ModelicaRes Documentation

Release 0.5

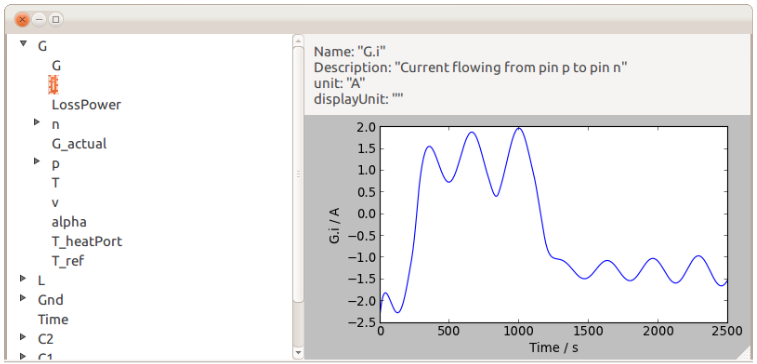
Kevin Davies

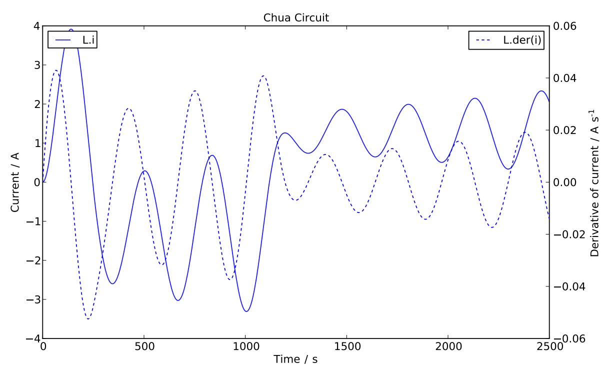
January 22, 2013

CONTENTS

Python utilities to set up and analyze Modelica simulation experiments

The goal of ModelicaRes is to provide an open-source tool to effectively manage Modelica simulations, interpret results, and create publishable figures. It is currently possible to auto-generate simulation scripts, run model executables with varying parameters, browse data, perform 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 command-line interpreter with math and matrix functions from NumPy.



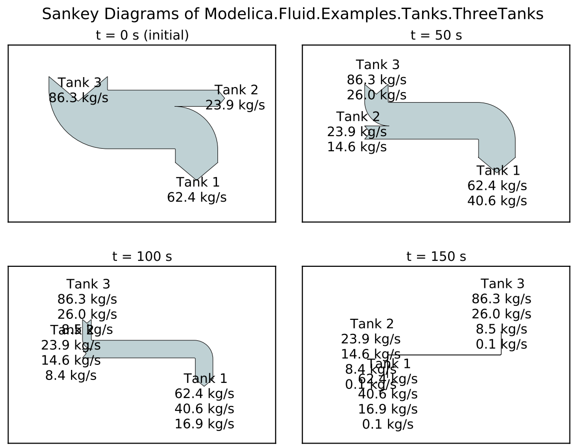


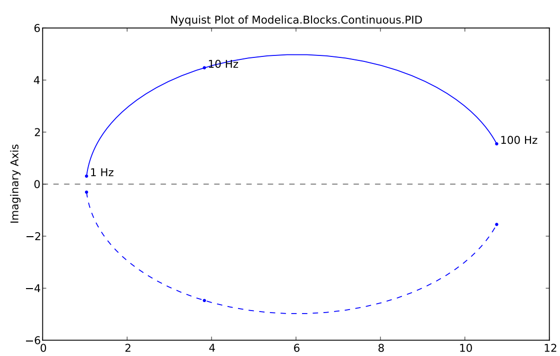
The following chapters describe the components of ModelicaRes. The top-level module, `modelicares`, provides direct access to the most important classes and functions. Others must be accessed through their submodules. The `modelicares.base` submodule has general supporting functions. The `modelicares.exps` submodule has classes and functions to set up and manage simulation experiments. The `modelicares.linres` submodule has a class to load, analyze, and plot results from linearizing a model. The `modelicares.simres` submodule has classes to load, analyze, and plot simulation results. The `modelicares.multi` submodule has functions to load and plot results from multiple data files at once. The last submodule, `modelicares.texunit`, has functions to translate Modelica *unit* and *displayUnit* strings into LaTeX-formatted strings. Finally, the `loadres` script loads data files and provides a Python command-line interpreter to help analyze them.

Updates to ModelicaRes may be available at the main project site. ModelicaRes is also listed in the Python Package Index (direct link). The development site is https://github.com/kdavies4/ModelicaRes.

See Also:

The pysimulator, BuildingsPy, and DyMat projects provide other Python modules that are related. pysimulator includes an elaborate GUI and supports the Functional Model Interface (FMI). BuildingsPy has a


Sankey Diagrams of Modelica.Fluid.Examples.Tanks.ThreeTanks
t = 0 s (initial)
Tank 3
86.3 kg/s
Tank 2
23.9 kg/s
Tank 1
62.4 kg/s
t = 50 s
Tank 3
86.3 kg/s
26.0 kg/s
Tank 2
23.9 kg/s
14.6 kg/s
Tank 1
62.4 kg/s
40.6 kg/s
t = 100 s
Tank 3
86.3 kg/s
26.0 kg/s
23.9 kg/s
14.6 kg/s
8.4 kg/s
Tank 1
62.4 kg/s
40.6 kg/s
16.9 kg/s
t = 150 s
Tank 3
86.3 kg/s
26.0 kg/s
8.5 kg/s
0.1 kg/s
Tank 2
23.9 kg/s
14.6 kg/s
40.6 kg/s
16.9 kg/s
0.1 kg/s


Nyquist Plot of Modelica.Blocks.Continuous.PID
10 Hz
100 Hz
1 Hz
Imaginary Axis
Real Axis
6
4
2
0
−2
−4
−6
0
2
4
6
8
10
12

`Tester` class that can be used for unit testing. DyMat has functions to export Modelica simulation data to comma separated values (CSV), Gnuplot, MATLAB®, and Network Common Data Form (netCDF).

CHAPTER

ONE

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

CHAPTER

TWO

MODELICARES.BASE

Basic methods to help plot and interpret experimental data

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)
saveall()
Saved examples/add_arrows.pdf
Saved examples/add_arrows.png
plt.show()
```

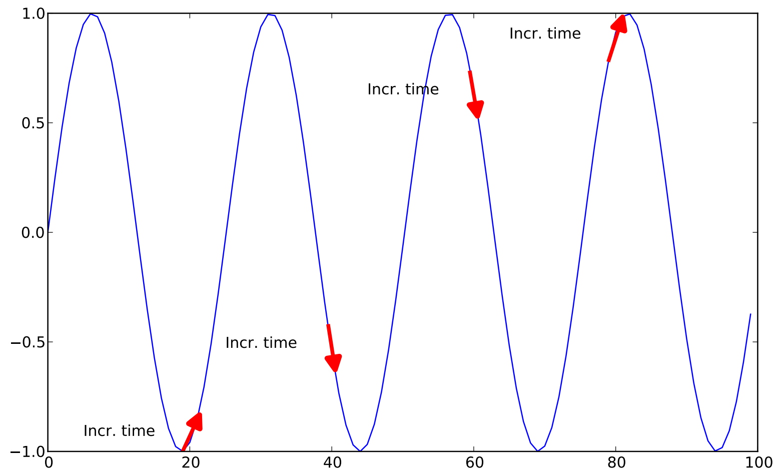


Figure 2.1: Example of add_arrows()

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

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

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='--')
saveall()
Saved examples/add_hlines.pdf
Saved examples/add_hlines.png
plt.show()
```

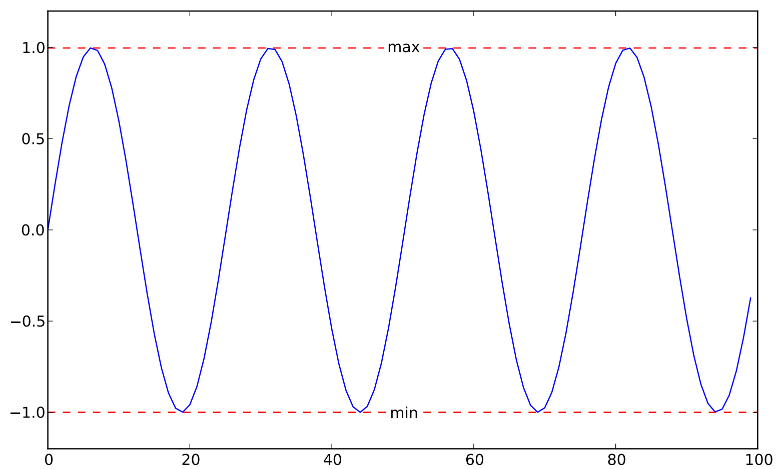


Figure 2.2: Example of add_hlines()

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

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

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='--')
saveall()
Saved examples/add_vlines.pdf
Saved examples/add_vlines.png
plt.show()
```

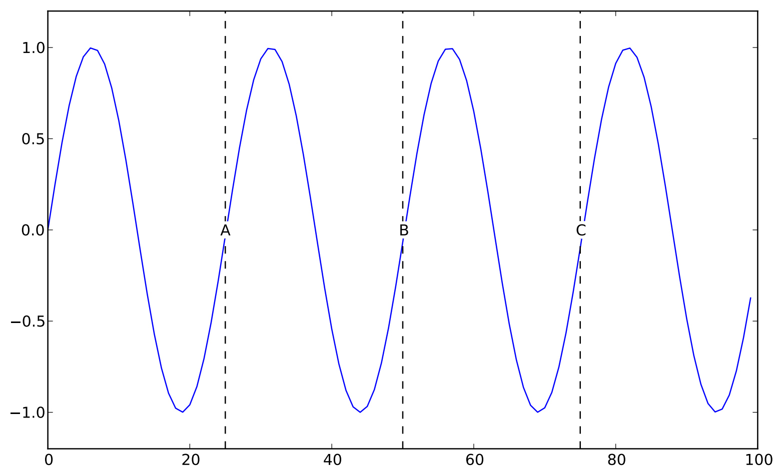


Figure 2.3: Example of add_vlines()

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

Encode a series of PNG images as a MPG movie.

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*: Propagated to `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...>
saveall()
Saved examples/color.pdf
Saved examples/color.png
plt.show()
```

modelicares.base.**convert**(*quantity*)

Convert the expression of a physical quantity between units.

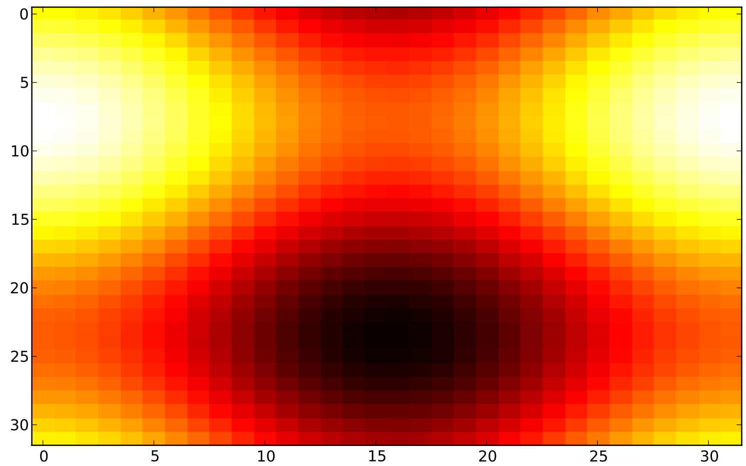


Figure 2.4: Example of color()

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*: Propagated to `matplotlib.pyplot.figure()`

Example:

```
fig = figure("velocity_vs_time")
plt.getp(fig, 'label')
'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 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*: Propagated to `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=[(1, 0), (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 (1,0) for a solid line.

  See Also:

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

- ***kwargs*: Propagated to `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...>]]
saveall()
Saved examples/plot.pdf
Saved examples/plot.png
plt.show()
```

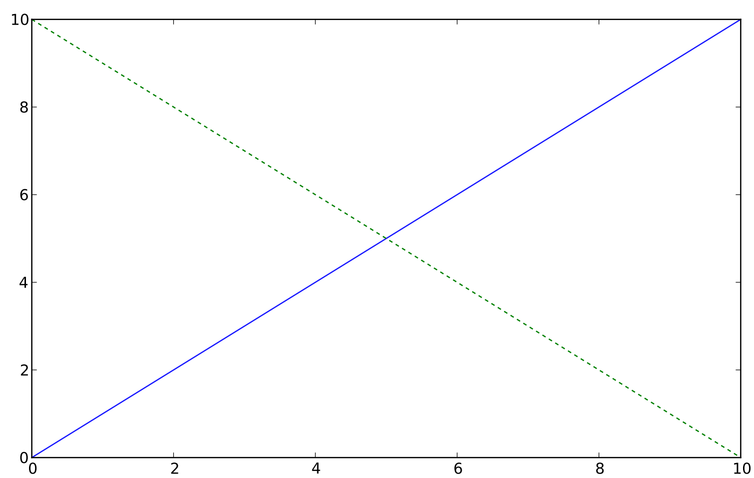


Figure 2.5: Example of plot()

`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*: Propagated to `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...>
saveall()
Saved examples/quiver.pdf
Saved examples/quiver.png
plt.show()
```

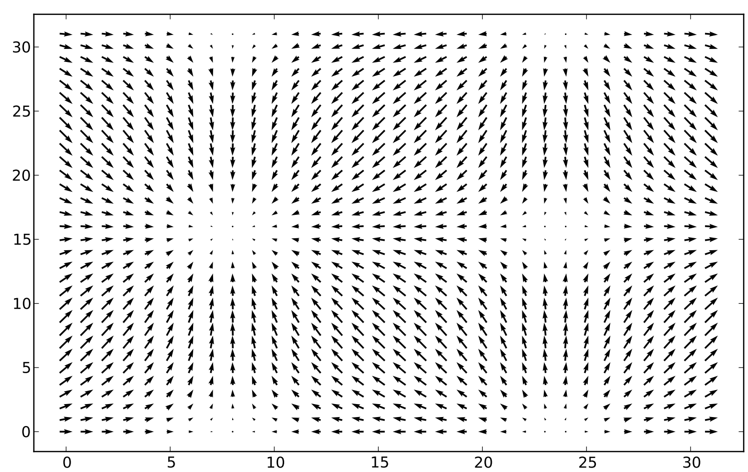


Figure 2.6: Example of quiver()

`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. 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.

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...>]
saveall()
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=0.098*, *margin_right=0.08999999999999997*, *margin_bottom=0.1*, *margin_top=0.08999999999999997*, *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...>, <matplotl
saveall()
Saved examples/setup_subplots.pdf
Saved examples/setup_subplots.png
plt.show()
```

Example of setup_subplots()

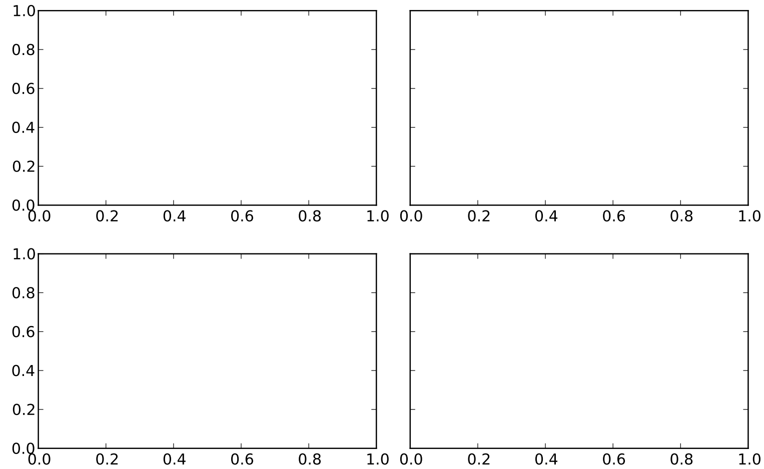
`modelicares.base.`**`shift_scale_x`**(*ax*, *eagerness=0.325*)

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])
saveall()
Saved examples/shift_scale_x.pdf
Saved examples/shift_scale_x.png
plt.show()
```

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 recen-

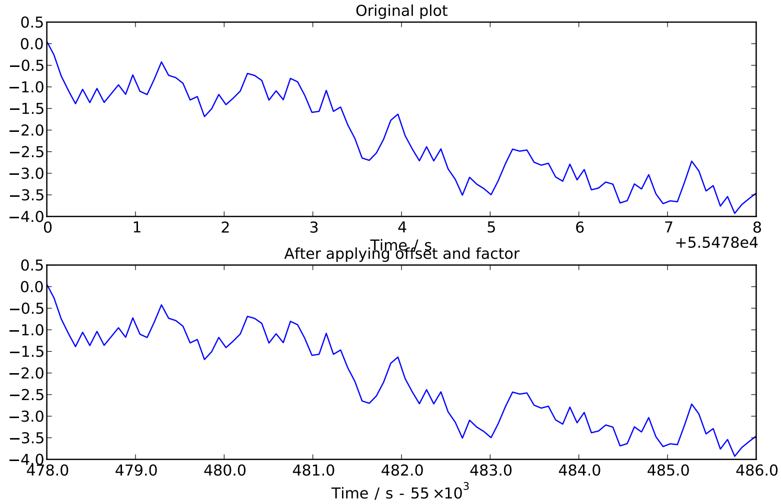


Figure 2.7: Example of shift_scale_x()

tered

–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])
```

```
saveall()
Saved examples/shift_scale_y.pdf
Saved examples/shift_scale_y.png
plt.show()
```

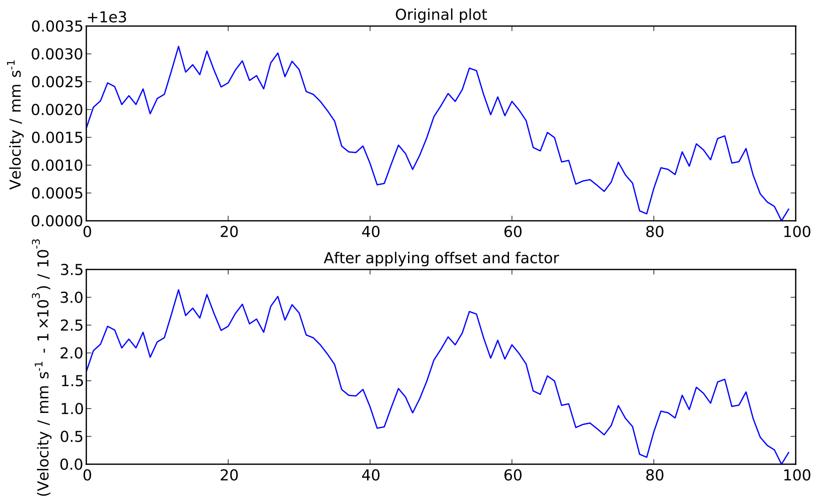


Figure 2.8: Example of shift_scale_y()

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

A matplotlib subclass to draw an arrowhead on a line

Initialize the line and arrow.

Arguments:

- *arrow* (=*None*): Type of arrow (e.g., '>')
- *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*: Propagated to `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)
saveall()
Saved examples/ArrowLine.pdf
Saved examples/ArrowLine.png
plt.show()
```

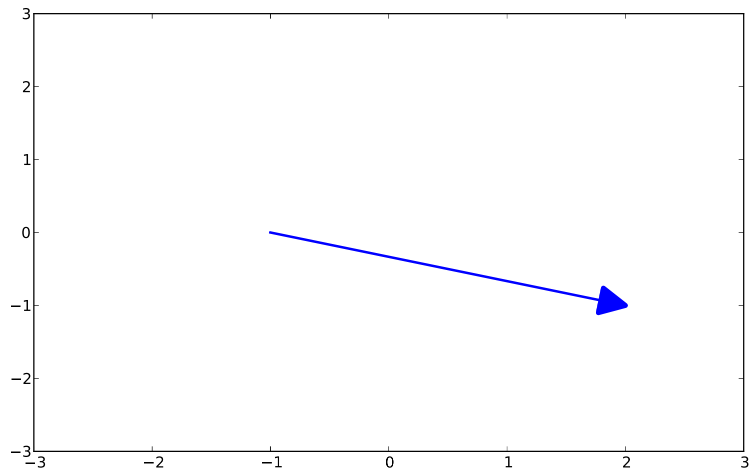


Figure 2.9: Example of ArrowLine

CHAPTER

THREE

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 "examples/ChuaCircuit.py"), 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).

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={}*, *permute=True*)

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.

- *permute*: *True*, if the lists of values (for *model* and the entries in *params* and *args*) should be permuted to generate a full-factorial design of experiments (DOE)

  If *permute* is *False*, then the lists of values will be iterated jointly (element-wise), terminating at end of the shortest list.

Example 1 (element-wise list of experiments):

```
from modelicares import *

doe = gen_experiments(['Modelica.Electrical.Analog.Examples.ChuaCircuit']*3,
                      {'L.L': [16, 18, 20], 'C2.C': [80, 100, 120]},
                      permute=False)
for experiment in doe:
    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 (permutation—full-factorial design of experiments):

```
from modelicares import *

doe = gen_experiments(['Modelica.Electrical.Analog.Examples.ChuaCircuit'],
                      {'L.L': [16, 18, 20], 'C2.C': [80, 100, 120]},
                      permute=True)
for experiment in doe:
    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 3 (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*)

Return a string representing a NumPy array in Modelica format.

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*)

Return a string representing an instance of Python's `bool` in Modelica.

This is simply 'true' or 'false' (in lowercase).

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.

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.

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/Dymola', fname='run-sims.mos', command='simulateModel', filemap={'dsin.txt': '%s_%i.in', 'dymosim%x': '%s_%i%x', 'dslog.txt': '%s_%i.log', 'dsres.mat': '%s_%i.mat'}*)

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* ("~/Documents/Dymola"): 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

  Each may be a "*.mo" file or a folder that contains a "package.mo" 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.

- ***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.

- *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 are relative to the working directory and the destinations are relative to the results directory (implied by *fname*). '%s' may be included in the destination to indicate the model name (*model*) without the full path. '%i' may be included to indicate the simulation number in the sequence of experiments. '%x' may be included in the source or destination to represent '.exe' if the operating system is Windows and '' otherwise.

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/run-sims1.mos")
(['ChuaCircuit'], '...examples')
```

In "examples/run-sims1.mos":

```
import Modelica.Utilities.Files.copy;
cd(".../Documents/Dymola")

// Experiment 1
ok = simulateModel("Modelica.Electrical.Analog.Examples.ChuaCircuit", stopTime=2500);
if ok then
   copy("dsin.txt", ".../examples/ChuaCircuit_1.in", true);
   copy("dslog.txt", ".../examples/ChuaCircuit_1.log", true);
   copy("dsres.mat", ".../examples/ChuaCircuit_1.mat", true);
   copy("dymosim", ".../examples/ChuaCircuit_1"", 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/run-sims2.mos")
(['ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit', 'ChuaCircuit'
```

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

CHAPTER

FOUR

MODELICARES.LINRES

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

This module 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

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 the `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(*ax=None, pairs=None, w_min=-1, w_max=3, label='bode', title=None, colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], styles=[(100, 0), (3, 3), (1, 1), (3, 2, 1, 2)], **kwargs*)

Create a Bode plot of the system's response.

Arguments:

- *ax*: Axis onto which the data should be plotted.
  If not provided, an axis 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.
- *w_min*: Common logarithm of the minimum angular frequency
- *w_max*: Common logarithm of the maximum angular frequency
- *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 "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*: Propagated to `control.matlab.bode()`

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

Example:

```
from modelicares import LinRes, saveall

lin = LinRes('examples/PID.mat')
lin.bode(label='examples/PID-bode',
         title="Bode Plot of Modelica.Blocks.Continuous.PID")
saveall()
Saved examples/PID-bode.pdf
Saved examples/PID-bode.png
Saved examples/PID-bode.pdf
Saved examples/PID-bode.png
```

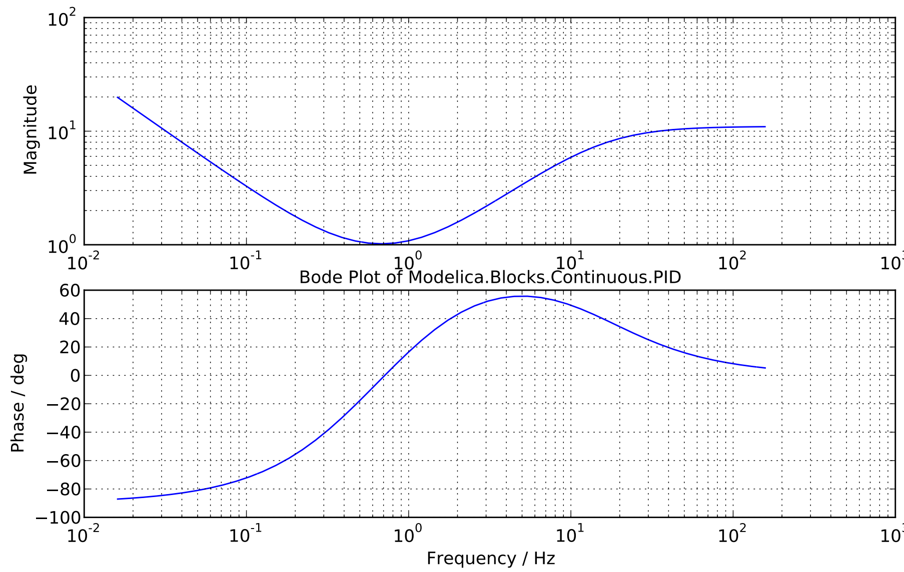


Figure 4.1: Results of example for `LinRes.bode()`.

nyquist(*ax=None, pairs=None, w_min=0.79817986835811505, w_max=2.7981798683581149, label='nyquist', title=None, xlabel='Real Axis', ylabel='Imaginary Axis', mark=True, colors=['b', 'g', 'r', 'c', 'm', 'y', 'k'], **kwargs*)

Create a Nyquist plot of the system's response.

Arguments:

- *ax*: Axis onto which the data should be plotted.
  If not provided, an axis 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.
- *w_min*: Common logarithm of the minimum angular frequency
- *w_max*: Common logarithm of the maximum angular frequency
- *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
- *mark*: *True* if frequencies should be labeled (with a dot and text denoting the frequency)
- *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*: Propagated to `control.matlab.nyquist()` **and** `control.matlab.nyquist_label()`

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

Example:

```
from modelicares import LinRes, saveall

lin = LinRes('examples/PID.mat')
lin.nyquist(label='examples/PID-nyquist',
            title="Nyquist Plot of Modelica.Blocks.Continuous.PID")
saveall()
Saved examples/PID-nyquist.pdf
Saved examples/PID-nyquist.png
```

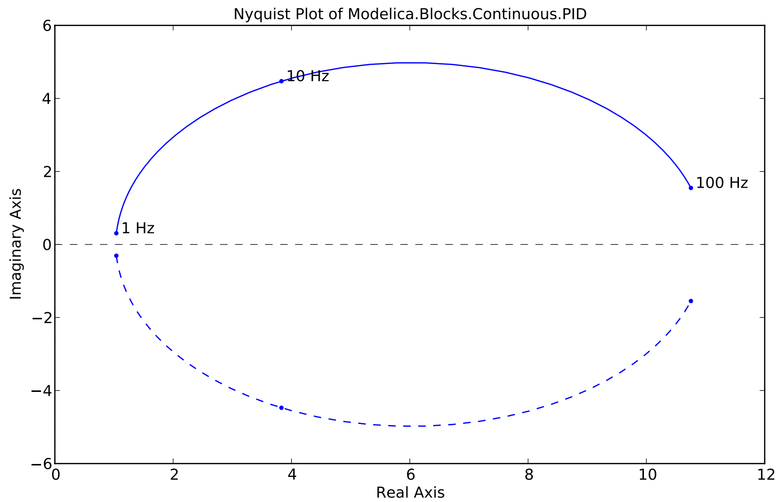


Figure 4.2: Results of example for `LinRes.nyquist()`.

CHAPTER

FIVE

MODELICARES.MULTI

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

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 filename(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', 'examples/PID'])
Valid: SimRes('...ChuaCircuit.mat')
Valid: LinRes('...PID.mat')
([SimRes('...ChuaCircuit.mat')], [LinRes('...PID.mat')])

# By directory:
multiload('examples')
Valid: SimRes('...ChuaCircuit.mat')
Valid: SimRes('...ChuaCircuit2.mat')...
Valid: LinRes('...PID.mat')...
([SimRes('...ChuaCircuit.mat'), SimRes('...ChuaCircuit2.mat'), SimRes('...ThreeTanks.mat')], [LinRes('...
```

modelicares.multi.**multiplot**(*sims, suffixes, dashes=[(1, 0), (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()`)

- ***kwargs*: Propagated to `simres.SimRes.plot()` (and thus to `base.plot()` and finally `matplotlib.pyplot.plot()`)

  The *dashes* sequence is iterated across all plots.

Example:

```
from modelicares import SimRes, multiplot, saveall

sims = map(SimRes, ['examples/ChuaCircuit', 'examples/ChuaCircuit2'])
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)
saveall()
Saved examples/ChuaCircuits.pdf
Saved examples/ChuaCircuits.png
```

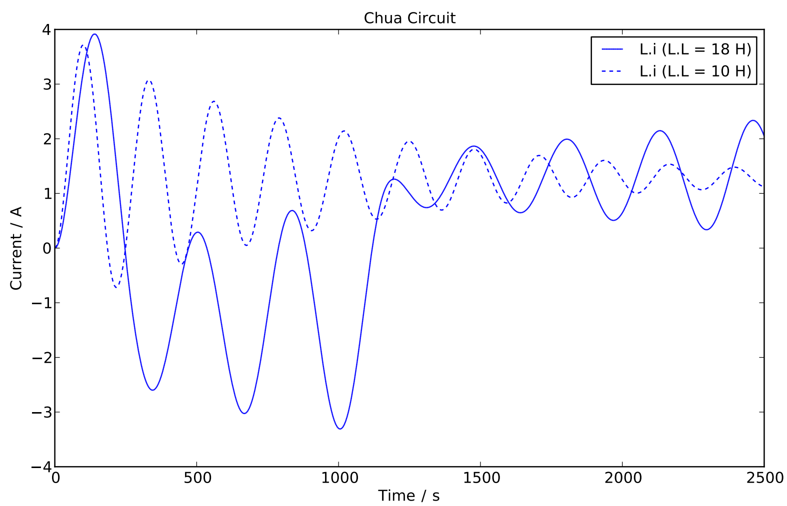


Figure 5.1: Plot of Chua circuit with varying inductance

CHAPTER

SIX

MODELICARES.SIMRES

Load, analyze, and plot results from Modelica simulations.

class `modelicares.simres.`**`Info`**

Shortcuts to the "get" methods in `SimRes`

`FV`(*names*, *f=<function <lambda> at 0x9bd125c>*)
Alias for `SimRes.get_FV()`

`IV`(*names*, *f=<function <lambda> at 0x9bd11ec>*)
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()`

`times`(*names*, *i=slice(0, -1, None)*)
Alias for `SimRes.get_times()`

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

`values`(*names*, *i=slice(0, -1, None)*, *f=<function <lambda> at 0x9bd133c>*)
Alias for `SimRes.get_values()`

`values_at_times`(*names*, *times*, *f=<function <lambda> at 0x9bd13ac>*)
Alias for `SimRes.get_description()`

class `modelicares.simres.`**`SimRes`**(*fname='dsres.mat'*)

Bases: `object`

Base class for Modelica-based simulation results and methods to analyze those results

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.

Example:

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

__call__(*names*, *action=<function get_values at 0x9bd1374>*, **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*: Propagated to *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()

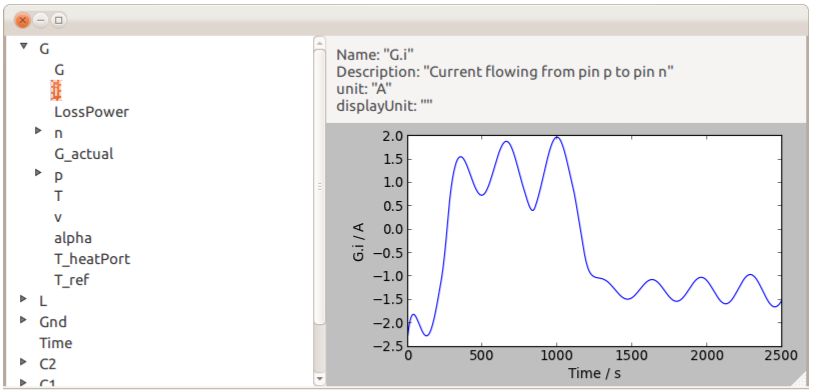
Launch a variable browser.

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 0x9bd125c>*)

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 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_FV('L.v')
-0.25352862
```

get_IV(*names*, *f=<function <lambda> at 0x9bd11ec>*)

Return the initial values 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 trajectory 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 trajectory 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_times`(*names*, *i=slice(0, -1, None)*)

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.

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_unit(*names*)

Return the *unit* attribute(s) of trajectory 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, -1, None)*, *f=<function <lambda> at 0x9bd133c>*)

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)
```

get_values_at_times(*names*, *times*, *f=<function <lambda> at 0x9bd13ac>*)

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

- *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()

Return a list of all variable names.

Example:

```
from modelicares import SimRes
sim = SimRes('examples/ChuaCircuit.mat')
sim.names()
[u'L.p.i', u'Ro.alpha', ... u'Ro.LossPower']
```

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*, ***kwargs*)

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

A new figure is created if necessary.

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 in parentheses at the end of the legend entries
- ***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, saveall

sim = SimRes('examples/ChuaCircuit')
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.Axes object at 0x...>)

saveall()
Saved examples/ChuaCircuit.pdf
Saved examples/ChuaCircuit.png
```

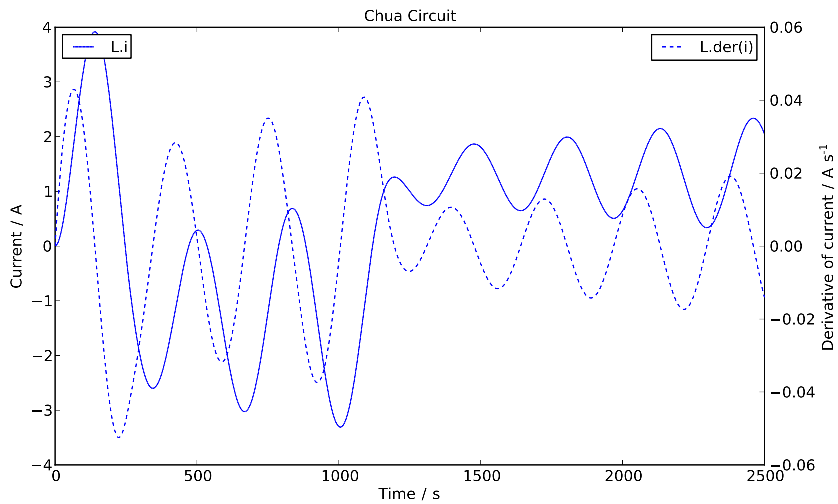


Figure 6.1: Plot of Chua circuit

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
- *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*: Propagated to `matplotlib.sankey.Sankey`

Returns:

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

Example:

```
from modelicares import SimRes, saveall

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=0.100, margin=6, offset=1.5,
    pathlengths=2, trunklength=10)
saveall()
Saved examples/ThreeTanks.pdf
Saved examples/ThreeTanks.png
```

`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)
```

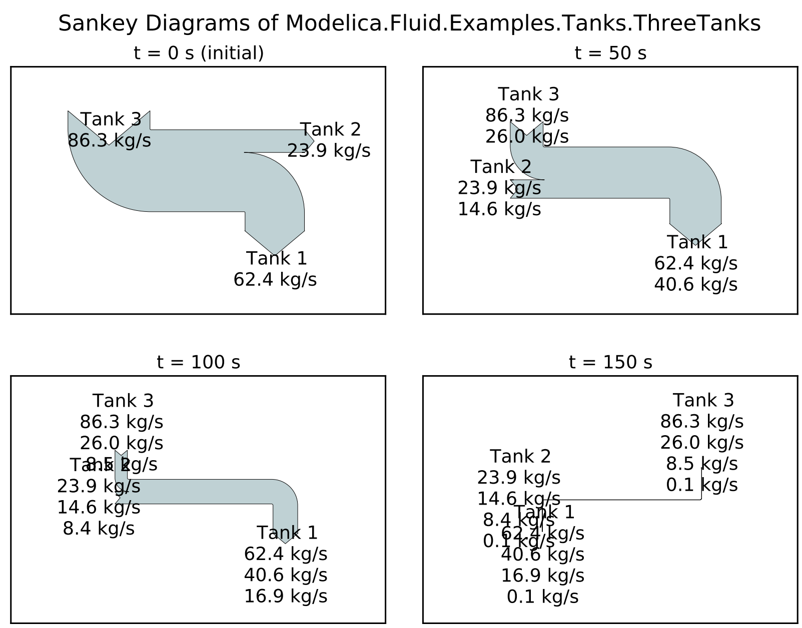


Figure 6.2: Sankey digarams of three-tank model

CHAPTER

SEVEN

MODELICARES.TEXUNIT

Methods to format numbers to support 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.
- *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 label 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: $1.234 \times 10^{-3}\,\mathrm{m}$

```
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: $1.0\,\mathrm{k\Omega}$

`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 ('^')).

Here, the unit may also contain LaTeX math mode commands. For example, '\Omega' becomes Ω. Use '\%' for percent, '\$' for dollar, and '\mathrm{^{\circ}C}' for degree Celsius (°C). Use '$' in pairs (without a leading '\') to escape and then return to the LaTeX math mode later in the string. Use '\!' to cancel a 3/18 quad space ('\,').

- *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

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()`).

Example:

```
$ loadres examples
Valid: SimRes('.../examples/ChuaCircuit.mat')
Valid: SimRes('.../examples/ChuaCircuit2.mat')
Valid: SimRes('.../examples/ThreeTanks.mat')
Valid: LinRes('.../examples/PID.mat')
Simulation results have been loaded into sims[0] through sims[2].
A linearization result has been loaded into lin.

In [1]:
```

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

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W