FONSim

Release 0.2a

The FONSim Project

Apr 28, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Welcome to the documentation of the FONSim library!

It is hoped that this documentation helps the reader in exploring FONSim. If something appears missing: no worries, please send an email to one of the developers and they'll kindly try to help you forward.

1.1 How to get started with FONSim

It is suggested to follow the tutorials in this documentation. These ought to familiarize the reader with the basic usage of FONSim. Many users will want to extend FONSim, as the included standard functionality is unlikely to fully support their research case. They will want to continue into the code documentation, which discusses the five main parts of FONSim.

Installation: FONSim is available on the PyPI and therefore can be easily installed with `pip install fonsim`. For more detailed intallation instructions, see *Installation*.

1.2 Features

- Powerful and fast simulation backend
  - Newton-Raphson method for handling nonlinear equations
  - Backward Euler time discretization for stability
- Implicit component equations
- Fluid class
  - custom fluids, e.g. non-Newtonian
  - fallback functionality
- Toolset focused on soft robotics (SoRo) research
  - read and process pv-curves
- Standard library of fluidic components
  - Tubes, nodes, pressure sources, volume sources, containers, one-way valves, ...
- Flow calculations
  - Compressible flow approximations
  - Laminar and turbulent flow based on Reynold number
  - Major and minor losses (Darcy, Haaland, K-factor etc.)

- Preconfigured custom plot methods
- Export data for further processing as JSON file
- Cross platform

1.3 Goal and philosophy

The goal of FONSim is to greatly ease simulation of fluidic systems in softrobotics research by providing a set of often-needed system components and analysis tools and by automating the construction and solving of the resulting component and network equations with an open-source, easily extensible library.

FONSim is designed in the first place for usage in research environments, where flexibility and easy-to-experiment-with software are key. Therefore, FONSim is fully written in relatively easily-read Python code such that it is doable to inspect and extend FONSim without investing too much time and effort. While this choice for Python (versus, for example, C or C++) means that FONSim will not excel in solving speed, this is compensated for by Python being much easier to use. Finally, the speed difference is not that large, as for most of the time-consuming numerical maths the Numpy library is used, which under the hood relies on highly-optimized C and FORTRAN code.

FONSim does not have a GUI (Graphical User Interface). While this results in a steeper learning curve in the beginning, defining systems using code (and not using a GUI) does have two mayor advantages. The first advantage is that code is far more powerful that a GUI. Defining and connecting hundreds of system components can be done in a few lines using appropriate constructs such as for-loops. The second advantages is that Python code is eay to collaborate on and to version manage, for example using industry-standard tools like Git.

FONSim consists of five parts. While these parts depend on each other, they are not finely ingrained to increase flexibility and ease maintainability. The first part is the core and is application-agnostic (!). It is used just as easily to simulate kinematics as fluidic networks. Two other parts are the standard components and fluids, which define most common components used in softrobotics research. Finally, the three last parts provide some smaller functionality for data visualization, some data IO and a function generator.

1.4 Contributing

1.4.1 Ways to contribute to FONSim

There are several ways to contribute to the FONSim project:

- Provide feedback about your usage experience.
- Ask a question.
- Suggest an improvement, be it to the documentation, the code or anything else.
- Implement an improvement yourself - please see *Contribution process*.

For all the above, users can email one of the developers or open a new issue on GitLab issues. Once there are more developers, a more appropriate discussion place will be opened.

1.4.2 Contribution process

1. Check for open issues or open a new issue.
2. Fork the FONSim repository on GitLab.
3. Make your changes. Don't forget to add yourself to `CONTRIBUTORS.txt`!
4. Send a pull request. If no response, email one of the maintainers.

Before submitting the pull request, as a general rule: - Any code change should come with appropriate tests and documentation. - All tests should pass.

1.5 FAQ

…

CHAPTER

TWO

DEVELOPMENT

2.1 Introduction

This file contains some generic information about the development of FONSim.

2.1.1 Branching

The FONS project branching is based on the [Driessen or git-flow model](https://nvie.com/posts/a-successful-git-branching-model/). Put simply, the `master` branch is reserved for production-ready code. All code in `master` should be stable and usable. The `dev` branch contains the latest developed features, yet as a result the software is not as stable. The actual features (and improvements in general) are developed in the feature branches, for example `feature-plotting`.

2.1.2 To get this repo locally

1. Clone the repo (notice the *$* - this means to do it in a terminal/console). The directory with the project will be located in the current working directory of the terminal.

```
$ git clone https://gitlab.com/abaeyens/fonsim.git
```

1. Go in the created directory (note: one can use *TAB* for autocompletion)

```
$ cd fonsim
```

2.1.3 Create a local install

A local install allows to try out the library locally. This can be useful during development. First, rename the project root directory to *fonsim* (default name after Git clone: *fons*). Second, run in the project root directory:

```
$ python -m pip install -e .
```

This installs the FONS package such that it is accessible like all other Python packages, e.g. using *import fonsim*. The *-e* option denotes that it uses a symbolic link: code changes in the project directory (including branch switching) take effect at the first following *import*. No re-installation is required.

Note: *python* should refer to Python 3. You may have to write *python3* to avoid using Python 2.

Note: there appear to be problems with this method on some Windows machines.

Note: if you want to install several versions of the same package on your system, for example a stable version from PyPi and a development version from a local install, you may want to use a [Python virtual environment](https://docs.python.org/3/tutorial/venv.html).

2.1.4 Development tools

A git repository history visualizer tool like [gitg](https://wiki.gnome.org/Apps/Gitg/) can be helpful in developing this software. It shows the relations between version branches visually, lists all commits and allows to see the exact changes were made in a particular commit. In addition, it can show uncommitted changes.

Gitlab provides similar tools as a web version like the [GitLab graph](https://gitlab.com/abaeyens/fonsim/-/network/master).

2.1.5 Style guide

https://google.github.io/styleguide/pyguide.html.

2.2 Documentation

Note: This document is a work in progress.

2.2.1 Introduction

The documentation of the FONSim project resides together with the code in the GitLab repository in the directory *docs/source*. This directory contains the documentation source files, written in ReStructuredText (RST). All RST files in the directory *autodoc* are generated using tools while the other RST files, such as the one used to render this document, are compiled manually.

These ReStructuredText files are rendered to HTML viewable in a browser using Sphinx. This HTML documentation is then hosted on the internet thanks to the ReadTheDocs organization. The ReadTheDocs servers look at the FONSim GitLab repository and more or less rebuilds the documentation every time the codebase on the GitLab repository changes.

The Sphinx + ReadTheDocs documentation has a nice "How To" guide: https://sphinx-rtd-tutorial.readthedocs.io/en/latest/index.html The FONSim project mostly adheres to the default choices.

The rest of this documents discusses how to build the documentation.

2.2.2 Local compilation

The output HTML files reside in `docs/build/` and can be viewed in a web browser. They closely resemble those that will be built by the ReadTheDocs server (their process is slightly different though).

First time:

```
cd docs
sphinx-build -b html source/ build/
```

Thereafter:

```
cd docs
make clean
make html
```

The `make clean` is preferable not omitted as doing so tends to lead to not updating files that should have been updated.

2.2.3 Compiling autodocs

In short, the Sphinx autodocs functionality takes the codebase, strips all the code away and dresses the docstrings in nice RST files. These RST files only contain references to the docstrings and therefore not the docstrings themselves. Therefore, it is not necessary to rerun the Sphinx autodocs unless major changes to the code structure are made. The generated autodocs reside in the directory `docs/source/autodocs`.

```
cd docs
sphinx-apidoc -o ./source/autodoc ../src/fonsim -f
rm source/autodoc/modules.rst
rm source/autodoc/fonsim.rst
```

The files `modules.rst` and `fonsim.rst` are deleted to avoid generating 'WARNING: document isn't included in any toctree'.

2.2.4 Updating ReadTheDocs

The ReadTheDocs server is configured to look at the GitLab repository and recompiles the documentation automatically. However, it will fail at the smallest error, so it is preferable to check the documentation source files first by doing a local compilation and observing the output.

CHAPTER

THREE

RELEASE LOG

3.1 Upcoming releases

3.1.1 0.3b: Spring 2023

The first beta release!

Changes

- Introduce the vastly improved solver. Currently (January 2023) resolving some minor issues and adding tests.

3.2 Previous releases

3.2.1 0.2a: 2023, February 04

The second release of FONSim focuses on cleaning up the codebase, improving documentation to make the codebase attractive to new users, and the introduction of the *autocall* and *autodiff* functionality.

Changes

- Introduce autocall: in component definition, reference variables and states by their label, removing the requirement to work with arrays and indices.
- Introduce autodiff: in component definition, calculate derivatives automatically using numerical differentiation, removing the requirement to specify them manually.
- Introduce more than 100 tests (PyTest framework) covering most of FONSim's codebase.
- Documentation (tutorials + formal codedoc) on readthedocs (went online spring 2022).
- Many bugfixes and code, UI and documentation improvements. Some breaking changes were made.

3.2.2 0.1a5: 2021, March 24

Initial release of FONSim on PyPI.

CHAPTER

FOUR

INTRODUCTION

TODO write some introduction to the tutorials here, perhaps with a short discussion of the learned FONSim features in each tutorial or so./ TODO remove index below once doc reaches maturity.

Index

If you have any thoughts, ideas, … for improving this tutorial, sharing them would be greatly appreciated.

CHAPTER

FIVE

INSTALLATION

To install FONSim, follow these four steps:

Step 0: Installing Python 3
FONSim is a Python library and therefore requires a Python interpreter. See www.python.org for the appropriate download and installation instructions for your operating system.

Notes

- On many Linux desktops Python 3 comes preinstalled and this step can be safely skipped.
- To test whether a Python interpreter has been installed, open a terminal window and run `python` or `python3`. If both return an error messages, there is no Python interpreter installed.

Step 1: Install PIP
TheF ONSim library will be installed using PIP. To install PIP, see https://pip.pypa.io/en/stable/installation/. To test whether PIP is installed, open a Python shell (for example, by starting a command line and running `python3` or `python`) and run `pip`.

Step 2: Installing FONSim using PIP
Installing FONSim is most easily done using PIP. This method installs from PyPI, the Python Package Index. In terminal, run `pip install fonsim`.

For installing FONSim from source, please see the development documentation.

Step 3: Testing the installation
Open a Python shell and run, in the Python shell, `import fonsim`. If an error is returned, there is an issue with the installation.

CHAPTER

SIX

A FIRST NETWORK

In this first tutorial a small fluidic network is built and simulated. While still limited in complexity, this simple example shows a typical usage of the FONSim library, including component creation, system definition, simulation and data visualization. In later tutorials, the many aspects of FONSim are explored more in depth.

Note: It might be relevant to note that Python variables are by reference if the variable is mutable. The code example below illustrates this. It prints '[4, 2, 3]' and not '[1, 2, 3]'. If it is desired to copy object data and not the reference to it, use the Python copy library.

```
# Create a List instance.
a = [1, 2, 3]
# Copy the _reference_ to the object.
b = a
# Change variable 'b', which will also change 'a',
# because they point to the same data.
b[0] = 4
# Print 'a' to show that it has been changed.
# This will print '[4, 2, 3]' and not '[1, 2, 3]'.
print(a)
```

Full script: `01.py`.

The first step is to import two required modules, Respectively FONSim and Matplotlib. The latter will be used for plotting the simulation results.

```
10 # Import fonsim package, for building and simulating a fluidic system
11 import fonsim as fons
12 # Import matplotlib package, for plotting the simulation results
13 import matplotlib.pyplot as plt
```

Next we create a *System* object called 'system' to which we will add components. The 'system' object will contain all components in the system and how these are connected to each other.

```
16 # Create a new pneumatic system
17 system = fons.System()
```

Here we create and add components to the system. One of the ways to create a *Component* instance is to use FONSim's standard library, as is done here with `mycomponent = fons.PressureSource...`. The object gets label

'source_00'. The label can be any string as long as it only occurs once per system. The line thereafter, the created object, named 'mycomponent', is added to the system using the *System.add()* method.

In the following two lines the same is done, yet no variable that references to the component, such as 'mycomponent', is kept. Keeping a variable that references to the component is not necessary because components can be easily retrieved by their label by calling the method :py:meth`.System.get` with as argument the component label (example: `system.get('source_00')`).

```
# Create components and add them to the system
# Note: for now, only take major (pipe friction) losses into account.
mycomponent = fons.PressureSource("source_00", pressure=2e5)
system += mycomponent
system += fons.Container("container_00", fluid=fluid, volume=50e-6)
system += fons.CircularTube("tube_00", fluid=fluid, diameter=2e-3, length=0.60)
```

The pressure source is instantiated with a constant pressure of 2e5 Pa. Time dependent pressures are also possible, which will be discussed in the next tutorial.

Now that the components are defined, they are connected to each other. Here, no specifics about how exactly the components should be connected to each other are given, and FONSim will therefore choose defaults. After this step, the networked system is fully defined.

```
# Connect the components to each other.
system.connect("tube_00", "source_00")
```

The next step is to simulate the system. A *Simulation* instance named 'sim' is created which will hold the system object and all parameters relevant for the simulation. Only one simulation parameter is given, namely the duration in seconds for which the system has to be simulated. Other parameters, such as the simulation timestep, are not specified and will therefore be chosen automatically by FONSim.

```
# Let's simulate the system!
sim = fons.Simulation(system, duration=0.3)
```

The simulation can now be run:

```
# Run the simulation
sim.run()
```

Finally, the simulation results are plotted using the Matplotlib library. For matplotlib tutorials, please refer to the Matplotlib Pyplot introduction.

There are currently two standard ways to plot simulation data. The first is to manually fetch the simulation data, plot it and add the legend and axis labels. The second is to FONSim's built-in *plotting* functionality, which results in quasi the same result as the first option but with far less work from the user. This tutorial discusses achieving the same result with both methods.

The first two code lines prepare the plot, after which the simulation data is plotted. The simulation data is discretized in the time domain. The time array is in the `Simulation.times` attribute and is accessed here using `sim.times`. The simulation data array is accessed through the components. First, the reference to the component is retrieved from the system, here using `system.get("source_00")`. Then, the array is retrieved from the component, again using 'get'. The arguments of the *Component.get()* method are the label of the variable and, for some simple components optional, the *port*. For example, the tube (*CircularTube*) has two ports, which are labeled 'a' and 'b'. All fluidic

components have at least one 'pressure' variable and one 'massflow' variable. A list of the variable and port names of each component type can be found in the docstring of that component type, which can be retrieved by calling the `help()` function on the object in a Python shell, for example `help(system.get("source_00"))`. Some IDEs show this docstring after hovering with the mouse over the code in question.

```
# Manual method
fig, axs = plt.subplots(3, sharex=True)
fig.suptitle("Simulation results")
axs[0].plot(sim.times, system.get("source_00").get('pressure')*1e-5, label='source_00')
axs[0].plot(sim.times, system.get("container_00").get('pressure')*1e-5, label='container_
↪00')
axs[0].set_ylabel('pressure [bar]')
axs[1].plot(sim.times, system.get("container_00").get_state('mass')*1e3, label=
↪'container_00')
axs[1].set_ylabel('mass [g]')
axs[2].plot(sim.times, system.get("source_00").get('massflow')*1e3, label='source_00')
axs[2].plot(sim.times, system.get("container_00").get('massflow')*1e3, label='container_
↪00')
axs[2].set_ylabel('mass flow [g/s]')
axs[2].set_xlabel('time [s]')
for a in axs: a.legend()
```

The simulation arrays denoting a pressure are multiplied with '1e-5' because the simulation results for pressure are stored in unit Pa while the plot y-axis unit is bar (1 bar = 1e5 Pa).

Next, almost the same result is achieved using the FONSim *plotting* functionality. The third line plots the *pressure* of the components labeled *source_00* and *container_00* with unit *bar*.

```
# Automatic method
fig, axs = plt.subplots(3, sharex=True)
fig.suptitle("Simulation results")
fons.plot(axs[0], sim, 'pressure', unit='bar', components=('source_00', 'container_00'))
fons.plot_state(axs[1], sim, 'mass', unit='g', components=('container_00',))
fons.plot(axs[2], sim, 'massflow', unit='g/s', components=('source_00', 'container_00'))
axs[2].set_xlabel('time [s]')
plt.show()
```

The figure below shows the image achieved with the second method.

To finish, a brief discussion of the simulation results shown in Figure 1. The container pressure in the top graph starts around 1 bar because the simulator assumes that at the start of the simulation there was already some air in the container such that its pressure equalled atmospheric pressure. Therefore the container also starts out with a mass of air of around 0.06 g (middle graph).

As the air flows through the tube from the source to the container, the container fills with air and its pressure increases until its pressure equals that of the source. It is apparent that the system by rough approximation behaves like a first-order linear low-pass filter exposed to a step input.

The bottom plot shows the massflow through the tube (which, the sign not considered, equals that of the source and the container). Of particular interest here are the two flow regimes, turbulent and laminar. Initially, the pressure difference is large, the flow speeds are high, resulting in a high Reynolds number and therefore turbulent flow. Around 0.14 s the massflow has decreased sufficiently for the flow regime to change to laminar, which causes the small 'bubble' around 0.16 s.

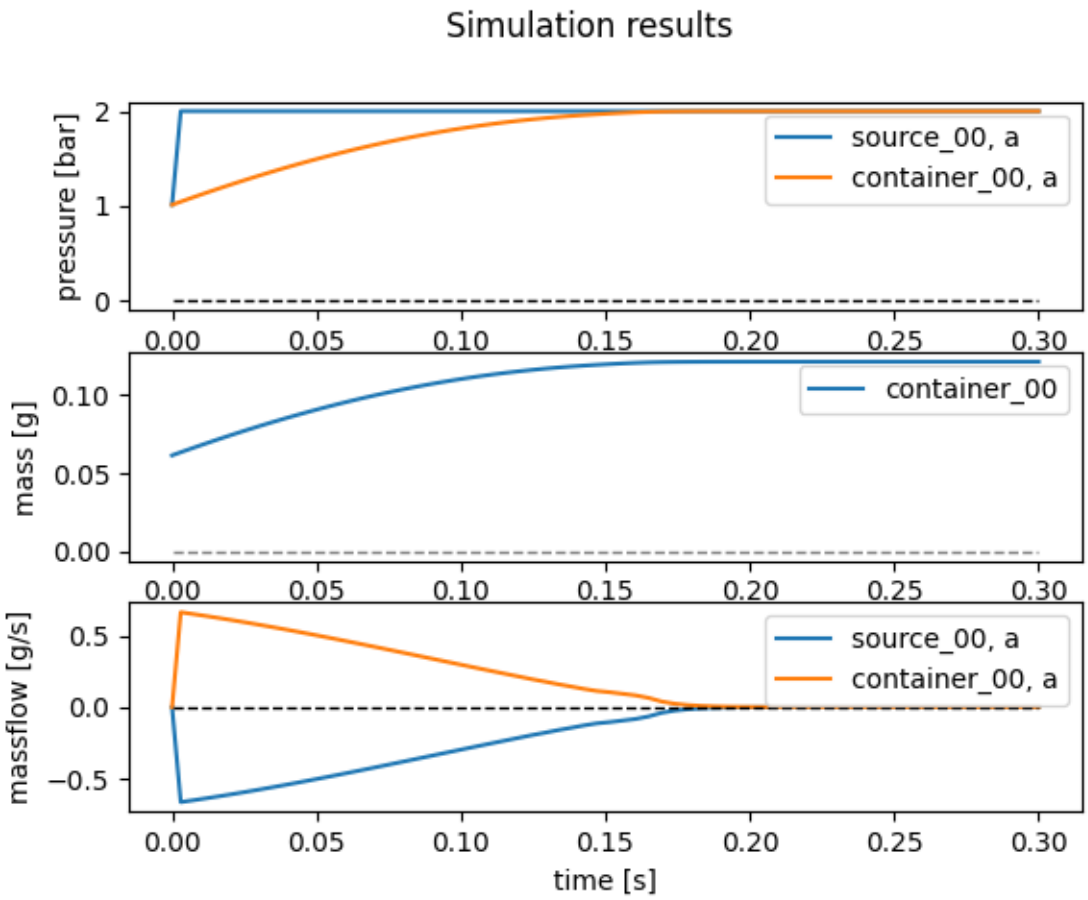

Fig. 1: Figure 1

It is hoped that this first tutorial provided a not-to-steep introduction to FONSim while still providing a good overview of how FONSim is designed to be used.

CHAPTER

SEVEN

A MORE COMPLEX NETWORK

This second tutorial extends the network seen in the first with two more tubes and a container. The component connection syntax is elaborated together with the concepts *terminal* and *node*. Furthermore, the usage of the built-in wave generator is demonstrated and an introduction to fluids is given.

The following schematic shows the network that will be built and simulated:

```
         -=- container
src -=- |
         -=- container
```

Full script: `03.py`.

In the first tutorial, the pressure gave a constant pressure. More useful is if the pressure varies over time. This is easily achieved using the *wave.custom.Custom* class included in FONSim. Here a simple rectangular wave is chosen. The desired wave is specified as a list of tuples, with each tuple of the form (x, y), with x the time (unit s) where the pressure rises to y (unitless). In this example, the pressure rises to 0.900 bar relative at t = 0 s, drops to 0.100 bar relative at t = 0.50 s and rises to 0.500 bar relative at t = 1.00 s. The multiplier 1e5 converts the pressures specified in bar to Pa and the atmospheric pressure of 1013 hPa (*pressure_atmospheric*) is added because FONSim works with absolute pressures while those specified in the wave are relative.

```
17 # Create wave function for pressure source
18 waves = [(0, 0.900), (0.5, 0.100), (1.0, 0.500)]
19 wave_function = fons.wave.Custom(waves)*1e5\
20                 + fons.pressure_atmospheric
```

The system is constructed, and the components are defined. The applicable component equations depend on the used fluid and therefore it is necessary to specify the fluid when creating the component. Given that all components are constant-volume, a simulation with an incompressible fluid would be rather boring, therefore the compressible fluid air is chosen. FONSim supports Newtonian ideal gasses with the *IdealCompressible* class. Several ideal gasses are included, see *fluids.IdealCompressible*.

```
22 # Create system
23 system = fons.System()
24 # Select fluid
25 fluid = fons.air
26 # Create components and add them to the system
27 system.add(fons.PressureSource('source', pressure=wave_function))
28 system.add(fons.Container('container_0', fluid=fluid, volume=100e-6))
29 system.add(fons.Container('container_1', fluid=fluid, volume=100e-6))
```

(continues on next page)

(continued from previous page)

```
system.add(fons.CircularTube('tube_0', fluid=fluid, diameter=2e-3, length=0.10))
system.add(fons.CircularTube('tube_1', fluid=fluid, diameter=2e-3, length=0.02))
system.add(fons.CircularTube('tube_2', fluid=fluid, diameter=2e-3, length=0.70))
```

Next, connect the components to each other. *Component* objects are connected to each other by assigning their *Terminal* objects to a common *Node*. Put differently, terminals that belong to a common node are considered to be connected to each other, and the components belonging to these terminals are also considered to be connected to each other.

In the previous tutorial, no terminals were specified and the choice was left to FONSim, which tends to select a yet unconnected terminal. Here this behaviour is undesired as we would like to connect the ends of three tubes to each other. The first code line asks to connect terminal 'a' of component 'tube_0' to a terminal of component 'source'. Terminal 'b' of 'tube_0' gets connected to terminal 'a' of 'tube_1' and terminal 'a' of 'tube_2'. The docstring of the component ought to contain the labels of the terminals, see for example *CircularTube*.

```
# Connect components to each other
system.connect(('tube_0', 'a'), 'source')
system.connect(('tube_0', 'b'), ('tube_1', 'a'))
system.connect(('tube_0', 'b'), ('tube_2', 'a'))
system.connect('tube_1', 'container_0')
system.connect('tube_2', 'container_1')
```

Note: The *Node* objects ought not to be confused with nodes as they occur in fluidic networks. FONSim sees a system as a unidirected graph with edges/links/lines (FONSim: *Component*) and nodes/vertices/points (FONSim: *Node*). Sometimes, the FONSim and fluidic meaning of 'node' coincide, such as when connecting the ends of three tubes together. However, most often they do not.

With the system fully defined, the simulation is run. The previous example showed the two manual and automatic plotting methods. The freedom offered by the former is not required here and therefore the latter is used. For plotting the massflow through 'tube_2', the terminal to take the massflow from is specified ('a') in a similar way as for connecting the components. This gives the plot shown in Figure 1.

```
# Plot results
fig, axs = plt.subplots(3, sharex=True)
fig.suptitle("Simulation results")
fons.plot(axs[0], sim, 'pressure', 'bar', ('source', 'container_0', 'container_1'))
fons.plot_state(axs[1], sim, 'mass', 'g', ('container_0', 'container_1'))
fons.plot(axs[2], sim, 'massflow', 'g/s', ('tube_0', 'tube_1', ('tube_2', 'a')))
plt.show()
```

To finish, a brief discussion of the simulation results shown in Figure 1. Perhaps first look back on the given tube lengths. The tube from the source to the Y-split has length 10 cm, the tube from the split to the first container ('container_0') has length 2 cm and the the tube from the split to the second actuator ('container_1') is with 70 cm far longer than the other two tubes.

The pressure (top plot) in the first container behaves similarly to the container in the previously tutorial. It was to be expected that the behaviour of this component is mostly governed by the source as it is much closer to the source than to the other container. Meanwhile, the pressure in the second container shows characteristics of a double first-order

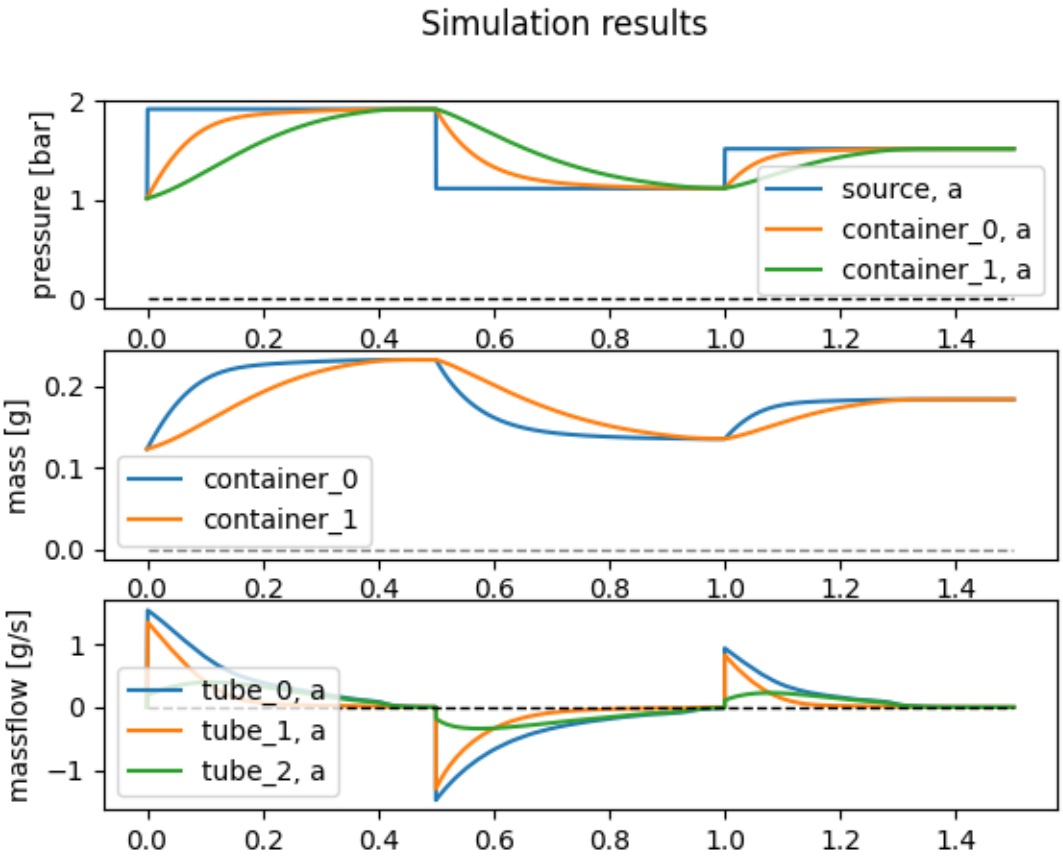

Fig. 1: Figure 1

low-pass filter, in particular that the pressure curve starts almost horizontally (in an ideal such a filter, it would start perfectly horizontally).

This also shows in massflow through the tubes (bottom plot): the flow through 'tube_2' starts from almost zero (even while the source has increased its pressure, little air has yet flown, so both containers maintain their initial pressure).

Feel free to play with the component parameters, such as the diameter and length of the tubes, and see how they influence the simulation results.

The next tutorial will focus on balloon actuators with their pv-curves and nonlinear behaviour.

CHAPTER

EIGHT

BALLOON ACTUATORS

The previous two examples already showed how to use several main features of FONSim, however, the observed behaviour of the container components was rather simple. Balloon actuators provide a more interesting component and are studied extensively in the field of soft robotics (SoRo). One of their special properties is their nonlinear behaviour. On one hand this makes them somewhat difficult to use for existing applications, but it also opens a wide array of possibilities for new applications such as hardware-encoded sequencing.

8.1 PV curves

The behaviour of these balloon actuators is specified as a pressure-volume curve or pv-curve. This curve is usually derived from measurements. FONSim contains an example measurements file of a balloon actuator: `Measurement_60mm_balloon_actuator_01.csv`. TODO add reference to paper.

Let's have a look at the first five lines of this CSV file:

```
Latex balloon actuator 60 mm,2017-11-16,MECH1A1
Time,Volume,Pressure
s,ml,mbar
14.31,1,22
17.01,2.01,280
```

The file is split in four parts (explained more in depth in *PVCurve*):

- First line: information about the file. For example, the measurement occurred in 2017, November 16.
- Second line: keys of data columns. Here we're only interested in volume and pressure, though time is listed as well.
- Third line: units of data columns.
- Fourth line and further: the actual numerical data.

Lets read this file using FONSim and plot the curve. First, the CSV file is loaded into a *PVCurve* object. The resulting object has two attributes, `v` and `p`, which respectively are 1D Numpy arrays of volume and pressures. Their units have been automatically adjusted such they conform to SI base units. The pressure has also been adjusted to absolute because the measured pressure in the CSV file was relative to atmospheric pressure.

```
# Load pvcurve in PVCurve object
pvcurve = fons.data.PVCurve(data=ppath, pressure_reference='relative')

# Plot the curve
fig, ax = plt.subplots(1)
ax.plot(pvcurve.v * 1e6, pvcurve.p * 1e-5)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('absolute pressure [bar]')
ax.set_xlabel('volume [ml]')
plt.show()
```

Full file: `06.py`. The plotting result:

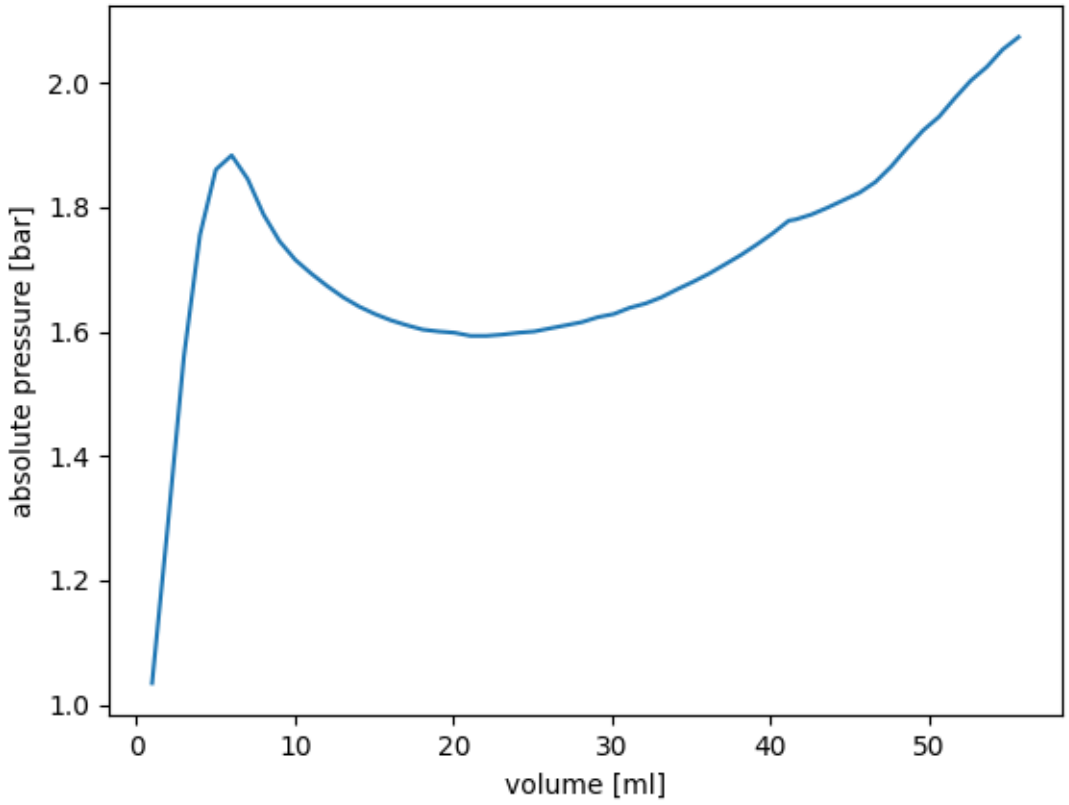

Fig. 1: Figure 1: PV curve of a balloon actuator

Figure 1 shows what makes these balloon actuators so interesting: the non-monotonically increasing pressure. As a result, for some pressures there are multiple solutions for the volume of the actuator, three in this case. Note the local pressure maximum of 1.87 bar around a volume of 6 ml (the 'hill') and a local pressure minimum of 1.58 bar around a volume of 21 ml (the 'dip').

8.2 Single balloon actuator

Let's do a simulation with this balloon actuator! Full file: `04.py`.

After the usual preamble, the pressure reference and the fluid are selected:

```
# Pressure reference
waves = [(0, 0.890), (1, 0.650), (2, 0.550), (3, 0.650)]
wave_function = fons.wave.Custom(waves)*1e5 + fons.pressure_atmospheric

# Select a fluid
# Previous examples required a compressible fluid.
# This one doesn't, feel free to try with an incompressible fluid (e.g. water)!
fluid = fons.air
```

The previous required a compressible fluid as all components were constant-volume. In this example, the balloon actuators can vary their volume, so feel free to try with an incompressible fluid (for example, water)!

Next, the system is built and the simulation is run. The system consists of one pressure source, one balloon actuator (FreeActuator) and a tube connecting the former two.

The FreeActuator component allows to simulate components characterized by a pv-curve and therefore provides an excellent fit for simulating balloon actuators. If the pvcurve is available as a CSV file, the first two emphasized lines can be omitted and the curve filepath can be passed directly to the parameter `curve` of *FreeActuator*.

```
# Create system
# The FreeActuator allows to simulate components characterized by a pvcurve.
# The curve argument can take a pvcurve specified as a CSV file.
system = fons.System()
system.add(fons.PressureSource('source', pressure=wave_function))
filepath = 'resources/Measurement_60mm_balloon_actuator_01.csv'
ppath = str(ir.files('fonsim').joinpath(filepath))
system.add(fons.FreeActuator('actu', fluid=fluid, curve=ppath))
system.add(fons.CircularTube('tube', fluid=fluid, diameter=2e-3, length=0.60))
system.connect('tube', 'source')
system.connect('tube', 'actu')

# Create and run simulation
sim = fons.Simulation(system, duration=4)
sim.run()
```

Let's plot the simulation results! Like in the previous example, we'll use the FONSim built-in plotting functionality:

```
# Plot simulation results
fig, axs = plt.subplots(3, sharex=True)
fons.plot(axs[0], sim, 'pressure', 'bar', ('source', 'actu'))
axs[0].set_ylim(1.4, 2.0)
fons.plot_state(axs[1], sim, 'mass', 'g', ['actu'])
fons.plot(axs[2], sim, 'massflow', 'g/s', ['tube'])
for a in axs: a.legend()
plt.show()
```

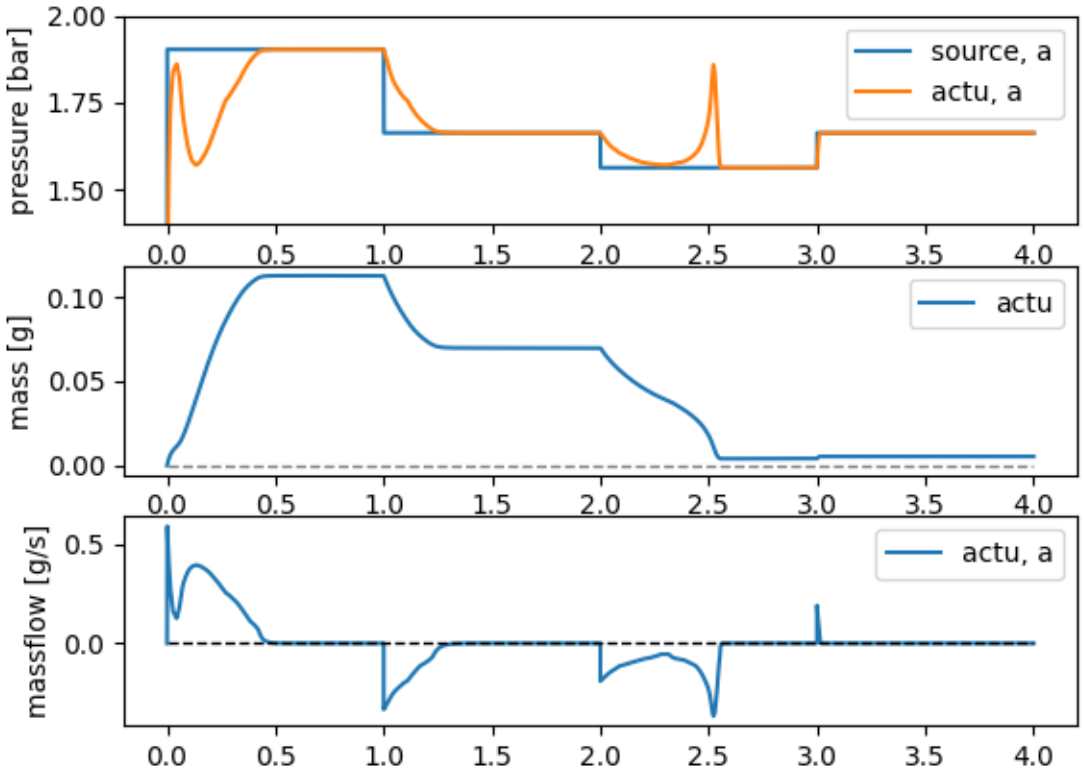

Fig. 2: Figure 2: Simulation output

This outputs the following plot figure:

To finish, a brief discussion of the simulation results shown in Figure 2. The simulation starts with an increase of absolute pressure to 1.89 bar, the actuator fully inflates as this pressure is larger than the pvcurve hill. Shortly after the pvcurve has been fully inflated, the pressure decreases back to 1.65 bar, just above the pvcurve dip. The actuator therefore stays mostly inflated, as is evidenced by more than half of the mass remaining, shown in the middle graph of Figure 2.

At time = 2.0 s the pressure drops just 0.1 bar, to 1.55 bar. Nevertheless, the actuator now deflates almost fully. Increasing the pressure again at time = 3.0 s has little effect on the actuator volume (middle graph of Figure 2).

8.3 Two balloon actuators

In this second simulation, we'll look at two balloon actuators in series and show a basic example of hardware sequencing achieved by exploiting a combination of actuator nonlinearity and flow restriction. It is based on TODO paper reference. The code is very similar to the previous simulation, so only the relevant differences are discussed here. Full file: `07.py`.

Pressure reference:

```
# Pressure reference
p1 = 0.890
p2 = 0.650
p3 = 0.200
waves = [(0.0, p2),
         (1.0, p1), (1.3, p2),
         (3.0, p1), (3.5, p2),
         (5.0, p3), (5.3, p2),
         (7.0, p3), (7.4, p2)]
wave_function = fons.wave.Custom(waves, time_notation='absolute')*1e5\
                + fons.pressure_atmospheric
```

tubes:

```
system.add(fons.CircularTube('tube_0', fluid=fluid, diameter=2e-3, length=1.20))
system.add(fons.CircularTube('tube_1', fluid=fluid, diameter=2e-3, length=0.60))
```

and the plotting:

```
# Plot simulation results
fig, axs = plt.subplots(3, sharex=True)
fons.plot(axs[0], sim, 'pressure', 'bar', ('source', 'actu_0', 'actu_1'))
axs[0].set_ylim(1.1, 2.0)
fons.plot_state(axs[1], sim, 'mass', 'g', ['actu_0', 'actu_1'])
fons.plot(axs[2], sim, 'massflow', 'g/s', ['tube_0', 'tube_1'])
for a in axs: a.legend()
plt.show(block=False)

# Also do a parametric plot
rho = fluid.rho if hasattr(fluid, 'rho') else fluid.rho_stp
p_atm = fons.pressure_atmospheric
m_a0 = system.get('actu_0').get_state('mass')
p_a0 = system.get('actu_0').get('pressure')
```

(continues on next page)

(continued from previous page)

```
v_a0 = m_a0 / rho * p_atm / p_a0
m_a1 = system.get('actu_1').get_state('mass')
p_a1 = system.get('actu_1').get('pressure')
v_a1 = m_a1 / rho * p_atm / p_a1

fig, ax = plt.subplots(1)
ax.plot(v_a0 * 1e6, v_a1 * 1e6)
ax.set_aspect('equal')
ax.set_xlabel('volume actuator 0 [ml]')
ax.set_ylabel('volume actuator 1 [ml]')
plt.show()
```

This gives the following two figures:

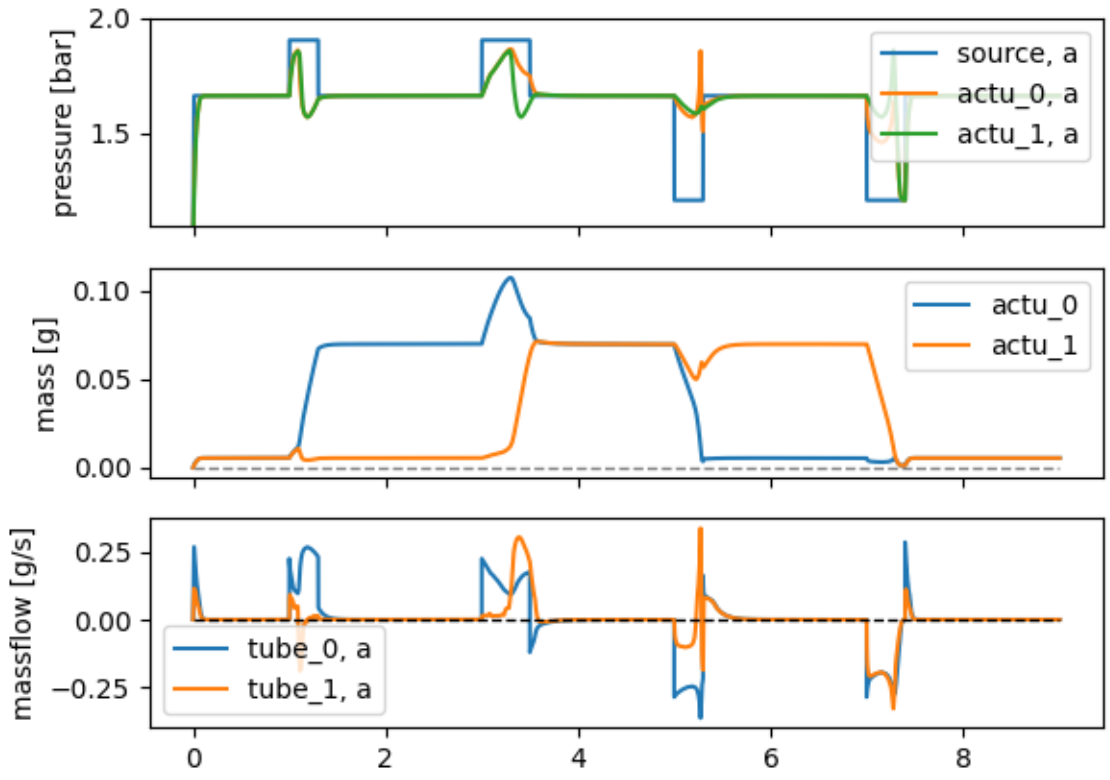

Fig. 3: Figure 3: Simulation output

Figures 3 and 4 show the simulation results. The middle graph of Figure 3 shows the achieved phase lag. Even while actuator 0 is inflated first, it is actuator 1 that is deflated last. Also note that the system state between the transitions is stable, as is shown by the constant pressure and mass in these region. These regions can thus be elongated or shortened by changing the pressure reference timings.

This phase lag is also well visible in Figure 4, which plots a virtual path traversed by the actuator volumes. In research

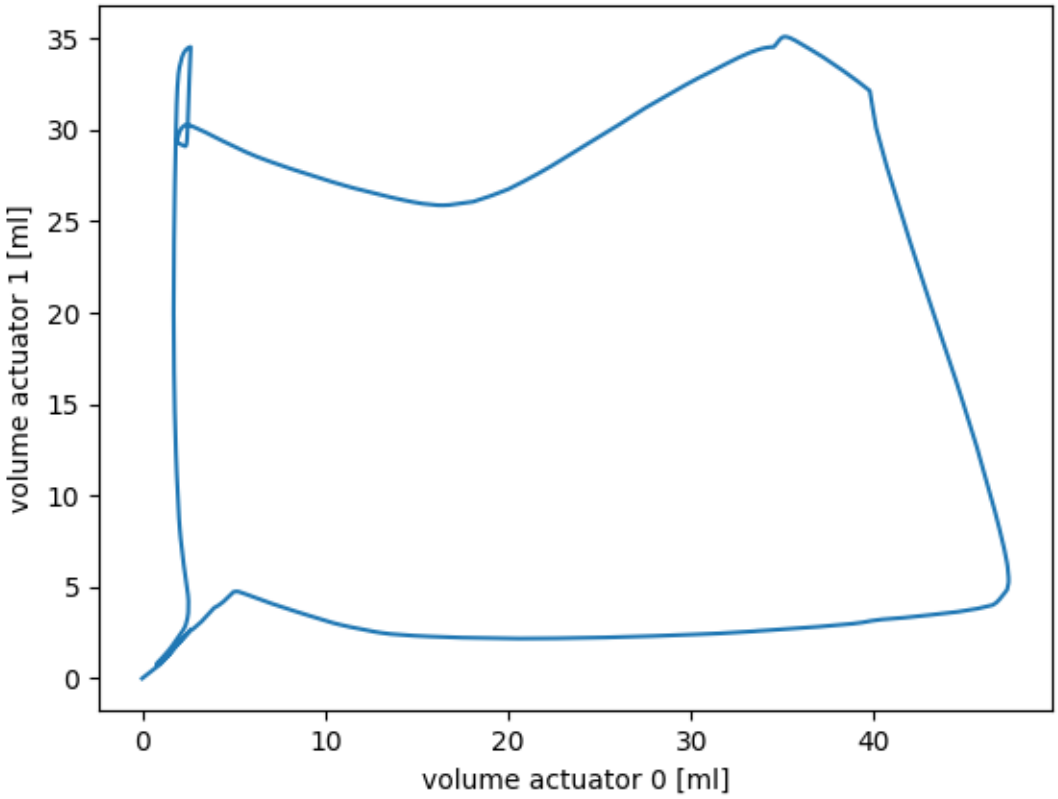

Fig. 4: Figure 4: Simulation output, actuator volumes

work TODO, this phase lag was used to produce a stepping motion for a legged robot, where one actuator moved the feet horizontally and another one moved the feet vertically.

CHAPTER

NINE

CUSTOM COMPONENTS

FONSim provides several ready-to-use components in its *standard library*. However, there's a good chance you will now and then want to define and use your own components. This tutorial discusses how to do this by redefining the *Container* component.

Full script, including usage example: `custom_component.py`.

9.1 Component definition

Components are defined as Python classes that inherit from the base class *Component*. The following snippet defines our new `Container` class and initializes the base class, and also provides some documentation. n addition to the mandatory label argument, we have added two custom arguments: the fluid and the container volume, as both of these influence the behavior of the component.

```
11 class Container(fons.Component):
12     """
13     A Container object is a (by default, empty) container or tank.
14     It has one terminal named 'a'.
15     It has one state named 'mass',
16     which represents the mass of the fluid inside the container.
17 
18     :param label: label
19     :param fluid: fluid, must be compressible
20     :param volume: volume of the container in m^3.
21     """
22     def __init__(self, label=None, fluid=None, volume=None):
23         super().__init__(label)
```

Next, we need to define the possibilities for the component to interact with other components by instantiating and adding a *Terminal*. This is done in the following five lines:

```
26         self.set_terminals(
27             fons.Terminal('a',
28                           [fons.Variable('pressure', 'across',
29                                          cnorm.pressure_atmospheric, label='p'),
30                            fons.Variable('massflow', 'through', label='mf')]))
```

The second line gives the terminal its label 'a', and the next three lines add the variables that constitute the terminal. Two terminals from two different components that are connected together should have the same variable _keys_. In line with other components in the FONSim standard library, we are working in a simple fluidical domain with the keys 'pressure' and 'massflow' (we assume the fluid temperature is constant throughout the system). Following classical

system modelling theory, the former is an 'across' variable while the latter is an 'through' variable. Both variables receive labels, respectively 'p' and 'mf', to refer to them in the further component definition.

Note: At each internal node, which internally is a group of terminals that are connected together, FONSim enforces that at any simulation time step all *across* variables are equal in value and that all *through* variables sum to zero. Internal nodes are normally not of relevance to the user and they do not represent physical elements (e.g. T-piece).

The container has a single state: the fluid mass 'm' inside. Its initial value is set to the fluid mass that corresponds to standard temperature and pressure (STP) (following the ideal gas law). This is easily added with the following two lines:

```
        self.set_states(fons.Variable('mass', 'local',
                                      volume*fluid.rho_stp, label='m'))
```

After the terminals and states have been defined, we need to describe the behaviour of the component. This is done by adding the two class methods `update_state` and `evaluate`.

The `update_state` method, shown in the excerpt below, defines how the state should change over time given particular values at the terminals. It is formulated as an explicit finite difference equation (hence the timestep `dt` is provided as argument). The two arguments 'mf' and 'm' should match with the variable labels given earlier. In the case of our container, this is a simple finite integral (the mass inside changes over time as there is massflow in- and out).

```
        @self.auto_state
        def update_state(dt, mf, m):
            m_new = m + mf*dt
            return {'m': m_new}
        self.update_state = update_state
```

The latter (`evaluate`) is shown in the following snippet and defines how the terminal and state variable values relate. It is formulated as an implicit equation. In the case of our container, it is the ideal gas law for a fluid with a constant temperature. One can add the argument `t` which will receive the value of the current simulation time. Here it is left out because the behaviour does not depend on time.

This is the end of the component definition. FONSim takes care of estimating the derivatives. Alternatively, we can specify them ourselves, which is discussed in the next section.

9.2 Manual derivative definition

Sometimes it is necessary for stability to manually specify the derivative. It also increases the simulation speed. The following example shows the two methods with the derivative definitions:

```
        # With derivative specified
        @self.auto_state
        def update_state(dt, mf, m):
            jacobian = {}
            m_new = m + dt * mf
            jacobian['m/p'] = 0
            jacobian['m/mf'] = dt
            return {'m': m_new}, jacobian
        self.update_state = update_state

        @self.auto
```

(continues on next page)

(continued from previous page)

```
        def evaluate(p, m):
            mass_stp = volume*fluid.rho_stp
            values, jacobian = np.zeros([1], dtype=float), [{}]
            values[0] = m*cnorm.pressure_atmospheric - mass_stp*p
            jacobian[0]['m'] = cnorm.pressure_atmospheric
            jacobian[0]['p'] = -mass_stp
            return values, jacobian
        self.evaluate = evaluate
```

For the state update, the derivatives are specified in the form of a dictionary `jacobian` with as keys strings of the partial derivatives. For the `evaluate` method, they are formulated as a list of dictionaries (also named `jacobian`).

9.3 Usage

To finish, the component can be used like any other component. For example,

```
system.add(Container('container', fluid=fluid, volume=250e-6))
```

creates and adds a container to the *system* object with a volume of 250 ml. If desired, you can run the simulation coded in the example file and view the resulting plot.

9.4 Conclusion

In this tutorial, we have discussed how to create a custom component in FONSim by redefining the Container component. We have explained the component definition process, including how to define the terminals, states, and behavior of the component. We have also demonstrated how to use the custom component by creating an instance of it, and using it in a system. With this information, you should now be able to create your own custom components and use them in your FONSim simulations.

CHAPTER

TEN

INTRODUCTION

Introduction to codedoc.

The codedoc has been generated using Sphinx from the docstrings in the code.

The text below provides a high-abstraction-level introduction to the FONSim codebase.

10.1 Main parts

FONSim consists of nine main parts. Each part is discussed briefly.

- `core`: This is the network simulation functionality. It is written agnostically of the domain (fluidic, eletric, ...) to be simulated. This core is discussed in detail further on.
- `components`: The standard library of fluidic components with actuators, tubes, restrictors, sources etc.
- `wave`: Signal generators: block wave, sine wave etc.
- `data`: Functionality for handling data. The files `curve.py`, `pvcurve.py`, `dataseries.py` and `interpolate.py` ease working with pv-curves, including reading them in from CSV files. The simulation data export functionality resides in `writeout.py`.
- `fluid`: Defines how fluids should be specified in the form of class definitions. Currently supported are ideal gasses and incompressible liquids.
- `fluids`: Standard library of fluids with air, water etc.
- `visual`: Plotting functionality specific to visualizing simulation results.
- `conversion`: String comparison and unit conversion.
- `constants`: Physical constants.
- `resources`: Data files such as pv-curves in CSV format, mostly used in examples and tests.

10.2 Core

The core contains all functionality that is required for simulating networked system but, as it is written domain-agnostic, it does not contain any code specific to fluidic system.

The core consists of seven files, each discussed here.

- `variable.py`

  Class definition of `Variable`. Variable objects are used to tell to the solver that there is a yet unknown timeseries of numerical scalar values that will have to be solved for during the simulation. After running the simulation, for

each Variable object a 1D array of values will have been derived that satisfies the given equations best. To put it simply, the solver uses the Variable objects to keep reference of what numerical value belongs to what equations.

A Variable has a key to indicate the medium it refers to (for example, pressure, massflow, or temperature). It also has an orientation to indicate which network equations should be applied when it is connected to other components (or when it is used stand-alone).

- `terminal.py`

  Class definition of `Terminal`. A Terminal contains one or more Variable instances that are often used alongside each other. A Terminal can be connected to another terminal. The solver automatically generates the network equations (which relates the Variables of different components to each other) from how the terminals are connected to each other. Therefore, the Variables in a Terminal should be defined such that they make physical sense.

  In practice this often leads to two Variables, one with orientation *across* and another with orientation *over*. Examples of typical pairs and their domain include pressure and massflow (fluidic), voltage and current (electric), linear displacement and force (mechanical, linear), rotational displacement and torque (mechanical, rotational).

- `component.py`

  Class definition of the base class `Component`. This class definition provides the functionality common to all components. Usable components such as actuators, sources etc. inherit from this base class and add the relevant Variables, Terminals and internal equations. In practice, the class Component is never instantiated, as doing so would create a component without any physical usefulness.

- `system.py` and `node.py`

  Class definitions of respectively `System` and `Node`. A System object holds information on which components it contains and how they are connected to each other. A connection between two terminals (a connection is always between terminals, not between components) is recorded in a Node object. A Node class instance contains of data only a list of Terminals it is connected to, the Node class mostly defines functionality to manipulate Node objects.

  While it is usual to say that x terminals are connected to each other, in the simulator this is handled by connecting those x terminals to a common node. Each Node instance therefore contains the Terminals the Node is connected to. A Terminal is always connected to a Node, and vice-versa. A Terminal may not be connected to more than one Node. A Terminal does not know to which Node it is connected, this information is only stored in the nodes.

- `simulation.py`

  Class definition of Simulation. This class provides functionality for:

  - Deriving mathematical constructs from a relationally-specified System object: matrices, listing all equations etc.
  - Evaluating all component equations
  - Memory management
  - Starting the the numerical simulation

  For solving the resulting system of equations it uses a solver. The solvers are defined in `solver.py`.

  Putting the relationally-specified System object in a mathematical form is a computationally intensive task and is done only once during the initialization thanks to a cache system.

- `solver.py`

  Numerical solvers. These solvers assume particular functionality of the class Simulation.

CHAPTER

ELEVEN

FONSIM.CORE PACKAGE

11.1 Submodules

11.2 fonsim.core.component module

Class Component

2020, July 21

class fonsim.core.component.**Component**(*label*)

Bases: object

Components to build a system with.

TODO

States and variables The state data (e.g. amount of fluid inside actuator) is saved as 2D-array in the components themselves (and the solver always refers to these) while the argument data (e.g. pressure, flow in/out) is saved as a list of _references_ to nameless 1D-arrays created by the solver. The component object provides functionality for the solver to allocate memory for the states, but not for allocating the variables. The solver takes care of calling these functions necessary. The object property `state_history` holds the 2D-array and object property `argument_history` holds a list with references to the 1D-arrays.

Parameters
label – Component name.

auto(*func*)

auto_state(*func*)

evaluate(*values*, *jacobian_state*, *jacobian_arguments*, *state*, *arguments*, *elapsed_time*)

Evaluates the component internal equations. **This method should be static**.

Note: only evaluate left-hand side (LH) of equation, equation should be structured such that RH is always zero.

Parameters

- **values** – array where the equation residuals will be stored.
- **jacobian_state** – array where the jacobian to the state will be stored.
- **jacobian_arguments** – array where the jacobian to the arguments will be stored.
- **state** – numerical values belonging to the state Variables.
- **arguments** – numerical values belonging to the Variables.

- **elapsed_time** – ? TODO.

Returns
None

get(*variable_key*, *terminal_label=None*)

Same as method `self.get_all`, but returns only the first return value.

get_all(*variable_key*, *terminal_label=None*)

Get simulation results. Supports 'smart matching' by comparing string distances.

Parameters

- **variable_key** – key of variable, e.g. 'pressure'
- **terminal_label** – label of terminal, e.g. 'a'

Returns
Numpy ndarray object and Terminal object

get_state(*label*)

Get simulation results. Supports 'smart matching' by comparing string distances.

Parameters
label – state label, e.g. 'volume'

Returns
Numpy ndarray object

get_terminal(*terminallabel=None*)

Returns Terminal object. If no label given, returns the first unconnected terminal. If label given, returns terminal with that label. If no terminal found, returns None.

Parameters
terminallabel – Label of terminal

Returns
Terminal object

set_arguments(**arguments*)

Overwrite component arguments list with the provided Variable objects.

Parameters
arguments – one or more Variable objects

Returns
None

set_states(**states*)

Overwrite component states list with the provided Variable objects.

Parameters
states – one or more Variable objects

Returns
None

set_terminals(**terminals*)

Overwrite component terminals list with the provided Terminal objects and attach those terminals to the component.

Parameters
terminals – one or more Terminal objects

Returns
None

`update_state`(*state_new*, *jacobian*, *state*, *arguments*, *dt*)

Evaluates the update to the component state. **This method should be static**.

Parameters

- **state_new** – array where the new state values will be stored.
- **jacobian** – array where the jacobian to the arguments will be stored.
- **state** – numerical values belonging to the state Variables.
- **arguments** – numerical values belonging to the Variables.
- **dt** – time discretization timestep.

Returns
None

11.3 fonsim.core.node module

Class Node

2021, January 14

`class fonsim.core.node.Node`(**terminals*)

Bases: `object`

Connection between multiple component terminals

`add_terminals`(**terminals*)

Connect terminals to the node

Parameters
terminals – Terminal object. Multiple can be provided

Returns
None

`contains_terminal`(*terminal*)

Check if the node contains the requested terminal

Parameters
terminal – Terminal object

Returns
Boolean

`get_variables`(*orientation=None*, *key=None*)

Get a list of all variables with the provided orientation and/or key associated with the node.

Parameters

- **orientation** – optional string specifying requested variable orientation
- **key** – optional string specifying requested variable key

Returns
list of Variable objects

merge_node(*node*)

Connect all terminals from another node to this node

Parameters
node – Node object

Returns
None

11.4 fonsim.core.setnumpythreads module

https://gitlab.com/abaeyens/fonsim/-/issues/19

2022, June 04

fonsim.core.setnumpythreads.**setnumpythreads**(*nb_threads=1*)

11.5 fonsim.core.simulation module

Class Simulation

2020, July 22

class fonsim.core.simulation.**Simulation**(*system_to_simulate*, *duration=10*, *step=None*, *step_min=None*, *step_max=None*, *max_steps=0*, *verbose=0*)

Bases: object

Class to convert network information (how the components are connected to each other) and component information (equations for each component) to a single non-linear system of equations (function vector + Jacobian) describing the whole system.

Solving this system is to be done by a solver. This object contains a solver object in the property self.solver. (Future functionality: possibility to select a solver manually etc.)

The Class Solver interacts heavily with the System class and expects the following methods to be available:

- evaluate_equations()
- update_state()
- equation_to_string()

as well as the following properties:

- system
- phi, H and A
- arguments
- nb_arguments and nb_network_equations
- times and duration and verbose

Parameters

- **system_to_simulate** – System object with components and how they are connected
- **duration** – amount of time the system will be simulated for

- **step** – initial time increment
- **step_min** – minimal time increment during the simulation
- **step_max** – maximal time increment during the simulation
- **max_steps** – maximum amount of time increments before the simulation is terminated prematurely. A value of 0 disables this behavior (default)
- **verbose** – level of information printed in the console during the simulation. All messages belonging to a level with a number lower than or equal to the provided parameter will be displayed, with the possible levels being:
  - -1: simulation start and end messages
  - 0 (default): simulation progress in % steps
  - 1: system matrices on iterations with bad convergence

`check_issolvable()`

Warns when number of equations doesn't equal number of unknowns. :return: None

`equation_to_string(`*equation_index*`)`

Return a string describing the equation with the provided index in a human-readable format. For a network equation, this string contains the involved variables and their coefficients. For a component equation, this string mentions the component label and the index of the equation in the list of equations of that particular component.

Parameters
equation_index – index of the row in the simulation equation matrix corresponding to the desired equation

Return eq_str
string representing the equation

`evaluate_equations(`*simstep*, *g*, *H*, *elapsed_time*, *dt*`)`

Evaluate component equations to obtain evaluated function vector (equation residuals) and jacobian. This method will call the method `Component.evaluate` on each component. This method will also call the method `Component.update_state` on each component.

Note: This function does not evaluate (or update) the network equations (upper part of the jacobian 'H')!

Parameters

- **simstep** – simulation timestep index to start from
- **g** – numpy ndarray for the evaluated function vector
- **H** – numpy ndarray for the evaluated jacobian
- **elapsed_time** – time elapsed
- **dt** – timestep (0 for explicit Euler, dt for implicit Euler)

Returns
None

`extend_memory(`*nb_extra_steps*`)`

Increase the size of the simulation memory without overwriting previous results. This deals with the same entities as described in the documentation of initialize_memory

Parameters
nb_extra_steps – amount of time steps by which to increase the memory

Returns
None

`fill_network_matrix`(*A*)

Fill the network matrix. The network matrix is constant over the simulation, at least supposing the network configuration does not change. It is thus sufficient to calculate it a single time.

There are two types of network equations, one for across variables and one for through variables. At each node, all across variables have to be equal. Thus for each node n-1 equations, n being the number of components attached to that node. Concerning the through variables, the sum of all through variables should be zero at each node, thus one equation for each node.

Parameters
A – system Jacobian matrix, numpy ndarray n x n, n=len(phi)

Returns
None

`init_matrixconstruction`()

Create some LUT-style lists and dicts so data can be moved around quickly in the simulation loop.

Returns
None

`initialize_memory`(*nb_steps*)

Initialize all arrays that will hold the simulation results through time. This includes - The vector with time values - Component argument and state histories (in Component class) - The simulation phi matrix with all arguments over time All previously stored results are overwritten. This method initializes the arrays with zeros and therefore does not use the `initial_value` attribute of the Variable objects.

Parameters
nb_steps – estimated amount of time steps in the simulation

Returns
None

`map_phi_to_components`(*phi*)

Send addresses of arguments over time to components, so one can get the data from the component without passing by the Simulation object. Furthermore, it avoids duplicating the data.

Parameters
phi – numpy ndarray m x n with the argument vector over time (m = nb timesteps and n = nb arguments)

Returns
None

`map_state_to_components`(*state*)

Send addresses of state over time to components, so one can get the data from the component without passing by the Simulation object. Furthermore, it avoids duplicating the data.

Parameters
state – numpy ndarray m x n with the state vector over time (m = nb timesteps and n = nb states)

Returns
None

`print_equations`()

Print a human-readable representation of the full system of equations to the console.

Returns
None

`run()`

Run simulation, with the parameters specified previously.

Returns
None

`set_initial_values_phi(step=0)`

Takes the initial values from the component argument Variable objects and writes them to the simulation memory (matrix 'phi').

Parameters
step – simulation step at which to write the initial value

Returns
None

`set_initial_values_state(step=0)`

Takes the initial values from the component state Variable objects and writes them to the simulation memory (matrix 'state').

Parameters
step – simulation step at which to write the initial value

Returns
None

`slice_memory(start_step, end_step)`

Decrease the size of the simulation memory by taking a slice out of it. This deals with the same entities as described in the documentation of initialize_memory

TODO update such that takes slice as argument

Parameters

- **start_step** – index of the first time step in the range to maintain
- **end_step** – index of the first time step outside the range to maintain

Returns
None

`update_state(simstep, dt)`

Update the state variables in all components using the arguments in self.phi at step n + 1 (n = 'simstep'). This method will call the method `Component.update_state` on each component.

Parameters

- **simstep** – simulation timestep index to start from
- **dt** – timestep

Returns
None

11.6 fonsim.core.solver module

Classes with available solvers

A solver takes care of solving the (non-linear) system of equations generated by the Simulation object. This object can interact with the Simulation object.

The Simulation class expects the following methods from the solver object: - `get_nb_steps_estimate()` - `run_step(simulation_step_index)`

Current solvers:

1. ImplicitEulerNewtonConstantTimeStep
2. ImplicitEulerNewtonAdaptiveTimeStep

2020, July 23

class fonsim.core.solver.**ImplicitEulerNewton**(*simulation*)

Bases: object

apply_solver_bias(*bias=1e-12*, *step=0*)

Give elements in the solution vector a small positive value, to ease starting the simulation

Parameters

- **bias** – bias value added to the solution vector entries
- **step** – simulation step for which this bias is applied

Returns
None

get_all_variables(*simstep*)

Parameters
simstep – simulation step at which variables are queried

Returns
np array with the values of all arguments and component states at the desired simulation step

get_residual(*simstep*, *res=None*)

Evaluate the simulation system of equations at the provided timestep to get the residual vector

Parameters

- **simstep** – index of simulation timestep at which the system of equations is evaluated
- **res** – optional initialized residual vector that will be overwritten by this function in place

Returns
evaluated residual vector

newton_solver(*step*, *iterations=100*, *alpha=1.0*)

Newton method for solving the system equations and storing the solution in the simulation phi vector at an time step

Parameters

- **step** – step index for which the system will be solved and the solution will be stored. The initialization for the solver can be done externally by setting self.sim.phi[step] to the initial guess

- **iterations** – maximum number of newton solver iterations. 100 (default) gives good results although it often converges in one step and otherwise mostly in less than ten
- **alpha** – correction constant for damped Newton method

Returns
flag indicating exit status of the solver for this step: * 0: maximum amount of iterations reached without convergence * 1: solver converged quickly * 2: solver converged slowly

print_report(*simstep*)

Print a report on the solver status at a certain simulation time step.

Parameters
simstep – index of simulation timestep for which the report is generated

Returns
None

class fonsim.core.solver.**ImplicitEulerNewtonAdaptiveTimeStep**(*simulation*, *step*, *step_min*, *step_max*, *max_attempts=200*)

Bases: *ImplicitEulerNewton*

delta_in_range(*delta*)

Check whether a change in simulation variables (both arguments and states) during a step is within an acceptable range compared to the other simulation step results or it is abnormally large

Parameters
delta – vector with change in all the arguments between consecutive simulation steps

Return in_range
True if delta is of an acceptable magnitude and False if it represents a change which is abnormally large

get_nb_steps_estimate()

Returns
number of timesteps the solver thinks it will need to finish the simulation

run_step(*simstep*)

Run a single step of the solver. Note: Calling it at step n ('simstep' = n) results in it writing to the next step, aka step n+1

Parameters
simstep – index of timestep with the last simulated results

Returns
list with as elements: * 0: boolean showing solver convergence for the simulation step * 1: string giving more information about the exit status

update_delta_range(*nb_points*, *delta*)

Update the average and variance of the changes of simulation variables (both arguments and states) over all simulation steps with the change at the current step

Parameters

- **nb_points** – amount of steps included in the last metrics
- **delta** – new change in simulation arguments between consecutive steps to include in the metrics

Returns
None

class fonsim.core.solver.**ImplicitEulerNewtonConstantTimeStep**(*simulation*, *step*)

Bases: *ImplicitEulerNewton*

get_nb_steps_estimate()

Returns
number of timesteps the solver needs

run_step(*simstep*)

Run a single step of the solver. Note: Calling it at step n ('simstep' = n) results in it writing to the next step, aka step n+1

Parameters
simstep – index of simulation timestep with the last results

Returns
None

11.7 fonsim.core.system module

Class System

2020, July 22

class fonsim.core.system.**System**(*label=None*)

Bases: object

A System is a collection of interconnected components. It contains the Component objects and keeps track of how they are connected to each other. A System with components is created by first creating the System object, then adding components to this object and finally connecting these components to each other.

Parameters
label – Label of the system, optional and currently not important.

add(**components*)

Add components to the system.

Parameters
components – Component object(s) to be added to the system

Returns
None

connect(**args*)

Connect two or more components together by connecting their terminals. In case terminals are not specified directly, the Component objects decide on which of their terminals to connect. The connections are made in sequential pairs using the method `System.connect_two_components` which in turn uses the method `self.connect_two_terminals`.

The components and/or terminals to connect should be as specified in the method `System.get_component_and_terminal`.

For instead connecting all components to a common Node, use the method `System.connect_common`.

Parameters
args – component terminals

Returns
None

connect_common(**args*)

Connect two or more component terminals together. In case terminals are not specified directly, the Component objects decide on which of their terminals to connect. All Terminals are connected to each other, aka to a **common Node**. Making the connection is handled with the method `self.connect_two_terminals`.

The components and/or terminals to connect should be as specified in the method `System.get_component_and_terminal`.

For instead making sequential connections, use the method `System.connect`.

Parameters
args – component terminals

Returns
None

connect_two_components(*component_a*, *component_b*)

Connect two Components to each other. The connection is made using the method `self.connect_two_terminals`.

The component arguments component_a and component_b can be as specified in the method `System.get_component_and_terminal`.

Parameters

- **component_a** – First component, see description
- **component_b** – Second component, see description

Returns
None

connect_two_terminals(*terminal_a*, *terminal_b*)

Connect two system terminals together in a Node. These nodes exist for the workings of the solver and have nothing to do with any nodes in the fluidic networks being simulated.

Parameters

- **terminal_a** – Terminal object
- **terminal_b** – Terminal object

Returns
None

get(*component_label*)

Parameters
component_label – Label of the desired component.

Returns
Component object with the given label.

get_component_and_terminal(*arg*)

Get a pair of a Component and a Terminal given an argument `arg`. This argument can be:

- a string specifying a component label present in the system
- a Component object
- a Terminal object
- a Tuple with first the component label and then the terminal label as strings

If multiple choices are available, this method may make an undefined choice.

Parameters
arg – see desription

Returns
Component object, Terminal object

`get_connectivity_message()`

Get a message describing the connectivity of the system in case not every component is connected together.

Returns
message string

11.8 fonsim.core.terminal module

Class Terminal

2020, July 22

`class fonsim.core.terminal.Terminal(label, variables, variable_labels={})`

Bases: `object`

Component connection point with local through and across variables.

Note: In any particular Terminal object, there cannot be more than one variable with the same key.

Parameters

- **label** – Label to refer to the Terminal later on. Free to choose.
- **variables** – Variable objects that will belong to the Terminal.

`get_variable(key)`

Return the variable object attached to the terminal with the provided key, for example 'pressure'. If there is no variable with the requested key, None is returned.

Parameters
key – key of the variable to return

Return variable
attached variable with the matching key

`get_variables(orientation=None)`

Get list of all terminal variables with the given orientation. The orientation can be either "through" or "across". If not provided or None, all variables regardless of orientation are returned

Parameters
orientation – optional string specifying desired variable orientation

Returns
list of Variable objects

11.9 fonsim.core.variable module

Class Variable

2020, July 21

class fonsim.core.variable.**Variable**(*key*, *orientation*, *initial_value=0.0*, *terminal=None*, *label='None'*)

Bases: object

A Variable object is used to denote the presence of a yet unknown numerical value. For each Variable object, the solver will search for the optimal numerical values over time. The solver does so by solving the system of equations that connect these variables together. The variables are connected by each other by connecting the Terminal objects that contain the values to each other.

The parameter 'key' indicates the type label. Only Variable objects with the same type label can exchange information.

The parameter 'orientation' should have value 'across' or 'through' or 'free'. 'across' indicates that the value of the Variable will be shared with the Variable belonging to the other Terminal while 'through' indicates that its negative will be shared. The former is typically used with nondirectional values, such as pressure, while the latter is typically used with directional values, such as a massflow. 'local' indicates that it will not be shared (feature WIP).

Parameters

- **key** – type label, e.g. 'pressure', 'massflow'
- **orientation** – 'across', 'through' or 'free'.
- **initial_value** – Initial value, default: 0
- **terminal** – Terminal object to which Variable object get connected, default: None

copy_and_attach(*terminal*)

Return a copy of the variable object attached to a given terminal. The returned variable has the same key, orientation and initial value but is otherwise unrelated to the Variable object it is called upon.

Parameters
terminal – Terminal object to attach the variable copy to

Return variable
attached copy of the variable object

short_str(*nb_var_chars=1*)

Return a short string describing the variable more as a symbol than in words. This string contains the first n letters of the variable name as well as (if applicable) the port and component it is attached to.

Parameters
nb_var_chars – number of characters with which the variable key is abbreviated. Set to 0 to avoid abbreviation.

Return var_str
short string representing the variable

11.10 Module contents

2020, September 17

CHAPTER

TWELVE

FONSIM.COMPONENTS PACKAGE

12.1 Submodules

12.2 fonsim.components.actuators module

2020, July 21

class fonsim.components.actuators.**FreeActuator**(*label=None*, *fluid=None*, *curve=None*, *initial_volume=None*)

Bases: *Component*

An actuator with a custom pressure-volume relationship specified as a pressure-volume curve or pv-curve. It is named 'free' because the actuator cannot drive anything, at least in this simulation. It has two terminals 'a' and 'b'. It has one state 'mass' that represents the mass of fluid inside the actuator.

The argument fluid should be one of the fluids defined in the module `fluids`.

The argument curve should point to a pressure-volume curve (pv-curve) that describes the pressure-volume relationship of the actuator. It can be:

- a filename of a CSV file
- a PVCurve object
- an object that behaves sufficiently like a PVCurve object

Concerning the latter option, the object should provide the following methods:

- `get_initial_volume(p0)`
- `fdf_volume(volume)`

Parameters

- **label** – label
- **fluid** – fluid
- **curve** – pressure-volume curve (pv-curve)

fonsim.components.actuators.**freeactuator_compressible**(*self:* FreeActuator)

Init function, part specifically for compressible fluids.

Parameters
self – FreeActuator object

Returns
None

fonsim.components.actuators.**freeactuator_incompressible**(*self:* FreeActuator)

Init function, part specifically for incompressible fluids.

Parameters
self – FreeActuator object

Returns
None

12.3 fonsim.components.circulartube_autodiff module

2020, July 21

class fonsim.components.circulartube_autodiff.**CircularTube_autodiff**(*label=None*, *fluid=None*, *length=0.6*, *diameter=0.002*, *roughness=1.5e-06*)

Bases: *Component*

Tube modeled as an elongated cylindrical shape. The terminal labels are 'a' and 'b'. It is stateless (the kinetic energy of the fluid in the tube is neglected).

The fluid should be one of the fluids defined in the module `fluids`.

TODO:

- include kinetic energy of fluid in tube

Parameters

- **label** – label
- **fluid** – fluid
- **length** – length in m
- **diameter** – internal diameter in m
- **roughness** – wall roughness in m

fonsim.components.circulartube_autodiff.**circulartube_compressible**(*self:* CircularTube_autodiff)

Init function, part specifically for compressible fluids.

Parameters
self – CircularTube object

Returns
None

fonsim.components.circulartube_autodiff.**circulartube_incompressible**(*self:* CircularTube_autodiff)

Init function, part specifically for incompressible fluids.

Parameters
self – CircularTube object

Returns
None

12.4 fonsim.components.containers module

2020, July 21

class fonsim.components.containers.**Container**(*label=None*, *fluid=None*, *volume=None*)

Bases: *Component*

A Container object is a (by default, empty) container or tank. It has one terminal named 'a'. It has one state named 'mass', which represents the mass of the fluid inside the container.

The fluid should be one of the fluids defined in the module `fluids`. A Container object is mostly useful with compressible fluids.

Parameters

- **label** – label
- **fluid** – fluid
- **volume** – volume of the container in m^3.

fonsim.components.containers.**container_compressible**(*self:* Container)

Init function, part specifically for compressible fluids.

Parameters
self – Container object

Returns
None

fonsim.components.containers.**container_incompressible**(*self:* Container)

Init function, part specifically for incompressible fluids.

Parameters
self – Container object

Returns
None

12.5 fonsim.components.containers_autodiff module

2020, July 21

class fonsim.components.containers_autodiff.**Container_autodiff**(*label=None*, *fluid=None*, *volume=None*)

Bases: *Component*

A Container object is a (by default, empty) container or tank. It has one terminal named 'a'. It has one state named 'mass', which represents the mass of the fluid inside the container.

The fluid should be one of the fluids defined in the module `fluids`. A Container object is mostly useful with compressible fluids.

Parameters

- **label** – label

- **fluid** – fluid
- **volume** – volume of the container in m^3.

fonsim.components.containers_autodiff.**container_compressible**(*self:* Container_autodiff)

Init function, part specifically for compressible fluids.

Parameters
self – Container object

Returns
None

fonsim.components.containers_autodiff.**container_incompressible**(*self:* Container_autodiff)

Init function, part specifically for incompressible fluids.

Parameters
self – Container object

Returns
None

12.6 fonsim.components.dummy module

Dummy component for testing. Has one terminal such that system connectivity can be tested. 2022, May 06

class fonsim.components.dummy.**Dummy**(*label=None*)

Bases: *Component*

evaluate(*values*, *jacobian_state*, *jacobian_arguments*, *state*, *arguments*, *elapsed_time*)

Evaluates the component internal equations. **This method should be static**.

Note: only evaluate left-hand side (LH) of equation, equation should be structured such that RH is always zero.

Parameters

- **values** – array where the equation residuals will be stored.
- **jacobian_state** – array where the jacobian to the state will be stored.
- **jacobian_arguments** – array where the jacobian to the arguments will be stored.
- **state** – numerical values belonging to the state Variables.
- **arguments** – numerical values belonging to the Variables.
- **elapsed_time** – ? TODO.

Returns
None

12.7 fonsim.components.restrictors module

2020, July 21

class fonsim.components.restrictors.**CircularTube**(*label=None*, *fluid=None*, *length=0.6*, *diameter=0.002*, *roughness=1.5e-06*)

Bases: *Component*

Tube modeled as an elongated cylindrical shape. The terminal labels are 'a' and 'b'. It is stateless (the kinetic energy of the fluid in the tube is neglected).

The fluid should be one of the fluids defined in the module `fluids`.

TODO:

- include kinetic energy of fluid in tube

Parameters

- **label** – label
- **fluid** – fluid
- **length** – length in m
- **diameter** – internal diameter in m
- **roughness** – wall roughness in m

class fonsim.components.restrictors.**FlowRestrictor**(*label=None*, *fluid=None*, *diameter=0.002*, *k=0.6*)

Bases: *Component*

Flow restrictor modeled as an orifice with a K-factor. Terminals are named 'a' and 'b'. It is stateless.

The fluid should be one of the fluids defined in the module `fluids`.

Parameters

- **label** – label
- **fluid** – fluid
- **diamter** – diameter of orifice
- **k** – K-factor

fonsim.components.restrictors.**circulartube_compressible**(*self:* CircularTube)

Init function, part specifically for compressible fluids.

Parameters
self – CircularTube object

Returns
None

fonsim.components.restrictors.**circulartube_incompressible**(*self:* CircularTube)

Init function, part specifically for incompressible fluids.

Parameters
self – CircularTube object

Returns
None

fonsim.components.restrictors.**flowrestrictor_compressible**(*self:* FlowRestrictor)

Init function, part specifically for compressible fluids.

Parameters
self – FlowRestrictor object

Returns
None

fonsim.components.restrictors.**flowrestrictor_incompressible**(*self:* FlowRestrictor)

Init function, part specifically for incompressible fluids.

Parameters
self – FlowRestrictor object

Returns
None

12.8 fonsim.components.sources module

2020, July 21

class fonsim.components.sources.**MassflowSource**(*label=None*, *fluid=None*, *massflow=None*)

Bases: *Component*

Ideal massflow source. The massflow exactly equals the desired massflow. The pressure is limited to positive values. It has one terminal 'a' and is stateless.

The argument 'massflow' should be either a constant value or a callable method that takes a single argument, the argument being the elapsed time since the simulation start.

The argument fluid should be one of the fluids defined in the module `fluids`.

Parameters

- **label** – label
- **fluid** – fluid
- **massflow** – desired massflow

evaluate(*values*, *jacobian_state*, *jacobian_arguments*, *state*, *arguments*, *elapsed_time*)

Evaluates the component internal equations. **This method should be static**.

Note: only evaluate left-hand side (LH) of equation, equation should be structured such that RH is always zero.

Parameters

- **values** – array where the equation residuals will be stored.
- **jacobian_state** – array where the jacobian to the state will be stored.
- **jacobian_arguments** – array where the jacobian to the arguments will be stored.
- **state** – numerical values belonging to the state Variables.
- **arguments** – numerical values belonging to the Variables.
- **elapsed_time** – ? TODO.

Returns
None

class fonsim.components.sources.**PressureSource**(*label=None*, *fluid=None*, *pressure=None*)

Bases: *Component*

Ideal pressure source. The pressure exactly equals the desired pressure and the flow is unlimited. It has one terminal 'a' and is stateless.

The argument 'pressure' should be either a constant value or a callable method that takes a single argument, the argument being the elapsed time since the simulation start.

The argument fluid should be one of the fluids defined in the module `fluids`.

TODO

- Limit pressure to positive values.

Parameters

- **label** – label
- **fluid** – fluid
- **pressure** – desired pressure

evaluate(*values*, *jacobian_state*, *jacobian_arguments*, *state*, *arguments*, *elapsed_time*)

Evaluates the component internal equations. **This method should be static**.

Note: only evaluate left-hand side (LH) of equation, equation should be structured such that RH is always zero.

Parameters

- **values** – array where the equation residuals will be stored.
- **jacobian_state** – array where the jacobian to the state will be stored.
- **jacobian_arguments** – array where the jacobian to the arguments will be stored.
- **state** – numerical values belonging to the state Variables.
- **arguments** – numerical values belonging to the Variables.
- **elapsed_time** – ? TODO.

Returns

None

class fonsim.components.sources.**VolumeflowSource**(*label=None*, *fluid=None*, *volumeflow=None*)

Bases: *Component*

Ideal volumeflow source. The volumeflow exactly equals the desired volume flow. The pressure is limited to positive values. It has one terminal 'a' and is stateless.

The value 'volumeflow' should be either a constant value or a callable method that takes a single argument, the argument being the elapsed time since the simulation start.

The fluid should be one of the fluids defined in the module `fluids`.

TODO

- Limit pressure to positive values.

Parameters

- **label** – label
- **fluid** – fluid

- **volumeflow** – desired volumeflow

fonsim.components.sources.**volumeflowsource_compressible**(*self:* VolumeflowSource)

Init function, part specifically for compressible fluids.

Parameters
self – VolumeflowSource object

Returns
None

fonsim.components.sources.**volumeflowsource_incompressible**(*self:* VolumeflowSource)

Init function, part specifically for incompressible fluids.

Parameters
self – VolumeflowSource object

Returns
None

12.9 Module contents

2020, September 9

CHAPTER

THIRTEEN

FONSIM.DATA PACKAGE

13.1 Submodules

13.2 fonsim.data.curve module

Class Curve

2020, September 1

class fonsim.data.curve.**Curve**(*data*, *key_x*, *key_f*, *convert_to_base_si=False*, *autocorrect=False*, ***interpolation_opts*)

Bases: object

Class to ease working with pv- and pn-curves and similar curves

Parameters
data – filepath to CSV file or DataSeries-like object

autocorrect(*arg*)

TODO implement autocorrect here I made this a separate function such that child classes can modify the data as they want before they use the autocorrect functionality.

Return None

fdf(*x*)

Readout f(x)

Parameters
x – x

Returns
f, df

13.3 fonsim.data.dataseries module

Class DataSeries

2020, September 1

class fonsim.data.dataseries.**DataSeries**(*filename*, *bytestring=None*)

Bases: object

Class to load and hold tabular data from CSV file. Numerical data is stored in numpy arrays as floats. Labels and units are stored in Python lists

Parameters

- **filename** – path to file to read or, if bytestring given, filetype
- **bytestring** – bytestring with file data

load_data(*filename*, *bytestring=None*)

Load in data. Provide a filename or byte string. If providing a byte string, provide the filetype extension (e.g. .csv) to the filename argument such that the formatting of the bytestring can be determined.

Parameters

- **filename** – path to file to read or, if bytestring given, filetype
- **bytestring** – bytestring with file data

Returns
None

13.4 fonsim.data.interpolate module

Function interpolate_fdf

2020, September 4

fonsim.data.interpolate.**interpolate_fdf**(*x*, *xs*, *ys*, *extrapolate=False*, *extrapolate_derivative=False*, *method='linear'*)

Quickly interpolate a dataseries and return both interpolated value and its derivative.

Arrays xs, ys can be Numpy arrays, yet any object that allows indexing can be used, such as Python lists.

Method: linear or quadratic Search: bisection

TODO

- Document pchip method.
- Document extrapolation options. The current naming is not great, as choosing 'False' for extrapolation results in a zero-order extrapolation. Better to use numbers, e.g. '-1' for no extrapolation (throw error), '0' for zero-order extrapolation and '1' for linear extrapolation?
- Shorten parameters `extrapolate` and `extrapolate_derivative`?

Parameters

- **x** – x value to interpolate at
- **xs** – x dataseries
- **ys** – y dataseries
- **extrapolate** – True or False
- **extrapolate_derivative** – True or False
- **method** – 'linear' or 'quadratic'

Returns
f and df/dx, both evaluated at x

Note: Given that this function is used very often, it is written with the eye on fast execution rather than modularity and good looks.

13.5 fonsim.data.pvcurve module

Class PVCurve

2020, September 4

class fonsim.data.pvcurve.**PVCurve**(*data*, *pressure_reference='relative'*, *autocorrect=False*, ***interpolation_opts*)

Bases: *Curve*

Class to ease working with pv-curves

Warning: original data in DataSeries object may be modified by this function. Take a deepcopy if modification undesirable.

The autocorrect functionality provides a little tool to correct measurement data. Parameter `autocorrect` should be a tuple of length two (respectively volume and pressure) or a scalar. The elements of this tuple, or the scalar, can be:

- False: No correction applied.
- True: Default correction applied. For volume, the volume at index 0 will equal zero. For pressure, the pressure at index 0 will equal standard atmospheric pressure.
- A scalar value: Default correction is applied whereafter an offset with the given value is applied. Units are m^3 for volume and Pa for pressure.
- A function: The function is applied to the value series. The function should take the value series as argument and should return the new value series.

Note: the pressure_reference parameter looses its effect when autocorrect is applied to pressure.

Note: The volume data sequence should be increasing or decreasing, otherwise the interpolation function will not work.

Example:

```
import fonsim

# Create PVCurve object
curve = fonsim.data.pvcurve.PVCurve('mypvcurvefile.csv',
                pressure_reference='relative', autocorrect=True)

# Readout the absolute pressure and its derivative to volume
# at volume 3.8e-6 m^3 (= 3.8 ml)
p, dp_dv = PVCurve.fdf_volume(3.8e-6)
```

TODO Discuss format of CSV file.

Parameters

- **data** – filepath to CSV file or DataSeries-like object
- **pressure_reference** – “relative” or “absolute”

- **autocorrect** – see description
- **interpolation_opts** – kwargs for interpolation function

fdf_volume(*volume*)

Readout the pressure for a given volume

Parameters
volume – volume in [m3]

Returns
f, df

get_initial_volume(*p0*)

Get the volume of the first datapoint on the curve that approaches the provided pressure value the closest

TODO what is this function used for?

Parameters
p0 – pressure at which to find the first matching volume

Returns
first closest matching volume

13.6 fonsim.data.writeout module

Function writeout_simulation

2020, September 9

class fonsim.data.writeout.**Bank**

Bases: object

add(*obj*)

Add an object. Object does not have to be hashable. :param obj: object :return: index of object, integer

indices()

Return all indices pointing to objects. :return: all indices, Python range

fonsim.data.writeout.**writeout_simulation**(*filename*, *simulation*)

Write out simulation data in components to a file. Supported formats: JSON. Format follows from filename extension.

Parameters

- **filename** – string with filepath
- **simulation** – Simulation-like object

Returns
None

===

JSON specification:

```
{
  "scheme": <string>,
  "general": {
    "date": <timestamp>,
```

(continues on next page)

(continued from previous page)

```
    "hostname": <computer name",
    "version": <version>
  },
  "simulation": {
    "system": {
      "label": <string>,
      "nb components": <integer>
    },
    "solver": {
      "name": <string>,
    },
    "time": <key in "data"
  },
  "components": {
    <component_label>: {
      "terminals": {
        <terminal_label>: {
          "over": {
            <over_label>: <key in "data">
          },
          through": {
            <through_label>: <key in "data">
          }
        },
        ...
      }
      "states": {
        <state_label>: <key in "data">,
        ...
      },
      "time": <key in "data">
    },
    ...
  },
  "data": {
    <key>: <list with numbers>,
    ...
  },
}
```

13.7 Module contents

2020, September 18

CHAPTER

FOURTEEN

FONSIM.FLUID PACKAGE

14.1 Submodules

14.2 fonsim.fluid.fallback module

Tool to easily select the most appropriate fallback fluid

2020, September 10

fonsim.fluid.fallback.**get_fluid**(*fluid*, *fluids_desired*)

Parameters

- **fluid** – Fluid instance
- **fluids_desired** – Ordered iterable (e.g. list, tuple) of Fluid types

Returns
Fluid instance

fonsim.fluid.fallback.**select_fallback**(*fluid*, *fluids_desired*)

Return given fluid if its type is in fluids_desired. Otherwise, from all its fallback fluids return the fluid that is first in fluids_desired.

Note: returns tuple

Parameters

- **fluid** – fluid object
- **fluids_desired** – ordered iterable (e.g. list, tuple) of fluid types

Returns
(fluid object, index of type of fluid object in fluids_desired)

14.3 fonsim.fluid.fluid module

Fluid classes for keeping fluid properties

Currently supported types:

- IdealIncompressible
  - newtonian
- IdealCompressible

- newtonian, ideal gas

Types in progress:

- Bingham - example of a non-newtonian one

2019, September 7

class fonsim.fluid.fluid.**Bingham**(*name*, *rho*, *mu_p*, *tau_y*, *fallbacks=None*)

Bases: `object`

Note: in progress!

Parameters

- **name** – name of fluid
- **rho** – density [kg/m**3]
- **mu_p** – plastic viscosity [Pa s] (sometimes called Poise, [P])
- **tau_y** – yield point (YP) (yield shear stress) [Pa]
- **fallback** – fallback fluid

class fonsim.fluid.fluid.**Fluid**(*name*)

Bases: `object`

Parameters
name – name of fluid

select_object_by_fluid(*object_by_fluids_compatible*)

Rely on fluid fallback functionality.

Note: select_fallback expects an ordered iterable by making it a list the dict fluids_desired becomes ordered, yet a dict has no order of it keys. The made order is thus not the same as the one defined in the components.

Solution: use OrderedDict in the component definition, those remember the order of the keys like they were defined.

The function does not crash when giving a standard Dict, but it won't be able to respect the order.

Parameters
object_by_fluids_compatible – OrderedDict with (type(fluid), object) pairs

Returns
object

class fonsim.fluid.fluid.**IdealCompressible**(*name*, *rho*, *mu*, *fallbacks=None*)

Bases: *Fluid*

Parameters

- **name** – name of fluid
- **rho** – density at STP conditions [kg/m**3]
- **mu** – dynamic viscosity [Pa s]
- **fallback** – fallback fluid

class fonsim.fluid.fluid.**IdealIncompressible**(*name*, *rho*, *mu*)

Bases: *Fluid*

Parameters

- **name** – name of fluid

- **rho** – density [kg/m**3]
- **mu** – dynamic viscosity [Pa s]

14.4 Module contents

CHAPTER

FIFTEEN

FONSIM.FLUIDS PACKAGE

15.1 Submodules

15.2 fonsim.fluids.Bingham module

TODO

15.3 fonsim.fluids.IdealCompressible module

Some common compressible fluids at T = 20 °C, p = 1 bar. These fluids are described as an ideal gas.

Available:

- air

In progress:

- helium (H2)
- nitrogen (N2)
- carbondioxide (CO2)
- oxygen (O2)

2020, September 9

`fonsim.fluids.IdealCompressible.air = <fonsim.fluid.fluid.IdealCompressible object>`

helium = fluid.IdealCompressible(name="helium", rho=0.167, nu=1.17e-4) nitrogen = fluid.IdealCompressible(name="nitrogen", rho=1.17, nu=1.51e-5) carbondioxide = fluid.IdealCompressible(name="carbondioxide", rho=1.81, nu=8.1e-6)

Source: https://www.engineeringtoolbox.com/ oxygen = fluid.IdealCompressible(name="oxygen", rho=1.31, nu=1.54e-5)

15.4 fonsim.fluids.IdealIncompressible module

Some common incomressible fluids at T = 20 °C, p = 1 bar.

Available:

- water

In progress:

- ethylene_glycol
- ethylene_glycol_30pct
- ethylene_glycol_50pct
- mineral_oil

2020, September 9

`fonsim.fluids.IdealIncompressible.water = <fonsim.fluid.fluid.IdealIncompressible object>`

ethylene_glycol = fluid.IdealIncompressible(name="ethylene glycol", rho=1116, nu=1.91e-5) ethylene_glycol_30pct = fluid.IdealIncompressible(name="ethylene glycol 30%", rho=1038, nu=2.089e-6) ethylene_glycol_50pct = fluid.IdealIncompressible(name="ethylene glycol 50%", rho=1056, nu=3.66e-6)

Source: http://www.fao.org/fileadmin/user_upload/jecfa_additives/docs/monograph13/additive-527-m13.pdf mineral_oil = fluid.IdealIncompressible(name="mineral oil", rho=850, nu=9.75e-4)

15.5 Module contents

CHAPTER

SIXTEEN

FONSIM.WAVE PACKAGE

16.1 Submodules

16.2 fonsim.wave.custom module

Class CustomWave

2020, September 5

class fonsim.wave.custom.**Custom**(*wave_array*, *time_notation='absolute'*, *kind='previous'*)

Bases: `object`

Custom wave

The argument for wave_array should be a 2D-indexable array-like object (List, Tuple, numpy.ndarray, etc.) and contain the time values and the corresponding output values. One dimension should have size two. The function is transpose-agnostic.

The argument for time_notation can be 'absolute' or 'relative'. In case of relative, each time value is relative to the one before it.

The default argument 'previous' for 'kind' results in a rectangular wave (zero-order interpolation). The interpolation is handled using the Scipy method `scipy.interpolate.interp1d` and the available interpolation kinds therefore are those supported by this Scipy method. For a complete reference, see https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html. From above site (copied 2020, September 5):

> Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', 'next', where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order; 'previous' and 'next' simply return the previous or next value of the point) or as an integer specifying the order of the spline interpolator to use.

To readout the value (and therefore call the interpolation function), call the created object.

Example:

```
import fonsim

# Create Custom wave object
# times: 0.0, 1.0, 1.5 and values: 12, 18, 15
wave_array = [[0.0, 12], [1.0, 18], [1.5, 15]]
mywave = fonsim.wave.custom.Custom(wave_array, time_notation='absolute', kind=
↪'previous')
```

(continues on next page)

(continued from previous page)

```
# Read it out by calling the object
y = mywave(1.2)  # y = array(18.)
```

Parameters

- **wave_array** – indexable object, shape 2 x N or N x 2
- **time_notation** – 'absolute' or 'relative'
- **kind** – interpolation kind

fonsim.wave.custom.**isincreasing**(*arr*)

Helper function. Returns True if array strictly increasing, False otherwise.

16.3 fonsim.wave.wave module

Wave generator functionality.

Available wave functions:

- square
- sine
- triangular
- sawtooth

The input range is [0, 2*pi] and output range is [-1, 1]. These functions are static and thus can be placed outside of the class definition.

Other functionality:

- Function time_to_angle: conversion elapsed time -> angle
- Function wave_custom: for custom waves

2020, September 5

fonsim.wave.wave.**sawtooth**(*angle*)

fonsim.wave.wave.**sine**(*angle*)

fonsim.wave.wave.**square**(*angle*)

fonsim.wave.wave.**time_to_angle**(*time*, *frequency*, *phase=0*)

Convert an elapsed time to an angle. Designed to be used with the wave functions that take an anle as input.

Equation:

```
angle = ((time · frequency + phase/(2·pi)) % 1) · 2·pi
```

Parameters

- **time** – elapsed time, in s
- **frequency** – frequency, in Hz
- **phase** – phase offset, in radians

Returns
angle, in radians

fonsim.wave.wave.**triangle**(*angle*)

fonsim.wave.wave.**unity**(*angle*)

16.4 Module contents

2020, September 18

CHAPTER

SEVENTEEN

FONSIM.VISUAL PACKAGE

17.1 Submodules

17.2 fonsim.visual.plotting module

Some tools to make plotting simulation results less repetitive.

2020, September 5

fonsim.visual.plotting.**plot**(*axs*, *sim*, *label*, *unit*, *components*)

Easily plot terminal data of components

Parameters

- **axs** – matplotlib plotting axis
- **sim** – simulation object
- **label** – variable label, e.g. 'pressure'
- **unit** – unit, e.g. 'bar'
- **components** – iterable with components or (component, terminal) pairs

Returns

None

fonsim.visual.plotting.**plot_state**(*axs*, *sim*, *label*, *unit*, *components*)

Easily plot state data of components

Parameters

- **axs** – matplotlib plotting axis
- **sim** – simulation object
- **label** – variable label, e.g. 'pressure'
- **unit** – unit, e.g. 'bar'
- **components** – iterable with components

Returns

None

17.3 Module contents

CHAPTER

EIGHTEEN

FONSIM.CONSTANTS PACKAGE

18.1 Submodules

18.2 fonsim.constants.norm module

Some constants at norm conditions, such as atmospheric pressure Please refer to the source of this file to see the defined constants.

2020, July 21

18.3 fonsim.constants.physical module

Some physical constants and parameters, such as the gas constant Please refer to the source of this file to see the defined constants.

2020, July 21

18.4 Module contents

CHAPTER

NINETEEN

FONSIM.CONVERSION PACKAGE

19.1 Submodules

19.2 fonsim.conversion.indexmatch module

Get index of best-matching labels

2020, September 4

fonsim.conversion.indexmatch.**get_index_of_best_match**(*unit*, *candidates*)

Get index of best-matching labels

Note: all labels are converted to strings and lowercase

Parameters

- **unit** – base label to compare with
- **candidates** – list of labels to compare against and get index from

Returns

index of best match

fonsim.conversion.indexmatch.**similar**(*a*, *b*)

19.3 Module contents

Go to the *Introduction* page.

Note: This documentation is a work in progress. Much content has yet to be written and some of the already existing content has to be reorganized.

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

F

G

I

L

M

N

P

R

S

T

U

V

W