bio_rtd

Jure Sencar

Mar 30, 2020

CONTENTS

CHAPTER ONE

ABOUT

bio_rtd library is a python library for modeling residence time distributions (RTD) of integrated continuous biomanufacturing processes.

Functionality

RTD models are based on volumetric flow rate and concentration profile.

Multiple process fluid components can be simulated. One can specify e.g. which components bind to the column, but (as for now) one cannot specify component specific binding properties.

Implemented unit operations can be simulated one by one. This allows easier workflow and faster evaluation. However, feedback process controls cannot be build on top of the model.

For more details check *Features*.

CHAPTER TWO

CONTACT

For technical issues, bug reports and feature request use the issue tracker on Github.

If you are using the library in your projects please let us know. This way we know how much interest there is, what is the scope of usage, the needs, etc. This information influences the future development of the project.

E-mail: jure.sencar@boku.ac.at.

CHAPTER

THREE

REFERENCING THE LIBRARY

- Senčar, J., (2020) GitHub Repository, https://github.com/open-biotech/bio-rtd.

CHAPTER FOUR

ACKNOWLEDGEMENTS

This work was supported by:

- The Federal Ministry for Digital and Economic Affairs (bmwd), the Federal Ministry for Transport, Innovation and Technology (bmvit), the Styrian Business Promotion Agency SFG, the Standortagentur Tirol, Government of Lower Austria, and ZIT - Technology Agency of the City of Vienna through the COMET-Funding Program managed by the Austrian Research Promotion Agency FFG
- Baxalta Innovations GmbH (now part of Takeda)
- Bilfinger Industrietechnik Salzburg GmbH

4.1 Getting Started

4.1.1 Requirements

Python 3.7.+

Packages:

```
numpy
scipy
bokeh
xlrd
pandas
```

- `numpy` and `scipy` are basic requirements.
- `bokeh` and `xlrd` are needed for plotting and/or GUI.
- `pandas` is needed for reading data from Excel in one of the examples.

It is generally recommended to use python virtual environment or conda virtual environment.

4.1.2 Setup

Clone git repo:

```
git clone https://github.com/open-biotech/bio-rtd.git
```

Go to repo folder:

```
cd bio-rtd
```

Install requirements:

```
pip install -r requirements.txt
```

4.1.3 Running first example

Run an example:

```
python example/models/single_pcc.py
```

If you see:

```
ModuleNotFoundError: No module named 'bio_rtd'
```

then you need to add repo to PYTHONPATH (either in your IDE or in terminal):

PyCharm

```
Check "Add source root to PYTHONPATH" under
"Run/Debug configuration".
```

Terminal

```
export PYTHONPATH=${PYTHONPATH}:`pwd`
```

4.1.4 Running example with bokeh serve

Examples that end with `_gui.py` are python scripts for creating an interactive web application.

In background a *Bokeh Server* instance is created. The server connects UI elements with python script (see more at Building Bokeh Applications).

Terminal

```
bokeh serve --show example/models/integrated_mab_gui.py
```

`bokeh` can also run as a python script (good for debugging):

```
python `which bokeh` serve --show example/models/integrated_mab_gui.py
```

PyCharm

To run `python bokeh serve` with PyCharm, set the following *Run Configuration*:

```
Configuration: python
Script path: /path_to/bokeh
Parameters: serve --show /example/models/integrated_mab_gui.py
```

where `/path_to/bokeh` can be obtained by running `which bokeh` command in terminal in PyCharm.

Also make sure the repo is added to the PYTHONPATH as described in *Running first example*.

Flag `--show` is optional and runs the newly created instance in a new tab in a web browser. Only one server instance can be run (on the same port). If you try to run another `bokeh serve` command while one is running, you will see the following exception:

```
Cannot start Bokeh server, port 5006 is already in use
```

In that case find and close the existing running `bokeh serve` process.

4.2 User Guide

4.2.1 Introduction

Residence time distribution (RTD)

Let's simulate a protein pulse response measurement on a small flow-through column:

```python
import numpy as np
from bokeh.plotting import figure, show
from bio_rtd import peak_shapes, utils

# Time vector.
t = np.linspace(0, 10, 201)

# Generate noisy data.
y = peak_shapes.emg(t, 2, 0.3, 1.0)  # clean signal
y_noisy = y + (np.random.random(y.shape) - 0.5) * y.max() / 10

# Determine peak start, end and max position.
i_start, i_end = utils.vectors.true_start_and_end(y > 0.1 * y.max())
i_max = y.argmax()

# Plot.
p = figure(plot_width=690, plot_height=350, title="Measurements",
           x_axis_label="t [min]", y_axis_label="c [mg/mL]")
p.line(t, y_noisy, line_width=2, color='green', alpha=0.6,
       legend_label='c [mg/mL]')
for i in [i_max, i_start, i_end]:  # plot circles
    p.circle(t[i], y[i], size=15, fill_alpha=0,
             line_color="blue", line_width=2)
show(p)
```

Let's fit an exponentially modified gaussian distribution to the peak using the reference points in blue circles.

> Probability distribution classes, such as *`bio_rtd.pdf.ExpModGaussianFixedDispersion`* allow the probability distribution functions (pdf) to be dependent on process parameters and inlet flow rate.

```python
import numpy as np
from bokeh.plotting import figure, show
from bio_rtd import pdf, uo, peak_shapes, peak_fitting

t = np.linspace(0, 10, 201)

# Calc rt_mean, sigma and skew from peak points.
rt, sigma, skew = peak_fitting.calc_emg_parameters_from_peak_shape(
    t_peak_max=1.4, t_peak_start=0.6, t_peak_end=3.9,
    relative_threshold=0.1
)

# Define pdf.
pdf_emg = pdf.ExpModGaussianFixedDispersion(t, sigma ** 2 / rt, skew)
pdf_emg.update_pdf(rt_mean=rt)
p_emg = pdf_emg.get_p()

# Generate noisy data.
y = peak_shapes.emg(t, 2, 0.3, 1.0)   # clean signal
y_noisy = y + (np.random.random(y.shape) - 0.5) * y.max() / 10

# Plot.
p = figure(plot_width=690, plot_height=350,
           title="Probability Distribution",
           x_axis_label="t [min]", y_axis_label="c [mg/mL]")
p.line(t, y_noisy, line_width=2, color='green', alpha=0.6,
       legend_label='c [mg/mL] (data)')
p.line(t, p_emg, line_width=2, color='black', alpha=1,
       legend_label='p (pdf)')
show(p)
```

Let’s expand the example by introducing flow-through unit operation which uses pdf:

```python
import numpy as np
from bokeh.plotting import figure, show
from bio_rtd import pdf, uo, peak_shapes, peak_fitting

t = np.linspace(0, 10, 201)

# Concentration with spike at t = 0
c_in = np.zeros([1, t.size])
c_in[0][0] = 1 / t[1]

# Flow rate
f = np.ones_like(t) * 3.5

# Calc rt_mean, sigma and skew from peak points
rt, sigma, skew = peak_fitting.calc_emg_parameters_from_peak_shape(
    t_peak_max=1.4, t_peak_start=0.6, t_peak_end=3.9,
    relative_threshold=0.1
)

# Define unit operation
ft_uo = uo.fc_uo.FlowThrough(
    t=t, uo_id="ft_example",
    pdf=pdf.ExpModGaussianFixedDispersion(t, sigma ** 2 / rt, skew),
)
```

(continues on next page)

(continued from previous page)

```
ft_uo.v_void = rt * f[0]  # set void volume as rt * flow rate

# Simulate.
f_out, c_out = ft_uo.evaluate(f, c_in)

# Define noisy data.
y = peak_shapes.emg(t, 2, 0.3, 1.0)  # clean signal
y_noisy = y + (np.random.random(y.shape) - 0.5) * y.max() / 10

# Plot.
p = figure(plot_width=690, plot_height=350,
           title="Unit Operation - Pulse Response",
           x_axis_label="t [min]", y_axis_label="c [mg/mL]")
p.line(t, y_noisy, line_width=2, color='green', alpha=0.6,
       legend_label='c [mg/mL] (data)')
p.line(t, c_out[0], line_width=2, color='black', alpha=1,
       legend_label='c [mg/mL] (model)')
show(p)
```

Simulating breakthrough profile with the same unit operation:

```
import numpy as np
from bokeh.plotting import figure, show
from bio_rtd import pdf, uo

# Define inlet profiles.
t = np.linspace(0, 10, 201)  # time
c_in = np.ones([1, t.size])  # concentration (constant)
f = np.ones_like(t) * 3.5  # flow rate

# Define unit operation.
ft_uo = uo.fc_uo.FlowThrough(
    t=t, uo_id="ft_example",
    pdf=pdf.ExpModGaussianFixedDispersion(t, 0.3 ** 2 / 2, 1.0))
ft_uo.v_void = 2 * f[0]  # set void volume (rt * flow rate)

# Simulation.
f_out, c_out = ft_uo.evaluate(f, c_in)

# Plot.
p = figure(plot_width=690, plot_height=350,
           title="Unit Operation - Breakthrough",
           x_axis_label="t [min]", y_axis_label="c [mg/mL]")
p.line(t, c_out[0], line_width=2, color='black',
       legend_label='c [mg/mL]')
show(p)
```

Simulating breakthrough profile with *bio_rtd.core.RtdModel* with inlet and two unit operations.

```
import numpy as np
from bokeh.plotting import figure, show
from bio_rtd import pdf, uo
from bio_rtd.core import RtdModel
from bio_rtd.inlet import ConstantInlet

t = np.linspace(0, 10, 201)  # time
```

(continues on next page)

(continued from previous page)

```
# Define inlet
inlet = ConstantInlet(t, inlet_id="sample_inlet",
                      f=3.5, c=np.array([1.0]),
                      species_list=['protein [mg/mL]'])

# Define unit operation.
ft_uo_1 = uo.fc_uo.FlowThrough(
    t=t, uo_id="ft_example",
    pdf=pdf.ExpModGaussianFixedDispersion(t, 0.3 ** 2 / 2, 1.0))
ft_uo_1.rt_target = 2.0

# Define another unit operation.
ft_uo_2 = uo.fc_uo.FlowThrough(t=t, uo_id="ft_example_2",
                               pdf=pdf.TanksInSeries(t, 3))
ft_uo_2.rt_target = 1.2

# Create an RtdModel.
rtd_model = RtdModel(inlet, dsp_uo_chain=[ft_uo_1, ft_uo_2])

# Run simulation.
rtd_model.recalculate()

# Plot.
p = figure(plot_width=690, plot_height=350,
           title="Model with 2 unit operations - Breakthrough",
           x_axis_label="t [min]", y_axis_label="c [mg/mL]")
p.line(t, inlet.get_result()[1][0], line_width=2, color='black',
       legend_label='inlet')
p.line(t, ft_uo_1.get_result()[1][0], line_width=2, color='green',
       legend_label='outlet of uo_1')
p.line(t, ft_uo_2.get_result()[1][0], line_width=2, color='blue',
       legend_label='outlet of uo_2')
p.legend.location = "bottom_right"
show(p)
```

See *Examples* section for more examples.

See *Templates* section on how to set up specific unit operations.

See *Features* section for brief overview of individual unit operations.

See *API Reference* for detailed info about parameters, attributes and methods for each unit operation.

Creating custom rtd model

We recommend checking *Models* and making a local copy of one that most closely resembles you needs and modify it accordingly. To define new instances of unit operations, use the parameter and attribute list from *Templates*.

Also check *Coding* section in order to better understand the coding style in *bio_rtd*.

4.2.2 Features

Inlet profiles

Available inlet profiles

- Constant flow rate, constant concentration `bio_rtd.inlet.ConstantInlet`
- Constant flow rate, box-shaped concentration profile `bio_rtd.inlet.IntervalInlet`
- Custom flow rate, custom concentration profile `bio_rtd.inlet.CustomInlet`

Unit operations

Unit operations are split in following groups:

- Fully-continuous `bio_rtd.uo.fc_uo` (accept and provide constant flow rate)
- Semi-continuous `bio_rtd.uo.sc_uo` (accept constant and provide periodic flow rate)
- Surge tank `bio_rtd.uo.surge_tank` (accept periodic and constant flow rate)
- Special `bio_rtd.uo.special_uo` (the ones that do not fit in categories above)

All unit operations can be instructed to discard parts of inlet or outlet process fluid stream in oder to optimize the start-up phase.

For common attributes among unit operations check `bio_rtd.core.UnitOperation` class. For complete parameter set of individual unit operation, check its API by clicking the class name.

Here are listed key features of unit operations:

`bio_rtd.uo.fc_uo.Dilution`

- Instant dilution of the process fluid stream.

`bio_rtd.uo.fc_uo.Concentration`

- Instant concentration of the process fluid stream.
- One can specify retained species and losses during concentration step.

`bio_rtd.uo.fc_uo.BufferExchange`

- Instant inline buffer exchange.
- One can specify retained species, losses and efficiency.

`bio_rtd.uo.fc_uo.FlowThrough`

- Propagation of the process fluid stream through a fixed unit operation (most common use case).
- A probability distribution function is specified to describe the propagation dynamics.
- Offers setting equilibration and wash buffer composition (for more parameters check the class link).

`bio_rtd.uo.fc_uo.FlowThroughWithSwitching`

- Extension of the *bio_rtd.uo.fc_uo.FlowThrough*.
- Allows switching unit operations during run (e.g. for alternating flow-through chromatography).

bio_rtd.uo.sc_uo.ACC

- Alternating column chromatography (without recycling of the overloaded material).
- Describing binding dynamics via `bio_rtd.core.BreakthroughProfile`.
- Option to specify load duration based on breakthrough material.
- Material in elution peak is homogenized. Various peak cut methods are available.

bio_rtd.uo.sc_uo.PCC

- Periodic counter-current chromatography.
- Extension of *bio_rtd.uo.sc_uo.ACC*.
- Option to recycle breakthrough material during load and/or wash step.

bio_rtd.uo.sc_uo.PCCWithWashDesorption

- Extension of *bio_rtd.uo.sc_uo.PCC*.
- Option to simulate desorption of captured material during wash step.

bio_rtd.uo.surge_tank.CSTR

- Ideal CSTR.
- Offers specifying initial fill level.
- Size can be determined based on specified 'safety margin', e.g. 10 %

bio_rtd.uo.special_uo.ComboUO

- Unit operation that combines several unit operations and presents them as one.

Some unit operations can be described with a set of simpler unit operation, but we might want to have them appear (e.g. in plots) as one. Typical use-case would be describing filtration or diafiltration step with a combination of Concentration, Dilution and/or FlowThrough unit operations. In such case, one can use ComboUO as a container.

Probability distribution functions

Available pdf peak shapes:

- Gaussian: `bio_rtd.peak_shape.gaussian()`
- Exponentially modified Gaussian: `bio_rtd.peak_shape.emg()`
- Skewed normal: `bio_rtd.peak_shape.skew_normal()`
- N tanks in series (N = 1 for exponential decay): `bio_rtd.peak_shape.tanks_in_series()`

Available PDF classes (wrappers around pdf peak shapes):

- *bio_rtd.pdf.GaussianFixedDispersion*
- *bio_rtd.pdf.GaussianFixedRelativeWidth*
- *bio_rtd.pdf.ExpModGaussianFixedDispersion*
- *bio_rtd.pdf.ExpModGaussianFixedRelativeWidth*
- *bio_rtd.pdf.TanksInSeries*

Fixed dispersion:

sigma = (void_volume * dispersion_index) ** 0.5

Fixed relative width:

sigma = void_volume * relative_width

Logging

Custom loggers are implemented in order to provide control over log messages and storing intermediate data.

See *RtdLogger* API for more info.

4.2.3 Coding

General guidelines

We recommend copying one of the *Models* that most closely resembles your needs and modifying it accordingly.

Use *Templates* to create instances of individual unit operations.

If you need to define new unit operations classes (or other elements of the library), then make sure they extend proper base classes.

Conventions

We follow `yx` convention for shaping numpy arrays. In our case `x` is typically time axis (`t`) and `y` corresponds to process fluid species.

Simulation time vector (`t`) is a 1D `np.ndarray`. It should start with `0` and have a fixed step size. The same time vector should be used across the model (for inlet and all the unit operations).

Vector for flow rate `f` is also a 1D `np.ndarray`.

Array for concentration profile is 2D `np.ndarray` with shape `(n_species, n_time_steps)`. In case of single specie, the shape is `(1, n_time_steps)` and not `(n_time_steps,)`.

Single underscore prefix _ annotates private functions and variables which should be only used inside the class or function.

Variable names

- `t` - simulation time vector
- `dt` - time step size
- `i` - time step index on time vector (`t[i] == dt * i`)
- `f` - process fluid flow rate
- `rt` - residence time
- `rt_mean` - mean residence time (= flow-through time)
- `rt_target` - target `rt_mean` at steady-state; typically used to determine the size of unit operations
- `v` - volume
- `v_void` - void volume; usually effective void volume, thus excluding hold-up zones (`rt_mean` = `v_void` / `f`)
- `v_init` - initial volume of fluid in unit operation (e.g. if surge tank starts with 50 % pre-fill, the `v_init ``= 0.5 * ``v_void`; 0.5 could also be specified as `v_init_ratio` = 0.5)

- `m` - mass
- `uo` - unit operation
- `fc_uo` - fully-continuous unit operation (accepts and provides constant flow rate)
- `sc_uo` - semi-continuous unit operation (accepts constant flow rate and provides irregular flow rate)
- `surge_tank` - surge tank (accepts irregular or constant flow rates and provides constant* flow rate)
- `pdf` - probability distribution function; `sum(pdf * time_step) == 1`
- `p` - probability distribution vector; `p = pdf(t)`

All names of the time dependent vectors or arrays are thus starting with `f_`, `c_`, `m_` or `p_`.

Constant flow rate profile can be clipped at the beginning or at the end, resulting in a box-shaped profile.

4.2.4 Development

Unit tests

Each file in `rtd_lib` has a corresponding test file with prefix `test_` placed in `rtd_lib_test` folder.

To run all tests and asses code coverage (the share of code tested) using the `coverage` package, run the following command in terminal:

```
coverage run --source=./bio_rtd -m unittest discover bio_rtd_test; coverage report
```

To see the detailed coverage analysis (e.g. to discover non-covered lines), run:

```
coverage html
```

and open `htmlcov/index.html` in web browser.

Running tests without code coverage:

```
python -m unittest discover bio_rtd_test
```

If you create a pull request, please add appropriate tests, make sure all tests succeed and keep complete (100 %) code coverage. If needed, also update the documentation.

Documentation

Dependencies (`pip` packages):

```
sphinx
sphinx_autodoc_typehints
sphinx_rtd_theme
```

To generate the documentation from script, run:

```
make html
```

and open `docs/build/html/index.html`.

Pull requests

To contribute to the library, please create a pull request on GitHub.

Checklist before making a pull request:

- all unit tests need to succeed
- ensure 100 % code coverage with unit tests
- update docstrings and documentation

4.3 Examples

Examples are split into two parts:

- *Models* contain runnable examples.
- *Templates* provide parameter and attribute lists for creating unit operation instances.

4.3.1 Models

Single unit operation - UF DF

```
"""Modeling UFDF

We will describe uf-df with a combination of three unit operations:
 1. `Concentration`
 2. `BufferExchange`
 3. `FlowThrough`
    (for describing residence time distribution of the product)

All three unit operations will be joined into single `ComboUO`.

"""

import numpy as np
from bokeh.io import show
from bokeh.layouts import column
from bokeh.models import Range1d, LinearAxis
from bokeh.plotting import figure

from bio_rtd import pdf
from bio_rtd.uo import fc_uo, special_uo

# Time step (`dt`) simulation time (`t`).
t = np.linspace(0, 100, 1000)
dt = t[1]

# ## CREATING INSTANCE ##

# Concentration.
conc = fc_uo.Concentration(
    t,
    flow_reduction=10,
    uo_id="concentration_sub_step"
```

(continues on next page)

(continued from previous page)

```
)
conc.relative_losses = 0.10  # 10 % losses
# BufferExchange
buffer_exchange = fc_uo.BufferExchange(
    t,
    exchange_ratio=0.95,
    uo_id="buffer_exchange_sub_step"
)
# FlowThrough
flow_through = fc_uo.FlowThrough(
    t,
    # Peak shape description (result of a pulse injection experiment)
    pdf=pdf.GaussianFixedRelativeWidth(t, relative_sigma=0.2),
    uo_id="rtd_sub_step"
)
# Peak position ( = first momentum of a peak at pulse injection experiment)
flow_through.rt_target = 5  # min

# UFDF
uf_df = special_uo.ComboUO(t,
                           sub_uo_list=[conc, buffer_exchange, flow_through],
                           uo_id="uf_df",
                           gui_title="UfDf step")

# ## SIMULATION ##

# Inlet flow rate and concentration profile with 3 species.
f = np.ones_like(t) * 200.0  # mL/min
c = np.zeros([3, t.size])
# We choose protein as a 1st specie and set inlet concentration to 14 mg/ml.
c[0] = 14  # mg/ml
# We 'label' a part of the product and treat is as separate specie.
c[1, 300:600] = 14
# Last component represent salt that we try to remove from the process fluid.
c[2] = 1000  # mM

# Update `ud_df` so it does not retain salt.
conc.non_retained_species = [2]
buffer_exchange.non_retained_species = [2]

# Evaluate ( = run simulation).
f_out, c_out = uf_df.evaluate(f, c)

# Plot results.
p1 = figure(plot_width=690, plot_height=350, title="UfDf",
            x_axis_label="t [min]", y_axis_label="c [mg/mL]")
# Add new axis for flow rate to the right.
p1.extra_y_ranges = {'f': Range1d(0, max(f.max(), f_out.max()) * 1.1)}
p1.add_layout(LinearAxis(y_range_name='f'), 'right')
p1.yaxis[1].axis_label = "f [mL/min]"
# Protein conc and flow rate.
p1.line(t, c[0],
        line_width=2, color='green',
        legend_label='c_in, protein [mg/mL]')
p1.line(t, c[1],
        line_width=2, color='red',
        legend_label='c_in, protein, labeled [mg/mL]')
```

(continues on next page)

(continued from previous page)

```
p1.line(t, f,
        y_range_name='f', line_width=1, color='black',
        legend_label='f_in')
# Flow.
p1.line(t, f_out,
        y_range_name='f', line_width=1, color='black',
        line_dash='dashed', legend_label='f_out')
p1.line(t, c_out[0],
        line_width=2, color='green', line_dash='dashed',
        legend_label='c_out, protein [mg/mL]')
p1.line(t, c_out[1],
        line_width=2, color='red', line_dash='dashed',
        legend_label='c_out, protein, labeled [mg/mL]')
p1.y_range.start = 0
p1.y_range.end = max(c[0:1].max(), c_out[0:1].max()) * 1.25
p1.legend.location = "center_right"

# Plot results.
p2 = figure(plot_width=690, plot_height=350, title="UfDf",
            x_axis_label="t [min]", y_axis_label="c [mM]")
# Add new axis for flow rate to the right.
p2.extra_y_ranges = {'f': Range1d(0, max(f.max(), f_out.max()) * 1.1)}
p2.add_layout(LinearAxis(y_range_name='f'), 'right')
p2.yaxis[1].axis_label = "f [mL/min]"
# Salt conc and flow rate.
p2.line(t, c[2], line_width=2, color='navy',
        legend_label='c_in, salt [mM]')
p2.line(t, f,
        y_range_name='f', line_width=1, color='black',
        legend_label='f_in')
p2.line(t, c_out[2], line_width=2, color='navy', line_dash='dashed',
        legend_label='c_out, salt [mM]')
p2.line(t, f_out,
        y_range_name='f', line_width=1, color='black',
        line_dash='dashed', legend_label='f_out')
p2.legend.location = "center_right"

# Plot both plots.
show(column(p1, p2))
```

Single unit operation - PCC

```
"""Example use of PCC unit operation.

`pcc` instance is taken from (and generated by) `pcc_template`.

In this example we simulate propagation of constant inlet flow rate
and concentration profiles throughout the pcc. Afterwards we create
a bunch of plots.

"""

import numpy as np

from bokeh.io import show
```

(continues on next page)

(continued from previous page)

```
from bokeh.layouts import gridplot
from bokeh.models import LinearAxis, Range1d
from bokeh.plotting import figure

from bio_rtd.logger import DataStoringLogger
from examples.templates.sc_uo.pcc_template import t, dt, pcc

"""Basic use."""
# Define inlet flow rate. Same shape at `t`.
f = np.ones_like(t) * 3.5
# Inlet concentration profile. Shape = (n_components, t.size).
c = np.ones([1, t.size]) * 4
# Set logger that stored data
pcc.log = DataStoringLogger()
# Simulate.
f_out, c_out = pcc.evaluate(f, c)

"""Plot inlet and outlet profiles."""
p1 = figure(plot_width=695, plot_height=350, title="PCC",
            x_axis_label="t [min]", y_axis_label="c [mg/mL]")
# Add new axis for flow rate to the right.
p1.extra_y_ranges = {'f': Range1d(0, max(f.max(), f_out.max()) * 1.1)}
p1.add_layout(LinearAxis(y_range_name='f'), 'right')
p1.yaxis[1].axis_label = "f [mL/min]"
# Set scale to CV.
x = np.cumsum(f) / pcc.cv * t[1]
p1.line(x, f, y_range_name='f',
        line_width=2, color='red', line_dash='dashed', legend_label='f_in')
p1.line(x, c[0],
        line_width=2, color='navy', line_dash='dashed', legend_label='c_in')
p1.line(x, f_out, y_range_name='f',
        line_width=2, color='red', legend_label='f_out')
p1.line(x, c_out[0],
        line_width=2, color='navy', legend_label='c_out')
# show(p1)

"""Print data from log."""
# get data tree from log
data_tree = pcc.log.get_data_tree(pcc.uo_id)
# print data
print(data_tree["t_cycle_optimization_loop_iter"])

"""Plot elution step profiles from log.

Notes
-----
Model stores parameters in log if
    `pcc.log.log_data == True` and
    `pcc.log.log_level_data <= pcc.log.INFO`.
Model stores parameters and time profiles in log if:
    `pcc.log.log_data == True` and
    `pcc.log.log_level_data <= pcc.log.DEBUG`.

"""
# Elution peak shape.
y = data_tree['p_elution_peak']
p2 = figure(plot_width=695, plot_height=350, title="Elution peak shape",
```

(continues on next page)

(continued from previous page)

```
            x_axis_label="t [min]", y_axis_label="p []")
p2.line(t[:y.size], y, line_width=2, legend_label='elution peak shape')
# Elution peak collection interval.
y_mask = data_tree['elution_peak_interval']
p2.line(t[:y_mask.size], y_mask * y.max() * 0.1, line_width=2, color="black",
        alpha=0.5, legend_label='elution peak collection range')
# Duration of elution step.
p2.line(data_tree['elution_t'] * np.ones(2), np.ones(2) * y.max(),
        line_width=2, line_dash='dotdash', line_color="black", line_alpha=0.7,
        legend_label='elution step end')
# show(p2)

# Plot breakthrough profile for 2 cycles worth of inlet material.
p3 = figure(plot_width=695, plot_height=350, title='Binding capacity',
            x_axis_label="t [min]", y_axis_label="c / c_load")
i_cycle = int(round(data_tree["cycle_t"] / dt))
p3.line(t[:i_cycle * 2], c[0, :i_cycle * 2],
        line_width=2, color="blue", legend_label='load material')
p3.line(t[:i_cycle * 2], pcc.load_bt.calc_c_bound(f, c)[0, :i_cycle * 2],
        line_width=2, color="black", legend_label='bound material')
p3.line((t[i_cycle], t[i_cycle]), (0, c.max()), line_width=2,
        line_dash='dotdash', color="black", alpha=0.7, legend_label='cycle')
p3.line((t[0], t[0]), (0, c.max()), line_width=2, line_dash='dotdash',
        color="black", alpha=0.7, legend_label='cycle')
# show(p3)

# Plot profiles for each cycle of load.
p4_group = []
for i, cycle in enumerate(data_tree['cycles']):
    # When column starts seeing recycled material.
    i_s = cycle['i_cycle_load_start']
    # When column starts seeing load material (beginning of load step).
    i_ss = cycle['i_cycle_load_step_start']
    # When the load step ends.
    i_e = cycle['i_cycle_load_end']

    y_max = 8.5  # Just so all plots have same y range.
    pc = figure(plot_width=695, plot_height=350, title='Cycle ' + str(i),
                x_axis_label="t [min]", y_axis_label="c [mg/mL]")
    # Profiles.
    pc.line(t[i_s:i_e], cycle['f_load'],
            line_width=2, color="blue", legend_label='load flow rate')
    pc.line(t[i_s:i_e], cycle['c_load'][0],
            line_width=2, color="black", alpha=0.5, legend_label='load conc')
    pc.line(t[i_s:i_e], cycle['c_bound'][0],
            line_width=2, color="green", alpha=0.5, line_dash='solid',
            legend_label='bound part of load conc')
    pc.line(t[i_s:i_e], cycle['c_unbound'][0],
            line_width=2, color="green", alpha=0.5, line_dash='dashed',
            legend_label='unbound part of load conc')
    pc.line(t[i_s:i_e], cycle['c_out_load'][0],
            line_width=2, color="red", alpha=1, legend_label='overload conc')
    # Vertical markers.
    pc.line(t[i_s] * np.ones(2), (0, y_max),
            line_width=2, line_dash='dotted', color="black", alpha=0.7,
            legend_label='previous load start')
    if i > 1:
```

(continues on next page)

(continued from previous page)

```
        i_wash_recycle = data_tree["cycles"][i - 1]["c_wash_recycle"].shape[1]
        pc.line(t[i_s + i_wash_recycle] * np.ones(2), (0, y_max),
                line_width=2, line_dash='dashed', color="black", alpha=0.4,
                legend_label='wash recycle end')
    pc.line(t[i_ss] * np.ones(2), (0, y_max),
            line_width=2, line_dash='dotdash', color="black", alpha=0.7,
            legend_label='load start')
    pc.line(t[i_e] * np.ones(2), (0, y_max),
            line_width=2, line_dash='dashdot', color="black", alpha=0.7,
            legend_label='load end')
    t_wash = data_tree["wash_t"]
    pc.line(t[i_e] * np.ones(2) + t_wash, (0, y_max),
            line_width=2, line_dash='dashed', color="black", alpha=0.7,
            legend_label='wash end')
    # ## WASH ##
    cvd = cycle['c_wash_desorbed']
    if cvd is not None and cvd.size > 0:
        pc.line(
            t[i_e] + t[:cvd.shape[1]],
            cvd[0],
            line_width=2, color="blue", line_dash='dotdash',
            legend_label='wash desorbed'
        )
    cow = cycle['c_out_wash']
    if cow is not None and cow.size > 0:
        pc.line(
            t[i_e] + t[:cow.shape[1]],
            cow[0],
            line_width=2, color="blue",
            legend_label='wash outlet'
        )
    pc.legend.location = "top_center"
    if i > 0:
        pc.legend.visible = False
    p4_group.append([pc])
# show(gridplot(p4_group))

show(gridplot([[p1], [p2], [p3], *p4_group]))
```

Integrated mAB model

```
"""Example `RtdModel` for mAB downstream process

In this example we define unit operations for a hypothetical mAB
process. We use only single species in the model in order to narrow
the scope of the sample.

Examples
--------
Additional species can be added to the model by following the steps:
>>> # Add new species to the inlet concentration vector.
>>> rtd_model.inlet.species_list = ['mAB', 'new_sp_1', 'new_sp_2']
>>> rtd_model.inlet.c = np.array([2.4, 20, 30])
>>> # Update unit operations.
>>> # E.g. PCC should not bind new species.
```

(continues on next page)

(continued from previous page)

```
>>> pcc_pro_a.non_binding_species = [1, 2]
>>> # E.g. UFDF should not retain not bind new species.
>>> conc.non_retained_species = [1, 2]
>>> buffer_exchange.non_retained_species = [1, 2]
>>> # Update parameters, like elution buffer composition, if needed.

Notes
-----
Process parameters
    Process parameters were chosen in a way that the workflow and the
    results can be easily interpreted by the user. This means that the
    process parameters might not always represent an actual mAB
    production process.

    See templates of individual unit operations for more information
    about which process parameters can be defined.

    See docstrings of individual unit operations for more details about
    individual process parameters.

Units
    RtdModel does not depend on any specific set of units.
    They just need to be consistent across the model. E.g.:
    Unit `base`:
      * time [min]
      * volume [mL]
      * weight [mg]
    Independent species can have different units, e.g.:
      * protein [IU]
      * acetate [mmol]
      * salt [mg]
    All derived units need to be a combination of 'basic' units.
      * flow rate [mL/min]
      * column height [cm]
      * column binding capacity (protein) [IU/mL]
      * etc.
    In presented model, [mL], [mg] and [min] were used as a 'base'.

"""

import numpy as np
from bokeh.io import show
from bokeh.layouts import column
from bokeh.models import Range1d, LinearAxis
from bokeh.plotting import figure

from bio_rtd import pdf, peak_fitting, inlet
from bio_rtd.chromatography import bt_load
from bio_rtd.uo import sc_uo, fc_uo, surge_tank, special_uo
from bio_rtd.core import RtdModel
from bio_rtd.logger import DataStoringLogger


"""Simulation time."""

dt = 0.5  # min
t = np.arange(0, 24.1 * 60, dt)
```

(continues on next page)

(continued from previous page)

```python
"""DSP train."""

# Cell removal.
ft_cell_removal = fc_uo.FlowThroughWithSwitching(
    t, pdf=pdf.GaussianFixedRelativeWidth(t, relative_sigma=0.1),
    uo_id="cell_removal_ft",
    gui_title="Cell removal"
)
ft_cell_removal.v_void = 200  # mL
ft_cell_removal.v_cycle = 20000  # mL; switch filter unit after 20 L

# ProteinA PCC.
# Describe binding dynamics during load.
load_bt = bt_load.ConstantPatternSolution(dt, dbc_100=50, k=0.12)
# Describe elution peak with EMG pdf.
ep_rt_mean, sigma, skew = peak_fitting.calc_emg_parameters_from_peak_shape(
    t_peak_start=9,  # experimental data - peak start
    t_peak_max=16,  # experimental data - peak position (max signal)
    t_peak_end=27.5,  # experimental data - peak end
    relative_threshold=0.05,  # 5 % threshold
)
# Assuming the above experiments were done with 10 mL column.
ep_rt_mean_cv = ep_rt_mean / 10
# Elution peak shape pdf.
peak_shape_pdf = pdf.ExpModGaussianFixedRelativeWidth(
    t, sigma_relative=sigma / ep_rt_mean, skew=skew
)
# Load recycle pdf. We try to describe the propagation of unbound and/or
# desorbed material throughout the column during load (and wash) step.
load_recycle_pdf = pdf.GaussianFixedDispersion(t, dispersion_index=2 * 2 / 30)
pcc_pro_a = sc_uo.PCC(
    t,
    load_bt=load_bt,
    peak_shape_pdf=peak_shape_pdf,
    load_recycle_pdf=load_recycle_pdf,
    # Porosity of the column for protein.
    column_porosity_retentate=0.64,
    uo_id="pro_a_pcc",
    gui_title="ProteinA PCC",
)
pcc_pro_a.cv = 100  # mL
# Equilibration step.
pcc_pro_a.equilibration_cv = 1.5
# Equilibration flow rate is same as load flow rate.
pcc_pro_a.equilibration_f_rel = 1
# Load until 70 % breakthrough.
pcc_pro_a.load_c_end_relative_ss = 0.7
# Automatically prolong first cycle in order to achieve steady-state faster.
pcc_pro_a.load_extend_first_cycle = True
# Define wash step. There is no desorption during wash step in this example.
pcc_pro_a.wash_cv = 5
pcc_pro_a.wash_recycle = True
pcc_pro_a.wash_recycle_duration_cv = 2
# Elution step.
pcc_pro_a.elution_cv = 3
```

(continues on next page)

(continued from previous page)

```
# 1st momentum of elution peak from data from above.
pcc_pro_a.elution_peak_position_cv = ep_rt_mean_cv
pcc_pro_a.elution_peak_cut_start_c_rel_to_peak_max = 0.05
pcc_pro_a.elution_peak_cut_end_c_rel_to_peak_max = 0.05
# Regeneration step.
pcc_pro_a.regeneration_cv = 1.5

# Surge tank - CSTR.
st_cstr = surge_tank.CSTR(t, uo_id="st_cstr", gui_title="Surge Tank - CSTR")
st_cstr.v_min_ratio = 0.1  # 10 % fill level remains after discharge
st_cstr.starts_empty = True

# Virus inactivation - FlowThrough, no switching
ft_vi = fc_uo.FlowThrough(
    t,
    pdf=pdf.GaussianFixedDispersion(t, dispersion_index=20 ** 2 / 100),
    uo_id="vi_ft", gui_title="Virus Inactivation - flow-through column")
ft_vi.rt_target = 68  # min

# AEX polishing step, FlowThroughWithSwitching.
ft_aex = fc_uo.FlowThroughWithSwitching(
    t,
    pdf=pdf.GaussianFixedDispersion(t, dispersion_index=2 ** 2 / 8),
    uo_id="aex_ft", gui_title="Polishing, AEX, flow-through")
ft_aex.v_void = 10  # mL
ft_aex.v_cycle = 50 * 10  # mL; switch column unit after 20 L

# UFDF
# Concentration.
conc = fc_uo.Concentration(t, flow_reduction=40, uo_id="uf_df_conc")
# BufferExchange
buffer_exchange = fc_uo.BufferExchange(t, exchange_ratio=0.995,
                                       uo_id="uf_df_buffer_exchange")
# FlowThrough
flow_through = fc_uo.FlowThrough(t, pdf=pdf.TanksInSeries(t, n_tanks=3),
                                 uo_id="uf_df_rtd")
flow_through.v_void = 0.5 * 3  # mL
uf_df = special_uo.ComboUO(
    t, sub_uo_list=[conc, buffer_exchange, flow_through],
    uo_id="uf_df", gui_title="UFDF")

# DSP train
dsp_train = [ft_cell_removal, pcc_pro_a, st_cstr, ft_vi, ft_aex, uf_df]


"""Inlet."""

const_inlet = inlet.ConstantInlet(t,
                                  f=1000 / 60,  # mL/min (= 1 L / h)
                                  c=np.array([2.4]),  # mg/mL
                                  species_list=["mAB [mg/mL]"],
                                  inlet_id="constant_inlet",
                                  gui_title="Constant inlet")


"""`RtdModel`."""
```

(continues on next page)

(continued from previous page)

```
rtd_model = RtdModel(inlet=const_inlet, dsp_uo_chain=dsp_train,
                     logger=DataStoringLogger(),
                     title="Sample model for mAB production process")

if __name__ != "examples.models.integrated_mab":
    # Prevent log from printing warning when not running script directly.
    rtd_model.log.log_level = rtd_model.log.ERROR

"""Running simulation."""

rtd_model.recalculate()


"""Display info about the losses during PCC step from log.

Notes
-----
Various additional process data and profiles are stored within the log.
`examples/models/single_pcc.py` contains an example of plotting
intermediate data, such as concentration profiles during the wash step.

"""


def print_pcc_info():
    print(f"\nDisplaying data about load material distribution during the pcc:")
    print(f"--" * 32)
    pcc_data = rtd_model.log.get_data_tree(pcc_pro_a.uo_id)

    for i, cycle_data in enumerate(pcc_data["cycles"]):
        print(f"Cycle {i + 1}:")
        m_load = cycle_data['m_load'][0]
        print(f"Loaded amount:"
              f" {m_load:.0f} mg")
        print(f"Captured protein:"
              f" {cycle_data['m_bound'][0] / m_load * 100:.0f} %")
        print(f"Load recycled during load step:"
              f" {cycle_data['m_load_recycle'][0] / m_load * 100:.0f} %")
        # Because wash gets recycled on 2nd column we pull data from next cycle.
        m_wash = pcc_data["cycles"][i + 1]['m_wash_recycle'][0] \
            if i + 1 < len(pcc_data["cycles"]) else 0
        print(f"Load recycled during wash step:"
              f" {m_wash / m_load * 100:.0f} %")
        m_elution = cycle_data['m_elution'][0]
        m_elution_peak = cycle_data['m_elution_peak_cut_loss'][0]
        print(f"Losses due to peak cut:"
              f" {m_elution_peak / m_elution * 100:.0f} %")
        if i == len(pcc_data["cycles"]) - 1:
            print(f"--" * 20)
        else:
            print(f"")


"""Plot flow rate and concentration profile."""


def plot_profiles():
```

(continues on next page)

(continued from previous page)

```
    plt_group = []
    for i, uo in enumerate([const_inlet, *dsp_train]):
        f, c = uo.get_result()  # get profiles
        # Prepare figure.
        plt = figure(plot_width=690, plot_height=300, title=uo.gui_title,
                     tools="crosshair,reset,save,wheel_zoom,box_zoom,hover",
                     x_axis_label="t [h]", y_axis_label="c [mg/mL]")
        plt.extra_y_ranges = {'f': Range1d(0, f.max() * 1.1)}
        plt.add_layout(LinearAxis(y_range_name='f'), 'right')
        plt.yaxis[1].axis_label = "f [mL/min]"
        plt.y_range.start = 0
        plt.y_range.end = c.max() * 1.25
        # Plot profiles.
        plt.line(t / 60, c[0], line_width=2, color='navy',
                 legend_label='mAB conc')
        plt.line(t / 60, f, y_range_name='f', color='green',
                 legend_label='flow rate')
        # Add plot to plot list.
        plt.legend.location = "bottom_right"
        plt_group.append(plt)
    # Show plots.
    show(column(*plt_group))


if __name__ == "__main__":
    plot_profiles()
    print_pcc_info()

if __name__ != "examples.models.integrated_mab":
    # Just plot, unless imported as module.
    plot_profiles()
```

Integrated mAB GUI model

```
"""Example with graphical user interface build on top of RtdModel

`RtdModel` instance is taken from `integrated_mab.py`.

`BokehServerGui` is implementation of `UserInterface` abstract class.
For more information see docstrings of both classes.

"""

import numpy as np

from bokeh.client import push_session
from bokeh.io import curdoc

from bio_rtd import adj_par
from examples.models.integrated_mab import rtd_model

from examples.models.util.gui_bokeh import BokehServerGui

"""Exposing process parameter for manipulation via GUI."""
```

(continues on next page)

(continued from previous page)

```
# Inlet
rtd_model.inlet.adj_par_list = [
    adj_par.AdjParSlider(
        var="c[0]",
        v_range=(0.2, 10, 0.2),
        par_name='mAB concentration [mg/mL]'),
    adj_par.AdjParSlider(
        var="f",
        v_range=(0.1, 2, 0.1),
        scale_factor=1000 / 60,
        par_name='flow rate [L/h]'),
]
# PCC column size
rtd_model.dsp_uo_chain[1].adj_par_list = [
    adj_par.AdjParSlider(
        var="cv",
        v_range=(50, 500, 50),
        par_name='column size [mL]'),
]
# CSTR size
rtd_model.dsp_uo_chain[2].adj_par_list = [
    adj_par.AdjParBoolean(
        var="starts_empty",
        par_name='starts empty'),
    adj_par.AdjParSlider(
        var="v_void",
        v_range=(10, 200, 10),
        par_name='void volume [mL]'),
]
# Set CSTR value definition to absolute.
rtd_model.dsp_uo_chain[2].v_void = 50
rtd_model.dsp_uo_chain[2].v_min_rel = -1
# AEX column size
rtd_model.dsp_uo_chain[4].adj_par_list = [
    adj_par.AdjParSlider(
        var="v_void",
        v_range=(1, 50, 1),
        par_name='void volume [mL]'),
]

"""Set up `UserInterface` instance."""
gui = BokehServerGui(rtd_model=rtd_model)

# Customize `BokehServerGui`.
gui.plot_height = 300
gui.x_scale_factor = 1 / 60  # from min -> hours
gui.x_label = 't [h]'
gui.y_label_f = 'f [mL/min]'
gui.y_label_c = 'c [mg/mL]'
gui.species_label = ['mAB concentration']
gui.custom_x_ticks = np.arange(25)
gui.legend_only_on_first = True
gui.dti_plot = 1  # do not reduce data points (increase for more responsive UI)
gui.line_colors = ['black', 'grey', 'black']
gui.line_dashes = ['dashed']
gui.font_size_pt = 14
```

(continues on next page)

(continued from previous page)

```
# Build GUI.
gui.build_ui()

# Run simulation.
gui.recalculate(True)

"""Run GUI.

To enable interactive session, the script must be run through
`python bokeh serve` or `bokeh serve` command with bio_rtd folder
added to the PYTHON_PATH.
In terminal go to the repo (bio-unit_test) folder and run:
 export PYTHONPATH="$PWD"
 python `which bokeh` serve --show examples/models/integrated_mab_gui.py

"""
```

Integrated Excel GUI model

```
"""RtdModel with process data read from Excel document

In this example we define a model of another hypothetical mAB process
defined in Excel spreadsheet: `examples/models/integrated_excel.xlsx`.

Step-by-step:
#. Reading parameters from Excel.
#. Binding read parameters to process `PARAMETERS` and `ATTRIBUTES`
    based on templates (`examples/templates/`) for each unit operation.
#. Instantiate `UnitOperation`s and `Inlet`.
#. Creating `RtdModel` from `Inlet` and list of unit operations.

In this example we build a GUI on top of the model. GUI is run on
`bokeh` server. Once the GUI is running, one can simply modify a value
of a parameter in spreadsheet file and refresh web page. Refreshing
web page will re-run the model and thus 'update' is based on new data.

Notes
-----
Excel spreadsheet file does not depend (of include information) on the
RTD modeling approach. It simply serves as a data holder, thus if user
adds any new parameter in spreadsheet file, then the data binding part
of the script needs to be updated accordingly.

Interactive Bokeh GUI can easily be removed from this example and
replaced by plots and/or reports as in `examples/integrated_mab.py`.

"""

import pathlib
import numpy as np
from bokeh.io import show
from bokeh.layouts import column
from bokeh.models import LinearAxis, Range1d
from bokeh.plotting import figure
```

(continues on next page)

(continued from previous page)

```
from bio_rtd import pdf, peak_fitting
from bio_rtd.uo import sc_uo, fc_uo, surge_tank, special_uo
from bio_rtd.core import RtdModel
from bio_rtd.inlet import IntervalInlet
from bio_rtd.logger import DataStoringLogger
from bio_rtd.adj_par import AdjParRange, AdjParSlider, AdjParBoolean
from bio_rtd.chromatography.bt_load import ConstantPatternSolution

from examples.models.util.gui_bokeh import BokehServerGui
from examples.models.util.xlsx_data import read_bio_process_xlsx_data


def create_uo_pars_and_attrs(t: np.ndarray, _uo_lib: dict):
    """Adds methods, parameters and attributes to `uo_list`."""

    # Assert proper simulation time vector.
    assert t[0] == 0
    dt = t[-1] / (t.size - 1)  # simulation time step

    def atd(uo: dict):
        """Function adds parameters and attributes to the uo dict.

        Copy dictionary content template for each unit operation from
        templates (found in `examples/templates/`).

        """
        d = uo['data']
        v_void = d['void volume [mL]']
        v_sigma = d['RTD sigma [mL]']

        uo['uo_class'] = fc_uo.FlowThrough
        uo['parameters'] = {
            "uo_id": uo['id'],
            "pdf": pdf.GaussianFixedDispersion(t, v_sigma ** 2 / v_void),
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "v_void": v_void,
        }

    def df(uo: dict):
        d = uo['data']
        v_void = d['void volume [mL]']
        v_sigma = d['RTD sigma [mL]']
        v_switch = d['switch volume [L]'] * 1000  # L -> mL

        uo['uo_class'] = fc_uo.FlowThroughWithSwitching
        uo['parameters'] = {
            "uo_id": uo['id'],
            "pdf": pdf.GaussianFixedDispersion(t, v_sigma ** 2 / v_void),
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "v_void": v_void,
            "v_cycle": v_switch,
        }
```

(continues on next page)

(continued from previous page)

```
    def cvi_column(uo: dict):
        d = uo['data']
        rt_target = d['mean RT [min]']
        v_sigma = d['peak sigma [mL]']
        v_peak = d['peak position [mL]']

        uo['uo_class'] = fc_uo.FlowThrough
        uo['parameters'] = {
            "uo_id": uo['id'],
            "pdf": pdf.GaussianFixedDispersion(t, v_sigma ** 2 / v_peak),
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "rt_target": rt_target,
        }

    def ftc_aex(uo: dict):
        d = uo['data']
        cv = d['CV [mL]']
        v_void = cv * d['porosity protein []']
        v_peak = d['peak position [mL]']
        v_peak_sigma = d['peak sigma [mL]']
        v_cycle = cv * d['life cycle [CV]']

        uo['uo_class'] = fc_uo.FlowThroughWithSwitching
        uo['parameters'] = {
            "uo_id": uo['id'],
            "pdf": pdf.GaussianFixedDispersion(t, v_peak_sigma ** 2 / v_peak),
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "v_void": v_void,
            "v_cycle": v_cycle,
        }

    def st_single(uo: dict):
        d = uo['data']
        v_min_rel = d['min fill level rel []']
        starts_empty = d['starts empty []']

        uo['uo_class'] = surge_tank.CSTR
        uo['parameters'] = {
            "uo_id": uo['id'],
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "v_min_ratio": v_min_rel,
            "starts_empty": starts_empty,
        }

    def uf_df(uo: dict):
        d = uo['data']
        n_tanks = d['n tanks []']
        rt = d['residence time [min]']
        volume_reduction = d['volume reduction []']
        t_switch = d['switch time [min]']
        efficiency = d['efficiency []']
```

(continues on next page)

(continued from previous page)

```
        concentration = fc_uo.Concentration(
            t, flow_reduction=volume_reduction,
            uo_id=f"{uo['id']}_concentration")
        buffer_exchange = fc_uo.BufferExchange(
            t, exchange_ratio=efficiency,
            uo_id=f"{uo['id']}_buffer_exchange")
        flow_through = fc_uo.FlowThroughWithSwitching(
            t, pdf=pdf.TanksInSeries(t, n_tanks),
            uo_id=f"{uo['id']}_rtd")
        flow_through.rt_target = rt
        flow_through.t_cycle = t_switch

        uo['uo_class'] = special_uo.ComboUO
        uo['parameters'] = {
            "uo_id": uo['id'],
            "sub_uo_list": [concentration, buffer_exchange, flow_through],
            "gui_title": uo['title'],
        }
        uo['attributes'] = {}

    # noinspection DuplicatedCode
    def acc_cex(uo: dict):
        d = uo['data']
        # Column volume.
        cv = d['CV [mL]']
        # Operating linear flow rate during load.
        v_lin_load = d['load flowrate [cm/h]'] / 60  # cm/min
        # Other steps info (durations and flow rates).
        f_eq_rel = d['equilibration / load flowrate []']
        f_wash_rel = d['wash / load flowrate []']
        f_elution_rel = d['elution / load flowrate []']
        f_reg_rel = d['regeneration / load flowrate []']
        v_eq_cv = d['equilibration [CV]']
        v_wash_cv = d['wash [CV]']
        v_elution_cv = d['elution [CV]']
        v_reg_cv = d['regeneration [CV]']
        # Switch columns at x % protein breakthrough during load.
        load_switch_c_rel = d['load outlet conc ratio []']
        # Breakthrough profile.
        dbc_10 = d['DBC_10% [mg/mL]']
        dbc_100 = d['DBC_100% [mg/mL]']
        k = np.log(9) / (dbc_100 - dbc_10)
        # Elution peak (experimental data with column of different size).
        ep_rt, ep_sigma, ep_skew = \
            peak_fitting.calc_emg_parameters_from_peak_shape(
                t_peak_start=d['peak_a [mL]'],
                t_peak_max=d['peak_max [mL]'],
                t_peak_end=d['peak_b [mL]'],
                relative_threshold=d['peak_ab / peak_max []'])
        v_elution_peak_cv = ep_rt / d['peak_eval_CV [mL]']
        # Elution peak cut.
        el_peak_cut_start_rel = d['peak_start / peak_max []']
        el_peak_cut_end_rel = d['peak_end / peak_max []']
        unaccounted_losses = d['unaccounted_losses []']

        uo['uo_class'] = sc_uo.ACC
```

(continues on next page)

(continued from previous page)

```
        uo['parameters'] = {
            "uo_id": uo['id'],
            "load_bt": ConstantPatternSolution(dt, dbc_100=dbc_100, k=k),
            "peak_shape_pdf": pdf.ExpModGaussianFixedRelativeWidth(
                t, sigma_relative=ep_sigma / ep_rt, skew=ep_skew),
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "cv": cv,
            "equilibration_cv": v_eq_cv,
            "equilibration_f_rel": f_eq_rel,
            "load_c_end_relative_ss": load_switch_c_rel,
            "load_c_end_estimate_with_iterative_solver": True,
            "load_target_lin_velocity": v_lin_load,
            "wash_cv": v_wash_cv,
            "wash_f_rel": f_wash_rel,
            "unaccounted_losses_rel": unaccounted_losses,
            "elution_cv": v_elution_cv,
            "elution_f_rel": f_elution_rel,
            "elution_peak_position_cv": v_elution_peak_cv,
            "elution_peak_cut_start_c_rel_to_peak_max": el_peak_cut_start_rel,
            "elution_peak_cut_end_c_rel_to_peak_max": el_peak_cut_end_rel,
            "regeneration_cv": v_reg_cv,
            "regeneration_f_rel": f_reg_rel,
        }

    # noinspection DuplicatedCode
    def pcc_pro_a(uo: dict):
        d = uo['data']
        # Column volume.
        cv = d['CV [mL]']
        # Operating linear flow rate during load.
        v_lin_load = d['load flowrate [cm/h]'] / 60  # cm/min
        # Other steps info (durations and flow rates).
        f_eq_rel = d['equilibration / load flowrate []']
        f_wash_rel = d['wash / load flowrate []']
        f_elution_rel = d['elution / load flowrate []']
        f_reg_rel = d['regeneration / load flowrate []']
        v_eq_cv = d['equilibration [CV]']
        v_wash_cv = d['wash [CV]']
        v_elution_cv = d['elution [CV]']
        v_reg_cv = d['regeneration [CV]']
        # Switch columns at x % protein breakthrough during load.
        load_switch_c_rel = d['load col1 outlet conc ratio []']
        # Breakthrough profile.
        dbc_10 = d['DBC_10% [mg/mL]']
        dbc_100 = d['DBC_100% [mg/mL]']
        k = np.log(9) / (dbc_100 - dbc_10)
        # Elution peak (experimental data with column of different size).
        ep_rt, ep_sigma, ep_skew = \
            peak_fitting.calc_emg_parameters_from_peak_shape(
                t_peak_start=d['peak_a [mL]'],
                t_peak_max=d['peak_max [mL]'],
                t_peak_end=d['peak_b [mL]'],
                relative_threshold=d['peak_ab / peak_max []'])
        v_elution_peak_cv = ep_rt / d['peak_eval_CV [mL]']
        # Elution peak cut.
```

(continues on next page)

(continued from previous page)

```
        el_peak_cut_start_cv = d['peak collection start [CV]']
        el_peak_cut_end_cv = d['peak collection end [CV]']
        protein_porosity = d['porosity protein []']
        hetp = d['HETP [cm]']
        unaccounted_losses = d['unaccounted_losses []']
        wash_recycle = d['recycle_wash []']
        extend_first_cycle = d['extend first load []']

        uo['uo_class'] = sc_uo.PCC
        uo['parameters'] = {
            "uo_id": uo['id'],
            "load_bt": ConstantPatternSolution(dt, dbc_100=dbc_100, k=k),
            "peak_shape_pdf": pdf.ExpModGaussianFixedRelativeWidth(
                t, sigma_relative=ep_sigma / ep_rt, skew=ep_skew),
            "load_recycle_pdf": pdf.GaussianFixedDispersion(
                t, dispersion_index=hetp / v_lin_load),
            # Porosity of the column for protein.
            "column_porosity_retentate": protein_porosity,
            "gui_title": uo['title'],
        }
        uo['attributes'] = {
            "cv": cv,
            "equilibration_cv": v_eq_cv,
            "equilibration_f_rel": f_eq_rel,
            "load_c_end_relative_ss": load_switch_c_rel,
            "load_c_end_estimate_with_iterative_solver": True,
            "load_extend_first_cycle": extend_first_cycle,
            "load_target_lin_velocity": v_lin_load,
            "wash_cv": v_wash_cv,
            "wash_f_rel": f_wash_rel,
            "wash_recycle": wash_recycle,
            "unaccounted_losses_rel": unaccounted_losses,
            "elution_cv": v_elution_cv,
            "elution_f_rel": f_elution_rel,
            "elution_peak_position_cv": v_elution_peak_cv,
            "elution_peak_cut_start_cv": el_peak_cut_start_cv,
            "elution_peak_cut_end_cv": el_peak_cut_end_cv,
            "regeneration_cv": v_reg_cv,
            "regeneration_f_rel": f_reg_rel,
        }

    # Map method names to functions.
    method_2_func = {
        'ATD': atd,
        'DF': df,
        'PCC_ProA': pcc_pro_a,
        'ST_Single': st_single,
        'CVI_Column': cvi_column,
        'ACC_Cex': acc_cex,
        'FTC_Aex': ftc_aex,
        'UFDF': uf_df,
    }

    try:
        for _uo in _uo_lib.values():
            method_2_func.get(_uo['method'])(_uo)
    except TypeError:
```

(continues on next page)

(continued from previous page)

```
        # handle errors
        # find missing methods
        mm = set([_uo['method'] for _uo in _uo_lib.values()
                  if _uo['method'] not in method_2_func.keys()])
        if mm.__len__() > 0:
            raise NameError(f"No parsing functions for methods:"
                            f" `{'`, `'.join(mm)}`")
        else:
            raise
    except KeyError as ke:
        # noinspection PyUnboundLocalVariable
        raise KeyError(f"Unit operation: `{_uo['id']}`"
                       f"` with title: `{_uo['title']}`"
                       f"` is missing filed: `{ke.args[0]}`")


def generate_dsp_uo_train(t, _uo_list, _uo_lib):
    _dsp_train = []
    for uo_id in _uo_list:
        uo_pars = _uo_lib[uo_id]['parameters']
        uo_attr = _uo_lib[uo_id]['attributes']
        uo_class = _uo_lib[uo_id]['uo_class']
        uo_instance = uo_class(t, **uo_pars)
        for key, value in uo_attr.items():
            # Make sure attribute exist.
            assert hasattr(uo_instance, key), f"`{key}` is wrong."
            # Override value.
            setattr(uo_instance, key, value)
        _dsp_train.append(uo_instance)
    return _dsp_train


def generate_inlet(t, inlet_data, species):
    inlet_c = inlet_data['Titer [g/L]']  # [mg/mL]
    inlet_f = inlet_data['Flow [RV/day]'] * inlet_data['V [L]']
    inlet_f *= 1000 / 24 / 60  # [L/day] -> [mL/min]
    inlet = IntervalInlet(
        t=t, f=inlet_f,
        c_inner=np.array([0.0, inlet_c]),  # modification not in excel
        c_outer=np.array([inlet_c, 0.0]),  # modification not in excel
        species_list=species,
        inlet_id="usp",
        # gui_title=inlet_data['Title'],
        gui_title="Inlet",
    )
    # Modifications not in Excel.
    inlet.t_start = 60 * 5
    inlet.t_end = 30 + inlet.t_start
    return inlet


def plot_profiles(_rtd_model):
    plt_group = []
    for i, uo in enumerate([_rtd_model.inlet, *_rtd_model.dsp_uo_chain]):
        f, c = uo.get_result()  # get profiles
        # Prepare figure.
        plt = figure(plot_width=690, plot_height=300, title=uo.gui_title,
```

(continues on next page)

(continued from previous page)

```
                     tools="crosshair,reset,save,wheel_zoom,box_zoom,hover",
                     x_axis_label="t [h]", y_axis_label="c [mg/mL]")
        plt.extra_y_ranges = {'f': Range1d(0, f.max() * 1.1)}
        plt.add_layout(LinearAxis(y_range_name='f'), 'right')
        plt.yaxis[1].axis_label = "f [mL/min]"
        plt.y_range.start = 0
        plt.y_range.end = c.max() * 1.25
        # Plot profiles.
        plt.line(_rtd_model.inlet.get_t() / 60, c[0],
                 line_width=2, color='green', legend_label='product')
        plt.line(_rtd_model.inlet.get_t() / 60, c[1], line_width=2,
                 color='red', legend_label='section of product')
        plt.line(_rtd_model.inlet.get_t() / 60, f, y_range_name='f',
                 color='blue', legend_label='flow rate')
        # Add plot to plot list.
        plt.legend.location = "bottom_right"
        plt_group.append(plt)
    # Show plots.
    show(column(*plt_group))


def add_adj_pars(_rtd_model):
    """Add adjustable parameters, exposed to gui."""
    # Inlet.
    uo = _rtd_model.inlet
    uo.adj_par_list = [
        AdjParRange(var_list=('t_start', 't_end'),
                    v_range=(0, _rtd_model.inlet.get_t()[-1], 30),
                    par_name='Inlet interval [min]'),
        AdjParSlider(var='c_outer[0]',
                     v_range=(0, 10, 0.5),
                     par_name='Titer outside interval'),
        AdjParSlider(var='c_inner[1]',
                     v_range=(0, 10, 0.5),
                     par_name='Titer in interval'),
    ]
    # PCC.
    _rtd_model.get_dsp_uo('pro_a_pcc').adj_par_list = [
        AdjParSlider(
            var='cv',
            v_range=(50, 500, 50),
            par_name='Column volume [mL]')
    ]
    # Surge tank 1.
    uo = _rtd_model.get_dsp_uo('surge_tank_1')
    uo.adj_par_list = [
        AdjParBoolean(
            var='starts_empty',
            par_name='Starts empty',
            v_init=uo.starts_empty),
        AdjParSlider(
            var='v_min_ratio',
            v_range=(0, 100, 5),
            par_name='Minimum fill level [%]',
            scale_factor=0.01)
    ]
    # FTC_AEX.
```

(continues on next page)

(continued from previous page)

```
    try:
        uo = _rtd_model.get_dsp_uo('aex_ftc')
        uo.adj_par_list = [
            AdjParSlider(
                var='v_void',
                v_range=(1, 30, 1),
                scale_factor=0.8,  # compensation for porosity
                par_name='Column volume [mL]')
        ]
    except KeyError:  # uo might not be present in current scenario
        pass
    # Surge tank 2.
    try:
        uo = _rtd_model.get_dsp_uo('surge_tank_2')
        uo.adj_par_list = [
            AdjParBoolean(
                var='starts_empty',
                par_name='Starts empty',
                v_init=uo.starts_empty),
            AdjParSlider(
                var='v_min_ratio',
                v_range=(0, 100, 5),
                par_name='Minimum fill level [%]',
                scale_factor=0.01)
        ]
    except KeyError:  # uo might not be present in current scenario
        pass


def customize_gui(gui):
    # customize GUI
    gui.plot_height = 280
    gui.line_colors = ['#3288bd', 'green', 'red']
    gui.x_scale_factor = 1 / 60
    gui.x_label = 't [h]'
    gui.y_label_f = 'f [mL/min]'
    gui.y_label_c = 'c [mg/mL]'
    gui.custom_x_ticks = np.arange(25)
    gui.legend_only_on_first = True
    gui.dti_plot = 1
    gui.plot_first_component_as_sum = False
    gui.line_dashes = ['dashed']
    gui.font_size_pt = 14


def generate_up_rtd_model():
    # Define species names and simulation time vector.
    dt = 0.5  # min
    t = np.arange(0, 24.1 * 60, dt)
    species = ['product', 'section of product']
    # Choose scenario for USP and DSP.
    usp_scenario = 1
    dsp_scenario = 'A'

    # Read data from Excel.
    uo_lib, dsp, usp = read_bio_process_xlsx_data(
        pathlib.Path(__file__).parent / 'integrated_excel.xlsx'
```

(continues on next page)

(continued from previous page)

```
    )
    # Determine parameters and attributes for unit operations.
    create_uo_pars_and_attrs(t, uo_lib)
    # Generate inlet.
    inlet = generate_inlet(t, usp[usp_scenario], species)
    # Generate DSP train.
    dsp_train = generate_dsp_uo_train(t, dsp[dsp_scenario], uo_lib)
    # Create `RtdModel`.
    _rtd_model = RtdModel(inlet, dsp_train, DataStoringLogger(),
                          'Sample integrated process',
                          'Data was sourced from Excel file')
    # Add Adjustable parameters to the model.
    add_adj_pars(_rtd_model)
    return _rtd_model


rtd_model = generate_up_rtd_model()

print(__name__[-4:])

if __name__[:9] == "bokeh_app_k":
    # Create GUI.
    gui = BokehServerGui(rtd_model)
    # Customize GUI.
    customize_gui(gui)
    # Build GUI.
    gui.build_ui()
    # Run simulation.
    gui.recalculate(True)
else:
    rtd_model.log.log_level = rtd_model.log.ERROR
    rtd_model.recalculate(-1)
    plot_profiles(rtd_model)
```

4.3.2 Templates

(fc_uo) Dilution

bio_rtd.uo.fc_uo.Dilution

Instantiation

a) direct

```
"""Direct instance creation."""

tmp_uo = Dilution(
    t=np.linspace(0, 10, 100),
    dilution_ratio=1.6,  # 60 % addition of dilution buffer
    uo_id="dilution_direct",
    gui_title="Dilution, direct instance"  # Optional.
)
# Optional. Dilution buffer composition.
tmp_uo.c_add_buffer = np.array([0, 0, 200])
```

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Required.
    # 1 = no dilution, 1.6 = 60 % addition of dilution buffer.
    "dilution_ratio": float,

    # Optional.
    "gui_title": str,  # default: Dilution
}
```

```
ATTRIBUTES = {
    # Optional. Composition of dilution buffer.
    # Default is empty array (equivalent to 0).
    "c_add_buffer": np.ndarray,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

"""

uo_pars = {
    # Required.
    "uo_id": "dilution_1",
    # Required.
    # 1 = no dilution, 1.6 = 20 % addition of dilution buffer.
    "dilution_ratio": 1.6,

    # Optional.
    # "gui_title": str,  # default: Dilution
}

uo_attr = {
    # Optional. Composition of dilution buffer.
    # Default is empty array (equivalent to 0).
    "c_add_buffer": np.array([0, 0, 200]),

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

(continues on next page)

(continued from previous page)

```python


"""3. Instantiate unit operation and populate attributes."""

dilution = Dilution(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(dilution, key), f"`{key}` is wrong."
    # Override value.
    setattr(dilution, key, value)

# Voila :)
```

(fc_uo) Concentration

bio_rtd.uo.fc_uo.Concentration

Instantiation

a) direct

```python
"""Direct instance creation."""

tmp_uo = Concentration(
    t=np.linspace(0, 10, 100),
    flow_reduction=8,  # f_out = f_in / flow_reduction
    uo_id="concentration",
    gui_title="Concentration, direct instance"  # Optional.
)
# Optional. Which species are non-retained (e.g. salt).
tmp_uo.non_retained_species = [2]
# Optional. Relative losses. Does not apply to `non_retained_species`.
tmp_uo.relative_losses = 0.05  # 5 % losses
```

b) using parameters and attributes

```python
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Required. Example: `flow_reduction` = 4 -> f_out = f_in / 4.
    "flow_reduction": float,
    # Optional.
    "gui_title": str,  # default: Concentration
}
```

```python
ATTRIBUTES = {
    # Optional. Which species are non-retained (e.g. salt).
    "non_retained_species": List,

    # Optional. Relative losses. Does not apply to `non_retained_species`.
    "relative_losses": float,

```

(continues on next page)

(continued from previous page)

```
    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

"""

uo_pars = {
    # Required.
    "uo_id": "concentration",
    # Required. Example: `flow_reduction` = 4 -> f_out = f_in / 4.
    "flow_reduction": 8,
    # Optional.
    # "gui_title": str,  # default: Concentration
}

uo_attr = {
    # Optional. Which species are non-retained (e.g. salt).
    "non_retained_species": [2],

    # Optional. Relative losses. Does not apply to `non_retained_species`.
    "relative_losses": 0.05,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

concentration = Concentration(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(concentration, key), f"`{key}` is wrong."
    # Override value.
    setattr(concentration, key, value)

# Voila :)
```

(fc_uo) BufferExchange

`bio_rtd.uo.fc_uo.BufferExchange`

Instantiation

a) direct

```
"""Direct instance creation."""

tmp_uo = BufferExchange(
    t=np.linspace(0, 10, 100),
    exchange_ratio=0.95,
    uo_id="buffer_exchange",
    gui_title="BufferExchange, direct instance"  # Optional.
)
# Optional. Which species are non-retained (e.g. salt).
tmp_uo.non_retained_species = [2]
# Optional. Relative losses. Does not apply to `non_retained_species`.
tmp_uo.relative_losses = 0.05  # 5 % losses
# Optional. Exchange buffer composition. Default = empty array (= 0).
tmp_uo.c_add_buffer = np.array([0, 0, 200])
```

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Required.
    "exchange_ratio": float,
    # Optional.
    "gui_title": str,  # default: BufferExchange
}
```

```
ATTRIBUTES = {
    # Optional. Which species are non-retained (e.g. salt).
    "non_retained_species": List,

    # Optional. Relative losses. Does not apply to `non_retained_species`.
    "relative_losses": float,

    # Optional. Composition of exchange buffer.
    # Default is empty array (equivalent to 0).
    "c_exchange_buffer": np.ndarray,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step
```

(continues on next page)

(continued from previous page)

```python
"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

"""

uo_pars = {
    # Required.
    "uo_id": "buffer_exchange",
    # Required.
    "exchange_ratio": 0.95,
    # Optional.
    # "gui_title": str,  # default: BufferExchange
}

uo_attr = {
    # Optional. Which species are non-retained (e.g. salt).
    "non_retained_species": [1],

    # Optional. Relative losses. Does not apply to `non_retained_species`.
    "relative_losses": 0.05,

    # Optional. Composition of exchange buffer.
    # Default is empty array (equivalent to 0).
    # "c_exchange_buffer": np.ndarray,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

buffer_exchange = BufferExchange(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(buffer_exchange, key), f"`{key}` is wrong."
    # Override value.
    setattr(buffer_exchange, key, value)

# Voila :)
```

(fc_uo) FlowThrough

bio_rtd.uo.fc_uo.FlowThrough
bio_rtd.uo.fc_uo.FlowThroughWithSwitching

Instantiation

a) direct

```
"""Direct instance creation."""

t = np.linspace(0, 10, 100)

tmp_uo = FlowThroughWithSwitching(
    t=t,
    pdf=pdf.GaussianFixedDispersion(t, 2 * 2 / 45),
    uo_id="dilution_direct",
    gui_title="Dilution, direct instance"  # Optional.
)
tmp_uo.v_void = 3.4
tmp_uo.t_cycle = 20
```

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Required.
    "pdf": core.PDF,
    # Optional.
    "gui_title": str,  # default: FlowThrough, FlowThroughWithSwitching
}
```

```
ATTRIBUTES = {
    # One of next two.
    "v_void": float,
    "rt_target": float,  # target mean residence time

    # One or none. If none, v_init == v_void is assumed.
    "v_init": float,
    "v_init_ratio": float,  # share of void volume (v_void)

    # Optional. Default = [] (empty).
    "c_init": np.ndarray,  # equilibration buffer composition

    # Optional. Default = 0.
    "losses_share": float,
    # List of species to which losses apply. Default = [] (none).
    "losses_species_list": List[int],

    # ============== For `FlowThroughWithSwitching` only ===============
    # One of those three.
    "t_cycle": float,
    "v_cycle": float,
```

(continues on next page)

(continued from previous page)

```
    "v_cycle_relative": float,
    # ==================================================================

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

Notes
-----
Process parameters in the following example were chosen for
demonstrating the model usability (rather than representing a real
chromatographic process).

"""

uo_pars = {
    # Required.
    "uo_id": "FlowThrough_template_implementation",
    # Required.
    "pdf": pdf.GaussianFixedDispersion(t, 2 * 2 / 45),
    # Optional.
    # "gui_title": str,  # default: FlowThrough, FlowThroughWithSwitching
}

uo_attr = {
    # One of next two.
    "v_void": 3.4,
    # "rt_target": float,  # target mean residence time

    # One or none. If none, v_init == v_void is assumed.
    # "v_init": float,
    # "v_init_ratio": float,  # share of void volume (v_void)

    # Optional. Default = [] (empty).
    "c_init": np.array([0, 0, 20]),  # equilibration buffer composition

    # Optional. Default = 0.
    "losses_share": 0.02,
    # List of species to which losses apply. Default = [] (none).
    "losses_species_list": [0, 1],

    # ============== For `FlowThroughWithSwitching` only ===============
    # One of those three.
```

(continues on next page)

(continued from previous page)

```
    # "t_cycle": float,
    "v_cycle": 20,
    # "v_cycle_relative": float,
    # ===================================================================

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

flow_through = FlowThroughWithSwitching(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(flow_through, key), f"`{key}` is wrong."
    # Override value.
    setattr(flow_through, key, value)

# Voila :)
```

(special_uo) Combo UO

bio_rtd.uo.special_uo.ComboUO

Instantiation

a) direct

```
"""Direct instance creation."""

tmp_uo = ComboUO(
    # All unit operations should have the same time vector.
    t=np.linspace(0, 100, 1001),
    sub_uo_list=[concentration_template.concentration,
                 buffer_exchange_template.buffer_exchange,
                 flow_through_template.flow_through],
    uo_id="combo_uo",
    gui_title="Combo UO, direct instance"  # Optional.
)
```

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Required.
    "sub_uo_list": List[UnitOperation],
    # Optional.
    "gui_title": str,  # default: ComboUO
}
```

```
ATTRIBUTES = {
    # `ComboUO` has no specific attributes,
    # apart from the ones in `PARAMETERS`.

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step

"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

"""

uo_pars = {
    # Required.
    "uo_id": "uf_df",
    # Required.
    "sub_uo_list": [concentration_template.concentration,
                    buffer_exchange_template.buffer_exchange,
                    flow_through_template.flow_through],
    # Optional.
    # "gui_title": str,  # default: ComboUO
}

uo_attr = {
    # `ComboUO` has no specific attributes,
    # apart from the ones in `PARAMETERS`.

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}

"""3. Instantiate unit operation and populate attributes."""

cstr = ComboUO(t, **uo_pars)

# Can be omitted.
for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(cstr, key), f"`{key}` is wrong."
    # Override value.
    setattr(cstr, key, value)

# Voila :)
```

(surge_tank) CSTR

bio_rtd.uo.surge_tank.CSTR

Instantiation

a) direct

```
"""Direct instance creation. Minimalistic example."""

tmp_uo = CSTR(
    t=np.linspace(0, 10, 100),
    uo_id="cstr_direct",
    gui_title="CSTR, direct instance"  # Optional.
)
# Size of the surge tank.
tmp_uo.v_void = 140
```

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    # Optional.
    "gui_title": str,  # default: CSTR
}
```

```
ATTRIBUTES = {
    # Required. One of the following four.
    # It determines the size of the CSTR.
    "rt_target": float,  # target mean residence time (first momentum)
    "v_void": float,
    # Next two require periodic inlet flow rate profile.
    "v_min": float,
    "v_min_ratio": float,  # 10 % safety margin

    # Optional. One of the following two. It determines initial fill volume.
    # If none are defined, the initial fill volume is the same as void volume.
    "v_init": float,
    "v_ratio": float,  # 20 % of the total (== void) volume

    # Optional. Default = False.
    # If True, then `v_init` and `v_init_ratio` are ignored.
    "starts_empty": bool,

    # Optional. Initial buffer composition.
    # Default is empty array (equivalent to 0).
    "c_add_buffer": np.ndarray,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 100, 1001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

"""

uo_pars = {
    # Required.
    "uo_id": "cstr_1",
    # Optional.
    # "gui_title": str,  # default: Cstr
}

uo_attr = {
    # Required. One of the following four.
    # It determines the size of the CSTR.
    # "rt_target": float,  # target mean residence time (first momentum)
    # "v_void": float,
    # Next two require periodic inlet flow rate profile.
    # "v_min": float,
    "v_min_ratio": 0.1,  # 10 % safety margin

    # Optional. One of the following two. It determines initial fill volume.
    # If none are defined, the initial fill volume is the same as void volume.
    # "v_init": float,
    # "v_ratio": float,  # 20 % of the total (== void) volume

    # Optional. Default = False.
    # If True, then `v_init` and `v_init_ratio` are ignored.
    "starts_empty": True,

    # Optional. Initial buffer composition.
    # Default is empty array (equivalent to 0).
    # "c_add_buffer": np.ndarray,

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

cstr = CSTR(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(cstr, key), f"`{key}` is wrong."
    # Override value.
    setattr(cstr, key, value)
```

(continues on next page)

(continued from previous page)

```
# Voila :)
```

(sc_uo) ACC

bio_rtd.uo.sc_uo.ACC

Instantiation

a) direct

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    "load_bt": core.ChromatographyLoadBreakthrough,
    "peak_shape_pdf": core.PDF,

    # Optional.
    "gui_title": str,  # default: ACC
}
```

```
ATTRIBUTES = {

    # One of next two.
    "cv": float,
    "ft_mean_retentate": float,  # Requires `column_porosity_retentate`.
    # Required if `ft_mean_retentate`, otherwise ignored.
    "column_porosity_retentate": float,

    # Optional. Default = [] (empty).
    "non_binding_species": List[int],  # indexing starts with 0

    # One or both. If both, the duration adds up.
    "equilibration_cv": float,
    "equilibration_t": float,
    # One of next two.
    "equilibration_f": float,
    "equilibration_f_rel": float,  # relative to inlet (load) flow rate

    # One of next three.
    "load_cv": float,
    "load_c_end_ss": np.ndarray,
    "load_c_end_relative_ss": float,
    # Optional. Default = False.
    "load_c_end_estimate_with_iterative_solver": bool,
    # Optional. Default = 1000.
    # Ignored if `load_c_end_estimate_with_iterative_solver == False`.
    "load_c_end_estimate_with_iterative_solver_max_iter": int,

    # Optional.
    "load_extend_first_cycle": bool,  # Default: False
    # Ignored if `load_extend_first_cycle == True`.
```

(continues on next page)

(continued from previous page)

```
    # If both, the duration is added together.
    # If none, the duration is estimated by the model
    "load_extend_first_cycle_cv": float,
    "load_extend_first_cycle_t": float,

    # Optional.
    "load_target_lin_velocity": float,

    # One or both. If both, the duration adds up.
    "wash_cv": float,
    "wash_t": float,
    # One of next two.
    "wash_f": float,
    "wash_f_rel": float,  # relative to inlet (load) flow rate

    # Optional. Default = 0.
    # Elution peak is scaled down by (1 - `unaccounted_losses_rel`).
    # Peak cut criteria is applied after the scale down.
    "unaccounted_losses_rel": float,

    # One or both. If both, the duration adds up.
    "elution_cv": float,
    "elution_t": float,
    # One of next two.
    "elution_f": float,
    "elution_f_rel": float,  # relative to inlet (load) flow rate
    # Optional. Default is empty array (-> all species are 0).
    "elution_buffer_c": np.ndarray,

    # One of next two.
    # Fist momentum relative to the beginning of elution step.
    "elution_peak_position_cv": float,
    "elution_peak_position_t": float,
    # One of next four.
    "elution_peak_cut_start_t": float,
    "elution_peak_cut_start_cv": float,
    "elution_peak_cut_start_c_rel_to_peak_max": float,
    "elution_peak_cut_start_peak_area_share": float,
    # One of next four.
    "elution_peak_cut_end_t": float,
    "elution_peak_cut_end_cv": float,
    "elution_peak_cut_end_c_rel_to_peak_max": float,
    "elution_peak_cut_end_peak_area_share": float,

    # One or both. If both, the duration adds up.
    "regeneration_cv": float,
    "regeneration_t": float,
    # One of next two.
    "regeneration_f": float,
    "regeneration_f_rel": float,  # relative to inlet (load) flow rate

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 1000, 10001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

Notes
-----
Process parameters in the following example were chosen for
demonstrating the model usability (rather than representing a real
chromatographic process).

"""

uo_pars = {
    # Required.
    "uo_id": "pcc_template_implementation",
    "load_bt": bt_load.ConstantPatternSolution(dt, dbc_100=240, k=0.05),
    "peak_shape_pdf": pdf.GaussianFixedDispersion(t, 8**2 / 30),

    # Optional.
    # "gui_title": str,  # default: ACC
}

uo_attr = {
    # One of next two.
    "cv": 13,
    # "ft_mean_retentate": float,  # Requires `column_porosity_retentate`.
    # Required if `ft_mean_retentate`, otherwise ignored.
    # "column_porosity_retentate": float,

    # Optional. Default = [].
    # "non_binding_species": List[int],  # indexing starts with 0

    # One or both. If both, the duration adds up.
    "equilibration_cv": 3,
    # "equilibration_t": float,

    # One of next two.
    # "equilibration_f": float,
    "equilibration_f_rel": 1,  # relative to inlet (load) flow rate

    # One of next three.
    # "load_cv": float,
    # "load_c_end_ss": np.ndarray,
    "load_c_end_relative_ss": 0.7,  # 70 % of breakthrough

    # Optional. Default = False.
    "load_c_end_estimate_with_iterative_solver": True,

    # Optional. Default = 1000.
    # Ignored if `load_c_end_estimate_with_iterative_solver == False`.
```

(continues on next page)

(continued from previous page)

```
    # "load_c_end_estimate_with_iterative_solver_max_iter": int,

    # Optional.
    "load_extend_first_cycle": True,  # Default: False

    # Ignored if `load_extend_first_cycle == True`.
    # If both, the duration is added together.
    # If none, the duration is estimated by the model
    # "load_extend_first_cycle_cv": float,
    # "load_extend_first_cycle_t": float,

    # Optional.
    # "load_target_lin_velocity": float,

    # One or both. If both, the duration adds up.
    "wash_cv": 5,
    # "wash_t": float,
    # One of next two.
    # "wash_f": float,
    "wash_f_rel": 1,  # relative to inlet (load) flow rate

    # Optional. Default = 0.
    # Elution peak is scaled down by (1 - `unaccounted_losses_rel`).
    # Peak cut criteria is applied after the scale down.
    "unaccounted_losses_rel": 0.2,

    # One or both. If both, the duration adds up.
    "elution_cv": 3,
    # "elution_t": float,

    # One of next two.
    # "elution_f": float,
    "elution_f_rel": 1 / 4,  # relative to inlet (load) flow rate
    # Optional. Default is empty array (-> all species are 0).
    # "elution_buffer_c": np.ndarray,

    # One of next two.
    # Fist momentum relative to the beginning of elution step.
    "elution_peak_position_cv": 1.6,
    # "elution_peak_position_t": float,

    # One of next four.
    # "elution_peak_cut_start_t": float,
    "elution_peak_cut_start_cv": 1.05,
    # "elution_peak_cut_start_c_rel_to_peak_max": float,
    # "elution_peak_cut_start_peak_area_share": float,

    # One of next four.
    # "elution_peak_cut_end_t": float,
    "elution_peak_cut_end_cv": 2.3,
    # "elution_peak_cut_end_c_rel_to_peak_max": float,
    # "elution_peak_cut_end_peak_area_share": float,

    # One or both. If both, the duration adds up.
    "regeneration_cv": 1,
    # "regeneration_t": float,
```

(continues on next page)

(continued from previous page)

```
    # One of next two.
    # "regeneration_f": float,
    "regeneration_f_rel": 1,  # relative to inlet (load) flow rate

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

acc = ACC(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(acc, key), f"`{key}` is wrong."
    # Override value.
    setattr(acc, key, value)

# Voila :)
```

(sc_uo) PCC

bio_rtd.uo.sc_uo.PCC
bio_rtd.uo.sc_uo.PCCWithWashDesorption

Instantiation

a) direct

b) using parameters and attributes

```
PARAMETERS = {
    # Required.
    "uo_id": str,
    "load_bt": core.ChromatographyLoadBreakthrough,
    "peak_shape_pdf": core.PDF,
    "load_recycle_pdf": core.PDF,
    # Porosity of the column for protein.
    "column_porosity_retentate": float,

    # Optional.
    "gui_title": str,  # default: PCC, PCCWithWashDesorption
}
```

```
ATTRIBUTES = {

    # One of next two.
    "cv": float,
    "ft_mean_retentate": float,

    # Optional. Default = [] (empty).
    "non_binding_species": List[int],  # indexing starts with 0
```

(continues on next page)

(continued from previous page)

```
    # One or both. If both, the duration adds up.
    "equilibration_cv": float,
    "equilibration_t": float,
    # Optional. One of next two. Default: `equilibration_f_rel` = 1
    "equilibration_f": float,
    "equilibration_f_rel": float,  # relative to inlet (load) flow rate

    # One of next three.
    "load_cv": float,
    "load_c_end_ss": np.ndarray,
    "load_c_end_relative_ss": float,
    # Optional. Default = False.
    "load_c_end_estimate_with_iterative_solver": bool,
    # Optional. Default = 1000.
    # Ignored if `load_c_end_estimate_with_iterative_solver == False`.
    "load_c_end_estimate_with_iterative_solver_max_iter": int,

    # Optional.
    "load_extend_first_cycle": bool,  # Default: False
    # Ignored if `load_extend_first_cycle == True`.
    # If both, the duration is added together.
    # If none, the duration is estimated by the model
    "load_extend_first_cycle_cv": float,
    "load_extend_first_cycle_t": float,

    # Optional.
    "load_target_lin_velocity": float,

    # One or both. If both, the duration adds up.
    "wash_cv": float,
    "wash_t": float,
    # Optional. One of next two. Default: `wash_f_rel` = 1
    "wash_f": float,
    "wash_f_rel": float,  # relative to inlet (load) flow rate

    # Optional.
    # Captures breakthrough material during wash step onto third column.
    # while 2nd column is loading.
    "wash_recycle": bool,
    # Optional. If both, the duration is added.
    # If none and `wash_recycle` is True, then the entire wash is recycled.
    "wash_recycle_duration_cv": float,
    "wash_recycle_duration_t": float,

    # Optional. Default = 0.
    # Elution peak is scaled down by (1 - `unaccounted_losses_rel`).
    # Peak cut criteria is applied after the scale down.
    "unaccounted_losses_rel": float,

    # One or both. If both, the duration adds up.
    "elution_cv": float,
    "elution_t": float,
    # Optional. One of next two. Default: `elution_f_rel` = 1
    "elution_f": float,
    "elution_f_rel": float,  # relative to inlet (load) flow rate
    # Optional. Default is empty array (-> all species are 0).
```

(continues on next page)

(continued from previous page)

```
    "elution_buffer_c": np.ndarray,

    # One of next two.
    # Fist momentum relative to the beginning of elution step.
    "elution_peak_position_cv": float,
    "elution_peak_position_t": float,
    # One of next four.
    "elution_peak_cut_start_t": float,
    "elution_peak_cut_start_cv": float,
    "elution_peak_cut_start_c_rel_to_peak_max": float,
    "elution_peak_cut_start_peak_area_share": float,
    # One of next four.
    "elution_peak_cut_end_t": float,
    "elution_peak_cut_end_cv": float,
    "elution_peak_cut_end_c_rel_to_peak_max": float,
    "elution_peak_cut_end_peak_area_share": float,

    # One or both. If both, the duration adds up.
    "regeneration_cv": float,
    "regeneration_t": float,
    # Optional. One of next two. Default: `regeneration_f_rel` = 1
    "regeneration_f": float,
    "regeneration_f_rel": float,  # relative to inlet (load) flow rate

    # ================ For `PCCWithWashDesorption` only ================
    # Required.
    "wash_desorption_tail_half_time_cv": float,
    # One of those two.
    "wash_desorption_desorbable_material_share": float,
    "wash_desorption_desorbable_above_dbc": float,
    # ==================================================================

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}
```

Additional *Add-On Attributes* are available for each `bio_rtd.core.UnitOperation`.

```
"""1. Define a time step and a simulation time vector."""
t = np.linspace(0, 1000, 10001)  # it must start with 0
dt = t[1]  # time step


"""2. Use `PARAMETERS` and `ATTRIBUTES` as a template.

Copy/Paste templates.
Replace variable types with values.
Remove or comment out the ones that are not needed.

Notes
-----
Process parameters in the following example were chosen for
demonstrating the model usability (rather than representing a real
chromatographic process).

"""
```

(continues on next page)

(continued from previous page)

```
uo_pars = {
    # Required.
    "uo_id": "pcc_template_implementation",
    "load_bt": bt_load.ConstantPatternSolution(dt, dbc_100=240, k=0.05),
    "peak_shape_pdf": pdf.GaussianFixedDispersion(t, 8**2 / 30),
    "load_recycle_pdf": pdf.GaussianFixedDispersion(t, 2 * 2 / 30),
    # Protein porosity of the column.
    "column_porosity_retentate": 0.64,

    # Optional.
    # "gui_title": str,  # default: PCC, PCCWithWashDesorption
}

uo_attr = {
    # One of next two.
    "cv": 13,
    # "ft_mean_retentate": float,

    # Optional. Default = [].
    # "non_binding_species": List[int],  # indexing starts with 0

    # Optional. One or both. If both, the duration adds up.
    "equilibration_cv": 3,
    # "equilibration_t": float,

    # Optional. One of next two. Default: `equilibration_f_rel` = 1
    # "equilibration_f": float,
    # "equilibration_f_rel": 1,  # relative to inlet (load) flow rate

    # One of next three.
    # "load_cv": float,
    # "load_c_end_ss": np.ndarray,
    "load_c_end_relative_ss": 0.7,  # 70 % of breakthrough

    # Optional. Default = False.
    "load_c_end_estimate_with_iterative_solver": True,

    # Optional. Default = 1000.
    # Ignored if `load_c_end_estimate_with_iterative_solver == False`.
    # "load_c_end_estimate_with_iterative_solver_max_iter": int,

    # Optional.
    "load_extend_first_cycle": True,  # Default: False

    # Ignored if `load_extend_first_cycle == True`.
    # If both, the duration is added together.
    # If none, the duration is estimated by the model
    # "load_extend_first_cycle_cv": float,
    # "load_extend_first_cycle_t": float,

    # Optional.
    # "load_target_lin_velocity": float,

    # One or both. If both, the duration adds up.
    "wash_cv": 5,
    # "wash_t": float,
```

(continues on next page)

(continued from previous page)

```
    # One of next two.
    # "wash_f": float,
    "wash_f_rel": 1,  # relative to inlet (load) flow rate

    # Optional.
    # Captures breakthrough material during wash step onto third column.
    # while 2nd column is loading.
    # "wash_recycle": bool,
    # Optional. If both, the duration is added.
    # If none and `wash_recycle` is True, then the entire wash is recycled.
    # "wash_recycle_duration_cv": float,
    # "wash_recycle_duration_t": float,

    # Optional. Default = 0.
    # Elution peak is scaled down by (1 - `unaccounted_losses_rel`).
    # Peak cut criteria is applied after the scale down.
    "unaccounted_losses_rel": 0.15,

    # One or both. If both, the duration adds up.
    "elution_cv": 3,
    # "elution_t": float,

    # One of next two.
    # "elution_f": float,
    "elution_f_rel": 1 / 4,  # relative to inlet (load) flow rate
    # Optional. Default is empty array (-> all species are 0).
    # "elution_buffer_c": np.ndarray,

    # One of next two.
    # Fist momentum relative to the beginning of elution step.
    "elution_peak_position_cv": 1.6,
    # "elution_peak_position_t": float,

    # One of next four.
    # "elution_peak_cut_start_t": float,
    "elution_peak_cut_start_cv": 1.05,
    # "elution_peak_cut_start_c_rel_to_peak_max": float,
    # "elution_peak_cut_start_peak_area_share": float,

    # One of next four.
    # "elution_peak_cut_end_t": float,
    "elution_peak_cut_end_cv": 2.3,
    # "elution_peak_cut_end_c_rel_to_peak_max": float,
    # "elution_peak_cut_end_peak_area_share": float,

    # One or both. If both, the duration adds up.
    "regeneration_cv": 1,
    # "regeneration_t": float,

    # One of next two.
    # "regeneration_f": float,
    "regeneration_f_rel": 1,  # relative to inlet (load) flow rate

    # ================ For `PCCWithWashDesorption` only ================
    "wash_desorption_tail_half_time_cv": 2,
    # One of those two.
    "wash_desorption_desorbable_material_share": 0.05,
```

(continues on next page)

(continued from previous page)

```
    # "wash_desorption_desorbable_above_dbc": float,
    # ==================================================================

    # Additional attributes are inherited from `UnitOperation`.
    # See `examples/templates/add_on_attributes.py`.
    # Add them to the list if needed.
}


"""3. Instantiate unit operation and populate attributes."""

pcc = PCCWithWashDesorption(t, **uo_pars)

for key, value in uo_attr.items():
    # Make sure attribute exist.
    assert hasattr(pcc, key), f"`{key}` is wrong."
    # Override value.
    setattr(pcc, key, value)

# Voila :)
```

Add-On Attributes

Add-On Attributes can be added to attribute list of any other template `bio_rtd.core.UnitOperation`.

```
ADD_ON_ATTRIBUTES = {
    # List of adjustable parameters exposed to the GUI.
    "adj_par_list": List[AdjustableParameter],

    # Hide plot of the unit operation (default False).
    "gui_hidden": bool,

    # Optional. One of next four.
    # Discard inlet until given time.
    "discard_inlet_until_t": float,
    # Discard inlet until given concentration is reached for each component.
    "discard_inlet_until_min_c": np.ndarray,
    # Discard inlet until specified concentration between inlet concentration
    # and steady-state inlet concentration is reached for each component.
    "discard_inlet_until_min_c_rel": np.ndarray,
    # Discard first n cycles of the periodic inlet flow rate profile.
    "discard_inlet_n_cycles": float,

    # Optional. One of next four.
    # Discard outlet until given time.
    "discard_outlet_until_t": float,
    # Discard outlet until given concentration is reached for each component.
    "discard_outlet_until_min_c": np.ndarray,
    # Discard outlet until specified concentration between outlet concentration
    # and steady-state outlet concentration is reached for each component.
    "discard_outlet_until_min_c_rel": np.ndarray,
    # Discard first n cycles of the periodic outlet flow rate profile.
    "discard_outlet_n_cycles": float,

    # Optional. Default = logger.DefaultLogger().
```

(continues on next page)

(continued from previous page)

```
    # If the unit operation is a part of `RtdModel`,  then the logger
    # is inherited from `RtdModel`.
    "log": RtdLogger,
}
```

4.4 API Reference

`bio_rtd.core.Inlet`

4.4.1 bio_rtd.inlet

ConstantInlet

class bio_rtd.inlet.**ConstantInlet**(*t*, *f*, *c*, *species_list*, *inlet_id*, *gui_title='ConstantInlet'*)

Bases: `bio_rtd.core.Inlet`

Constant flow rate and constant process fluid composition

Parameters

- **t** (`array`) – Simulation time vector
- **f** (`float`) – Constant flow rate.
- **c** (`ndarray`) – Constant concentration for each specie. For single specie use *np.array([c_value])*.

get_n_species()

Get number of process fluid species.

Return type `int`

get_result()

Get flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Flow rate profile.
- *c_out* – Concentration profile.

get_t()

Get simulation time vector.

Return type `ndarray`

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

IntervalInlet

class bio_rtd.inlet.**IntervalInlet**(*t*, *f*, *c_inner*, *c_outer*, *species_list*, *inlet_id*, *gui_title*)

Bases: *bio_rtd.core.Inlet*

Constant flow rate profile and box shaped concentration profile

Variables

- **t_start** – Start position of box, inclusive (>=). Initial == t[0] == 0
- **t_end** – End position of box, excluding (<). Initial == t[-1] + t[1]
- **c_inner** (*np.ndarray*) – Concentrations inside the box.
- **c_outer** (*np.ndarray*) – Concentrations outside the box.

get_n_species()

Get number of process fluid species.

Return type int

get_result()

Get flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Flow rate profile.
- *c_out* – Concentration profile.

get_t()

Get simulation time vector.

Return type ndarray

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

CustomInlet

class bio_rtd.inlet.**CustomInlet**(*t*, *f*, *c*, *species_list*, *inlet_id*, *gui_title*)

Bases: *bio_rtd.core.Inlet*

Custom flow rate profile and concentration profiles

Variables

- **t** – Simulation time vector.
- **f** – Custom flow rate profile. Should have the same size az *t*.
- **c** – Custom concentration profiles. *c.shape == (len(species_list), t.size)*

get_n_species()

Get number of process fluid species.

Return type int

get_result()

Get flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Flow rate profile.
- *c_out* – Concentration profile.

get_t()

Get simulation time vector.

Return type ndarray

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

---

bio_rtd.core.UnitOperation

4.4.2 bio_rtd.uo.fc_uo

Dilution

class `bio_rtd.uo.fc_uo.`**`Dilution`**(*t*, *dilution_ratio*, *uo_id*, *gui_title='Dilution'*)

Bases: *`bio_rtd.core.UnitOperation`*

Dilute process fluid stream in constant ratio.

Variables

- **`dilution_ratio`** (*`int`*) – Dilution ratio. Must be >= 1. Dilution ratio of 1.2 means adding 20 % of dilution buffer. It is applied before the *delay_inlet*.
- **`c_add_buffer`** (*`np.ndarray`*) – Concentration of species in dilution buffer.

`evaluate`(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **`c_in`** (`ndarray`) – Inlet concentration profile with shape (n_species, n_time_steps).
- **`f_in`** (`array`) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple`[`ndarray`, `ndarray`]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

`get_result`()

Returns existing flow rate and concentration profiles.

Return type `Tuple`[`ndarray`, `ndarray`]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *`RtdLogger`*

`set_logger_from_parent`(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **`parent_id`** (`str`) –
- **`logger`** (*`RtdLogger`*) –

Concentration

class bio_rtd.uo.fc_uo.**Concentration**(*t*, *flow_reduction*, *uo_id*, *gui_title='Concentration'*)

Bases: *bio_rtd.core.UnitOperation*

Concentrate process fluid stream

Concentration step can be used before or after the *FlowThrough* or *FlowThroughWithSwitching* steps in order to simulate unit operations such as SPTFF or UFDF

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

BufferExchange

class bio_rtd.uo.fc_uo.**BufferExchange**(*t*, *exchange_ratio*, *uo_id*, *gui_title='BufferExchange'*)

Bases: *bio_rtd.core.UnitOperation*

Buffer exchange

Can be combined with *Concentration* and one of the *FlowThrough* or *FlowThroughWithSwitching* steps in order to simulate unit operations such as SPTFF or UFDF

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

FlowThrough

class bio_rtd.uo.fc_uo.**FlowThrough**(*t*, *pdf*, *uo_id*, *gui_title='FlowThrough'*)

Bases: *bio_rtd.core.UnitOperation*

Fully continuous unit operation without inline switching

FlowThrough has a constant PDF, which depends on process parameters. It assumes a constant inlet flow rate (apart from initial delay or early stop). It does not depend on concentration.

If initial volume (*v_init*) < void volume (*v_void*), then the unit operation is first filled up. During the fill-up an ideal mixing is assumed.

Variables

- **v_void** (*float*) – Effective void volume of the unit operations (v_void = rt_target / f)

  Values > 0 are accepted.

- **rt_target** (*float*) – Specified flow-through time as alternative way to define *v_void* (v_void = rt_target / f) Values > 0 are accepted in v_void <= 0
- **v_init** (*float*) – Effective void volume of the unit operations (v_void = rt_target / f) Values (v_void >= v_init >= 0) are accepted. Values > v_void result in error. If v_init and v_init_ratio are both undefined or out of range, v_init = v_void is assumed.

- **v_init_ratio** (*float*) – Specified flow-through time as alternative way to define *v_void* (v_void = rt_target / f) Values (0 >= v_init_ratio >= 1) are accepted if v_init if < 0. Values > 1 result in error. If v_init and v_init_ratio are both < 0, v_init == v_void is assumed.
- **c_init** (*np.ndarray*) – Concentration in v_init for each process fluid component. If left blank it is assumed that all components are 0.
- **pdf** (PDF) – Steady-state probability distribution function (see PFD class). pdf is updated based on passed *v_void* and *f* at runtime

evaluate (*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result ()

Returns existing flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent (*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

FlowThroughWithSwitching

class bio_rtd.uo.fc_uo.**FlowThroughWithSwitching**(*t*, *pdf*, *uo_id*, *gui_title='FlowThroughWithSwitching'*)

Bases: *bio_rtd.uo.fc_uo.FlowThrough*

Fully continuous unit operation with inline switching (= piece-wise FC UO)

FlowThroughWithSwitching has a constant PDF, which depends on process parameters. It assumes a constant inlet flow rate (apart from initial delay or early stop). Its operation does not depend on concentration values It is periodically interrupted

If initial volume (*v_init*) < void volume (*v_void*), then the unit operation is first filled up. During the fill-up an ideal mixing is assumed. First cycle starts when the inlet flow rate is turned on (possible initial delays are covered in UnitOperation)

Variables

- **t_cycle** (*float*) – Duration of the cycle between switches
- **v_cycle** (*float*) – Alternative way to define *t_cycle t_cycle* = *v_cycle* / *f*
- **v_cycle_relative** (*float*) – Alternative way to define *t_cycle t_cycle* = *v_cycle_relative* * *v_void* / *f* = *v_cycle_relative* * *rt_mean*
- **pdf** (PDF) – Steady-state probability distribution function (see PFD class). pdf is updated based on passed *v_void* and *f* at runtime

Parameters

- **v_void** (*float*) – Effective void volume of the unit operations (v_void = rt_target / f)

  Values > 0 are accepted.
- **rt_target** (*float*) – Specified flow-through time as alternative way to define *v_void* (v_void = rt_target / f) Values > 0 are accepted in v_void <= 0
- **c_init** (*np.ndarray*) – Concentration in v_init for each process fluid component. If left blank it is assumed that all components are 0.

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (`RtdLogger`) –

4.4.3 bio_rtd.uo.sc_uo

ACC

class `bio_rtd.uo.sc_uo.`**ACC**(*t*, *uo_id*, *load_bt*, *peak_shape_pdf*, *gui_title='ACC'*)
Bases: `bio_rtd.uo.sc_uo.AlternatingChromatography`

evaluate(*f_in*, *c_in*)
Evaluate the propagation through the unit operation.

Parameters

- **c_in** (`ndarray`) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (`array`) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()
Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (`RtdLogger`) –

PCC

class bio_rtd.uo.sc_uo.**PCC**(*t*, *uo_id*, *load_bt*, *load_recycle_pdf*, *column_porosity_retentate*, *peak_shape_pdf*, *gui_title='PCC'*)

Bases: *bio_rtd.uo.sc_uo.AlternatingChromatography*

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type Tuple[ndarray, ndarray]

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

PCCWithWashDesorption

class bio_rtd.uo.sc_uo.**PCCWithWashDesorption**(*t*, *uo_id*, *load_bt*, *load_recycle_pdf*, *column_porosity_retentate*, *peak_shape_pdf*, *gui_title='PCCWithWashDesorption'*)

Bases: *bio_rtd.uo.sc_uo.PCC*

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (ndarray) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (array) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

AlternatingChromatography

class bio_rtd.uo.sc_uo.**AlternatingChromatography**(*t*, *uo_id*, *load_bt*, *peak_shape_pdf*, *gui_title='ACC'*)

Bases: *bio_rtd.core.UnitOperation*

column_porosity_retentate

Column rtm

cv: float

Column volume

evaluate(*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (`ndarray`) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (`array`) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result()

Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent (*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

4.4.4 bio_rtd.uo.surge_tank

CSTR

class bio_rtd.uo.surge_tank.**CSTR** (*t*, *uo_id*, *gui_title='CSTR'*)
Bases: *bio_rtd.core.UnitOperation*

Simulation of CSTR with ideal mixing

Target upper fill volume can be defined via - *v_void* : upper target volume - *v_min* : minimum target volume (if periodic_inlet == True) - *v_min_ratio* : the ratio between v_min and v_void (if periodic_inlet == True) - *rt_target* : target residence time (_target_upper_fill_volume = rt_target * _f_out_target) If *v_void == -1*, then the *v_min* is considered If *v_min == -1*, then the *v_min_ratio* is considered If *v_min_ratio == -1*, then the *rt_target* is considered If *rt_target == -1*, then the previous *_target_upper_fill_volume* is used

Initial fill volume can be defined via - v_init : init fill volume (_v_init = v_init) - v_init_ratio : the ratio between _v_init and v_void If v_init < 0, then the v_init_ratio is considered If v_init_ratio < 0, then the _v_init = v_void The concentration of pre-filled part is defined by *c_init*

evaluate (*f_in*, *c_in*)
Evaluate the propagation through the unit operation.

Parameters

- **c_in** (`ndarray`) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (`array`) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result ()
Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (`RtdLogger`) –

4.4.5 bio_rtd.uo.special_uo

ComboUO

class bio_rtd.uo.special_uo.**ComboUO**(*t*, *sub_uo_list*, *uo_id*, *gui_title='ComboUO'*)
Bases: `bio_rtd.core.UnitOperation`

A class that can hold a list of unit subsequent operation and present them as one

evaluate(*f_in*, *c_in*)
Evaluates all child unit operations.

get_result()
Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (`RtdLogger`) –

`bio_rtd.core.PDF`

4.4.6 bio_rtd.pdf

GaussianFixedDispersion

class bio_rtd.pdf.**GaussianFixedDispersion**(*t*, *dispersion_index*, *cutoff=0.0001*, *pdf_id='GaussianFixedDispersion'*)

Bases: *bio_rtd.core.PDF*

Gaussian PDF with fixed dispersion

Parameters

- **dispersion_index** (*float*) – Dispersion index, defined as *sigma * sigma / rt_mean*. Where *rt_mean* is a mean residence time and *sigma* is a standard deviation.
- **cutoff** (*float*) – Cutoff limit for trimming front and end tailing. Cutoff limit is relative to the peak max value.

See also:

`The`

Examples

```
>>> t = _np.linspace(0, 100, 1001)
>>> dt = t[1]
>>> pdf = GaussianFixedDispersion(t, dispersion_index=0.2)
>>> pdf.update_pdf(rt_mean=40)
>>> p = pdf.get_p()
>>> print(round(p.sum() * dt, 8))
1.0
>>> t[p.argmax()]
40.0
```

assert_and_get_provided_kv_pairs(***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

get_p()

Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

update_pdf(***kwargs*)
Re-calculate PDF based on specified parameters.

Parameters **kwargs** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

GaussianFixedRelativeWidth

class bio_rtd.pdf.**GaussianFixedRelativeWidth**(*t*, *relative_sigma*, *cutoff=0.0001*, *pdf_id=''*)
Bases: *bio_rtd.core.PDF*

Gaussian PDF with fixed relative peak width

relative_sigma = sigma / rt_mean

assert_and_get_provided_kv_pairs(***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

get_p()
Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

update_pdf(***kwargs*)
Re-calculate PDF based on specified parameters.

Parameters **kwargs** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

ExpModGaussianFixedDispersion

class bio_rtd.pdf.**ExpModGaussianFixedDispersion**(*t*, *dispersion_index*, *skew*, *pdf_id=''*)
Bases: *bio_rtd.core.PDF*

Exponentially Modified Gaussian PDF with fixed dispersion

*dispersion_index = sigma**2 / rt_mean pdf(rt_mean) = calc_emg(t, rt_mean, sigma, skew)*

set_pdf_pars(*dispersion_index: float*, *skew: float*)
Sets the dispersion index and skew factor

Abstract Methods

----

update_pdf(***kwargs*)
Calculate new pdf for a given set of parameters. *pdf_update_keys* contains groups of possible parameter combinations.

_possible_key_groups : Sequence[Sequence[str]]
List of keys for key-value argument combinations that could be specified to *update_pdf(**kwarg)* function

assert_and_get_provided_kv_pairs(***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

get_p()
Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

update_pdf(***kwargs*)
Re-calculate PDF based on specified parameters.

Parameters **`kwargs`** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

ExpModGaussianFixedRelativeWidth

class `bio_rtd.pdf.`**`ExpModGaussianFixedRelativeWidth`**(*t*, *sigma_relative*, *skew*, *pdf_id=''*)

Bases: *`bio_rtd.core.PDF`*

Exponentially Modified Gaussian PDF with fixed relative peak width

`assert_and_get_provided_kv_pairs`(***kwargs*)

Parameters **`kwargs`** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **`ValueError`** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

`get_p`()

Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *`RtdLogger`*

`set_logger_from_parent`(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **`parent_id`** (`str`) –
- **`logger`** (*`RtdLogger`*) –

`update_pdf`(***kwargs*)

Re-calculate PDF based on specified parameters.

Parameters **`kwargs`** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

TanksInSeries

class `bio_rtd.pdf.`**`TanksInSeries`**(*t*, *n_tanks*, *pdf_id=''*)

Bases: `bio_rtd.core.PDF`

Tanks in series PDF

rt_mean means flow-through time through entire unit operation (all tanks)

`set_pdf_pars`(*n_tanks: float*)

Sets number of tanks

`Abstract Methods`

`----------------`

`update_pdf`(***kwargs*)

Calculate new pdf for a given set of parameters. *pdf_update_keys* contains groups of possible parameter combinations.

`_possible_key_groups : Sequence[Sequence[str]]`

List of keys for key-value argument combinations that could be specified to *update_pdf(**kwarg)* function

`assert_and_get_provided_kv_pairs`(***kwargs*)

Parameters **`kwargs`** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **`ValueError`** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

`get_p`()

Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

`property log`

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

`set_logger_from_parent`(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **`parent_id`** (`str`) –
- **`logger`** (`RtdLogger`) –

`update_pdf`(***kwargs*)

Re-calculate PDF based on specified parameters.

Parameters **`kwargs`** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

`bio_rtd.logger.RtdLogger`

4.4.7 bio_rtd.logger

DefaultLogger

class `bio_rtd.logger.`**`DefaultLogger`**(*log_level=30*, *log_data=False*, *log_level_data=10*)

Bases: `bio_rtd.logger.RtdLogger`

Prints warnings to terminal and raises errors.

Does not store data.

`DEBUG = 10`

`ERROR = 40`

`INFO = 20`

`WARNING = 30`

`d_data`(*tree*, *key*, *value*)

Log larger intermediate data (time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *DEBUG* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

`e`(*msg*)

Log message at ERROR level

ERROR level is meant for signaling events that are very likely the source of or the sign of inaccuracies in the model, but not severe enough to raise an exception in all situations (e.g. in an interactive session via GUI).

If the issue is big enough to warrant an exception, then raise the exception instead of using this log.

Examples

Probability distribution with only few data points. Probability distribution with a high cut-off at time 0. Load phase of PCC being shorter than the rest of the process.

`get_data_tree`(*data_tree_id*)

Returns reference to the registered data tree.

Return type `dict`

`get_entire_data_tree`()

Returns reference to the dict with all logged data.

`i`(*msg*)

Log message at INFO level

INFO level is meant for events that might indicate potential mishaps the model, but can also occur normally.

Examples

Reporting that surge tank ran dry. This info might help explain the spectra, but it might just as well be a normal occurrence during the shut-down phase.

`i_data` (*tree*, *key*, *value*)
Log smaller data (no time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *INFO* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

`log` (*level*, *msg*)
Log messages at specific log level.

See documentation of *self.e()*, *self.w()* and *self.i()* on info about what belongs under which log level.

`set_branch` (*data_tree*, *branch_name*, *branch*)
Seeds branch into data tree.

Typically used to store data from nested unit operations or parts of unit operations (evaluation data of probability distributions, breakthrough profiles, etc.).

Parameters

- **`data_tree`** (`dict`) – Parent data tree.
- **`branch_name`** (`dict`) – Branch (child tree) name.
- **`branch`** (`dict`) – Branch (child tree).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> branch_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.set_branch(data_tree, "pdf", branch_tree)
>>> log.i_data(branch_tree, "par_b", 8.4)
>>> data_tree["pdf"]["par_b"]
8.4
>>> branch_tree["par_b"]
8.4
```

`set_data_tree` (*data_tree_id*, *data_tree*)
Seeds data tree in root dictionary.

Typically *tree_id* = *uo.id* and *tree_root* = *dict()*.

Unit operation needs to save a reference to the *tree_root* in order to store data in this logger.

Parameters

- **`data_tree_id`** (`str`) – Id of root tree (typically id of unit operation).
- **`data_tree`** (`dict`) – Root for data logging (for that unit operation).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.i_data(data_tree, "par_a", 10.4)
>>> data_tree["par_a"]
10.4
```

w(*msg*)

Log message at WARNING level

WARNING level is meant for signaling events that might be the source of inaccuracies in the model.

Examples

Suspicious probability distributions. Empty concentration of flow rate profiles. Assumptions that might impact the results, such as assumptions during steady-state estimation.

DataStoringLogger

class bio_rtd.logger.**DataStoringLogger**(*log_level=30*, *log_data=True*, *log_level_data=10*)

Bases: *bio_rtd.logger.RtdLogger*

Prints messages to terminal. Stores all data.

DEBUG = 10

ERROR = 40

INFO = 20

WARNING = 30

d_data(*tree*, *key*, *value*)

Log larger intermediate data (time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *DEBUG* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

e(*msg*)

Log message at ERROR level

ERROR level is meant for signaling events that are very likely the source of or the sign of inaccuracies in the model, but not severe enough to raise an exception in all situations (e.g. in an interactive session via GUI).

If the issue is big enough to warrant an exception, then raise the exception instead of using this log.

Examples

Probability distribution with only few data points. Probability distribution with a high cut-off at time 0. Load phase of PCC being shorter than the rest of the process.

get_data_tree(*data_tree_id*)
Returns reference to the registered data tree.

Return type dict

get_entire_data_tree()
Returns reference to the dict with all logged data.

i(*msg*)
Log message at INFO level

INFO level is meant for events that might indicate potential mishaps the model, but can also occur normally.

Examples

Reporting that surge tank ran dry. This info might help explain the spectra, but it might just as well be a normal occurrence during the shut-down phase.

i_data(*tree*, *key*, *value*)
Log smaller data (no time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *INFO* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

log(*level*, *msg*)
Log messages at specific log level.

See documentation of *self.e()*, *self.w()* and *self.i()* on info about what belongs under which log level.

set_branch(*data_tree*, *branch_name*, *branch*)
Seeds branch into data tree.

Typically used to store data from nested unit operations or parts of unit operations (evaluation data of probability distributions, breakthrough profiles, etc.).

Parameters

- **data_tree** (*dict*) – Parent data tree.
- **branch_name** (*dict*) – Branch (child tree) name.
- **branch** (*dict*) – Branch (child tree).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> branch_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.set_branch(data_tree, "pdf", branch_tree)
>>> log.i_data(branch_tree, "par_b", 8.4)
>>> data_tree["pdf"]["par_b"]
```

(continues on next page)

(continued from previous page)

```
8.4
>>> branch_tree["par_b"]
8.4
```

set_data_tree(*data_tree_id*, *data_tree*)

Seeds data tree in root dictionary.

Typically *tree_id* = *uo.id* and *tree_root* = *dict()*.

Unit operation needs to save a reference to the *tree_root* in order to store data in this logger.

Parameters

- **data_tree_id** (*str*) – Id of root tree (typically id of unit operation).
- **data_tree** (*dict*) – Root for data logging (for that unit operation).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.i_data(data_tree, "par_a", 10.4)
>>> data_tree["par_a"]
10.4
```

w(*msg*)

Log message at WARNING level

WARNING level is meant for signaling events that might be the source of inaccuracies in the model.

Examples

Suspicious probability distributions. Empty concentration of flow rate profiles. Assumptions that might impact the results, such as assumptions during steady-state estimation.

StrictLogger

class bio_rtd.logger.**StrictLogger**

Bases: *bio_rtd.logger.RtdLogger*

Raises RuntimeError on warning and error messages.

DEBUG = 10

ERROR = 40

INFO = 20

WARNING = 30

d_data(*tree*, *key*, *value*)

Log larger intermediate data (time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *DEBUG* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

e(*msg*)
Log message at ERROR level

ERROR level is meant for signaling events that are very likely the source of or the sign of inaccuracies in the model, but not severe enough to raise an exception in all situations (e.g. in an interactive session via GUI).

If the issue is big enough to warrant an exception, then raise the exception instead of using this log.

Examples

Probability distribution with only few data points. Probability distribution with a high cut-off at time 0. Load phase of PCC being shorter than the rest of the process.

get_data_tree(*data_tree_id*)
Returns reference to the registered data tree.

Return type `dict`

get_entire_data_tree()
Returns reference to the dict with all logged data.

i(*msg*)
Log message at INFO level

INFO level is meant for events that might indicate potential mishaps the model, but can also occur normally.

Examples

Reporting that surge tank ran dry. This info might help explain the spectra, but it might just as well be a normal occurrence during the shut-down phase.

i_data(*tree*, *key*, *value*)
Log smaller data (no time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *INFO* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

log(*level*, *msg*)
Log messages at specific log level.

See documentation of *self.e()*, *self.w()* and *self.i()* on info about what belongs under which log level.

set_branch(*data_tree*, *branch_name*, *branch*)
Seeds branch into data tree.

Typically used to store data from nested unit operations or parts of unit operations (evaluation data of probability distributions, breakthrough profiles, etc.).

Parameters

- **data_tree** (*dict*) – Parent data tree.
- **branch_name** (*dict*) – Branch (child tree) name.
- **branch** (*dict*) – Branch (child tree).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> branch_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.set_branch(data_tree, "pdf", branch_tree)
>>> log.i_data(branch_tree, "par_b", 8.4)
>>> data_tree["pdf"]["par_b"]
8.4
>>> branch_tree["par_b"]
8.4
```

set_data_tree(*data_tree_id*, *data_tree*)
Seeds data tree in root dictionary.

Typically *tree_id* = *uo.id* and *tree_root* = *dict()*.

Unit operation needs to save a reference to the *tree_root* in order to store data in this logger.

Parameters

- **data_tree_id** (*str*) – Id of root tree (typically id of unit operation).
- **data_tree** (*dict*) – Root for data logging (for that unit operation).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.i_data(data_tree, "par_a", 10.4)
>>> data_tree["par_a"]
10.4
```

w(*msg*)
Log message at WARNING level

WARNING level is meant for signaling events that might be the source of inaccuracies in the model.

Examples

Suspicious probability distributions. Empty concentration of flow rate profiles. Assumptions that might impact the results, such as assumptions during steady-state estimation.

RtdLogger

class bio_rtd.logger.**RtdLogger**(*log_level=30*, *log_data=False*, *log_level_data=10*)
Bases: `abc.ABC`

Abstract class for log and log data in rtd models.

Logger has different levels: DEBUG, INFO, WARNING, ERROR. Logger has option to hold copies of intermediate data.

Variables

- **log_level** (*int*) – Verbosity level for messages. Default = *WARNING*.

- **`log_data`** – If True, logger collects copies of intermediate data. Default = False.
- **`log_level_data`** – 'Verbosity level' for data. Default = *DEBUG*. Ignored if *log_data* is False.

`DEBUG = 10`

Log level DEBUG

DEBUG level is meant for keeping also large intermediate data (time series) in logger.

`ERROR = 40`

Log level ERROR

ERROR level is meant for signaling events (with messages) that very likely impact the accuracy of the model.

`INFO = 20`

Log level INFO

INFO level is meant for events that might indicate potential mishaps in the model, but can also occur normally (e.g. if surge tank runs dry). This info might help explain the spectra, but it might also be a normal occurrence during the shut-down phase.

INFO level is also meant for keeping small intermediate results (no time series) in logger.

`WARNING = 30`

Log level WARNING

WARNING level is meant for signaling events (with messages) that might be the source of inaccuracies in the model.

`d_data`(*tree*, *key*, *value*)

Log larger intermediate data (time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *DEBUG* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

`e`(*msg*)

Log message at ERROR level

ERROR level is meant for signaling events that are very likely the source of or the sign of inaccuracies in the model, but not severe enough to raise an exception in all situations (e.g. in an interactive session via GUI).

If the issue is big enough to warrant an exception, then raise the exception instead of using this log.

Examples

Probability distribution with only few data points. Probability distribution with a high cut-off at time 0. Load phase of PCC being shorter than the rest of the process.

`get_data_tree`(*data_tree_id*)

Returns reference to the registered data tree.

Return type `dict`

`get_entire_data_tree`()

Returns reference to the dict with all logged data.

`i`(*msg*)

Log message at INFO level

INFO level is meant for events that might indicate potential mishaps the model, but can also occur normally.

Examples

Reporting that surge tank ran dry. This info might help explain the spectra, but it might just as well be a normal occurrence during the shut-down phase.

`i_data`(*tree*, *key*, *value*)
Log smaller data (no time series).

Populates (key, value.copy()) pair into *tree* if *log_level_data* >= *INFO* and *log_data* is True.

If the copy of the value is stored, then the *self._on_data_stored* function is called.

`abstract log`(*level*, *msg*)
Log messages at specific log level.

See documentation of *self.e()*, *self.w()* and *self.i()* on info about what belongs under which log level.

`set_branch`(*data_tree*, *branch_name*, *branch*)
Seeds branch into data tree.

Typically used to store data from nested unit operations or parts of unit operations (evaluation data of probability distributions, breakthrough profiles, etc.).

Parameters

- **`data_tree`** (`dict`) – Parent data tree.
- **`branch_name`** (`dict`) – Branch (child tree) name.
- **`branch`** (`dict`) – Branch (child tree).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> branch_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.set_branch(data_tree, "pdf", branch_tree)
>>> log.i_data(branch_tree, "par_b", 8.4)
>>> data_tree["pdf"]["par_b"]
8.4
>>> branch_tree["par_b"]
8.4
```

`set_data_tree`(*data_tree_id*, *data_tree*)
Seeds data tree in root dictionary.

Typically *tree_id* = *uo.id* and *tree_root* = *dict()*.

Unit operation needs to save a reference to the *tree_root* in order to store data in this logger.

Parameters

- **`data_tree_id`** (`str`) – Id of root tree (typically id of unit operation).
- **`data_tree`** (`dict`) – Root for data logging (for that unit operation).

Examples

```
>>> log = DataStoringLogger()
>>> data_tree = dict()
>>> log.set_data_tree("my_uo", data_tree)
>>> log.i_data(data_tree, "par_a", 10.4)
>>> data_tree["par_a"]
10.4
```

w (*msg*)
Log message at WARNING level

WARNING level is meant for signaling events that might be the source of inaccuracies in the model.

Examples

Suspicious probability distributions. Empty concentration of flow rate profiles. Assumptions that might impact the results, such as assumptions during steady-state estimation.

---

bio_rtd.core.ChromatographyLoadBreakthrough

4.4.8 bio_rtd.chromatography.bt_load

ConstantPatternSolution

class bio_rtd.chromatography.bt_load.**ConstantPatternSolution** (*dt*, *dbc_100*, *k*, *bt_profile_id=''*)
Bases: *bio_rtd.core.ChromatographyLoadBreakthrough*

assert_and_get_provided_kv_pairs (***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

calc_c_bound (*f_load*, *c_load*)
Calculates what parts of load bin to the column.

Parameters

- **f_load** (ndarray) – Load flow rate profile.
- **c_load** (ndarray) – Load concentration profile. Concentration profile should include only species which bind to the column.

Returns Parts of the load that binds to the column during the load step. *c_bound* has the same shape as *c_load*.

Return type c_bound

Notes

This is default implementation. The user is welcome to override this function in a custom child class.

get_total_bc()
Total binding capacity

Useful for determining column utilization.

Return type `float`

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

update_btc_parameters(***kwargs*)
Update binding dynamics for a given set of parameters.

Return type `None`

`bio_rtd.chromatography.bt_curve`

4.4.9 bio_rtd.chromatography.bt_curve

btc_constant_pattern_solution

`bio_rtd.chromatography.bt_curve.`**btc_constant_pattern_solution**(*m_load*, *dbc_100*, *k*, *cv*, *logger=None*)

Return type `ndarray`

`bio_rtd.utils.convolution`

4.4.10 bio_rtd.utils.convolution

time_conv

`bio_rtd.utils.convolution.`**time_conv**(*dt*, *c_in*, *rtd*, *c_equilibration=None*, *logger=None*)
Perform time convolution

First time-point of *c_in* and ‘c_rtd’ is at t == 0 (and not at t == dt). Convolution is applied to all columns (= species) of the *c_in*

Parameters

- **dt** (*float*) – Time step.

- **c_in** (*np.ndarray*) – Starting concentration profile for each specie
- **rtd** (ndarray) – Residence time distribution (= unit impulse response)
- **c_equilibration** (Optional[ndarray]) – Initial concentrations of species inside the void volume of the unit operation. E. g. the composition of equilibration buffer for flow-through column.
- **logger** (Optional[*RtdLogger*]) – Logger can be passed from unit operations

Returns Final concentration profile for each specie

Return type np.ndarray

piece_wise_conv_with_init_state

bio_rtd.utils.convolution.**piece_wise_conv_with_init_state**(*dt*, *f_in*, *c_in*, *t_cycle*, *rt_mean*, *rtd*, *c_equilibration=None*, *c_wash=None*, *logger=None*)

Return type ndarray

---

bio_rtd.utils.vectors

4.4.11 bio_rtd.utils.vectors

true_start

bio_rtd.utils.vectors.**true_start**(*boolean_vector*)
accepts array of booleans and returns index of first true

Return type int

true_end

bio_rtd.utils.vectors.**true_end**(*boolean_vector*)
accepts array of booleans and returns index AFTER last true

Return type int

true_start_and_end

bio_rtd.utils.vectors.**true_start_and_end**(*boolean_vector*)
accepts array of booleans and returns indexes of first and AFTER last true

Return type (<class 'int'>, <class 'int'>)

---

bio_rtd.adj_par.AdjustableParameter

4.4.12 bio_rtd.adj_par

AdjParSlider

class `bio_rtd.adj_par.`**`AdjParSlider`**(*var*, *v_range*, *v_init=None*, *scale_factor=1*, *par_name=None*, *gui_type='slider'*)

Bases: *`bio_rtd.adj_par.AdjustableParameter`*

A value slider version of *AdjustableParameter*

Parameters

- **`var`** (`str`) – Variable of a UnitOperation affected by the parameter.
- **`or Sequence[bool]`** (*`v_init`*) –
- **`or (float, float) or (float, float, float)`** (*`v_range`*) – Defines range end or [start, end] or [start, end, step]. If step is not defined, the *step = (start - end) / 10*. If start is not defined the *start = 0*.

Notes

For more info see docstring of *AdjustableParameter* superclass.

AdjParRange

class `bio_rtd.adj_par.`**`AdjParRange`**(*var_list*, *v_range*, *v_init=None*, *scale_factor=1*, *par_name=None*, *gui_type='range'*)

Bases: *`bio_rtd.adj_par.AdjustableParameter`*

Defines a range slider version of *AdjustableParameter*

Variables

- **`var_list`** (*`Tuple[str, str]`*) – Defines attributed affected by the *AdjustableParameter*. First has value of the start of the interval and second of the end of the interval.
- **`or Tuple[float, float] or Tuple[float, float, float]`** (*`v_range`*) – Defines range end or [start, end] or [start, end, step]. If step is not defined, the *step = (start - end) / 10*. If start is not defined the *start = 0*.
- **`v_init`** (*`Tuple[float, float]`*) – Initial value [interval start, interval end]. If None (default), then the initial values are assigned from the initial conditions at the start of the simulation.

AdjParBoolean

class `bio_rtd.adj_par.`**`AdjParBoolean`**(*var*, *v_init=None*, *par_name=None*, *scale_factor=1*, *gui_type='checkbox'*)

Bases: *`bio_rtd.adj_par.AdjustableParameter`*

A boolean variation of *`AdjustableParameter`*

Parameters

- **`var`** (`str`) – Variable of a UnitOperation affected by the parameter.
- **`or list of bool`** (*`v_init`*) – Initial value or list of initial values for each item in *var_list*.

- **or list of str** (*par_name*) – List of parameter names. If defined, the shape of the should match *var_list*.

AdjParBooleanMultiple

class bio_rtd.adj_par.**AdjParBooleanMultiple**(*var_list*, *v_init=None*, *par_name=None*, *scale_factor=1*, *gui_type='radio_group'*)

Bases: *bio_rtd.adj_par.AdjustableParameter*

A radio group variation of *AdjustableParameter*

Parameters

- **var_list** (*list of str*) – A list of attributes of a *UnitOperation* affected by the *AdjustableParameter*.
- **v_init** (Optional[Sequence[bool]]) – Initial value for each entry in *var_list*. If None (default), the value is inherited from the initial attribute values at the start of the runtime. If specified, the shape should match the shape of *var_list*.
- **par_name** (*list of str*) – Attribute title (e.g. 'Initial delay [min]') for each option. The shape should batch the shape of *var_list*.

AdjustableParameter

class bio_rtd.adj_par.**AdjustableParameter**(*var_list*, *v_range=None*, *v_init=None*, *scale_factor=1*, *par_name=None*, *gui_type=None*)

Bases: abc.ABC

Adjustable parameter for a UnitOperation (or Inlet).

The class does not provide any logic. It just provides a unified way to specify an adjustable attribute.

Parameters

- **var_list** (List[str]) – A list of attributes of a *UnitOperation* affected by the *AdjustableParameter*.
- **v_range** (Union[Tuple[float, float, float], Sequence[str], Sequence[float], Sequence[Tuple[str, str]], None]) – Range of values. For boolean set None. For range specify [start, end, step]. For list set list. If the list has two columns, the first one is the title and the second is the value.
- **v_init** (Union[float, bool, Sequence[float], None]) – Initial value. If None (default), the value is inherited from the initial attribute value (at the start of the runtime).
- **scale_factor** (float) – The scale factor to compensate for differences in units. E.g.: Setting time in [h] for attribute set in [min] requires a *scale_factor* of 60.
- **par_name** (Union[str, Sequence[str], None]) – Attribute title (e.g. 'Initial delay [min]') or list of titles in case of multiple options (e.g. radio button group).
- **gui_type** (Optional[str]) – Preferred method for setting parameter in gui parameters. If None (default) or the gui does not support the type, then the gui should decide the method based on *v_init* and/or *var_list*. Example values: 'checkbox', 'range', 'slider', 'select', 'multi-select'

Notes

Technically the class relates to instance attributes rather than parameters. However, the term 'process parameters' is well established in biotech, hence the class name.

`bio_rtd.peak_fitting`

4.4.13 bio_rtd.peak_fitting

calc_emg_parameters_from_peak_shape

`bio_rtd.peak_fitting.`**`calc_emg_parameters_from_peak_shape`**(*t_peak_max*, *t_peak_start*, *t_peak_end*, *relative_threshold*)

Calculates mean_rt, sigma and skew based on peak start, peak position and peak end

Parameters

- **`t_peak_max`** (*`float`*) – Peak max position.
- **`t_peak_start`** (*`float`*) – Peak start position.
- **`t_peak_end`** (*`float`*) – Peak end position.
- **`relative_threshold`** (*`float`*) – Relative signal (compared to peak max) for given start and end positions

Returns Calculated *rt_mean*.

Return type (rt_mean, sigma, skew)

`bio_rtd.peak_shapes`

4.4.14 bio_rtd.peak_shapes

gaussian

`bio_rtd.peak_shapes.`**`gaussian`**(*t*, *rt_mean*, *sigma*, *logger=None*)

Return type `ndarray`

emg

`bio_rtd.peak_shapes.`**`emg`**(*t*, *rt_mean*, *sigma*, *skew*, *logger=None*)

skew_normal

`bio_rtd.peak_shapes.`**`skew_normal`**(*t*, *rt_mean*, *sigma*, *skew*, *logger=None*)

tanks_in_series

`bio_rtd.peak_shapes.`**`tanks_in_series`**(*t*, *rt_mean*, *n_tanks*, *logger=None*)
rt_mean is for entire unit operation (all tanks together)

Return type `ndarray`

`bio_rtd.core`

4.4.15 bio_rtd.core

Inlet

`class` `bio_rtd.core.`**`Inlet`**(*t*, *species_list*, *inlet_id*, *gui_title*)
Bases: *`bio_rtd.core.DefaultLoggerLogic`*, `abc.ABC`

Generates starting flow rate and concentration profiles.

Parameters

- **`t`** (`ndarray`) – Simulation time vector

  Starts with 0 and has a constant time step.

- **`species_list`** (`Sequence[str]`) – List with names of simulating process fluid species.
- **`inlet_id`** (`str`) – Unique identifier of an instance. It is stored in `uo_id`.
- **`gui_title`** (`str`) – Readable title of an instance.

Variables

- **`species_list`** (*`list of str`*) – List with names of simulating process fluid species.
- **`uo_id`** (*`str`*) – Unique identifier of the *`Inlet`* instance.
- **`gui_title`** (*`str`*) – Readable title of the *`Inlet`* instance.
- **`adj_par_list`** (list of *`bio_rtd.adj_par.AdjustableParameter`*) – List of adjustable parameters exposed to the GUI.

`get_n_species`()
Get number of process fluid species.

Return type `int`

`get_result`()
Get flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Flow rate profile.
- *c_out* – Concentration profile.

get_t()

Get simulation time vector.

Return type `ndarray`

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

UnitOperation

class bio_rtd.core.**UnitOperation**(*t*, *uo_id*, *gui_title=''*)

Bases: *bio_rtd.core.DefaultLoggerLogic*, `abc.ABC`

Processes flow rate and concentration profiles.

Parameters

- **t** (`ndarray`) – Global simulation time vector. It must starts with 0 and have a constant time step.
- **uo_id** (`str`) – Unique identifier.
- **gui_title** (`str`) – Readable title for GUI.

Variables

- ***adj_par_list*** – List of adjustable parameters exposed to the GUI.
- **gui_hidden** – Hide the of the unit operation (default False).
- **discard_inlet_until_t** – Discard inlet until given time.
- **discard_inlet_until_min_c** – Discard inlet until given concentration is reached.
- **discard_inlet_until_min_c_rel** – Discard inlet until given concentration relative to is reached. Specified concentration is relative to the max concentration.
- **discard_inlet_n_cycles** – Discard first n cycles of the periodic inlet flow rate profile.
- **discard_outlet_until_t** – Discard outlet until given time.
- **discard_outlet_until_min_c** – Discard outlet until given concentration is reached.
- **discard_outlet_until_min_c_rel** – Discard outlet until given concentration relative to is reached. Specified concentration is relative to the max concentration.
- **discard_outlet_n_cycles** – Discard first n cycles of the periodic outlet flow rate profile.

adj_par_list: _typing.Sequence[_adj_par.AdjustableParameter]

Settings

evaluate (*f_in*, *c_in*)

Evaluate the propagation through the unit operation.

Parameters

- **c_in** (`ndarray`) – Inlet concentration profile with shape (n_species, n_time_steps).
- **f_in** (`array`) – Inlet flow rate profile with shape (n_time_steps,).

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

get_result ()

Returns existing flow rate and concentration profiles.

Return type `Tuple[ndarray, ndarray]`

Returns

- *f_out* – Outlet flow rate profile.
- *c_out* – Outlet concentration profile.

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent (*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

RtdModel

class bio_rtd.core.**RtdModel** (*inlet*, *dsp_uo_chain*, *logger=None*, *title='RtdModel'*, *desc=''*)

Bases: *bio_rtd.core.DefaultLoggerLogic*, `abc.ABC`

Combines inlet and a train of unit operations into a model.

The logger assigned to the instance of RtdModel is propagated throughout *inlet* and unit operations in *dsp_uo_chain*.

Parameters

- **inlet** (*Inlet*) – Inlet profile.
- **dsp_uo_chain** (`Sequence[`*UnitOperation*`]`) – Sequence of unit operations. The sequence needs to be in order.
- **logger** (`Optional[`*RtdLogger*`]`) – Logger that the model uses for sending status messages and storing intermediate data.
- **title** (`str`) – Title of the model.
- **desc** (`str`) – Description of the model.

recalculate(*start_at*, *on_update_callback*)
Recalculates the process fluid propagation, starting at *start_at* unit operation (-1 for inlet and entire process). Callback function can be specified. It receives the index of the just updated unit operation.

get_dsp_uo(*uo_id*)
Get reference to a *UnitOperation* with specified *uo_id*.

Return type `UnitOperation`

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type `RtdLogger`

recalculate(*start_at=0*, *on_update_callback=None*)
Recalculate process fluid propagation.

Parameters

- **start_at** (`int`) – The index of first unit operation that needs to be re-evaluated. Default = 0. If -1, then the inlet profile is also re-evaluated.
- **on_update_callback** (`Optional[Callable[[int], None]]`) – Optional callback function which receives an integer.

  The integer corresponds to the index of re-evaluated unit operation (-1 for inlet). This can serve as a trigger for updating UI after re-evaluation of individual unit operations.

set_logger_from_parent(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (`RtdLogger`) –

UserInterface

class bio_rtd.core.**UserInterface**(*rtd_model*)
Bases: `abc.ABC`

Wrapper around RtdModel suitable for building GUI on top of it.

Parameters **rtd_model** (`RtdModel`) – RTD model.

Variables

- **species_label** (*`list of str`*) – Labels of the species in concentration array.
- **x_label** – Label of x axis (time). Default = 't'
- **y_label_c** – Label of y axis (concentration). Default = 'c'
- **y_label_f** – Label of y axis (flow rate). Default = 'f'
- **start_at** (*`int`*) – The index of unit operation (starting with 0) at which the re-evaluation starts. The value of -1 means that the inlet profile is also reevaluated.

abstract build_ui()
Build the UI from scratch.

recalculate (*forced=False*)
Re-evaluates the model from the *start_at* index onwards.

Parameters **forced** – If true, the entire model (inlet + unit operations) is re-evaluated. The same can be achieved by setting *self.start_at* to -1.

PDF

class bio_rtd.core.**PDF** (*t*, *pdf_id=''*)
Bases: *bio_rtd.core.ParameterSetList*, *bio_rtd.core.DefaultLoggerLogic*, abc.ABC

Abstract class for defining probability distribution functions.

Parameters

- **t** (ndarray) – Simulation time vector.
- **pdf_id** (str) – Unique identifier of the PDF instance.

Variables

- **trim_and_normalize** – Trim edges of the peak by the threshold at the relative signal specified by *cutoff_relative_to_max*. Default = True.
- **cutoff_relative_to_max** – Cutoff as a share of max value of the pdf (default 0.0001). It is defined to avoid very long tails of the distribution.

get_p ()
Get calculated PDF.

update_pdf (***kwargs*)
Re-calculate PDF based on specified parameters.

Abstract Methods

_calc_pdf (*kw_pars: dict*)
Calculate new pdf for a given set of parameters. The keys of the *kw_pars* include keys from one of the group in *_possible_key_groups* and any optional subset of keys from *_optional_keys*.

assert_and_get_provided_kv_pairs (***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

get_p ()
Get probability distribution.

Returns **p** – Evaluated probability distribution function. *sum(p * self._dt) == 1* Corresponding time axis starts with 0 and has a fixed step of self._dt.

Return type np.ndarray

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

`set_logger_from_parent`(*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **`parent_id`** (`str`) –
- **`logger`** (*RtdLogger*) –

`update_pdf`(***kwargs*)
Re-calculate PDF based on specified parameters.

Parameters **`kwargs`** – Should contain keys from one of the group in *self._possible_key_groups*. It may contain additional keys from *self._optional_keys*.

ChromatographyLoadBreakthrough

class `bio_rtd.core.`**`ChromatographyLoadBreakthrough`**(*dt*, *bt_profile_id='ChromatographyLoadBreakthrough'*)
Bases: *bio_rtd.core.ParameterSetList*, *bio_rtd.core.DefaultLoggerLogic*, `abc.ABC`

What parts of the load bind to the column.

Parameters **`bt_profile_id`** (`str`) – Unique identifier of the PDF instance. Used for logs.

Notes

See docstring of *ParameterSetList* for information about key groups.

`assert_and_get_provided_kv_pairs`(***kwargs*)

Parameters **`kwargs`** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **`ValueError`** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

`calc_c_bound`(*f_load*, *c_load*)
Calculates what parts of load bin to the column.

Parameters

- **`f_load`** (`ndarray`) – Load flow rate profile.
- **`c_load`** (`ndarray`) – Load concentration profile. Concentration profile should include only species which bind to the column.

Returns Parts of the load that binds to the column during the load step. *c_bound* has the same shape as *c_load*.

Return type c_bound

Notes

This is default implementation. The user is welcome to override this function in a custom child class.

abstract get_total_bc()

Total binding capacity

Useful for determining column utilization.

Return type `float`

property log

Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent(*parent_id*, *logger*)

Inherit logger from parent.

Parameters

- **parent_id** (`str`) –
- **logger** (*RtdLogger*) –

update_btc_parameters(***kwargs*)

Update binding dynamics for a given set of parameters.

Return type `None`

ParameterSetList

class bio_rtd.core.**ParameterSetList**

Bases: `abc.ABC`

Abstract class for asserting keys in key-value pairs.

Key-value pairs passed to *assert_and_get_provided_kv_pairs* should contain all keys from (at least) one of the key groups in *_possible_key_groups*. The method returns key-value pars with keys from that group and all passed keys that can be also found in *_optional_keys*.

Examples

```
>>> class DummyClass(ParameterSetList):
...     _possible_key_groups = [['par_1'], ['par_2a', 'par_2b']]
...     _optional_keys = ['key_plus_1', 'key_plus_2']
>>>
>>> dc = DummyClass()
>>> dc.assert_and_get_provided_kv_pairs(par_1=1, par_2a=2)
{'par_1': 1}
>>> dc.assert_and_get_provided_kv_pairs(par_2a=1, par_2b=2,
...                                     key_plus_1=3, key_plus_9=2)
{'par_2a': 1, 'par_2b': 2, 'key_plus_1': 3}
>>> dc.assert_and_get_provided_kv_pairs(
...     key_plus_1=1)
Traceback (most recent call last):
KeyError: "Keys ... do not contain any of the required groups: ...
```

assert_and_get_provided_kv_pairs(***kwargs*)

Parameters **kwargs** – Inputs to *calc_pdf(**kwargs)* function

Returns Filtered *kwargs* so the keys contain first possible key group in *_possible_key_groups* and any number of optional keys from *_optional_keys*.

Return type dict

Raises **ValueError** – If ***kwargs* do not contain keys from any of the groups in *_possible_key_groups*.

DefaultLoggerLogic

class bio_rtd.core.**DefaultLoggerLogic**(*logger_parent_id*)

Bases: abc.ABC

Default binding of the *RtdLogger* to a class.

The class holds a reference to a *RtdLogger* logger instance and plants a data tree in the logger.

Parameters **logger_parent_id** (str) – Custom unique id that belongs to the instance of the class. It is used to plant a data tree in the *RtdLogger*.

Examples

```
>>> logger_parent_id = "parent_unit_operation"
>>> l = DefaultLoggerLogic(logger_parent_id)
>>> isinstance(l.log, _logger.DefaultLogger)
True
>>> l.log.e("Error Description")  # log error
Traceback (most recent call last):
RuntimeError: Error Description
>>> l.log.w("Warning Description")  # log waring
Warning Description
>>> l.log.i("Info")  # log info
>>> l.log.log_data = True
>>> l.log.log_level = _logger.RtdLogger.DEBUG
>>> l.log.i_data(l._log_tree, "a", 3)  # store value in logger
>>> l.log.d_data(l._log_tree, "b", 7)  # store at DEBUG level
>>> l.log.get_data_tree(logger_parent_id)["b"]
7
>>> l.log = _logger.StrictLogger()
>>> l.log.w("Warning Info")
Traceback (most recent call last):
RuntimeError: Warning Info
```

Notes

See the documentation of the *RtdLogger*.

property log
Logger.

If logger is not set, then a *DefaultLogger* is instantiated. Setter also plants a data tree into passed logger.

Return type *RtdLogger*

set_logger_from_parent (*parent_id*, *logger*)
Inherit logger from parent.

Parameters

- **parent_id** (str) –
- **logger** (*RtdLogger*) –

INDEX