samples will be returned. If necessary, the values will be interpolated over time. The function *f* is applied before interpolation.

Example:

```
>>> from modelicares import SimRes

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.get_values_at_times('L.v', [0, 2000])
array([ 0.        ,  0.15459341])
```

names(*pattern=None*, *re=False*)

Return a list of variable names that match a pattern.

By default, all names are returned.

Arguments:

- *pattern*: Case-sensitive string used for matching
  - If *re* is *False* (next argument), then the pattern follows the Unix shell style:

Character(s)	Role
*	Matches everything
?	Matches any single character
[seq]	Matches any character in seq
[!seq]	Matches any char not in seq

Wildcard characters (‘*’) are not automatically added at the beginning or the end of the pattern. For example, ‘x*’ matches variables that begin with “x”, whereas ‘*x*’ matches all variables that contain “x”.

–If *re* is *True*, regular expressions are used a la Python’s re module. See also http://docs.python.org/2/howto/regex.html#regex-howto.

Since `re.search` is used to produce the matches, it is as if wildcards (‘.*’) are automatically added at the beginning and the end. For example, ‘x’ matches all variables that contain “x”. Use ‘^x$’ to match only the variables that begin with “x” and ‘x$’ to match only the variables that end with “x”.

Note that ‘.’ is a subclass separator in Modelica but a wildcard in regular expressions. Escape the subclass separator as ‘\.’.

- *re*: *True* to use regular expressions (*False* to use shell style)

Examples:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')

>>> # Names starting with "L.p", using shell-style matching:
>>> sim.names('L.p*')
[u'L.p.i', u'L.p.v']

>>> # Names ending with "p.v", using re matching:
>>> sim.names('p\.v$', re=True)
[u'C2.p.v', u'Gnd.p.v', u'L.p.v', u'Nr.p.v', u'C1.p.v', u'Ro.p.v', u'G.p.v']
```

nametree()

Return a tree of all variable names with respect to the path names.

The tree represents the structure of the Modelica model. It is returned as a nested dictionary. The keys are the path elements and the values are sub-dictionaries or variable names.

There are no arguments.

Example:

```
>>> from modelicares import SimRes
>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.nametree()
{u'G': {u'G': u'G.G', ... u'n': {u'i': u'G.n.i', u'v': u'G.n.v'}, ...}, u'L': {...}, ...}
```

plot(*ynames1=[]*, *ylabel1=None*, *legends1=[]*, *leg1_kwargs={‘loc’: ‘best’}*, *ax1=None*, *ynames2=[]*, *ylabel2=None*, *legends2=[]*, *leg2_kwargs={‘loc’: ‘best’}*, *ax2=None*, *xname=’Time’*, *xlabel=None*, *title=None*, *label=’xy’*, *incl_prefix=False*, *suffix=None*, *use_paren=True*, ***kwargs*)

Plot data as points and/or curves in 2D Cartesian coordinates.

Arguments:

- *ynames1*: Names of variables for the primary y axis

  If any names are invalid, then they will be skipped.

- *ylabel1*: Label for the primary y axis

  If *ylabel1* is *None* (default) and all of the variables have the same Modelica description string, then the common description will be used. Use ‘’ for no label.

- *legends1*: List of legend entries for variables assigned to the primary y axis

If *legends1* is an empty list ([]), ynames1 will be used. If *legends1* is *None* and all of the variables on the primary axis have the same unit, then no legend will be shown.

- *leg1_kwargs*: Dictionary of keyword arguments for the primary legend
- *ax1*: Primary y axes

  If *ax1* is not provided, then axes will be created in a new figure.
- *ynames2*, *ylabel2*, *legends2*, *leg2_kwargs*, and *ax2*: Similar to *ynames1*, *ylabel1*, *legends1*, *leg1_kwargs*, and *ax1* but for the secondary y axis
- *xname*: Name of the x-axis data
- *xlabel*: Label for the x axis

  If *xlabel* is *None* (default), the variable's Modelica description string will be applied. Use '' for no label.
- *title*: Title for the figure

  If *title* is *None* (default), then the title will be the base filename. Use '' for no title.
- *label*: Label for the figure (ignored if ax is provided)

  This will be used as a base filename if the figure is saved.
- *incl_prefix*: If *True*, prefix the legend strings with the base filename of the class.
- *suffix*: String that will be added at the end of the legend entries
- *use_paren*: Add parentheses around the suffix
- ***kwargs*: Propagated to `base.plot()` (and thus to `matplotlib.pyplot.plot()`)

  If both y axes are used (primary and secondary), then the *dashes* argument is ignored. The curves on the primary axis will be solid and the curves on the secondary axis will be dotted.

Returns:

1. *ax1*: Primary y axes
2. *ax2*: Secondary y axes

Example:

```
>>> from modelicares import SimRes, save

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> sim.plot(ynames1='L.i', ylabel1="Current",
             ynames2='L.der(i)', ylabel2="Derivative of current",
             title="Chua Circuit", label='examples/ChuaCircuit')
(<matplotlib.axes...AxesSubplot object at 0x...>, <matplotlib.axes...AxesSubplot object at 0x...>)

>>> save()
Saved examples/ChuaCircuit.pdf
Saved examples/ChuaCircuit.png
```

sankey(*names=[], times=[0], n_rows=1, title=None, subtitles=[], label='sankey', margin_left=0.05, margin_right=0.05, margin_bottom=0.05, margin_top=0.1, wspace=0.1, hspace=0.25, **kwargs*)

Create a figure with Sankey diagram(s).

Arguments:

- *names*: List of names of the flow variables

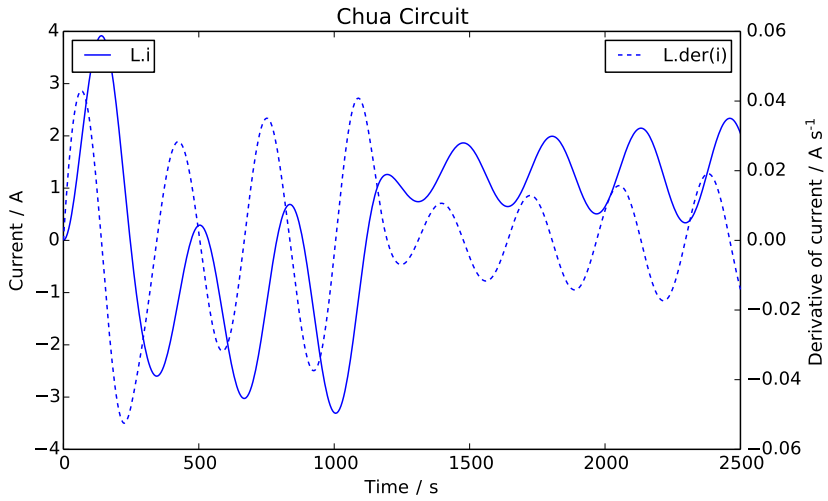


Figure 3.1: Plot of Chua circuit

- *times*: List of times at which the data should be sampled

  If multiple times are given, then subfigures will be generated, each with a Sankey diagram.

- *n_rows*: Number of rows of (sub)plots

- *title*: Title for the figure

  If *title* is *None* (default), then the title will be "Sankey Diagram of *fbase*", where *fbase* is the base filename of the data. Use '' for no title.

- *subtitles*: List of subtitles (i.e., titles for each subplot)

  If not provided, "t = xx s" will be used, where *xx* is the time of each entry. "(initial)" or "(final)" is appended if appropriate.

- *label*: Label for the figure

  This will be used as the base filename if the figure is saved.

- *margin_left*: Left margin

- *margin_right*: Right margin

- *margin_bottom*: Bottom margin

- *margin_top*: Top margin

- *wspace*: The amount of width reserved for blank space between subplots

- *hspace*: The amount of height reserved for white space between subplots

- ***kwargs*: Additional arguments for `matplotlib.sankey.Sankey`

Returns:

1. List of `matplotlib.sankey.Sankey` instances of the subplots

Example:

```
>>> from modelicares import SimRes, save

>>> sim = SimRes('examples/ThreeTanks')
```

```
>>> sankeys = sim.sankey(label='examples/ThreeTanks',
    title="Sankey Diagrams of Modelica.Fluid.Examples.Tanks.ThreeTanks",
    times=[0, 50, 100, 150], n_rows=2, format='%.1f ',
    names=['tank1.ports[1].m_flow', 'tank2.ports[1].m_flow',
           'tank3.ports[1].m_flow'],
    labels=['Tank 1', 'Tank 2', 'Tank 3'],
    orientations=[-1, 0, 1],
    scale=0.1, margin=6, offset=1.5,
    pathlengths=2, trunklength=10)
>>> save()
Saved examples/ThreeTanks.pdf
Saved examples/ThreeTanks.png
```

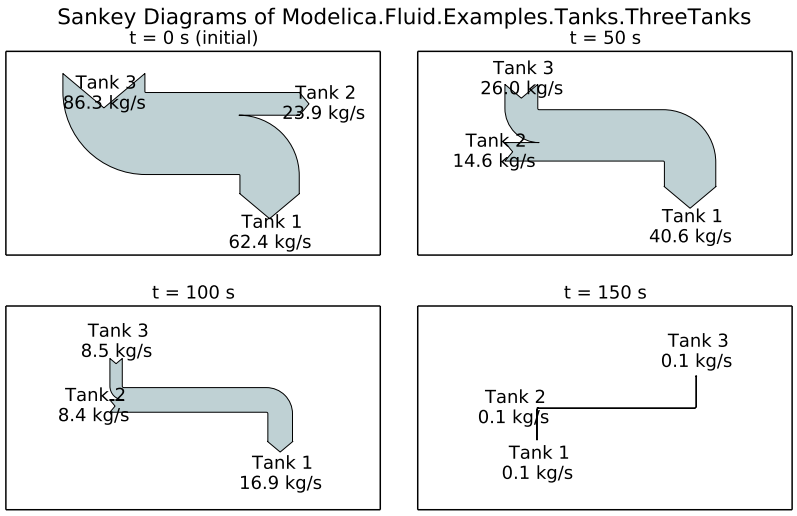


Figure 3.2: Sankey digarams of three-tank model

modelicares.simres.**merge_times**(*times_list*)

Merge a list of multiple time vectors into one vector.

Example:

```
>>> from modelicares.simres import SimRes, merge_times

>>> sim = SimRes('examples/ChuaCircuit.mat')
>>> times_list = sim.get_times(['L.v', 'G.T_heatPort'])
>>> merge_times(times_list)
array([    0.        , ... 2500.        ], dtype=float32)
```

CHAPTER

FOUR

MODELICARES.LINRES

Load, analyze, and plot the result of linearizing a Modelica model.

This module contains one class: `LinRes`. It relies on python-control, which is included in the distribution.

class `modelicares.linres.`**`LinRes`**(*fname='dslin.mat'*)

Bases: `object`

Class for Modelica-based linearization results and methods to analyze those results

This class contains two user-accessible methods:

- `bode()` - Create a Bode plot of the system's response
- `nyquist()` - Create a Nyquist plot of the system's response

On initialization, load and preprocess a linearized Modelica model (MATLAB® format).

The model is in state space:

```
der(x) = A*x + B*u;
     y = C*x + D*u;
```

The linear system is stored as *sys* within this class. It is an instance of `control.StateSpace`, which emulates the structure of a continuous-time model in MATLAB® (e.g., the output of `ss()` in MATLAB®). It contains:

- *A*, *B*, *C*, *D*: Matrices of the linear system
- *stateName*: List of name(s) of the states (x)
- *inputName*: List of name(s) of the inputs (u)
- *outputName*: List of name(s) of the outputs (y)

Arguments:

- *fname*: Name of the file (may include the path)

  The file extension ('.mat') is optional. The file must contain four matrices: *Aclass* (specifies the class name, which must be "AlinearSystem"), *nx*, *xuyName*, and *ABCD*.

Example:

```
>>> from modelicares import LinRes
>>> lin = LinRes('examples/PID')
```

`__repr__`()

Return a formal description of the `LinRes` instance.

Example:

```
>>> from modelicares import LinRes
>>> lin = LinRes('examples/PID.mat')
>>> lin
LinRes('...PID.mat')
```

__str__()

Return an informal description of the `LinRes` instance.

Example:

```
>>> from modelicares import LinRes
>>> lin = LinRes('examples/PID.mat')
>>> print(lin)
Modelica linearization results from "...PID.mat"
```

bode(*axes=None, pairs=None, label='bode', title=None, colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], styles=[(None, None), (3, 3), (1, 1), (3, 2, 1, 2)], **kwargs*)

Create a Bode plot of the system's response.

The Bode plots of a MIMO system are overlayed. This is different than MATLAB®, which creates an array of subplots.

Arguments:

- *axes*: Tuple (pair) of axes for the magnitude and phase plots

  If *axes* is not provided, then axes will be created in a new figure.

- *pairs*: List of (input index, output index) tuples of each transfer function to be evaluated

  If not provided, all of the transfer functions will be plotted.

- *label*: Label for the figure (ignored if *axes* is provided)

  This will be used as the base filename if the figure is saved.

- *title*: Title for the figure

  If *title* is *None* (default), then the title will be "Bode Plot of *fbase*", where *fbase* is the base filename of the data. Use '' for no title.

- *colors*: Color or list of colors that will be used sequentially

  Each may be a character, grayscale, or rgb value.

  See also:

  http://matplotlib.sourceforge.net/api/colors_api.html

- *styles*: Line/dash style or list of line/dash styles that will be used sequentially

  Each style is a string representing a linestyle (e.g., "–") or a tuple of on/off lengths representing dashes. Use "" for no line and "-" for a solid line.

  See also:

  http://matplotlib.sourceforge.net/api/collections_api.html

- ***kwargs*: Additional arguments for `control.freqplot.bode()`

Returns:

1. *axes*: Tuple (pair) of axes for the magnitude and phase plots

Example:

```
>>> from modelicares import LinRes, save
>>> from numpy import pi, logspace

>>> lin = LinRes('examples/PID.mat')
>>> lin.bode(label='examples/PID-bode', omega=2*pi*logspace(-2, 3),
         title="Bode Plot of Modelica.Blocks.Continuous.PID")
(<matplotlib.axes...AxesSubplot object at 0x...>, <matplotlib.axes...AxesSubplot object at 0x...>)
>>> save()
Saved examples/PID-bode.pdf
Saved examples/PID-bode.png
```

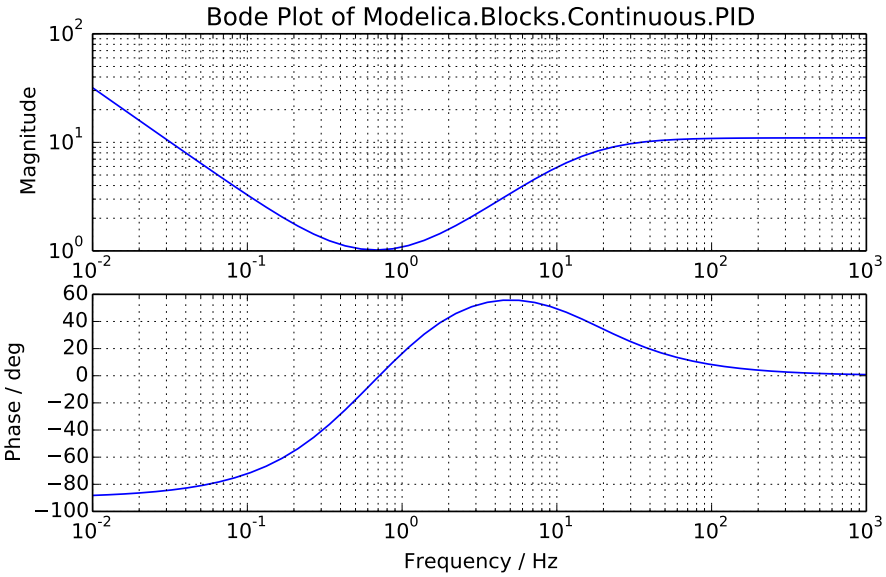


Figure 4.1: Results of example for `LinRes.bode()`.

nyquist(*ax=None, pairs=None, label='nyquist', title=None, xlabel='Real Axis', ylabel='Imaginary Axis', colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], **kwargs*)

Create a Nyquist plot of the system's response.

The Nyquist plots of a MIMO system are overlayed. This is different than MATLAB®, which creates an array of subplots.

Arguments:

- *ax*: Axes onto which the Nyquist diagram should be plotted

  If *ax* is not provided, then axes will be created in a new figure.

- *pairs*: List of (input index, output index) tuples of each transfer function to be evaluated

  If not provided, all of the transfer functions will be plotted.

- *label*: Label for the figure (ignored if ax is provided)

  This will be used as the base filename if the figure is saved.

- *title*: Title for the figure

  If *title* is *None* (default), then the title will be "Nyquist Plot of *fbase*", where *fbase* is the base filename of the data. Use '' for no title.

- *xlabel*: x-axis label
- *ylabel*: y-axis label
- *colors*: Color or list of colors that will be used sequentially

  Each may be a character, grayscale, or rgb value.

  See also:

  http://matplotlib.sourceforge.net/api/colors_api.html

- ***kwargs*: Additional arguments for `control.freqplot.nyquist()`

Returns:

1. *ax*: Axes of the Nyquist plot

Example:

```
>>> from modelicares import LinRes, save
>>> from numpy import pi, logspace

>>> lin = LinRes('examples/PID.mat')
>>> lin.nyquist(label='examples/PID-nyquist',
            omega=2*pi*logspace(0, 3, 61), labelFreq=20,
            title="Nyquist Plot of Modelica.Blocks.Continuous.PID")
<matplotlib.axes...AxesSubplot object at 0x...>
>>> save()
Saved examples/PID-nyquist.pdf
Saved examples/PID-nyquist.png
```

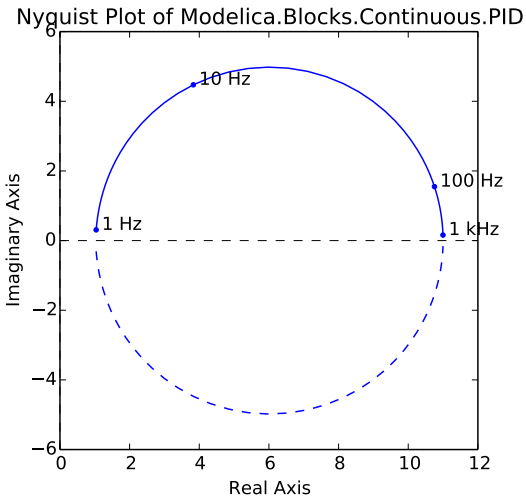


Figure 4.2: Results of example for `LinRes.nyquist()`.

CHAPTER

FIVE

MODELICARES.MULTI

Functions to load and plot data from multiple simulation and linearization files at once

- `multiload()` - Load multiple Modelica simulation and/or linearization results
- `multiplot()` - Plot data from multiple simulations in 2D Cartesian coordinates
- `multibode()` - Plot multiple linearizations onto a single Bode diagram
- `multinyquist()` - Plot multiple linearizations onto a single Nyquist diagram

`modelicares.multi.`**`multibode`**`(`*lins, axes=None, pair=(0, 0), label='bode', title='Bode Plot', labels='', colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], styles=[(None, None), (3, 3), (1, 1), (3, 2, 1, 2)], leg_kwargs={}, **kwargs*`)`

Plot multiple linearizations onto a single Bode diagram.

Arguments:

•*lins*: Linearization result or list of results (instances of `linres.LinRes`)

•*axes*: Tuple (pair) of axes for the magnitude and phase plots

If *axes* is not provided, then axes will be created in a new figure.

•*pair*: Tuple of (input index, output index) for the transfer function to be chosen from each system (applied to all)

This is ignored if the system is SISO.

•*label*: Label for the figure (ignored if axes is provided)

This will be used as the base filename if the figure is saved.

•*title*: Title for the figure

•*labels*: Label or list of labels for the legends

If *labels* is *None*, then no label will be used. If it is an empty string (''), then the base filenames will be used.

•*colors*: Color or list of colors that will be used sequentially

Each may be a character, grayscale, or rgb value.

See also:

http://matplotlib.sourceforge.net/api/colors_api.html

•*styles*: Line/dash style or list of line/dash styles that will be used sequentially

Each style is a string representing a linestyle (e.g., "–") or a tuple of on/off lengths representing dashes. Use "" for no line and "-" for a solid line.

See also:

http://matplotlib.sourceforge.net/api/collections_api.html

- *leg_kwargs*: Dictionary of keyword arguments for `matplotlib.pyplot.legend()`

  If *leg_kwargs* is *None*, then no legend will be shown.

- ***kwargs*: Additional arguments for `control.freqplot.bode()`

Returns:

1. *axes*: Tuple (pair) of axes for the magnitude and phase plots

Example:

```
>>> import os

>>> from glob import glob
>>> from modelicares import LinRes, multibode, save, read_params
>>> from numpy import pi, logspace

>>> lins = map(LinRes, glob('examples/PID/*/*.mat'))
>>> labels = ["Ti=%g" % read_params('Ti', os.path.join(lin.dir, 'dsin.txt'))
          for lin in lins]
>>> multibode(lins,
          title="Bode Plot of Modelica.Blocks.Continuous.PID",
          label='examples/PIDs-bode', omega=2*pi*logspace(-2, 3),
          labels=labels, leg_kwargs=dict(loc='lower right'))
(<matplotlib.axes...AxesSubplot object at 0x...>, <matplotlib.axes...AxesSubplot object at 0x...>)

>>> save()
Saved examples/PIDs-bode.pdf
Saved examples/PIDs-bode.png
```

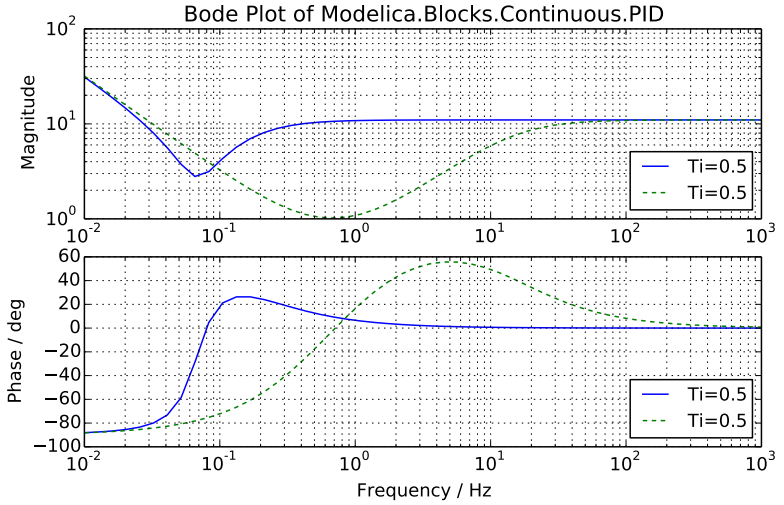


Figure 5.1: Bode plot of PID with varying parameters

`modelicares.multi.`**`multiload`**(*locations='*'*)

Load multiple Modelica simulation and/or linearization results.

Arguments:

- *locations*: Input filename, directory, or list of these

Wildcards ('*') may be used in the path(s).

Returns:

1.List of simulations (`simres.SimRes` instances)

2.List of linearizations (`linres.LinRes` instances)

Either may be an empty list.

Example:

```
>>> from modelicares import *

# By file:
>>> multiload(['examples/ChuaCircuit.mat', 'examples/PID/*/*.mat'])
Valid: SimRes('.../examples/ChuaCircuit.mat')
Valid: LinRes('.../examples/PID/1/dslin.mat')
Valid: LinRes('.../examples/PID/2/dslin.mat')
([SimRes('.../examples/ChuaCircuit.mat')], [LinRes('.../examples/PID/1/dslin.mat'), LinRes('.../examples/PID/2


# By directory:
>>> multiload('examples')
Valid: SimRes('...ChuaCircuit.mat')
Valid: LinRes('...PID.mat')...
Valid: SimRes('...ThreeTanks.mat')
([SimRes('...ChuaCircuit.mat'), SimRes('...ThreeTanks.mat')], [LinRes('...PID.mat')])
```

modelicares.multi.**multinyquist**(*lins, ax=None, pair=(0, 0), label='nyquist', title='Nyquist Plot', xlabel='Real Axis', ylabel='Imaginary Axis', labels='', colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], leg_kwargs={}, **kwargs*)

Plot multiple linearizations onto a single Nyquist diagram.

Arguments:

- *lins*: Linearization result or list of results (instances of `linres.LinRes`)
- *ax*: Axes onto which the Nyquist diagrams should be plotted

  If *ax* is not provided, then axes will be created in a new figure.
- *pair*: Tuple of (input index, output index) for the transfer function to be chosen from each system (applied to all)

  This is ignored if the system is SISO.
- *label*: Label for the figure (ignored if axes is provided)

  This will be used as the base filename if the figure is saved.
- *title*: Title for the figure
  - *xlabel*: x-axis label
  - *ylabel*: y-axis label
- *labels*: Label or list of labels for the legends

  If *labels* is *None*, then no label will be used. If it is an empty string (''), then the base filenames will be used.
- *colors*: Color or list of colors that will be used sequentially

  Each may be a character, grayscale, or rgb value.

See also:

http://matplotlib.sourceforge.net/api/colors_api.html

- *leg_kwargs*: Dictionary of keyword arguments for `matplotlib.pyplot.legend()`

  If *leg_kwargs* is *None*, then no legend will be shown.

- ***kwargs*: Additional arguments for `control.freqplot.nyquist()`

  If *textFreq* is not specified, then only the frequency points of the first system will have text labels.

Returns:

1. *ax*: Axes of the Nyquist plot

Example:

```
>>> import os

>>> from glob import glob
>>> from modelicares import LinRes, multinyquist, save, read_params
>>> from numpy import pi, logspace

>>> lins = map(LinRes, glob('examples/PID/*/*.mat'))
>>> labels = ["Td=%g" % read_params('Td', os.path.join(lin.dir, 'dsin.txt'))
          for lin in lins]
>>> multinyquist(lins,
             title="Nyquist Plot of Modelica.Blocks.Continuous.PID",
             label='examples/PIDs-nyquist', textFreq=True,
             omega=2*pi*logspace(-1, 3, 81), labelFreq=20,
             labels=labels)
<matplotlib.axes...AxesSubplot object at 0x...>

>>> save()
Saved examples/PIDs-nyquist.pdf
Saved examples/PIDs-nyquist.png
```

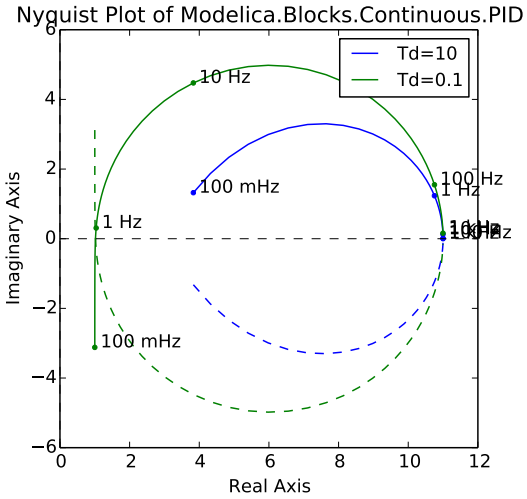


Figure 5.2: Nyquist plot of PID with varying parameters

`modelicares.multi.`**`multiplot`**(*sims, suffixes=''', color=['b', 'g', 'r', 'c', 'm', 'y', 'k'], dashes=[(None, None), (3, 3), (1, 1), (3, 2, 1, 2)], **kwargs*)

Plot data from multiple simulations in 2D Cartesian coordinates.

This method simply calls `simres.SimRes.plot()` from multiple instances of `simres.SimRes`.

A new figure is created if necessary.

Arguments:

- *sims*: Simulation result or list of results (instances of `simres.SimRes`)
- *suffixes*: Suffix or list of suffixes for the legends (see `simres.SimRes.plot()`)

  If *suffixes* is *None*, then no suffix will be used. If it is an empty string (''), then the base filenames will be used.
- *color*: Single entry, list, or `itertools.cycle` of colors that will be used sequentially

  Each entry may be a character, grayscale, or rgb value.

  See also:

  http://matplotlib.sourceforge.net/api/colors_api.html
- *dashes*: Single entry, list, or `itertools.cycle` of dash styles that will be used sequentially

  Each style is a tuple of on/off lengths representing dashes. Use (0, 1) for no line and (None ,None) for a solid line.

  See also:

  http://matplotlib.sourceforge.net/api/collections_api.html
- ***kwargs*: Propagated to `simres.SimRes.plot()` (and thus to `base.plot()` and finally `matplotlib.pyplot.plot()`)

Returns:

1. *ax1*: Primary y axes
2. *ax2*: Secondary y axes

Example:

```
>>> from glob import glob
>>> from modelicares import SimRes, multiplot, save

>>> sims = map(SimRes, glob('examples/ChuaCircuit/*/*.mat'))
>>> multiplot(sims, title="Chua Circuit", label='examples/ChuaCircuits',
          suffixes=['L.L = %.0f H' % sim.get_IV('L.L')
                    for sim in sims], # Read legend parameters.
          ynames1='L.i', ylabel1="Current")
(<matplotlib.axes...AxesSubplot object at 0x...>, None)

>>> save()
Saved examples/ChuaCircuits.pdf
Saved examples/ChuaCircuits.png
```

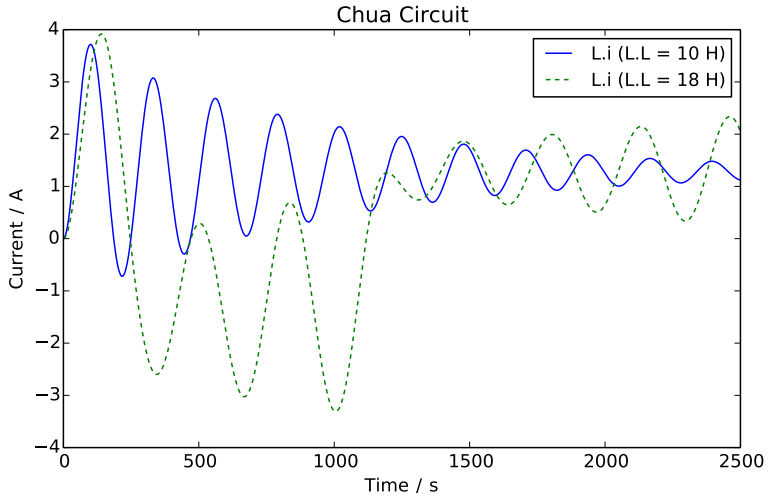


Figure 5.3: Plot of Chua circuit with varying inductance

CHAPTER

SIX

MODELICARES.EXPS

Set up and help run Modelica simulation experiments.

This module supports two approaches for managing simulations. The first is to create a Modelica script (using `write_script()`) and run it within a Modelica environment (see the scripts in "examples/ChuaCircuit/"), which translates and simulates the models with the prescribed settings. The second approach is to execute pre-translated models. The `run_models()` method handles this by writing to initialization file(s) (e.g, "dsin.txt") and launching the appropriate model executables. The advantage of the first approach is that formal parameters (those that are hard-coded during translation) can be adjusted. However, the second approach is faster because it does not require a model to be recompiled when only tunable parameters (those that are not hard-coded during translation) are changed.

The first step in either case is to create a dictionary to specify model parameters and other settings for simulation experiment. A single model parameter may have multiple possible values. The dictionary is passed to the `gen_experiments()` function (see that function for a description of the dictionary format), which combines the values of all the variables (by piecewise alignment or permutation) and returns a generator to step through the experiments. Finally, the generator is passed to the `write_script()` or `run_models()` function (see first paragraph).

Classes:

- `ParamDict` - Dictionary that prints its items as nested tuple-based modifiers, formatted for Modelica
- `Experiment` - Named tuple class to represent a simulation experiment

Functions:

- `gen_experiments()` - Return a generator for a set of simulation experiments using permutation or simple element-wise grouping
- `modelica_array()` - Convert a NumPy array to a Modelica-formatted string
- `modelica_boolean()` - Convert a Boolean variable to a Modelica string
- `read_params()` - Read parameter values from an initialization or final values file
- `run_models()` - Run Modelica models via pairs of executables and initialization files (not yet implemented)
- `write_params()` - Write parameter values to a simulation initialization file
- `write_script()` - Write a Modelica script to run simulations

Submodules:

- `doe` - Functions for design of experiments (DOE)

class `modelicares.exps.`**Experiment**

Bases: `tuple`

Named tuple class to represent a simulation experiment

Instances of this class may be used in the *experiments* argument of `write_script()` and `run_models()`, although there are some differences in the entries (see those functions for details).

Example:

```
>>> from modelicares import *

>>> experiment = Experiment('ChuaCircuit', params={'L.L': 18}, args={})
>>> experiment.model
'ChuaCircuit'
```

__repr__()
Return a nicely formatted representation string

args
Alias for field number 2

model
Alias for field number 0

params
Alias for field number 1

class modelicares.exps.**ParamDict**
Bases: dict

Dictionary that prints its items (string mapping) as nested tuple-based modifiers, formatted for Modelica

Otherwise, this class is the same as dict. The underlying structure is not nested or reformatted—only the informal representation (ParamDict.__str__()).

__str__()
Map the ParamDict instance to a string using tuple-based modifiers formatted for Modelica.

Each key is interpreted as a parameter name (including the full model path in Modelica dot notation) and each entry is a parameter value. The value may be a number (integer or float), Boolean constant (in Python format—*True* or *False*, not 'true' or 'false'), string, or NumPy arrays of these. Modelica strings must be given with double quotes included (e.g., "'hello"'). Enumerations may be used as values (e.g., 'Axis.x'). Values may include functions, but the entire value must be expressed as a Python string (e.g., 'fill(true, 2, 2)'). Items with a value of *None* are not shown.

Redeclarations and other prefixes must be included in the key along with the name of the instance (e.g., 'redeclare Region regions[n_x, n_y, n_z]'). The single quotes must be explicitly included for instance names that contain symbols (e.g., "'H+"').

Note that Python dictionaries do not preserve order.

Example:

```
>>> from numpy import array
>>> from modelicares import *

>>> d = ParamDict({'a': 1, 'b.c': array([2, 3]), 'b.d': False,
                   'b.e': '"hello"', 'b.f': None})
>>> print(d)
(a=1, b(c={2, 3}, e="hello", d=false))

# The formal representation (and the internal structure) is unaffected:
>>> d
{'a': 1, 'b.c': array([2, 3]), 'b.f': None, 'b.e': '"hello"', 'b.d': False}

# An empty dictionary prints as an empty string (not "()"):
print(ParamDict({}))
```

modelicares.exps.**gen_experiments**(*models=None*, *params={}*, *args={}*, *design=<function fullfact at 0x7f9da9f42f50>*)

Return a generator for a set of simulation experiments using permutation or simple element-wise grouping.

The generator yields instances of `Experiment`—named tuples of (*model*, *params*, *args*), where *model* is the name of a single model (type `str`), *params* is a specialized dictionary (`ParamDict`) of model parameter names and values, and *arg_dict* is a dictionary (`dict`) of command arguments (keyword and value) for the Modelica tool or environment.

Arguments:

- *models*: List of model names (including the full model path in Modelica dot notation)
- *params*: Dictionary of model parameters

  Each key is a variable name and each entry is a list of values. The keys must indicate the hierarchy within the model—either in Modelica dot notation or via nested dictionaries.
- *args*: Dictionary of command arguments for the Modelica tool or environment (e.g., to the `simulateModel()` command in Dymola)

  Each key is an argument name and each entry is a list of settings.
- *design*: Method of generating the simulation experiments (i.e., design of experiments)

  This is a function that returns a iterable object that contains or generates the simulation settings. Several options are available in `modelicares.doe`.

Example 1 (element-wise list of experiments):

```
>>> from modelicares import *

>>> experiments = gen_experiments(
                  ['Modelica.Electrical.Analog.Examples.ChuaCircuit']*3,
                  {'L.L': [16, 18, 20], 'C2.C': [80, 100, 120]},
                  design=doe.aslisted)
>>> for experiment in experiments:
    print(experiment.model + str(experiment.params))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=100), L(L=18))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=120), L(L=20))
>>> # Note that the model name must be repeated in the models argument.
```

Example 2 (one-factor-at-a-time; first entries are baseline):

```
>>> from modelicares import *

>>> experiments = gen_experiments(
                  ['Modelica.Electrical.Analog.Examples.ChuaCircuit'],
                  {'L.L': [16, 18, 20], 'C2.C': [80, 100, 120]},
                  design=doe.ofat)
>>> for experiment in experiments:
    print(experiment.model + str(experiment.params))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=18))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=20))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=100), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=120), L(L=16))
```

Example 3 (permutation—full-factorial design of experiments):

```
>>> from modelicares import *

>>> experiments = gen_experiments(
                  ['Modelica.Electrical.Analog.Examples.ChuaCircuit'],
                  {'L.L': [16, 18, 20], 'C2.C': [80, 100, 120]},
                  design=doe.fullfact)
>>> for experiment in experiments:
    print(experiment.model + str(experiment.params))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=100), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=120), L(L=16))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=18))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=100), L(L=18))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=120), L(L=18))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=80), L(L=20))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=100), L(L=20))
Modelica.Electrical.Analog.Examples.ChuaCircuit(C2(C=120), L(L=20))
```

Example 4 (parameters given in nested form):

```
>>> from modelicares import *

>>> models = ['Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.oneAxis']
>>> params = dict(axis=dict(motor=dict(i_max=[5, 15],
                                       Ra=dict(R=[200, 300]))))
>>> for experiment in gen_experiments(models, params):
    print(experiment.model + str(experiment.params))
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.oneAxis(axis(motor(i_max=5, Ra(R=200))))
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.oneAxis(axis(motor(i_max=15, Ra(R=200))))
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.oneAxis(axis(motor(i_max=5, Ra(R=300))))
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.oneAxis(axis(motor(i_max=15, Ra(R=300))))

>>> # Note that the underlying representation of the parameters is
>>> # actually flat:
>>> for experiment in gen_experiments(models, params):
    experiment.params
{'axis.motor.Ra.R': 200, 'axis.motor.i_max': 5}
{'axis.motor.Ra.R': 200, 'axis.motor.i_max': 15}
{'axis.motor.Ra.R': 300, 'axis.motor.i_max': 5}
{'axis.motor.Ra.R': 300, 'axis.motor.i_max': 15}
>>> # Also note that Python dictionaries do not preserve order (and it
>>> # is not necessary here).
```

modelicares.exps.**modelica_array**(*x*)

Convert a NumPy array to a Modelica-formatted string.

Square brackets are curled and Booleans are cast to lowercase.

Example:

```
>>> from numpy import array
>>> from modelicares import *

>>> x = array([[1, 2], [3, 4]])
>>> modelica_array(x)
'{{1, 2}, {3, 4}}'

>>> modelica_array(array([[True, True], [False, False]]))
'{{true, true}, {false, false}}'
```

modelicares.exps.**modelica_boolean**(*x*)

Convert a Boolean variable (`bool`) to a Modelica string ('true' or 'false').

Example:

```
>>> from modelicares import *

>>> modelica_boolean(True)
'true'
>>> modelica_boolean(False)
'false'
```

modelicares.exps.**read_params**(*names*, *fname='dsin.txt'*)

Read parameter values from an initialization or final values file (e.g., dsin.txt or dsfinal.txt).

Arguments:

- *names*: Parameter name or list of names (with full model path in Modelica dot notation)

  A parameter name includes array indices (if any) in Modelica representation (1-base indexing); the values are scalar.

- *fname*: Name of the file (may include the file path)

Example:

```
>>> from modelicares import *

>>> read_params(['L.L', 'C1.C'], 'examples/dsin.txt')
[18.0, 10.0]
```

modelicares.exps.**run_models**(*experiments=[(None, {}, {})]*, *filemap={'dslog.txt': '%s_%i.log', 'dsres.mat': '%s_%i.mat'}*)

Run Modelica models via pairs of executables and initialization files.

Warning: This function has not yet been implemented.

Arguments:

- *experiments*: Tuple or (list or generator of) tuples specifying the simulation experiment(s)

  The first entry of each tuple is the name of the model executable. The second is a dictionary of model parameter names and values. The third is a dictionary of simulation settings (keyword and value).

  Each tuple may be (optionally) an instance of the tuple subclass `Experiment`, which names the entries as *model*, *params*, and *args*. These designations are used below for clarity.

  model may include the file path. It is not necessary to include the extension (e.g., ".exe"). There must be a corresponding model initialization file on the same path with the same base name and the extension ".in". For Dymola®, the executable is the "dymosim" file (possibly renamed) and the initialization file is a renamed 'dsin.txt' file.

  The keys or variable names in the *params* dictionary must indicate the hierarchy within the model—either in Modelica dot notation or via nested dictionaries. The items in the dictionary must correspond to parameters in the initialization file. In Dymola, these are integers or floating point numbers. Therefore, arrays must be broken into scalars by indicating the indices (Modelica 1-based indexing) in the key along with the variable name. Enumerations and Booleans must be given as their unsigned integer equivalents (e.g., 0 for *False*). Strings and prefixes are not supported.

  Items with values of *None* in *params* and *args* are skipped.

- *filemap*: Dictionary of result file mappings

  Each key is the path/name of a file that is generated during simulation (source) and each value is the path/name it will be copied as (destination). The sources and destinations are relative to the directory indicated by the *model* subargument. '%s' may be included in the destination to indicate the model name (*model*) without the full path or extension. '%i' may be included to indicate the simulation number in the sequence of experiments.

There are no return values.

modelicares.exps.**write_params**(*params*, *fname='dsin.txt'*)

Write parameter values to a simulation initialization file (e.g., dsin.txt).

Arguments:

- *params*: Dictionary of parameters

  Each key is a parameter name (including the full model path in Modelica dot notation) and each entry is a parameter value. The parameter name includes array indices (if any) in Modelica representation (1-bases indexing). The values must be representable as scalar numbers (integer or floating point). *True* and *False* (not 'true' and 'false') are automatically mapped to 1 and 0. Enumerations must be given explicitly as the unsigned integer equivalent. Strings, functions, redeclarations, etc. are not supported.

- *fname*: Name of the file (may include the file path)

Example:

```
>>> from modelicares import *

>>> write_params({'L.L': 10, 'C1.C': 15}, 'examples/dsin.txt')
```

This updates the appropriate lines in "examples/dsin.txt":

```
-1      10                  0       0                   1  280   # L.L
...
-1      15                  0  1.000000000000000E+100   1  280   # C1.C
```

modelicares.exps.**write_script**(*experiments=[(None, {}, {})]*, *packages=[]*, *working_dir='~/Documents/Modelica'*, *fname='run-sims.mos'*, *command='simulateModel'*, *results=['dsin.txt', 'dslog.txt', 'dsres.mat', 'dymosim%x', 'dymolalg.txt']*)

Write a Modelica script to run simulations.

Arguments:

- *experiments*: Tuple or (list or generator of) tuples specifying the simulation experiment(s)

  The first entry of each tuple is the name of the model to be simulated, including the full path in Modelica dot notation. The second is a dictionary of parameter names and values. The third is a dictionary of command arguments (keyword and value) for the Modelica tool or environment (see below for Dymola®).

  Each tuple may be (optionally) an instance of the tuple subclass `Experiment`, which names the entries as *model*, *params*, and *args*. These designations are used below for clarity.

  The keys or variable names in the *params* dictionary must indicate the hierarchy within the model—either in Modelica dot notation or via nested dictionaries. If *model* is *None*, then *params* is not used. Python values are automatically mapped to their Modelica equivalent (see `ParamDict.__str__()`). Redeclarations and other prefixes must be included in the keys along with the variable names.

  `gen_experiments()` can be used to create a generator for this argument.

Items with values of *None* in *params* and *args* are skipped.

- *working_dir*: Working directory (for the executable, log files, etc.)

  '~' may be included to represent the user directory.

- *packages*: List of Modelica packages that should be preloaded or scripts that should be run

  Each may be a "*.mo" file, a folder that contains a "package.mo" file, or a "*.mos" file. The path may be absolute or relative to *working_dir*. It may be necessary to include in *packages* the file or folder that contains the model specified by the *model* subargument, but the Modelica Standard Library generally does not need to be included. If an entry is a script ("*.mos"), it is run from its folder.

- ***fname*: Name of the script file to be written (usually in the form** "*.mos")

  This may include the path ('~' for user directory). The results will be stored relative to the same folder. If the folder does not exist, it will be created.

- *command*: Simulation or other command to the Modelica tool or environment

  Instead of the default ('simulateModel'), this could be 'linearizeModel' to create a state space representation or 'translateModel' to create model executables without running them.

- *results*: List of files to copy to the results folder

  Each entry is the path/name of a file that is generated during simulation. The path is relative to the working directory. '%x' may be included in the filename to represent '.exe' if the operating system is Windows and '' otherwise. The result folders are named by the number of the simulation run and placed within the folder that contains the simulation script (*fname*).

If *command* is 'simulateModel' and the Modelica environment is Dymola®, then the following keywords may be used in *args* (see *experiments* above). The defaults (shown in parentheses) are applied by Dymola®—not by this function.

- *startTime* (0): Start of simulation
- *stopTime* (1): End of simulation
- *numberOfIntervals* (0): Number of output points
- *outputInterval* (0): Distance between output points
- *method* ("Dassl"): Integration method
- *tolerance* (0.0001): Tolerance of integration
- *fixedstepsize* (0): Fixed step size for Euler
- *resultFile* ("dsres.mat"): Where to store result

Note that *problem* is not listed. It is generated from *model* and *params*. If *model* is *None*, the currently/previously translated model will be simulated.

Returns:

1. List of model names without full model paths
2. Directory where the script has been saved

Example 1 (single simulation):

```
>>> from modelicares import *

>>> experiment = Experiment(model='Modelica.Electrical.Analog.Examples.ChuaCircuit',
                            params={},
```

```
                        args=dict(stopTime=2500))
>>> write_script(experiment,
             fname="examples/ChuaCircuit/run-sims1.mos")
(['ChuaCircuit'], '...examples/ChuaCircuit')
```

In "examples/ChuaCircuit/run-sims1.mos":

```
import Modelica.Utilities.Files.copy;
import Modelica.Utilities.Files.createDirectory;
Advanced.TranslationInCommandLog = true "Also include translation log in command log";
cd(".../Documents/Modelica");
destination = ".../examples/ChuaCircuit/";

// Experiment 1
ok = simulateModel("Modelica.Electrical.Analog.Examples.ChuaCircuit", stopTime=2500);
if ok then
    savelog();
    createDirectory(destination + "1");
    copy("dsin.txt", destination + "1/dsin.txt", true);
    copy("dslog.txt", destination + "1/dslog.txt", true);
    copy("dsres.mat", destination + "1/dsres.mat", true);
    copy("dymosim", destination + "1/dymosim", true);
    copy("dymolalg.txt", destination + "1/dymolalg.txt", true);
end if;

exit();
```

where "..." depends on the local system.

Example 2 (full-factorial design of experiments):

```
>>> from modelicares import *

>>> experiments = gen_experiments(
    models=["Modelica.Electrical.Analog.Examples.ChuaCircuit"],
    params={'L.L': [18, 20],
            'C1.C': [8, 10],
            'C2.C': [80, 100, 120]})
>>> write_script(experiments, fname="examples/ChuaCircuit/run-sims2.mos")
(['ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'Ch
```

In "examples/ChuaCircuit/run-sims2.mos", there are commands to run and save results from 12 simulation experiments.

CHAPTER

SEVEN

MODELICARES.TEXUNIT

Functions to format numbers to support LaTeX

- `label_number()` - Generate text to label a number as a quantity expressed in a unit
- `label_quantity()` - Generate text to write a quantity as a number times a unit
- `unit2tex()` - Convert a Modelica unit string to LaTeX

`modelicares.texunit.`**`label_number`**(*quantity=''*, *unit=None*, *times='\\, '*, *per='\\,/\\, '*, *roman=False*)

Generate text to label a number as a quantity expressed in a unit.

The unit is formatted with LaTeX as needed.

Arguments:

•*quantity*: String describing the quantity

•*unit*: String specifying the unit

This is expressed in extended Modelica notation. See `unit2tex()`.

•*times*: LaTeX math string to indicate multiplication

times is applied between the number and the first unit and between units. The default is 3/18 quad space. The multiplication between the significand and the exponent is always indicated by "×".

•*per*: LaTeX math string to indicate division

It is applied between the quantity and the units. The default is a 3/18 quad space followed by '/; and another 3/18 quad space. The division associated with the units on the denominator is always indicated by a negative exponential.

If the unit is not a simple scaling factor, then "in" is used instead. For example,

```
>>> label_number("Gain", "dB")
'Gain in $dB$'
```

•*roman*: *True*, if the units should be typeset in Roman text (rather than italics)

Examples:

```
>>> label_number("Mobility", "m2/(V.s)", roman=True)
'Mobility$\\,/\\,\\mathrm{m^{2}\\,V^{-1}\\,s^{-1}}$'
```

in LaTeX: Mobility $\,/\,\mathrm{m^{2}\,V^{-1}\,s^{-1}}$

```
>>> label_number("Mole fraction", "1")
'Mole fraction'
```

modelicares.texunit.**label_quantity**(*number*, *unit=''*, *format='%G'*, *times='\\, '*, *roman=False*)

Generate text to write a quantity as a number times a unit.

If an exponent is present, then either a LaTeX-formatted exponential or a System International (SI) prefix is applied.

Arguments:

- *number*: Floating point or integer number
- *unit*: String specifying the unit

  unit uses extended Modelica notation. See `unit2tex()`.
- *format*: Modified Python number formatting string

  If LaTeX-formatted exponentials should be applied, then then use an uppercase exponential formatter ('E' or 'G'). A lowercase exponential formatter ('e' or 'g') will result in a System International (SI) prefix, if applicable.

  See also:

  http://docs.python.org/release/2.5.2/lib/typesseq-strings.html and http://en.wikipedia.org/wiki/SI_prefix
- *times*: LaTeX math string to indicate multiplication

  times is applied between the number and the first unit and between units. The default is 3/18 quad space. The multiplication between the significand and the exponent is always indicated by "×".
- *roman*: *True*, if the units should be typeset in Roman text (rather than italics)

Examples:

```
>>> label_quantity(1.2345e-3, 'm', format='%.3e', roman=True)
'1.234$\\,\\mathrm{mm}$'
```

in LaTeX: 1.234 mm

```
>>> label_quantity(1.2345e-3, 'm', format='%.3E', roman=True)
'1.234$\\times10^{-3}$$\\,\\mathrm{m}$'
```

in LaTeX: 1.234×10^{-3} m

```
>>> label_quantity(1.2345e6)
'1.2345$\\times10^{6}$'
```

in LaTeX: 1.2345×10^{6}

```
>>> label_quantity(1e3, '\Omega', format='%.1e', roman=True)
'1.0$\\,\\mathrm{k\\Omega}$'
```

in LaTeX: 1.0 kΩ

modelicares.texunit.**unit2tex**(*unit*, *times='\\, '*, *roman=False*)

Convert a Modelica unit string to LaTeX.

Arguments:

- *unit*: Unit string in extended Modelica notation

  See also:

  Modelica Specification, version 3.2, p. 209 (https://www.modelica.org/documents)

In summary, '.' indicates multiplication. The denominator is enclosed in parentheses and begins with a '/'. Exponents directly follow the significand (e.g., no carat ('^')).

- *times*: LaTeX math string to indicate multiplication

  times is applied between the number and the first unit and between units. The default is 3/18 quad space.

- *roman*: *True*, if the units should be typeset in Roman text (rather than italics)

Example:

```
>>> unit2tex("m/s2", roman=True)
'\\mathrm{m\\,s^{-2}}'
```

which will render in LaTeX math as $\mathrm{m\,s^{-2}}$

CHAPTER

EIGHT

MODELICARES.BASE

Classes and functions to help plot and interpret experimental data

Classes:

- `ArrowLine` - A matplotlib subclass to draw an arrowhead on a line
- `Quantity` - Named tuple class for a constant physical quantity

Functions:

- `add_arrows()` - Overlay arrows with annotations on top of a pre-plotted line
- `add_hlines()` - Add horizontal lines to a set of axes with optional labels
- `add_vlines()` - Add vertical lines to a set of axes with optional labels
- `animate()` - Encode a series of PNG images as a MPG movie
- `color()` - Plot 2D scalar data on a color axis in 2D Cartesian coordinates
- `closeall()` - Close all open figures
- `convert()` - Convert the expression of a physical quantity between units
- `expand_path()` - Expand a file path by replacing '~' with the user directory and makes the path absolute
- `flatten_dict()` - Flatten a nested dictionary
- `flatten_list()` - Flatten a nested list
- `figure()` - Create a figure and set its label
- `get_indices()` - Return the pair of indices that bound a target value in a monotonically increasing vector
- `get_pow10()` - Return the exponent of 10 for which the significand of a number is within the range [1, 10)
- `get_pow1000()` - Return the exponent of 1000 for which the significand of a number is within the range [1, 1000)
- `load_csv()` - Load a CSV file into a dictionary
- `plot()` - Plot 1D scalar data as points and/or line segments in 2D Cartesian coordinates
- `quiver()` - Plot 2D vector data as arrows in 2D Cartesian coordinates
- `save()` - Save the current figures as images in a format or list of formats
- `saveall()` - Save all open figures as images in a format or list of formats
- `setup_subplots()` - Create an array of subplots and return their axes
- `shift_scale_x()` - Apply an offset and a factor as necessary to the x axis
- `shift_scale_y()` - Apply an offset and a factor as necessary to the y axis

class modelicares.base.**Quantity**

Named tuple class for a constant physical quantity

The factor and then the offset are applied to the number to arrive at the quantity expressed in terms of the unit.

__repr__()

Return a nicely formatted representation string

factor

Alias for field number 1

number

Alias for field number 0

offset

Alias for field number 2

unit

Alias for field number 3

modelicares.base.**add_arrows**(*p, x_locs=[0], xstar_offset=0, ystar_offset=0, lstar=0.05, label='', orientation='tangent', color='r'*)

Overlay arrows with annotations on top of a pre-plotted line.

Arguments:

- *p*: A plot instance (`matplotlib.lines.Line2D` object)
- *x_locs*: x-axis locations of the arrows
- *xstar_offset*: Normalized x-axis offset from the middle of the arrow to the text
- *ystar_offset*: Normalized y-axis offset from the middle of the arrow to the text
- *lstar*: Length of each arrow in normalized xy axes
- *label*: Annotation text
- *orientation*: 'tangent', 'horizontal', or 'vertical'
- *color*: Color of the arrows (from `matplotlib.colors`)

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> # Create a plot.
>>> figure('examples/add_arrows')
<matplotlib.figure.Figure object at 0x...>
>>> x = np.arange(100)
>>> p = plt.plot(x, np.sin(x/4.0))

>>> # Add arrows and annotations.
>>> add_arrows(p[0], x_locs=x.take(np.arange(20,100,20)),
           label="Incr. time", xstar_offset=-0.15)
>>> save()
Saved examples/add_arrows.pdf
Saved examples/add_arrows.png
>>> plt.show()
```

modelicares.base.**add_hlines**(*ax=None, positions=[0], labels=[], **kwargs*)

Add horizontal lines to a set of axes with optional labels.

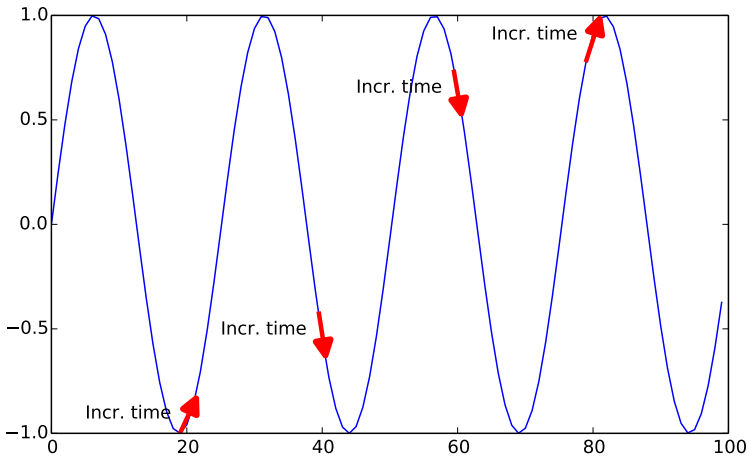


Figure 8.1: Example of add_arrows()

Arguments:

- *ax*: Axes (`matplotlib.axes` object)
- *positions*: Positions (along the x axis)
- *labels*: List of labels for the lines
- ***kwargs*: Line properties (propagated to `matplotlib.pyplot.axhline()`)

  E.g., `color='k', linestyle='--', linewidth=0.5`

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> # Create a plot.
>>> figure('examples/add_hlines')
<matplotlib.figure.Figure object at 0x...>
>>> x = np.arange(100)
>>> y = np.sin(x/4.0)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-1.2, 1.2])
(-1.2, 1.2)

>>> # Add horizontal lines and labels.
>>> add_hlines(positions=[min(y), max(y)], labels=["min", "max"],
           color='r', ls='--')
>>> save()
Saved examples/add_hlines.pdf
Saved examples/add_hlines.png
>>> plt.show()
```

`modelicares.base.`**`add_vlines`**(*ax=None, positions=[0], labels=[], **kwargs*)

Add vertical lines to a set of axes with optional labels.

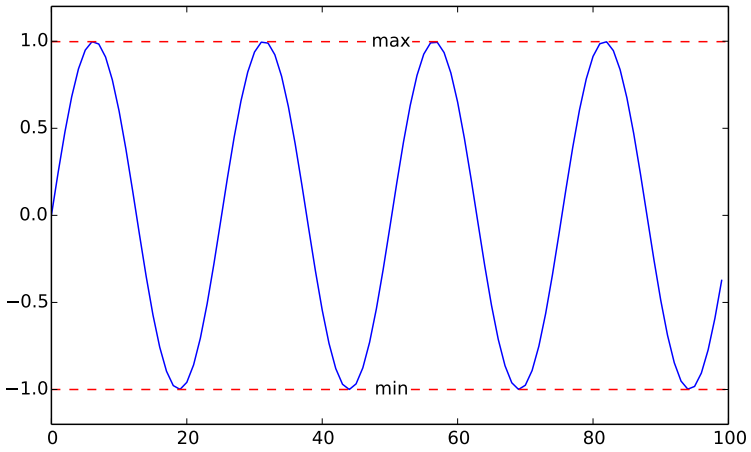


Figure 8.2: Example of add_hlines()

Arguments:

- *ax*: Axes (matplotlib.axes object)
- *positions*: Positions (along the x axis)
- *labels*: List of labels for the lines
- ***kwargs*: Line properties (propagated to `matplotlib.pyplot.axvline()`)

  E.g., `color='k', linestyle='--', linewidth=0.5`

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> # Create a plot.
>>> figure('examples/add_vlines')
<matplotlib.figure.Figure object at 0x...>
>>> x = np.arange(100)
>>> y = np.sin(x/4.0)
>>> plt.plot(x, y)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-1.2, 1.2])
(-1.2, 1.2)

>>> # Add horizontal lines and labels.
>>> add_vlines(positions=[25, 50, 75], labels=["A", "B", "C"],
           color='k', ls='--')
>>> save()
Saved examples/add_vlines.pdf
Saved examples/add_vlines.png
>>> plt.show()
```

`modelicares.base.`**`animate`**(*imagebase='_tmp'*, *fname='animation'*, *fps=10*, *clean=False*)

Encode a series of PNG images as a MPG movie.

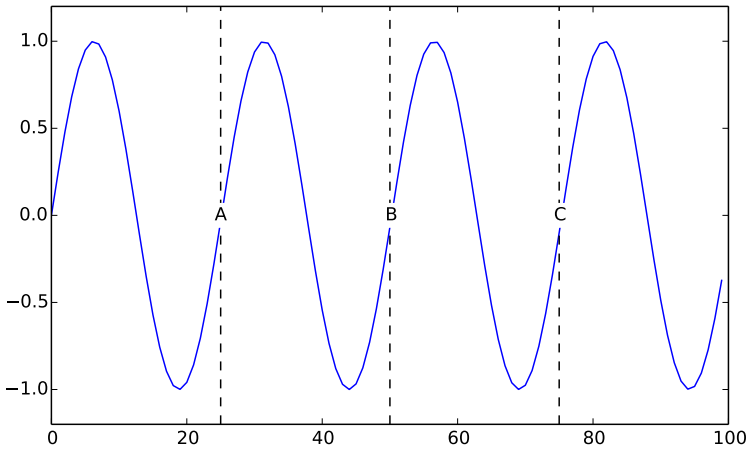


Figure 8.3: Example of add_vlines()

Arguments:

- *imagebase*: Base filename for the PNG images

  The images should be located in the current directory as an "*imagebase***xx*.png" sequence, where *xx* is a frame index.

- *fname*: Filename for the movie

  ".mpg" will be appended if necessary.

- *fps*: Number of frames per second

- *clean*: *True*, if the PNG images should be deleted afterward

---

Note: This function requires mencoder. On Linux, install it with the following command: `sudo apt-get install mencoder`. Currently, this function is not supported on Windows.

Example:

```
import matplotlib.pyplot as plt
from numpy.random import rand
from modelicares import *

# Create the frames.
fig = plt.figure(figsize=(5,5))
ax = fig.add_subplot(111)
for i in range(50):  # 50 frames
    ax.cla()
    ax.imshow(rand(5,5), interpolation='nearest')
    fname = '_tmp%02d.png' % i
    print("Saving frame %i (file %s)" % (i, fname))
    fig.savefig(fname) # doctest: +ELLIPSIS

# Assemble the frames into a movie.
animate(clean=True)
```

modelicares.base.**closeall**()

Close all open figures.

This is a shortcut for the following:

```
>>> from matplotlib._pylab_helpers import Gcf
>>> Gcf.destroy_all()
```

modelicares.base.**color**(*ax*, *c*, **args*, ***kwargs*)

Plot 2D scalar data on a color axis in 2D Cartesian coordinates.

This uses a uniform grid.

Arguments:

- *ax*: Axis onto which the data should be plotted
- *c*: color- or c-axis data (2D array)
- **args*, ***kwargs*: Additional arguments for `matplotlib.pyplot.imshow()`

Example:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from modelicares import *

>>> figure('examples/color')
<matplotlib.figure.Figure object at 0x...>
>>> x, y = np.meshgrid(np.arange(0, 2*np.pi, 0.2),
                       np.arange(0, 2*np.pi, 0.2))
>>> c = np.cos(x) + np.sin(y)
>>> ax = plt.subplot(111)
>>> color(ax, c)
<matplotlib.image.AxesImage object at 0x...>
>>> save()
Saved examples/color.pdf
Saved examples/color.png
>>> plt.show()
```

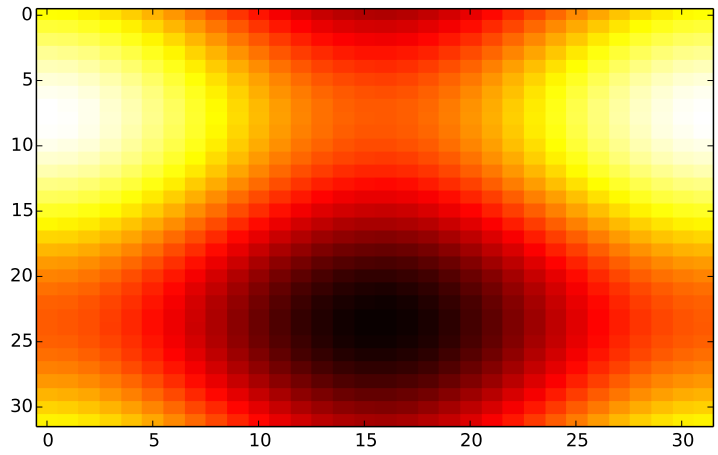


Figure 8.4: Example of color()

modelicares.base.**convert**(*quantity*)
Convert the expression of a physical quantity between units.

Arguments:

- *quantity*: Instance of `Quantity`

Example:

```
>>> from modelicares import *

>>> T = 293.15 # Temperature in K
>>> T_degC = convert(Quantity(T, factor=1, offset=-273.15, unit='C'))
>>> print(str(T) + " K is " + str(T_degC) + " degC.")
293.15 K is 20.0 degC.
```

modelicares.base.**expand_path**(*path*)
Expand a file path by replacing '~' with the user directory and making the path absolute.

Example:

```
>>> from modelicares import *

>>> expand_path('~/Documents')
'...Documents'
>>> # where ... is '/home/user/' on Linux or 'C:\Users\user\' on
>>> # Windows (and "user" is the user id).
```

modelicares.base.**figure**(*label=''*, **args*, ***kwargs*)
Create a figure and set its label.

Arguments:

- *label*: String to apply to the figure's *label* property
- **args*, ***kwargs*: Additional arguments for `matplotlib.pyplot.figure()`

Example:

```
>>> fig = figure("velocity_vs_time")
>>> plt.getp(fig, 'label')
u'velocity_vs_time'
```

Note: The *label* property is used as the base filename in the `saveall()` method.

---

modelicares.base.**flatten_dict**(*d*, *parent_key=''*, *separator='.'*)
Flatten a nested dictionary.

Arguments:

- *d*: Dictionary (may be nested to an arbitrary depth)
- *parent_key*: Key of the parent dictionary, if any
- *separator*: String or character that joins elements of the keys or path names

Example:

```
>>> from modelicares import *
>>> flatten_dict(dict(a=1, b=dict(c=2, d='hello')))
{'a': 1, 'b.c': 2, 'b.d': 'hello'}
```

modelicares.base.**flatten_list**(*l*, *ltypes=(<type 'list'>, <type 'tuple'>)*)

Flatten a nested list.

Arguments:

- *l*: List (may be nested to an arbitrary depth)

  If the type of *l* is not in ltypes, then it is placed in a list.

- *ltypes*: Tuple (not list) of accepted indexable types

Example:

```
>>> from modelicares import *
>>> flatten_list([1, [2, 3, [4]]])
[1, 2, 3, 4]
```

modelicares.base.**get_indices**(*x*, *target*)

Return the pair of indices that bound a target value in a monotonically increasing vector.

Arguments:

- *x*: Vector

- *target*: Target value

Example:

```
>>> from modelicares import *
>>> get_indices([0,1,2],1.6)
(1, 2)
```

modelicares.base.**get_pow10**(*num*)

Return the exponent of 10 for which the significand of a number is within the range [1, 10).

Example:

```
>>> get_pow10(50)
1
```

modelicares.base.**get_pow1000**(*num*)

Return the exponent of 1000 for which the significand of a number is within the range [1, 1000).

Example:

```
>>> get_pow1000(1e5)
1
```

modelicares.base.**load_csv**(*fname*, *header_row=0*, *first_data_row=None*, *types=None*, ***kwargs*)

Load a CSV file into a dictionary.

The strings from the header row are used as dictionary keys.

Arguments:

- *fname*: Path and name of the file

- *header_row*: Row that contains the keys (uses zero-based indexing)

- *first_data_row*: First row of data (uses zero-based indexing)

  If *first_data_row* is not provided, then it is assumed that the data starts just after the header row.

- *types*: List of data types for each column

  `int` and `float` data types will be cast into a `numpy.array`. If *types* is not provided, attempts will be made to cast each column into `int`, `float`, and `str` (in that order).

- ***kwargs*: Additional arguments for `csv.reader()`

Example:

```
>>> from modelicares import *
>>> data = load_csv("examples/load-csv.csv", header_row=2)
>>> print("The keys are: %s" % data.keys())
The keys are: ['Price', 'Description', 'Make', 'Model', 'Year']
```

`modelicares.base.`**`plot`**(*y, x=None, ax=None, label=None, color=['b', 'g', 'r', 'c', 'm', 'y', 'k'], marker=None, dashes=[(None, None), (3, 3), (1, 1), (3, 2, 1, 2)], **kwargs*)

Plot 1D scalar data as points and/or line segments in 2D Cartesian coordinates.

This is similar to `matplotlib.pyplot.plot()` (and actually calls that method), but provides direct support for plotting an arbitrary number of curves.

Arguments:

- *y*: y-axis data

  This may contain multiple series.

- *x*: x-axis data

  If *x* is not provided, the y-axis data will be plotted versus its indices. If *x* is a single series, it will be used for all of the y-axis series. If it is a list of series, each x-axis series will be matched to a y-axis series.

- *ax*: Axis onto which the data should be plotted.

  If *ax* is *None* (default), axes are created.

- *label*: List of labels of each series (to be used later for the legend if applied)

- *color*: Single entry, list, or `itertools.cycle` of colors that will be used sequentially

  Each entry may be a character, grayscale, or rgb value.

  See also:

  http://matplotlib.sourceforge.net/api/colors_api.html

- *marker*: Single entry, list, or `itertools.cycle` of markers that will be used sequentially

  Use *None* for no marker. A good assortment is [”o”, “v”, “^”, “<”, “>”, “s”, “p”, “*”, “h”, “H”, “D”, “d”]. All of the possible entries are listed at: http://matplotlib.sourceforge.net/api/artist_api.html#matplotlib.lines.Line2D.set_marker.

- *dashes*: Single entry, list, or `itertools.cycle` of dash styles that will be used sequentially

  Each style is a tuple of on/off lengths representing dashes. Use (0, 1) for no line and (None, None) for a solid line.

  See also:

  http://matplotlib.sourceforge.net/api/collections_api.html

- ***kwargs*: Additional arguments for `matplotlib.pyplot.plot()`

Returns: List of `matplotlib.lines.Line2D` objects

Example:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from modelicares import *
```

```
>>> figure('examples/plot')
<matplotlib.figure.Figure object at 0x...>
>>> ax = plt.subplot(111)
>>> plot([range(11), range(10, -1, -1)], ax=ax)
[[<matplotlib.lines.Line2D object at 0x...>], [<matplotlib.lines.Line2D object at 0x...>]]
>>> save()
Saved examples/plot.pdf
Saved examples/plot.png
>>> plt.show()
```

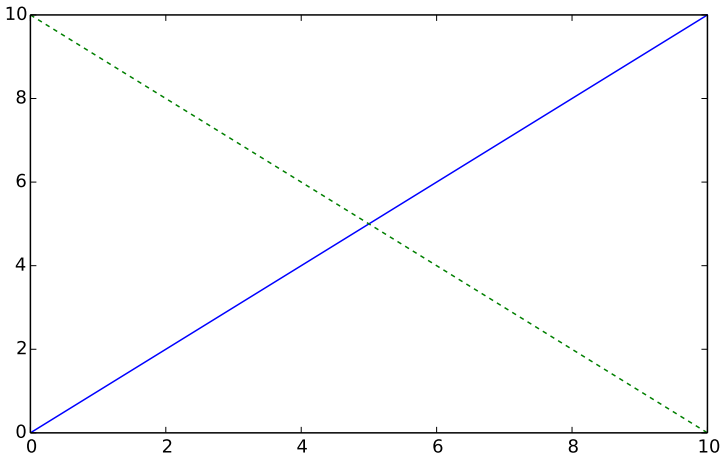


Figure 8.5: Example of plot()

modelicares.base.**quiver**(*ax*, *u*, *v*, *x=None*, *y=None*, *pad=0.05*, *pivot='middle'*, ***kwargs*)

Plot 2D vector data as arrows in 2D Cartesian coordinates.

Uses a uniform grid.

Arguments:

- *ax*: Axis onto which the data should be plotted
- *u*: x-direction values (2D array)
- *v*: y-direction values (2D array)
- *pad*: Amount of white space around the data (relative to the span of the field)
- *pivot*: "tail" | "middle" | "tip" (see `matplotlib.pyplot.quiver()`)
- ***kwargs*: Additional arguments for `matplotlib.pyplot.quiver()`

Example:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from modelicares import *

>>> figure('examples/quiver')
<matplotlib.figure.Figure object at 0x...>
>>> x, y = np.meshgrid(np.arange(0, 2*np.pi, 0.2),
                       np.arange(0, 2*np.pi, 0.2))
```

```
>>> u = np.cos(x)
>>> v = np.sin(y)
>>> ax = plt.subplot(111)
>>> quiver(ax, u, v)
<matplotlib.quiver.Quiver object at 0x...>
>>> save()
Saved examples/quiver.pdf
Saved examples/quiver.png
>>> plt.show()
```

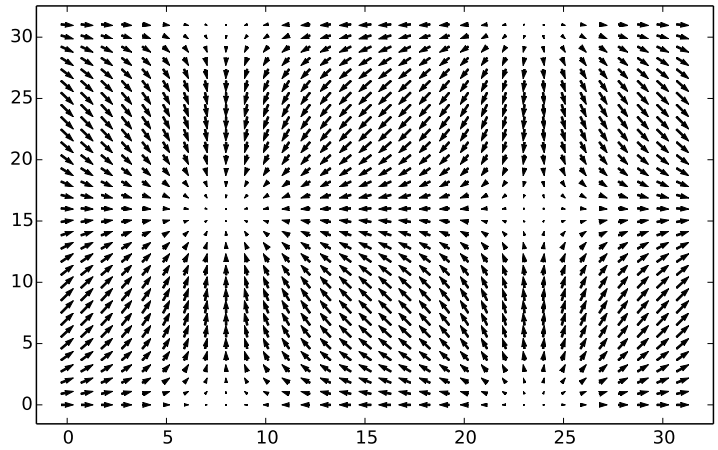


Figure 8.6: Example of quiver()

modelicares.base.**save**(*formats=['pdf', 'png']*, *fbase='1'*)

Save the current figures as images in a format or list of formats.

The directory and base filenames are taken from the *label* property of the figures. A slash ("/") can be used as a path separator, even if the operating system is Windows. Folders are created as needed. If the *label* property is empty, then a directory dialog is opened to chose a directory.

Arguments:

- *formats*: Format or list of formats in which the figure should be saved
- *fbase*: Default directory and base filename

  This is used if the *label* attribute of the figure is empty ('').

Note: In general, `save()` should be called before `matplotlib.pyplot.show()` so that the figure(s) are still present in memory.

Example:

```
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> figure('temp_plot')
<matplotlib.figure.Figure object at 0x...>
>>> plt.plot(range(10))
[<matplotlib.lines.Line2D object at 0x...>]
```

```
>>> save()
Saved temp_plot.pdf
Saved temp_plot.png
```

Note: The `figure()` method can be used to directly create a figure with a label.

modelicares.base.**saveall**(*formats=['pdf', 'png']*)

Save all open figures as images in a format or list of formats.

The directory and base filenames are taken from the *label* property of the figures. A slash ("/") can be used as a path separator, even if the operating system is Windows. Folders are created as needed. If the *label* property is empty, then a directory dialog is opened to chose a directory. In that case, the figures are saved as a sequence of numbers.

Arguments:

- *formats*: Format or list of formats in which the figures should be saved

Note: In general, `saveall()` should be called before `matplotlib.pyplot.show()` so that the figure(s) are still present in memory.

Example:

```
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> figure('temp_plot')
<matplotlib.figure.Figure object at 0x...>
>>> plt.plot(range(10))
[<matplotlib.lines.Line2D object at 0x...>]
>>> save()
Saved temp_plot.pdf
Saved temp_plot.png
```

Note: The `figure()` method can be used to directly create a figure with a label.

modelicares.base.**setup_subplots**(*n_plots*, *n_rows*, *title=''*, *subtitles=None*, *label='multiplot'*, *xlabel=''*, *xticklabels=None*, *xticks=None*, *ylabel=''*, *yticklabels=None*, *yticks=None*, *ctype=None*, *clabel=''*, *margin_left=0.125*, *margin_right=0.09999999999999998*, *margin_bottom=0.1*, *margin_top=0.125*, *margin_cbar=0.2*, *wspace=0.1*, *hspace=0.25*, *cbar_space=0.1*, *cbar_width=0.05*)

Create an array of subplots and return their axes.

Arguments:

- *n_plots*: Number of (sub)plots
- *n_rows*: Number of rows of (sub)plots
- *title*: Title for the figure
- *subtitles*: List of subtitles (i.e., titles for each subplot) or *None* for no subtitles
- *label*: Label for the figure

  This will be used as a base filename if the figure is saved.
- *xlabel*: Label for the x-axes (only shown for the subplots on the bottom row)

- *xticklabels*: Labels for the x-axis ticks (only shown for the subplots on the bottom row)

  If *None*, then the default is used.

- *xticks*: Positions of the x-axis ticks

  If *None*, then the default is used.

- *ylabel*: Label for the y-axis (only shown for the subplots on the left column)

- *yticklabels*: Labels for the y-axis ticks (only shown for the subplots on the left column)

  If *None*, then the default is used.

- *yticks*: Positions of the y-axis ticks

  If *None*, then the default is used.

- *ctype*: Type of colorbar (*None*, 'vertical', or 'horizontal')

- *clabel*: Label for the color- or c-bar axis

- *margin_left*: Left margin

- *margin_right*: Right margin (ignored if `cbar_orientation == 'vertical'`)

- *margin_bottom*: Bottom margin (ignored if `cbar_orientation == 'horizontal'`)

- *margin_top*: Top margin

- *margin_cbar*: Margin reserved for the colorbar (right margin if `cbar_orientation == 'vertical'` and bottom margin if `cbar_orientation == 'horizontal'`)

- *wspace*: The amount of width reserved for blank space between subplots

- *hspace*: The amount of height reserved for white space between subplots

- *cbar_space*: Space between the subplot rectangles and the colorbar

  If *cbar* is *None*, then this is ignored.

- *cbar_width*: Width of the colorbar if vertical (or height if horizontal)

  If *cbar* is *None*, then this is ignored.

Returns:

1.List of subplot axes

2.Colorbar axis (returned iff `cbar != None`)

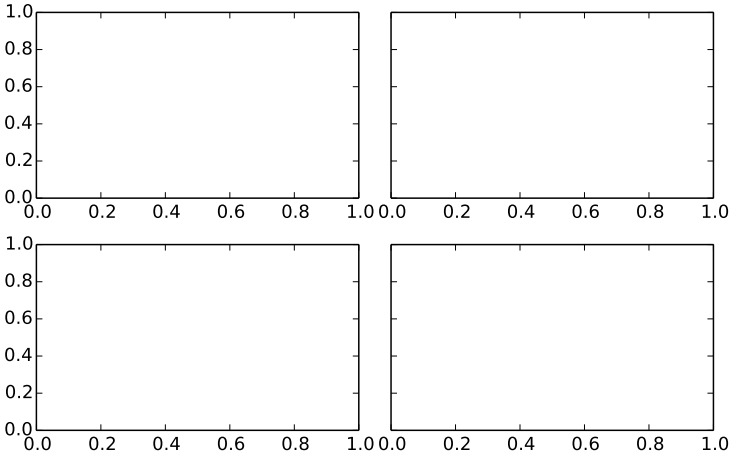
3.Number of columns of subplots

Example:

```
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> setup_subplots(4, 2, label='examples/setup_subplots')
([<matplotlib.axes...AxesSubplot object at 0x...>, <matplotlib.axes...AxesSubplot object at 0x...>, <matplotli
>>> save()
Saved examples/setup_subplots.pdf
Saved examples/setup_subplots.png
>>> plt.show()
```

Example of setup_subplots()



modelicares.base.**shift_scale_x**(*ax*, *eagerness=0.325*)

Apply an offset and a factor as necessary to the x axis.

Arguments:

- *ax*: matplotlib.axes object
- *eagerness*: Parameter to adjust how little of an offset is required before the label will be recentered
  - 0: Offset is never applied.
  - 1: Offset is always applied if it will help.

Example:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from texunit import label_number
>>> from modelicares import *

>>> # Generate some random data.
>>> x = np.linspace(55478, 55486, 100) # Small range and large offset
>>> xlabel = label_number('Time', 's')
>>> y = np.cumsum(np.random.random(100) - 0.5)

>>> # Plot the data.
>>> ax = setup_subplots(2, 2, label='examples/shift_scale_x')[0]
>>> for a in ax:
    a.plot(x, y)
    a.set_xlabel(xlabel)
[<matplotlib.lines.Line2D object at 0x...>]
<matplotlib.text.Text object at 0x...>
[<matplotlib.lines.Line2D object at 0x...>]
<matplotlib.text.Text object at 0x...>

>>> # Shift and scale the axes.
>>> ax[0].set_title('Original plot')
<matplotlib.text.Text object at 0x...>
>>> ax[1].set_title('After applying offset and factor')
<matplotlib.text.Text object at 0x...>
>>> shift_scale_x(ax[1])
```

```
>>> save()
Saved examples/shift_scale_x.pdf
Saved examples/shift_scale_x.png
>>> plt.show()
```

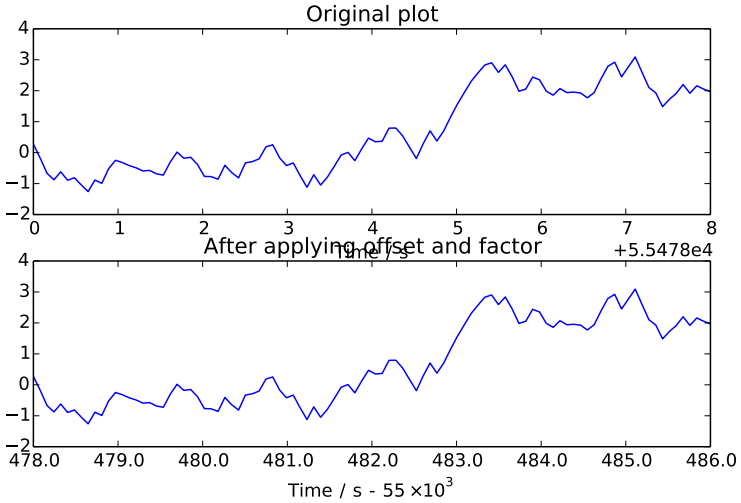


Figure 8.7: Example of shift_scale_x()

modelicares.base.**shift_scale_y**(*ax*, *eagerness=0.325*)

Apply an offset and a factor as necessary to the y axis.

Arguments:

- *ax*: matplotlib.axes object
- *eagerness*: Parameter to adjust how little of an offset is required before the label will be recentered
  - 0: Offset is never applied.
  - 1: Offset is always applied if it will help.

Example:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from texunit import label_number
>>> from modelicares import *

>>> # Generate some random data.
>>> x = range(100)
>>> y = np.cumsum(np.random.random(100) - 0.5)
>>> y -= y.min()
>>> y *= 1e-3
>>> y += 1e3 # Small magnitude and large offset
>>> ylabel = label_number('Velocity', 'mm/s')

>>> # Plot the data.
>>> ax = setup_subplots(2, 2, label='examples/shift_scale_y')[0]
>>> for a in ax:
    a.plot(x, y)
    a.set_ylabel(ylabel)
```

```
[<matplotlib.lines.Line2D object at 0x...>]
<matplotlib.text.Text object at 0x...>
[<matplotlib.lines.Line2D object at 0x...>]
<matplotlib.text.Text object at 0x...>

>>> # Shift and scale the axes.
>>> ax[0].set_title('Original plot')
<matplotlib.text.Text object at 0x...>
>>> ax[1].set_title('After applying offset and factor')
<matplotlib.text.Text object at 0x...>
>>> shift_scale_y(ax[1])
>>> save()
Saved examples/shift_scale_y.pdf
Saved examples/shift_scale_y.png
>>> plt.show()
```

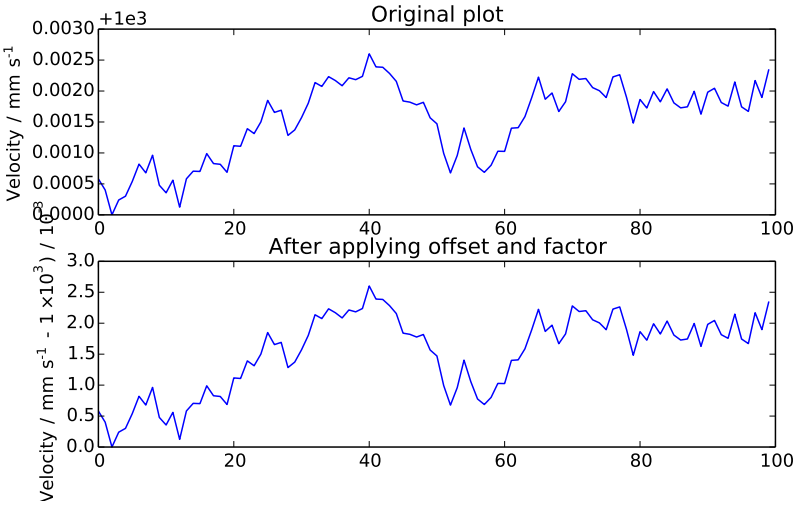


Figure 8.8: Example of shift_scale_y()

class modelicares.base.**ArrowLine**(**args*, ***kwargs*)

A matplotlib subclass to draw an arrowhead on a line

Initialize the line and arrow.

Arguments:

- *arrow* (='-'): Type of arrow ('<' | '-' | '>')
- *arrowsize* (=2*4): Size of arrow
- *arrowedgecolor* ='b'): Color of arrow edge
- *arrowfacecolor* ='b'): Color of arrow face
- *arrowedgewidth* (=4): Width of arrow edge
- *arrowheadwidth* (=*arrowsize*): Width of arrow head
- *arrowheadlength* (=*arrowsize*): Length of arrow head
- **args*, ***kwargs*: Additional arguments for `matplotlib.lines.Line2D`

Example:

```
>>> import matplotlib.pyplot as plt
>>> from modelicares import *

>>> fig = figure('examples/ArrowLine')
>>> ax = fig.add_subplot(111, autoscale_on=False)
>>> t = [-1,2]
>>> s = [0,-1]
>>> line = ArrowLine(t, s, color='b', ls='-', lw=2, arrow='>',
                     arrowsize=20)
>>> ax.add_line(line)
<modelicares.base.ArrowLine object at 0x...>
>>> ax.set_xlim(-3, 3)
(-3, 3)
>>> ax.set_ylim(-3, 3)
(-3, 3)
>>> save()
Saved examples/ArrowLine.pdf
Saved examples/ArrowLine.png
>>> plt.show()
```

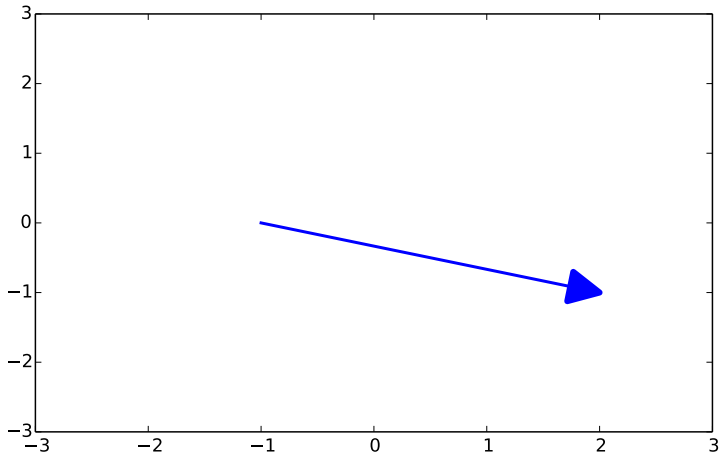


Figure 8.9: Example of ArrowLine

m

modelicares, 6
modelicares.base, 41
modelicares.exps, 30
modelicares.linres, 20
modelicares.multi, 24
modelicares.simres, 7
modelicares.texunit, 38

Symbols

__call__() (modelicares.simres.SimRes method), 9
__contains__() (modelicares.simres.SimRes method), 9
__getitem__() (modelicares.simres.SimRes method), 9
__len__() (modelicares.simres.SimRes method), 10
__repr__() (modelicares.base.Quantity method), 42
__repr__() (modelicares.exps.Experiment method), 31
__repr__() (modelicares.linres.LinRes method), 20
__repr__() (modelicares.simres.SimRes method), 10
__str__() (modelicares.exps.ParamDict method), 31
__str__() (modelicares.linres.LinRes method), 21
__str__() (modelicares.simres.SimRes method), 10

A

add_arrows() (in module modelicares.base), 42
add_hlines() (in module modelicares.base), 42
add_vlines() (in module modelicares.base), 43
animate() (in module modelicares.base), 44
args (modelicares.exps.Experiment attribute), 31
ArrowLine (class in modelicares.base), 56

B

bode() (modelicares.linres.LinRes method), 21
browse() (modelicares.simres.SimRes method), 10

C

closeall() (in module modelicares.base), 45
color() (in module modelicares.base), 46
convert() (in module modelicares.base), 46

D

description() (modelicares.simres.Info method), 7
displayUnit() (modelicares.simres.Info method), 7

E

expand_path() (in module modelicares.base), 47
Experiment (class in modelicares.exps), 30

F

factor (modelicares.base.Quantity attribute), 42
figure() (in module modelicares.base), 47
flatten_dict() (in module modelicares.base), 47
flatten_list() (in module modelicares.base), 47
FV() (modelicares.simres.Info method), 7

G

gen_experiments() (in module modelicares.exps), 31
get_description() (modelicares.simres.SimRes method), 11
get_displayUnit() (modelicares.simres.SimRes method), 12
get_FV() (modelicares.simres.SimRes method), 11
get_indices() (in module modelicares.base), 48
get_indices_wi_times() (modelicares.simres.SimRes method), 12
get_IV() (modelicares.simres.SimRes method), 11
get_max() (modelicares.simres.SimRes method), 12
get_mean() (modelicares.simres.SimRes method), 13
get_min() (modelicares.simres.SimRes method), 13
get_pow10() (in module modelicares.base), 48
get_pow1000() (in module modelicares.base), 48
get_times() (modelicares.simres.SimRes method), 13
get_tuple() (modelicares.simres.SimRes method), 14
get_unit() (modelicares.simres.SimRes method), 14
get_values() (modelicares.simres.SimRes method), 14
get_values_at_times() (modelicares.simres.SimRes method), 15

I

indices_wi_times() (modelicares.simres.Info method), 7
Info (class in modelicares.simres), 7
IV() (modelicares.simres.Info method), 7

L

label_number() (in module modelicares.texunit), 38
label_quantity() (in module modelicares.texunit), 38
LinRes (class in modelicares.linres), 20
load_csv() (in module modelicares.base), 48

M

max() (modelicares.simres.Info method), 7
mean() (modelicares.simres.Info method), 7
merge_times() (in module modelicares.simres), 19
min() (modelicares.simres.Info method), 7

model (modelicares.exps.Experiment attribute), 31
modelica_array() (in module modelicares.exps), 33
modelica_boolean() (in module modelicares.exps), 33
modelicares (module), 6
modelicares.base (module), 41
modelicares.exps (module), 30
modelicares.linres (module), 20
modelicares.multi (module), 24
modelicares.simres (module), 7
modelicares.texunit (module), 38
multibode() (in module modelicares.multi), 24
multiload() (in module modelicares.multi), 25
multinyquist() (in module modelicares.multi), 26
multiplot() (in module modelicares.multi), 27

N

names() (modelicares.simres.SimRes method), 15
nametree() (modelicares.simres.SimRes method), 16
number (modelicares.base.Quantity attribute), 42
nyquist() (modelicares.linres.LinRes method), 22

O

offset (modelicares.base.Quantity attribute), 42

P

ParamDict (class in modelicares.exps), 31
params (modelicares.exps.Experiment attribute), 31
plot() (in module modelicares.base), 49
plot() (modelicares.simres.SimRes method), 16

Q

Quantity (class in modelicares.base), 41
quiver() (in module modelicares.base), 50

R

read_params() (in module modelicares.exps), 34
run_models() (in module modelicares.exps), 34

S

sankey() (modelicares.simres.SimRes method), 17
save() (in module modelicares.base), 51
saveall() (in module modelicares.base), 52
setup_subplots() (in module modelicares.base), 52
shift_scale_x() (in module modelicares.base), 53
shift_scale_y() (in module modelicares.base), 55
SimRes (class in modelicares.simres), 8

T

times() (modelicares.simres.Info method), 7
tuple() (modelicares.simres.Info method), 7

U

unit (modelicares.base.Quantity attribute), 42
unit() (modelicares.simres.Info method), 8
unit2tex() (in module modelicares.texunit), 39

V

values() (modelicares.simres.Info method), 8
values_at_times() (modelicares.simres.Info method), 8

W

write_params() (in module modelicares.exps), 35
write_script() (in module modelicares.exps), 35