dclab Documentation

Release 0.58.6

Benedikt Hartmann Eoghan O'Connell
Maik Herbig Maximilian Schlögel Paul Müller
Philipp Rosendahl Raghava Alajangi

Apr 24, 2024

CONTENTS:

This is dclab, a Python library for the post-measurement analysis of real-time deformability cytometry (RT-DC) datasets. This is the documentation of dclab version 0.58.6.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

To install dclab, use one of the following methods:

- **from PyPI:**
  `pip install dclab[all]`
- **from sources:**
  `pip install .[all]`

The extra key `[all]` installs all possible dependencies in any context of RT-DC data analysis. You might prefer to only install a subset of these:

- `pip install dclab`: for the basic dclab functionalities
- `pip install dclab[dcor]`: to *access online data* from DCOR
- `pip install dclab[export]`: for .avi and .fcs export
- `pip install dclab[lme4]`: for *linear mixed effects model analysis* using R/lme4
- `pip install dclab[ml]`: for *machine-learning applications*
- `pip install dclab[s3]`: for accessing .rtdc (HDF5) data on S3-compatible storage
- `pip install dclab[tdms]`: for the (outdated) .tdms file format

You may also combine these dependencies, i.e. `pip install dclab[dcor,s3]` for DCOR and S3 support.

In addition, dclab already comes with code from OpenCV (computation of moments) and scikit-image (computation of contours and points in polygons) to reduce the list of dependencies (these libraries are not required to run dclab).

Note that if you are installing from source or if no binary wheel is available for your platform and Python version, Cython will be installed to build the required dclab extensions. If this process fails, please request a binary wheel for your platform (e.g. Windows 64bit) and Python version (e.g. 3.12) by creating a new issue.

1.2 Use cases

If you are a frequent user of RT-DC, you might run into problems that cannot (yet) be addressed with the graphical user interface Shape-Out. Here is a list of use cases that would motivate an installation of dclab.

- You would like to convert old .tdms-based datasets to the new .rtdc file format, because of enhanced speed in Shape-Out and reduced disk usage. What you are looking for is the command line program *dclab-tdms2rtdc* that comes with dclab. It allows to batch-convert multiple measurements at a time. Note that you should keep the original .tdms files backed-up somewhere, because there might be future improvements or bug fixes from which you would like to benefit. Please note that DCKit offers a graphical user interface for batch conversion from .tdms to .rtdc.
- You have built your own deformability cytometry device and would like to use the powers of dclab to analyze your data. There is user-convenient `RTDCWriter` class that allows you to convert your tabular data to *.rtdc* files.
- You would like to apply a simple set of filters (e.g. polygon filters that you exported from within Shape-Out) to every new measurement you take and apply a custom data analysis pipeline to the filtered data. This is a straightforward Python coding problem with dclab. After reading the basic usage section below, please have a look at the *polygon filter reference*.
- You would like to do advanced statistics or combine your RT-DC analysis with other fancy approaches such as machine-learning. It would be too laborious to do the analysis in Shape-Out, export the data as text files, and then open them in your custom Python script. If your initial analysis step with Shape-Out only involves tasks that can be automated, why not use dclab from the beginning?
- You simulated RT-DC data and plan to import them in Shape-Out for testing. Once you have loaded your data as a numpy array, you can instantiate an `RTDC_Dict` class and then use the `Export` class to create an .rtdc data file.

If you are still unsure about whether to use dclab or not, you might want to look at the *example section*. If you need advice, do not hesitate to create an issue.

1.3 Basic usage

Experimental RT-DC datasets are always loaded with the `new_dataset()` method:

```
import numpy as np
import dclab

# .tdms file format
ds = dclab.new_dataset("/path/to/measurement/Online/M1.tdms")
# .rtdc file format
ds = dclab.new_dataset("/path/to/measurement/M2.rtdc")
# DCOR data
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0")
```

The object returned by `new_dataset()` is always an instance of `RTDCBase`. It enables read-only (!) access to all features (e.g. “deform”, “area_um”) of the dataset. To show all available features, use:

```
print(ds.features)
```

This will list all scalar features (e.g. “area_um” and “deform”) and all non-scalar features (e.g. “contour” and “image”). Scalar features can be filtered by editing the configuration of `ds` and calling `ds.apply_filter()`:

```
# register filtering operations
amin, amax = ds["area_um"].min(), ds["area_um"].max()
ds.config["filtering"]["area_um min"] = (amax + amin) / 2
ds.config["filtering"]["area_um max"] = amax
ds.apply_filter()  # this step is important!
```

This will update the binary, read-only array `ds.filter.all` which can be used to extract the filtered data:

```
area_um_filtered = ds["area_um"][ds.filter.all]
```

It is also possible to create a hierarchy child of this dataset that only contains the filtered data.

```
ds_child = dclab.new_dataset(ds)
```

The hierarchy child `ds_child` is dynamic, i.e. when the filters in `ds` change, then `ds_child` also changes after calling `ds_child.rejuvenate()`. Note that calling *rejuvenate* may be computationally expensive, so dclab does not call it automatically. It is your own responsibility to call it after updating the parent dataset.

Non-scalar features may not (depending on the file format) support fancy indexing (i.e. `ds["image"][ds.filter.all]` might not work. Use a for-loop to extract them.

```
for ii in range(len(ds)):
    image = ds["image"][ii]
    mask = ds["mask"][ii]
    # this is equivalent to ds["bright_avg"][ii]
    bright_avg = np.mean(image[mask])
    print("average brightness of event {}: {:.1f}".format(ii, bright_avg))
```

If you need more information to get started on your particular problem, you might want to check out the *examples section* and the *advanced scripting section*.

1.4 How to cite

If you use dclab in a scientific publication, please cite it with:

> Paul Müller and others (2015), dclab version X.X.X: Python library for the post-measurement analysis of real-time deformability cytometry data sets [Software]. Available at https://github.com/DC-analysis/dclab.

If the journal does not accept `and others`, you can fill in the missing names from the pyproject.toml file.

CHAPTER

TWO

COMMAND-LINE INTERFACE

Note: You may also call all of these command-line functions from within Python. For instance, to compress a dataset, you would use *dclab.cli.compress()*:

```
import dclab.cli
dclab.cli.compress(
    path_out="/path/to/compressed_file.rtdc",
    path_in="/path/to/original.rtdc")
```

For more information please take a look at the code reference of the *CLI submodule*.

2.1 dclab-compress

Create a compressed version of an .rtdc file. This can be used for saving disk space (loss-less compression). The data generated during an experiment is usually not compressed.

```
usage: dclab-compress [-h] [--force] [--version] INPUT OUTPUT
```

required arguments:

- `INPUT` Input path (.rtdc file)
- `OUTPUT` Output path (.rtdc file)

optional arguments:

- `--force` *(disabled by default)* DEPRECATED
- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.2 dclab-condense

Reduce an RT-DC measurement to its scalar-only features (i.e. without *contour*, *image*, *mask*, or *trace*). All available ancillary features are computed.

```
usage: dclab-condense [-h] [--no-ancillary-features] [--no-basin-features]
                      [--version]
                      INPUT OUTPUT
```

required arguments:

- `INPUT` Input path (.tdms or .rtdc file)
- `OUTPUT` Output path (.rtdc file)

optional arguments:

- `--no-ancillary-features` *(disabled by default)* Do not compute expensive ancillary features such as volume
- `--no-basin-features` *(disabled by default)* Do not store basin-based feature data from the input file in the output file
- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.3 dclab-join

Join two or more RT-DC measurements. This will produce one larger .rtdc file. The meta data of the dataset that was recorded earliest will be used in the output file. Please only join datasets that were recorded in the same measurement run.

```
usage: dclab-join [-h] -o OUTPUT [--version] [INPUT ...]
```

required arguments:

- `INPUT` Input paths (.tdms or .rtdc files)
- `OUTPUT` Output path (.rtdc file)

optional arguments:

- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.4 dclab-repack

Repack an .rtdc file. The difference to dclab-compress is that no logs are added. Other logs can optionally be stripped away. Repacking also gets rid of old clutter data (e.g. previous metadata stored in the HDF5 file).

```
usage: dclab-repack [-h] [--strip-basins] [--strip-logs] [--version]
                    INPUT OUTPUT
```

required arguments:

- `INPUT` Input path (.rtdc file)
- `OUTPUT` Output path (.rtdc file)

optional arguments:

- `--strip-basins` *(disabled by default)* Do not copy any basin information to the output file.
- `--strip-logs` *(disabled by default)* Do not copy any logs to the output file.
- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.5 dclab-split

Split an RT-DC measurement file (.tdms or .rtdc) into multiple smaller .rtdc files.

```
usage: dclab-split [-h] [--path_out PATH_OUT] [--split-events SPLIT_EVENTS]
                   [--include-empty-boundary-images] [--version]
                   PATH_IN
```

required arguments:

- `PATH_IN` Input path (.tdms or .rtdc file)

optional arguments:

- `--path_out` *(default: SAME)* Output directory (defaults to same directory)
- `--split-events` *(default: 10000)* Maximum number of events in each output file
- `--include-empty-boundary-images` *(disabled by default)* In old versions of Shape-In, the first or last images were sometimes not stored in the resulting .avi file. In dclab, such images are represented as zero-valued images. Set this option, if you wish to include these events with empty image data.
- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.6 dclab-tdms2rtdc

Convert RT-DC .tdms files to the hdf5-based .rtdc file format. Note: Do not delete original .tdms files after conversion. The conversion might be incomplete.

```
usage: dclab-tdms2rtdc [-h] [--compute-ancillary-features]
                       [--include-empty-boundary-images] [--version]
                       TDMS_PATH RTDC_PATH
```

required arguments:

- `TDMS_PATH` Input path (tdms file or folder containing tdms files)
- `RTDC_PATH` Output path (file or folder), existing data will be overridden

optional arguments:

- `--compute-ancillary-features` *(disabled by default)* Compute features, such as volume or emodulus, that are otherwise computed on-the-fly. Use this if you want to minimize analysis time in e.g. Shape-Out. CAUTION: ancillary feature recipes might be subject to change (e.g. if an error is found in the recipe). Disabling this option maximizes compatibility with future versions and allows to isolate the original data.
- `--include-empty-boundary-images` *(disabled by default)* In old versions of Shape-In, the first or last images were sometimes not stored in the resulting .avi file. In dclab, such images are represented as zero-valued images. Set this option, if you wish to include these events with empty image data.
- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

2.7 dclab-verify-dataset

Check experimental datasets for completeness. This command is used e.g. to enforce data integrity with Shape-In. The following exit codes are defined: `0: valid dataset`, `1: alerts encountered`, `2: violations encountered`, `3: alerts and violations`, `4: other error`.

```
usage: dclab-verify-dataset [-h] [--version] PATH
```

required arguments:

- `PATH` Path to experimental dataset

optional arguments:

- `--version` *(default: ==SUPPRESS==)* show program's version number and exit

CHAPTER

THREE

EXAMPLES

3.1 Dataset overview plot

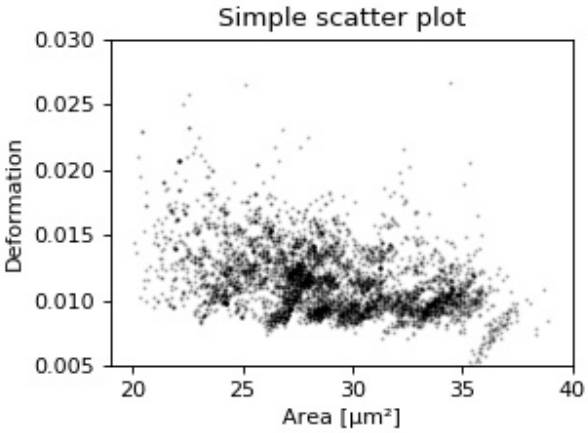
This example demonstrates basic data visualization with dclab and matplotlib. To run this script, download the reference dataset *calibration_beads.rtdc* [RHMG19] and place it in the same directory.

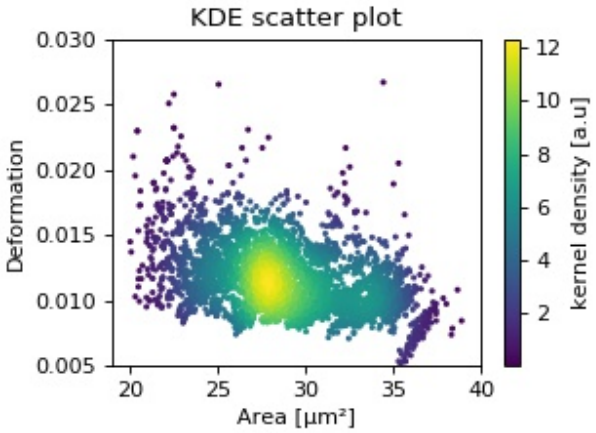
You will find more examples in the *advanced usage* section of this documentation.

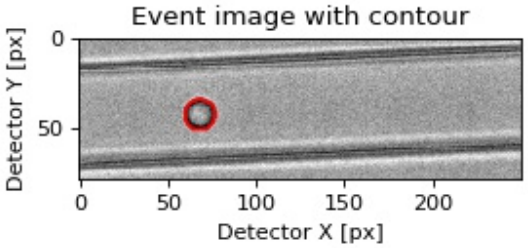
overview_plot.py

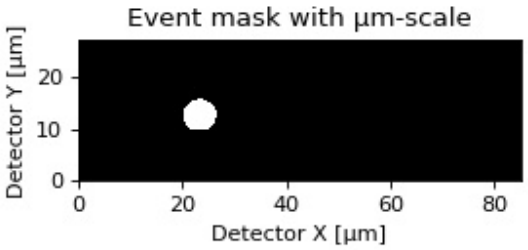
```
import matplotlib.pylab as plt
import numpy as np

import dclab

# Dataset to display
DATASET_PATH = "calibration_beads.rtdc"
# Features for scatter plot
SCATTER_X = "area_um"
SCATTER_Y = "deform"
# Event index to display
EVENT_INDEX = 100

xlabel = dclab.dfn.get_feature_label(SCATTER_X)
ylabel = dclab.dfn.get_feature_label(SCATTER_Y)

ds = dclab.new_dataset(DATASET_PATH)

fig = plt.figure(figsize=(8, 7))


ax1 = plt.subplot(221, title="Simple scatter plot")
ax1.plot(ds[SCATTER_X], ds[SCATTER_Y], "o", color="k", alpha=.2, ms=1)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(ylabel)
ax1.set_xlim(19, 40)
ax1.set_ylim(0.005, 0.03)

ax2 = plt.subplot(222, title="KDE scatter plot")
sc = ax2.scatter(ds[SCATTER_X], ds[SCATTER_Y],
                 c=ds.get_kde_scatter(xax=SCATTER_X,
```

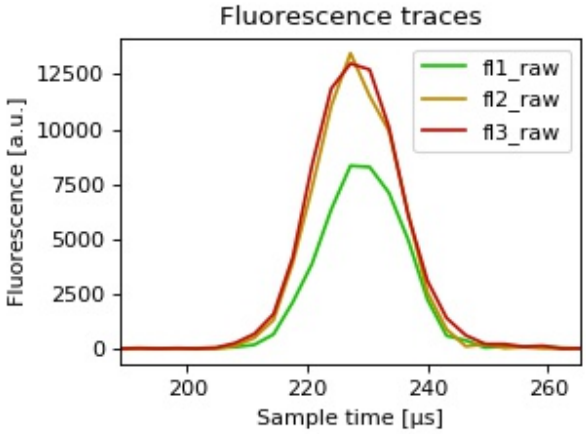
(continues on next page)


Simple scatter plot
Deformation
Area [µm²]
0.030
0.025
0.020
0.015
0.010
0.005
20
25
30
35
40


KDE scatter plot
Deformation
Area [µm²]
kernel density [a.u]
0.030
0.025
0.020
0.015
0.010
0.005
20
25
30
35
40
12
10
8
6
4
2


Event image with contour
Detector Y [px]
Detector X [px]
0
50
0
50
100
150
200


Event mask with µm-scale
Detector Y [µm]
Detector X [µm]
20
10
0
0
20
40
60
80


Fluorescence traces
Fluorescence [a.u.]
Sample time [µs]
fl1_raw
fl2_raw
fl3_raw
12500
10000
7500
5000
2500
0
200
220
240
260

(continued from previous page)

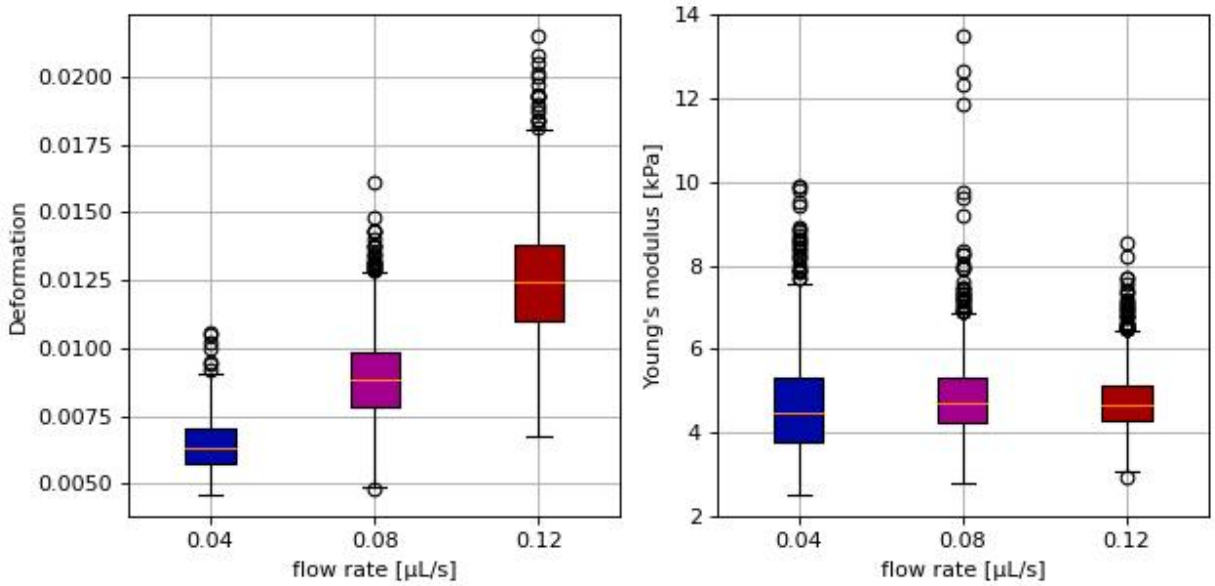
```
                                    yax=SCATTER_Y,
                                    kde_type="multivariate"),
                s=3)
plt.colorbar(sc, label="kernel density [a.u]", ax=ax2)
ax2.set_xlabel(xlabel)
ax2.set_ylabel(ylabel)
ax2.set_xlim(19, 40)
ax2.set_ylim(0.005, 0.03)

ax3 = plt.subplot(425, title="Event image with contour")
ax3.imshow(ds["image"][EVENT_INDEX], cmap="gray")
ax3.plot(ds["contour"][EVENT_INDEX][:, 0],
         ds["contour"][EVENT_INDEX][:, 1],
         c="r")
ax3.set_xlabel("Detector X [px]")
ax3.set_ylabel("Detector Y [px]")

ax4 = plt.subplot(427, title="Event mask with µm-scale")
pxsize = ds.config["imaging"]["pixel size"]
ax4.imshow(ds["mask"][EVENT_INDEX],
           extent=[0, ds["mask"].shape[2] * pxsize,
                   0, ds["mask"].shape[1] * pxsize],
           cmap="gray")
ax4.set_xlabel("Detector X [µm]")
ax4.set_ylabel("Detector Y [µm]")

ax5 = plt.subplot(224, title="Fluorescence traces")
flsamples = ds.config["fluorescence"]["samples per event"]
flrate = ds.config["fluorescence"]["sample rate"]
fltime = np.arange(flsamples) / flrate * 1e6
# here we plot "fl?_raw"; you may also plot "fl?_med"
ax5.plot(fltime, ds["trace"]["fl1_raw"][EVENT_INDEX],
         c="#15BF00", label="fl1_raw")
ax5.plot(fltime, ds["trace"]["fl2_raw"][EVENT_INDEX],
         c="#BF8A00", label="fl2_raw")
ax5.plot(fltime, ds["trace"]["fl3_raw"][EVENT_INDEX],
         c="#BF0C00", label="fl3_raw")
ax5.legend()
ax5.set_xlim(ds["fl1_pos"][EVENT_INDEX] - 2*ds["fl1_width"][EVENT_INDEX],
             ds["fl1_pos"][EVENT_INDEX] + 2*ds["fl1_width"][EVENT_INDEX])
ax5.set_xlabel("Event time [µs]")
ax5.set_ylabel("Fluorescence [a.u.]")

plt.tight_layout()

plt.show()
```

3.2 Young's modulus computation from data on DCOR

This example reproduces the lower right subplot of figure 10 in [Her17]. It illustrates how the Young's modulus of elastic beads can be retrieved correctly (independent of the flow rate, with correction for pixelation and shear-thinning) using the area-deformation look-up table implemented in dclab (right plot). For comparison, the flow-rate-dependent deformation is also shown (left plot).

Note that this example uses the 'buyukurganci-2022' model for computing the viscosity, which was introduced in dclab 0.48.0.

The dataset is loaded directly from DCOR and thus an active internet connection is required for this example.



emodulus_dcor.py

```
import dclab
import matplotlib.pylab as plt

# The dataset is also available on figshare
# (https://doi.org/10.6084/m9.figshare.12721436.v1), but we
# are accessing it through the DCOR API, because we do not
# have the time to download the entire dataset. The dataset
# name is figshare-12721436-v1. These are the resource IDs:
ds_loc = ["e4d59480-fa5b-c34e-0001-46a944afc8ea",
          "2cea205f-2d9d-26d0-b44c-0a11d5379152",
          "2cd67437-a145-82b3-d420-45390f977a90",
          ]
ds_list = []  # list of opened datasets
labels = []  # list of flow rate labels

# load the data
for loc in ds_loc:
    ds = dclab.new_dataset(loc)
    labels.append("{:.2f}".format(ds.config["setup"]["flow rate"]))
```

(continues on next page)

(continued from previous page)

```
    # emodulus computation
    ds.config["calculation"]["emodulus lut"] = "LE-2D-FEM-19"
    ds.config["calculation"]["emodulus medium"] = ds.config["setup"]["medium"]
    ds.config["calculation"]["emodulus temperature"] = \
        ds.config["setup"]["temperature"]
    ds.config["calculation"]["emodulus viscosity model"] = 'buyukurganci-2022'
    # filtering
    ds.config["filtering"]["area_ratio min"] = 1.0
    ds.config["filtering"]["area_ratio max"] = 1.1
    ds.config["filtering"]["deform min"] = 0
    ds.config["filtering"]["deform max"] = 0.035
    # This option will remove "nan" events that appear in the "emodulus"
    # feature. If you are not working with DCOR, this might lead to a
    # longer computation time, because all available features are
    # computed locally. For data on DCOR, this computation already has
    # been done.
    ds.config["filtering"]["remove invalid events"] = True
    ds.apply_filter()
    # Create a hierarchy child for convenience reasons
    # (Otherwise we would have to do e.g. ds["deform"][ds.filter.all]
    # everytime we need to access a feature)
    ds_list.append(dclab.new_dataset(ds))

# plot
fig = plt.figure(figsize=(8, 4))

# box plot for deformation
ax1 = plt.subplot(121)
ax1.set_ylabel(dclab.dfn.get_feature_label("deform"))
data_deform = [di["deform"] for di in ds_list]
# Uncomment this line if you are not filtering invalid events (above)
# data_deform = [d[~np.isnan(d)] for d in data_deform]
bplot1 = ax1.boxplot(data_deform,
                     vert=True,
                     patch_artist=True,
                     labels=labels,
                     )

# box plot for Young's modulus
ax2 = plt.subplot(122)
ax2.set_ylabel(dclab.dfn.get_feature_label("emodulus"))
data_emodulus = [di["emodulus"] for di in ds_list]
# Uncomment this line if you are not filtering invalid events (above)
# data_emodulus = [d[~np.isnan(d)] for d in data_emodulus]
bplot2 = ax2.boxplot(data_emodulus,
                     vert=True,
                     patch_artist=True,
                     labels=labels,
                     )

# colors
colors = ["#0008A5", "#A5008D", "#A50100"]
```

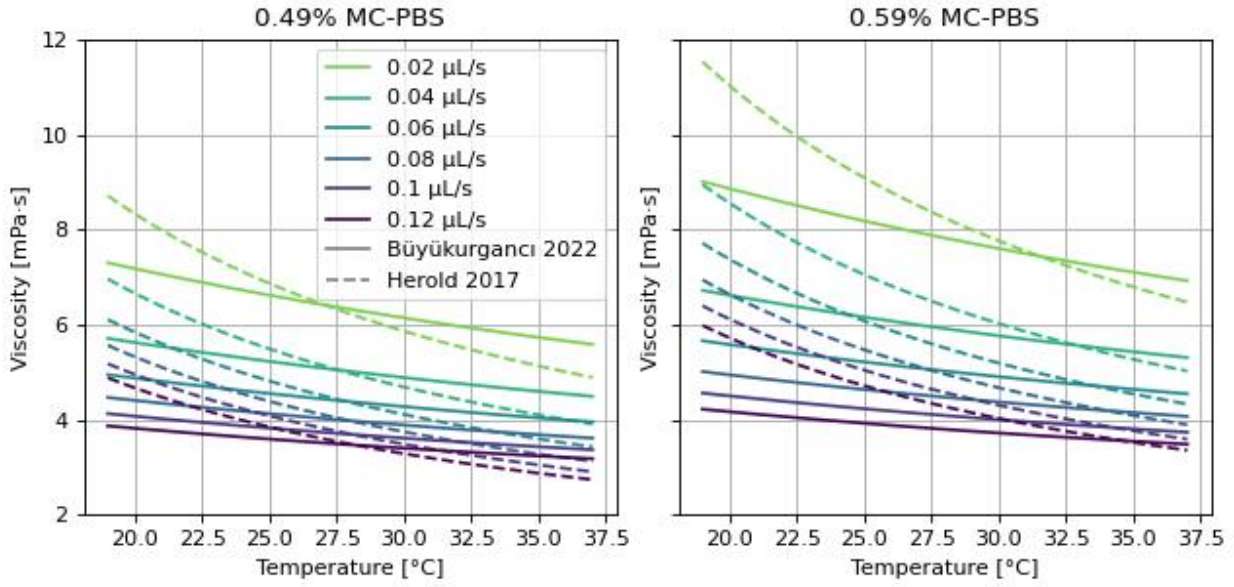
(continues on next page)

(continued from previous page)

```
for bplot in (bplot1, bplot2):
    for patch, color in zip(bplot['boxes'], colors):
        patch.set_facecolor(color)

# axes
for ax in [ax1, ax2]:
    ax.grid()
    ax.set_xlabel("flow rate [µL/s]")

plt.tight_layout()
plt.show()
```

3.3 Viscosity models for Young's modulus estimation

This example visualizes the different viscosity models for the MC-PBS media implemented in dclab. We reproduce the lower left part of figure 3 in [RB23] (channel width is 20 µm).



viscosity_models.py

```
import matplotlib.pylab as plt
import matplotlib.lines as mlines
from matplotlib import cm
import numpy as np

from dclab.features.emodulus import viscosity


visc_res = {}

for medium in ["0.49% MC-PBS", "0.59% MC-PBS"]:
```

(continues on next page)

(continued from previous page)

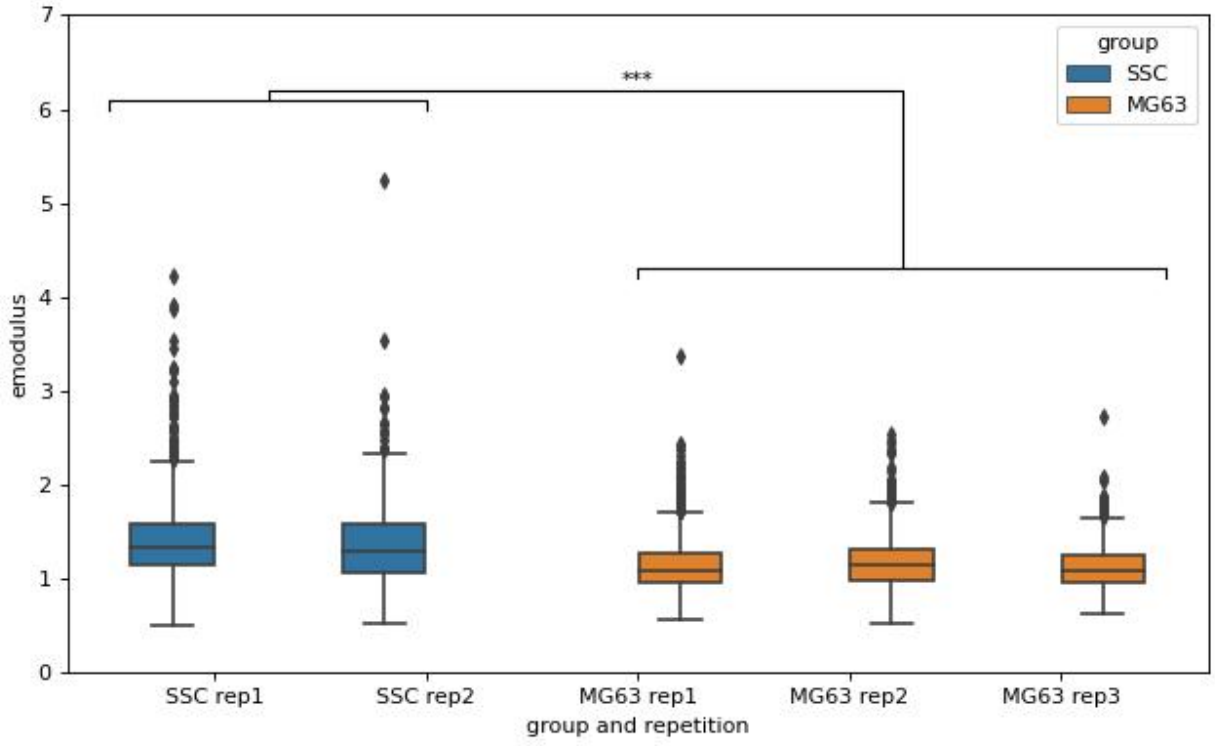
```
    visc_her = {}
    visc_buy = {}

    kwargs = {
        "medium": medium,
        "channel_width": 20.0,
        "temperature": np.linspace(19, 37, 100, endpoint=True),
    }

    flow_rate = np.arange(0.02, 0.13, 0.02)

    for fr in flow_rate:
        visc_her[fr] = viscosity.get_viscosity_mc_pbs_herold_2017(
            flow_rate=fr, **kwargs)
        visc_buy[fr] = viscosity.get_viscosity_mc_pbs_buyukurganci_2022(
            flow_rate=fr, **kwargs)

    visc_res[medium] = [visc_her, visc_buy]


fig, axes = plt.subplots(1, 2, figsize=(8, 4), sharey="all", sharex="all")
colors = [cm.get_cmap("viridis")(x) for x in np.linspace(.8, 0,
                                                         len(flow_rate))]

for ii, medium in enumerate(visc_res):
    visc_her, visc_buy = visc_res[medium]
    ax = axes.flatten()[ii]
    ax.set_title(medium)

    for jj, fr in enumerate(flow_rate):
        ax.plot(kwargs["temperature"], visc_her[fr], color=colors[jj], ls="--")
        ax.plot(kwargs["temperature"], visc_buy[fr], color=colors[jj], ls="-")

    ax.set_xlabel("Temperature [°C]")
    ax.set_ylabel("Viscosity [mPa·s]")
    ax.grid()
    ax.set_ylim(2, 12)

handles = []
for jj, fr in enumerate(flow_rate):
    handles.append(
        mlines.Line2D([], [], color=colors[jj], label=f'{fr:.4g} µL/s'))
handles.append(
    mlines.Line2D([], [], color='gray', label='Büyükurgancı 2022'))
handles.append(
    mlines.Line2D([], [], color='gray', ls="--", label='Herold 2017'))
axes[0].legend(handles=handles)


plt.tight_layout()
plt.show()
```

3.4 lme4: Linear mixed-effects models

We would like to quantify the difference between human skeletal stem cells (SSC) and the human osteosarcoma cell line MG-63 (which is often used as a model system for SSCs) using a likelihood ratio test based on LMM.

This example illustrates a basic LMM analysis. The data are loaded from DCOR ([XRM+20], DCOR:figshare-11662773-v2). We treat SSC as our "treatment" and MG-63 as our "control" group. These are just names that remind us that we are comparing one type of sample against another type.

We are interested in the p-value, which is 0.01256 for deformation. We repeat the analysis with area (0.0002183) and Young's modulus (0.0002771). The p-values indicate that MG-63 (mean elastic modulus 1.26 kPa) cells are softer than SSCs (mean elastic modulus 1.54 kPa). The figure reproduces the last subplot of figure 6b im [HMMO18].



lme4_lmer.py

```
import dclab
from dclab import lme4

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


# https://dcor.mpl.mpg.de/dataset/figshare-11662773-v2
# SSC_16uls_rep1_20150611.rtdc
ds_ssc_rep1 = dclab.new_dataset("86cc5a47-364b-cf58-f9e3-cc114dd38e55")
# SSC_16uls_rep2_20150611.rtdc
ds_ssc_rep2 = dclab.new_dataset("ab95c914-0311-6a46-4eba-8fabca7d27d6")
```

(continues on next page)

(continued from previous page)

```
# MG63_pure_16uls_rep1_20150421.rtdc
ds_mg63_rep1 = dclab.new_dataset("42cb33d4-2f7c-3c22-88e1-b9102d64d7e9")
# MG63_pure_16uls_rep2_20150422.rtdc
ds_mg63_rep2 = dclab.new_dataset("a4a98fcb-1de1-1048-0efc-b0a84d4ab32e")
# MG63_pure_16uls_rep3_20150422.rtdc
ds_mg63_rep3 = dclab.new_dataset("0a8096ce-ea7a-e36d-1df3-42c7885cd71c")

datasets = [ds_ssc_rep1, ds_ssc_rep2, ds_mg63_rep1, ds_mg63_rep2, ds_mg63_rep3]
for ds in datasets:
    # perform filtering
    ds.config["filtering"]["area_ratio min"] = 0
    ds.config["filtering"]["area_ratio max"] = 1.05
    ds.config["filtering"]["area_um min"] = 120
    ds.config["filtering"]["area_um max"] = 550
    ds.config["filtering"]["deform min"] = 0
    ds.config["filtering"]["deform max"] = 0.1
    ds.apply_filter()
    # enable computation of Young's modulus
    ds.config["calculation"]["emodulus lut"] = "LE-2D-FEM-19"
    ds.config["calculation"]["emodulus medium"] = "CellCarrier"
    ds.config["calculation"]["emodulus temperature"] = 23.0
    ds.config["calculation"]["emodulus viscosity model"] = 'buyukurganci-2022'

# setup lme4 analysis
rlme4 = lme4.Rlme4(model="lmer")
rlme4.add_dataset(ds_ssc_rep1, group="treatment", repetition=1)
rlme4.add_dataset(ds_ssc_rep2, group="treatment", repetition=2)
rlme4.add_dataset(ds_mg63_rep1, group="control", repetition=1)
rlme4.add_dataset(ds_mg63_rep2, group="control", repetition=2)
rlme4.add_dataset(ds_mg63_rep3, group="control", repetition=3)

# perform analysis for deformation
for feat in ["area_um", "deform", "emodulus"]:
    res = rlme4.fit(feature=feat)
    print("Results for {}:".format(feat))
    print("  p-value", res["anova p-value"])
    print("  mean of MG-63", res["fixed effects intercept"])
    print("  fixed effect size", res["fixed effects treatment"])

# prepare for plotting
df = pd.DataFrame()
for ds in datasets:
    group = ds.config["experiment"]["sample"].split()[0]
    rep = ds.config["experiment"]["sample"].split()[-1]
    dfi = pd.DataFrame.from_dict(
        {"area_m": ds["area_um"][ds.filter.all],
         "deform": ds["deform"][ds.filter.all],
         "emodulus": ds["emodulus"][ds.filter.all],
         "group and repetition": [group + " " + rep] * ds.filter.all.sum(),
         "group": [group] * ds.filter.all.sum(),
         })
    df = df.append(dfi)
```

(continues on next page)

(continued from previous page)

```
# plot
fig = plt.figure(figsize=(8, 5))
ax = sns.boxplot(x="group and repetition", y="emodulus", data=df, hue="group")
# note that `res` is still the result for "emodulus"
numstars = sum([res["anova p-value"] < .05,
                res["anova p-value"] < .01,
                res["anova p-value"] < .001,
                res["anova p-value"] < .0001])
# significance bars
h = .1
y1 = 6
y2 = 4.2
y3 = 6.2
ax.plot([-.5, -.5, 1, 1], [y1, y1+h, y1+h, y1], lw=1, c="k")
ax.plot([2, 2, 4.5, 4.5], [y2, y2+h, y2+h, y2], lw=1, c="k")
ax.plot([.25, .25, 3.25, 3.25], [y1+h, y1+2*h, y1+2*h, y2+h], lw=1, c="k")
ax.text(2, y3, "*"*numstars, ha='center', va='bottom', color="k")
ax.set_ylim(0, 7)

plt.tight_layout()
plt.show()
```

3.5 lme4: Generalized linear mixed-effects models with differential deformation

This example illustrates how to perform a differential feature (including reservoir data) GLMM analysis. The example data are taken from DCOR ([XRM+20], DCOR:figshare-11662773-v2). As in the *previous example*, we treat SSC as our "treatment" and MG-63 as our "control" group.

The p-value for the differential deformation is magnitudes lower than the p-value for the (non-differential) deformation in the previous example. This indicates that there is a non-negligible initial deformation of the cells in the reservoir.

lme4_glmer_diff.py

```
from dclab import lme4, new_dataset

# https://dcor.mpl.mpg.de/dataset/figshare-11662773-v2
datasets = [
    # SSC channel
    [new_dataset("86cc5a47-364b-cf58-f9e3-cc114dd38e55"), "treatment", 1],
    [new_dataset("ab95c914-0311-6a46-4eba-8fabca7d27d6"), "treatment", 2],
    # SSC reservoir
    [new_dataset("761ab515-0416-ede8-5137-135c1682580c"), "treatment", 1],
    [new_dataset("3b83d47b-d860-4558-51d6-dcc524f5f90d"), "treatment", 2],
    # MG-63 channel
    [new_dataset("42cb33d4-2f7c-3c22-88e1-b9102d64d7e9"), "control", 1],
    [new_dataset("a4a98fcb-1de1-1048-0efc-b0a84d4ab32e"), "control", 2],
    [new_dataset("0a8096ce-ea7a-e36d-1df3-42c7885cd71c"), "control", 3],
    # MG-63 reservoir
```

(continues on next page)

(continued from previous page)

```
    [new_dataset("56c449bb-b6c9-6df7-6f70-6744b9960980"), "control", 1],
    [new_dataset("387b5ac9-1cc6-6cac-83d1-98df7d687d2f"), "control", 2],
    [new_dataset("7ae49cd7-10d7-ef35-a704-72443bb32da7"), "control", 3],
]

# perform filtering
for ds, _, _ in datasets:
    ds.config["filtering"]["area_ratio min"] = 0
    ds.config["filtering"]["area_ratio max"] = 1.05
    ds.config["filtering"]["area_um min"] = 120
    ds.config["filtering"]["area_um max"] = 550
    ds.config["filtering"]["deform min"] = 0
    ds.config["filtering"]["deform max"] = 0.1
    ds.apply_filter()

# perform LMM analysis for differential deformation
# setup lme4 analysis
rlme4 = lme4.Rlme4(feature="deform")
for ds, group, repetition in datasets:
    rlme4.add_dataset(ds, group=group, repetition=repetition)

# LMM
lmer_result = rlme4.fit(model="lmer")
print("LMM p-value", lmer_result["anova p-value"])  # 0.00000351

# GLMM with log link function
glmer_result = rlme4.fit(model="glmer+loglink")
print("GLMM p-value", glmer_result["anova p-value"])  # 0.000868
```

3.6 ML: Using RT-DC data with tensorflow

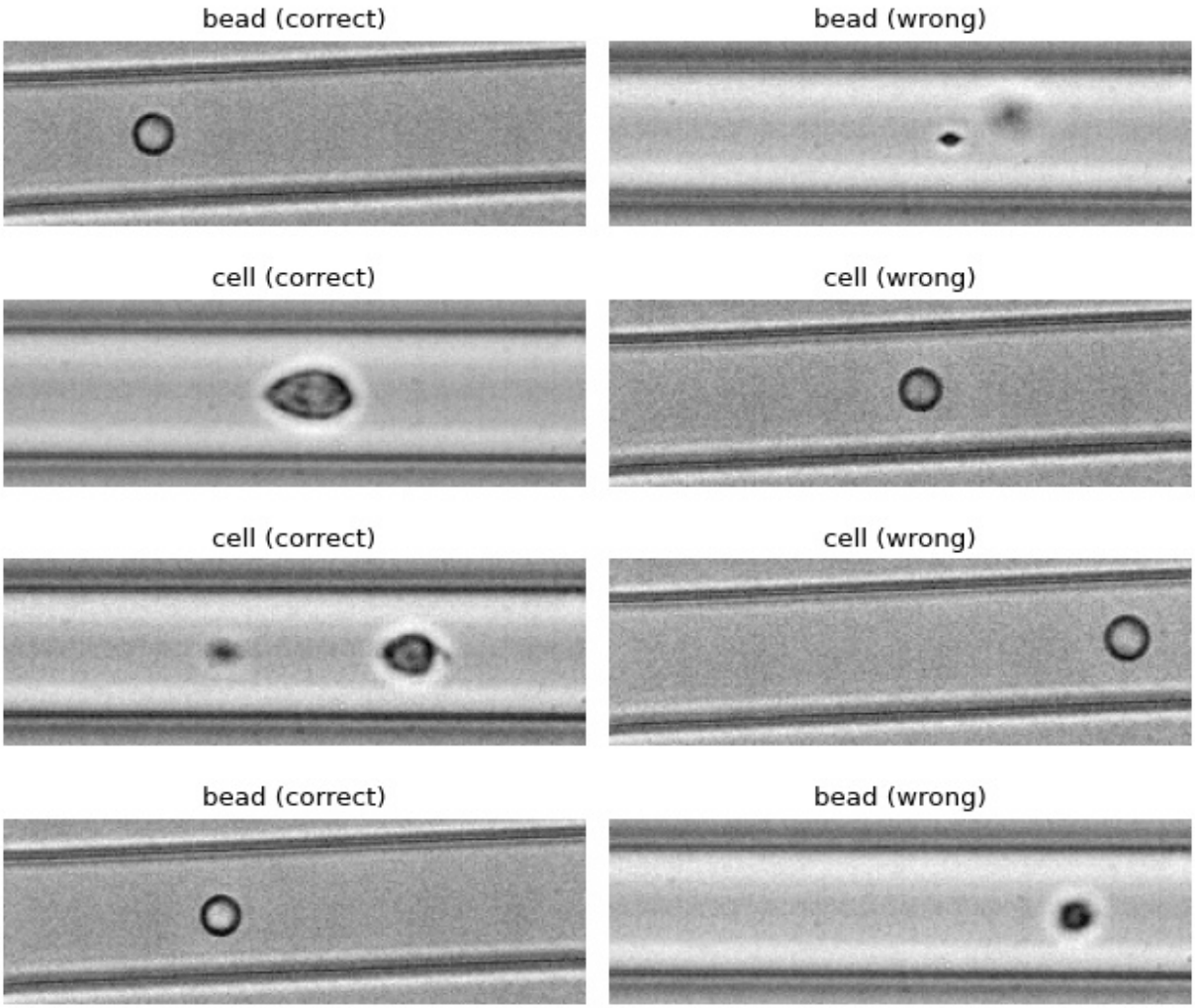
We use tensorflow to distinguish between beads and cells using scalar features only. The example data is taken from a reference dataset on DCOR. The classification accuracy using only the inputs `area_ratio`, `area_um`, `bright_sd`, and `deform` reaches values above 95%.

Warning: This example neglects a lot of important aspects of machine learning with RT-DC data (e.g. brightness normalization) and it is a very easy task (beads are smaller than cells). Thus, this example should only be considered as a technical guide on how tensorflow can be used with RT-DC data.

Note: What happens when you add `"bright_avg"` to the `features` list? Can you explain the result?

Apparently, debris in the cell dataset is classified as beads. We could have gotten around that by filtering the input data before inference. In addition, some beads get classified as cells as well. This is a result of the limited features used for training/inference. Under normal cirumstances, you would investigate other features in order to improve the model prediction.

`ml_tensorflow.py`


bead (correct)
bead (wrong)
cell (correct)
cell (wrong)
cell (correct)
cell (wrong)
bead (correct)
bead (wrong)

```python
import matplotlib.pylab as plt
import numpy as np
import tensorflow as tf
from dclab.rtdc_dataset.feat_anc_ml import hook_tensorflow

tf.random.set_seed(42)  # for reproducibility

# https://dcor.mpl.mpg.de/dataset/figshare-7771184-v2
dcor_ids = ["fb719fb2-bd9f-817a-7d70-f4002af916f0",
            "f7fa778f-6abd-1b53-ae5f-9ce12601d6f8"]
labels = [0, 1]  # 0: beads, 1: cells
features = ["area_ratio", "area_um", "bright_sd", "deform"]

# obtain train and test datasets
train, test = hook_tensorflow.assemble_tf_dataset_scalars(
    dc_data=dcor_ids,  # can also be list of paths or datasets
    labels=labels,
    feature_inputs=features,
    split=.8)

# build the model
model = tf.keras.Sequential(
    layers=[
        tf.keras.layers.Input(shape=(len(features),)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(32),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(2)
    ],
    name="scalar_features"
)

# fit the model to the training data
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])
model.fit(train, epochs=5)

# show accuracy using test data (loss: 0.1139 - accuracy: 0.9659)
model.evaluate(test, verbose=2)

# predict classes of the test data
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
y_test = np.concatenate([y for x, y in test], axis=0)
predict = np.argmax(probability_model.predict(test), axis=1)

# take a few exemplary events from true and false classification
false_cl = np.where(predict != y_test)[0]
true_cl = np.where(predict == y_test)[0]
num_events = min(4, min(len(true_cl), len(false_cl)))

false_images = hook_tensorflow.get_dataset_event_feature(
    dc_data=dcor_ids,
    feature="image",
```

(continues on next page)

(continued from previous page)

```
    tf_dataset_indices=false_cl[:num_events],
    split_index=1,
    split=.8)

true_images = hook_tensorflow.get_dataset_event_feature(
    dc_data=dcor_ids,
    feature="image",
    tf_dataset_indices=true_cl[:num_events],
    split_index=1,
    split=.8)

fig = plt.figure(figsize=(8, 7))

for ii in range(num_events):
    title_true = ("cell" if y_test[true_cl[[ii]]] else "bead") + " (correct)"
    title_false = ("cell" if predict[false_cl[ii]] else "bead") + " (wrong)"
    ax1 = plt.subplot(num_events, 2, 2*ii+1, title=title_true)
    ax2 = plt.subplot(num_events, 2, 2*(ii + 1), title=title_false)
    ax1.axis("off")
    ax2.axis("off")
    ax1.imshow(true_images[ii], cmap="gray")
    ax2.imshow(false_images[ii], cmap="gray")

plt.tight_layout()
plt.show()
```

3.7 ML: Creating built-in models for dclab with tensorflow

The *tensorflow example* already showcased a few convenience functions for machine learning implemented in dclab. In this example, we want to go even further and transform the predictions of an ML model into an *ancillary feature* (which is then globally available in dclab).

A few things are different from the other example:

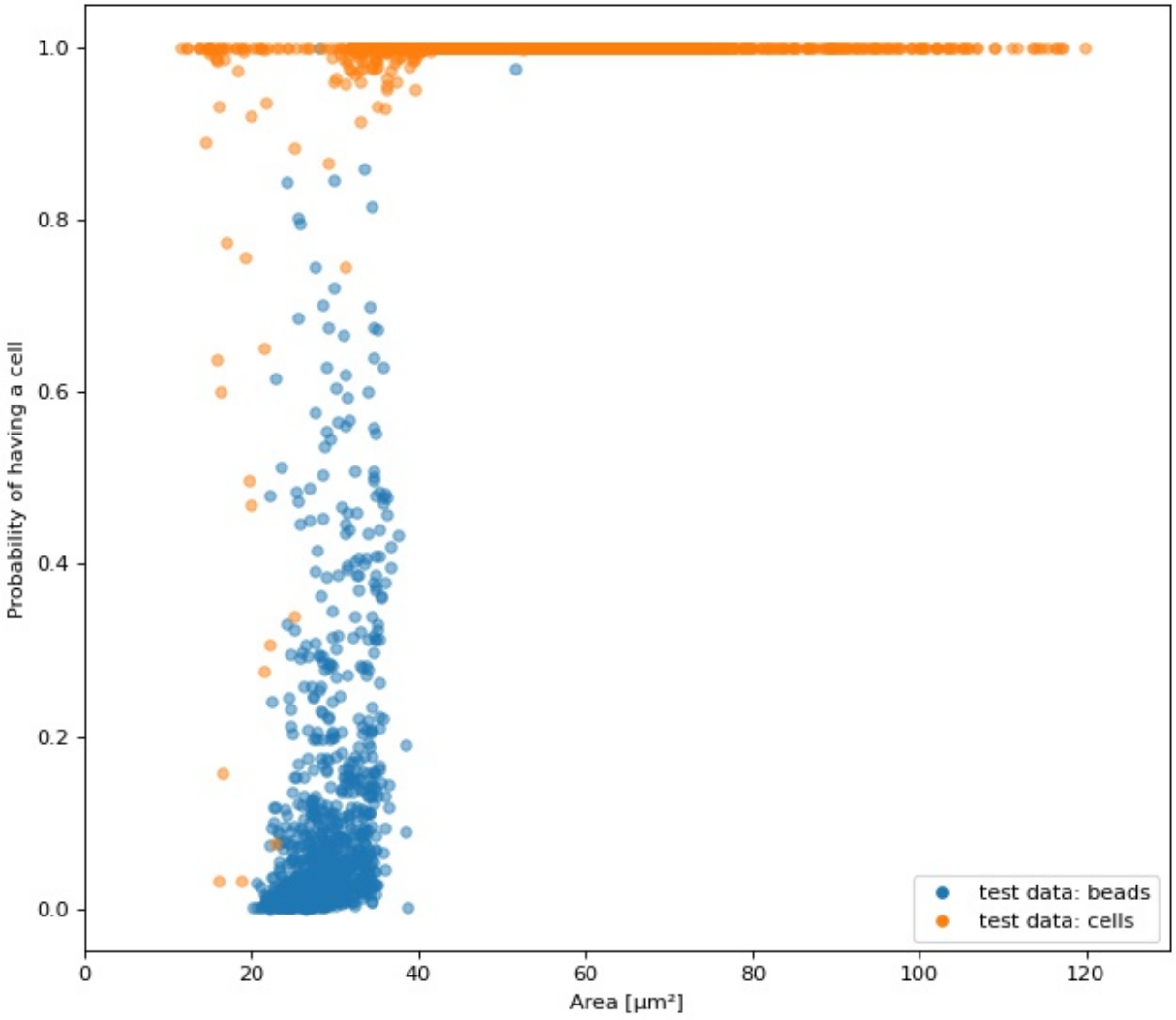
- We rename `model` to `bare_model` to make a clear distinction between the actual ML model (from tensorflow) and the model wrapper (see *Using models in dclab*).
- We turn the two-class problem into a regression problem for one feature only. Consequently, the loss function changes to "binary crossentropy" and for some inexplicable reason we have to train for 20 epochs instead of the previously 5 to achieve convergence in accuracy.
- Finally, and this is the whole point of this example, we register the model as an ancillary feature and perform inference indirectly by simply accessing the `ml_score_cel` feature of the test dataset.

The plot shows the test fraction of the dataset. The x-axis is (arbitrarily) set to area. The y-axis shows the sigmoid (dclab automatically applies a sigmoid activation if it is not present in the final layer; see `dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow.TensorflowModel.predict()`) of the model's output logits.

`ml_builtin.py`

```
import matplotlib.pylab as plt
import numpy as np
import tensorflow as tf
```

(continues on next page)


1.0
0.8
0.6
0.4
0.2
0.0
Probability of having a cell
0
20
40
60
80
100
120
Area [µm²]
test data: beads
test data: cells

(continued from previous page)

```
import dclab
from dclab.rtdc_dataset.feat_anc_ml import hook_tensorflow


tf.random.set_seed(42)  # for reproducibility

# https://dcor.mpl.mpg.de/dataset/figshare-7771184-v2
dcor_ids = ["fb719fb2-bd9f-817a-7d70-f4002af916f0",
            "f7fa778f-6abd-1b53-ae5f-9ce12601d6f8"]
labels = [0, 1]  # 0: beads, 1: cells
features = ["area_ratio", "area_um", "bright_sd", "deform"]

tf_kw = {"dc_data": dcor_ids,
         "split": .8,
         "shuffle": True,
         }

# obtain train and test datasets
train, test = hook_tensorflow.assemble_tf_dataset_scalars(
    labels=labels, feature_inputs=features, **tf_kw)

# build the model
bare_model = tf.keras.Sequential(
    layers=[
        tf.keras.layers.Input(shape=(len(features),)),
        tf.keras.layers.Dense(128),
        tf.keras.layers.Dense(32),
        tf.keras.layers.Dropout(0.3),
        tf.keras.layers.Dense(1)
    ],
    name="scalar_features"
)

# fit the model to the training data
# Note that we did not add a "sigmoid" activation function to the
# final layer and are training with logits here. We also don't
# have to manually add it in a later step, because dclab will
# add it automatically (if it does not exist) before prediction.
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
bare_model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])
bare_model.fit(train, epochs=20)

# show accuracy using test data (loss: 0.0725 - accuracy: 0.9877)
bare_model.evaluate(test, verbose=2)

# register the ancillary feature "ml_score_cel" in dclab
dc_model = hook_tensorflow.TensorflowModel(
    bare_model=bare_model,
    inputs=features,
    outputs=["ml_score_cel"],
    info={
        "description": "Distinguish between cells and beads",
```

(continues on next page)

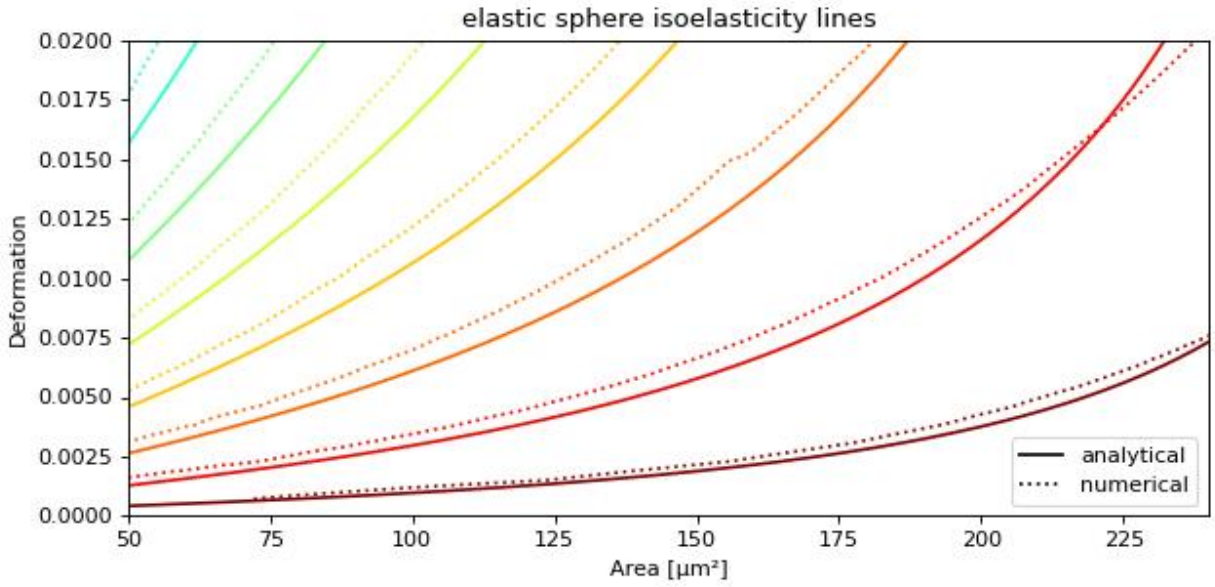
(continued from previous page)

```
        "output labels": ["Probability of having a cell"],
    }
)
dclab.MachineLearningFeature(feature_name="ml_score_cel",
                             dc_model=dc_model)

# Now we are actually done already. The only thing left to do is to
# visualize the prediction for the test-fraction of our dataset.
# This involves a bit of data shuffling (obtaining the dataset indices
# from the "index" feature (which starts at 1 and not 0) and creating
# hierarchy children after applying the corresponding manual filters)
# which is less complicated than it looks.

# create dataset hierarchy children for bead and cell test data
bead_train_indices = hook_tensorflow.get_dataset_event_feature(
    feature="index", dc_data_indices=[0], split_index=0, **tf_kw)
ds_bead = dclab.new_dataset(dcor_ids[0])
ds_bead.filter.manual[np.array(bead_train_indices) - 1] = False
ds_bead.apply_filter()
ds_bead_test = dclab.new_dataset(ds_bead)  # hierarchy child with test fraction

cell_train_indices = hook_tensorflow.get_dataset_event_feature(
    feature="index", dc_data_indices=[1], split_index=0, **tf_kw)
ds_cell = dclab.new_dataset(dcor_ids[1])
ds_cell.filter.manual[np.array(cell_train_indices) - 1] = False
ds_cell.apply_filter()
ds_cell_test = dclab.new_dataset(ds_cell)  # hierarchy child with test fraction

fig = plt.figure(figsize=(8, 7))
ax = plt.subplot(111)

plt.plot(ds_bead_test["area_um"], ds_bead_test["ml_score_cel"], ".",
         ms=10, alpha=.5, label="test data: beads")
plt.plot(ds_cell_test["area_um"], ds_cell_test["ml_score_cel"], ".",
         ms=10, alpha=.5, label="test data: cells")
leg = plt.legend()
for lh in leg.legendHandles:
    lh._legmarker.set_alpha(1)

ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("ml_score_cel"))
ax.set_xlim(0, 130)

plt.tight_layout()
plt.show()
```

3.8 Plotting isoelastics

This example illustrates how to plot dclab isoelastics by reproducing figure 3 (lower left) of [MMM+17].

Warning: This example does not work anymore since dclab 0.46.0, because the isoelasticity lines of the analytical model have different Young's moduli than the ones of the revised LE-2D-FEM-19 model. For the sake of completeness, we keep this example here. If you would like to extract lines at specific Young's moduli, please take a look at the next example.



isoelastics.py

```python
import matplotlib.pylab as plt
import matplotlib.lines as mlines
from matplotlib import cm
import numpy as np

import dclab

# parameters for isoelastics
kwargs = {"col1": "area_um",  # x-axis
          "col2": "deform",  # y-axis
          "channel_width": 20,  # [um]
          "flow_rate": 0.04,  # [ul/s]
          "viscosity": 15,  # [mPa s]
          "add_px_err": False  # no pixelation error
          }

isos = dclab.isoelastics.get_default()
analy = isos.get(lut_identifier="LE-2D-ana-18", **kwargs)
numer = isos.get(lut_identifier="LE-2D-FEM-19", **kwargs)

```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(8, 4))
ax = plt.subplot(111, title="elastic sphere isoelasticity lines")
colors = [cm.get_cmap("jet")(x) for x in np.linspace(0, 1, len(analy))]
for aa, nn, cc in zip(analy, numer, colors):
    ax.plot(aa[:, 0], aa[:, 1], color=cc)
    ax.plot(nn[:, 0], nn[:, 1], color=cc, ls=":")

line = mlines.Line2D([], [], color='k', label='analytical')
dotted = mlines.Line2D([], [], color='k', ls=":", label='numerical')
ax.legend(handles=[line, dotted])

ax.set_xlim(50, 240)
ax.set_ylim(0, 0.02)
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))

plt.tight_layout()
plt.show()
```

3.9 Plotting custom isoelastics

This example illustrates how to extract custom isoelasticity lines from the dclab look-up tables by reproducing figure 3 (right) of [WRM+22].

Note that at the boundary of the support of a look-up table, the isoelasticity lines may break away in perpendicular directions. The underlying reason is that the look-up table is first mapped onto a grid from which the constant isoelasticity lines are extracted. Since the Young's modulus values are linearly interpolated from the LUT onto that grid, there can be inaccuracies for pixels that are at the LUT boundary.

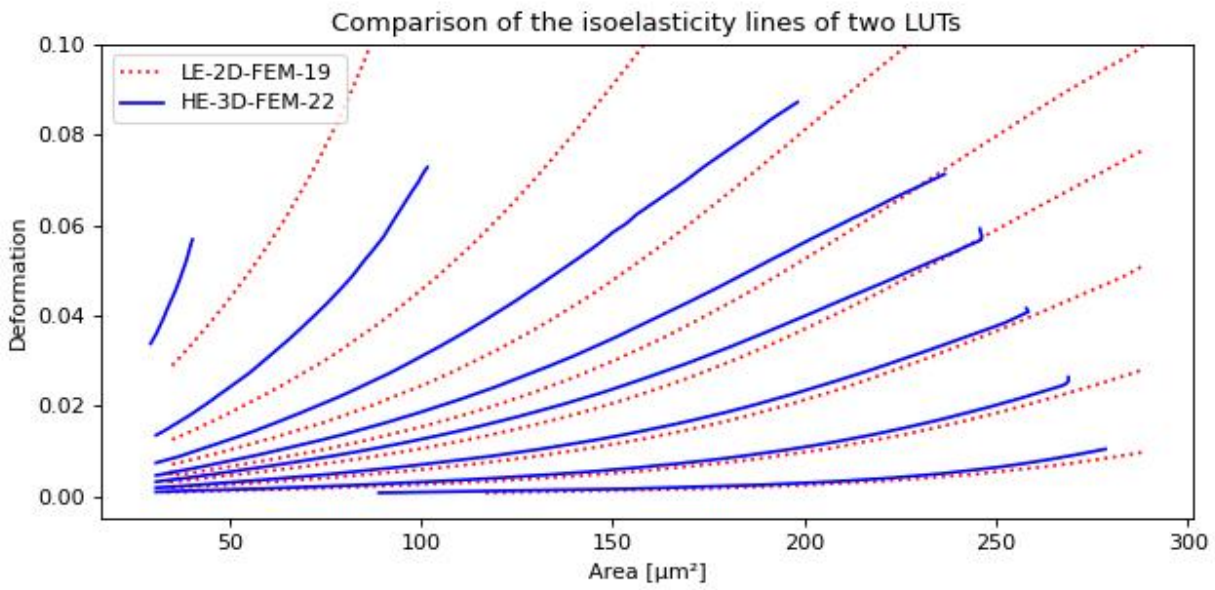
An elaborate way of getting rid of these inaccuracies (and this is how the isoelasticity lines for dclab are extracted), is to extend the LUT by fitting a polynomial to isoelasticity lines which are well-defined within the LUT and extrapolating these lines beyond the boundary of the LUT. This technique is documented in the *scripts* directory of the dclab repository.

A quicker and much less elaborate way of getting around this issue is to simply crop the individual isoelasticity lines where necessary.

`isoelastics_custom.py`

```
import matplotlib.pylab as plt
import numpy as np
import skimage

import dclab
from dclab.features import emodulus


colors = ["r", "b"]
linestyles = [":", "-"]

plt.figure(figsize=(8, 4))
ax = plt.subplot(111,
```

(continues on next page)



(continued from previous page)

```
                  title="Comparison of the isoelasticity lines of two LUTs")

grid_sie = 250

for ii, lut_name in enumerate(["LE-2D-FEM-19", "HE-3D-FEM-22"]):
    area_um = np.linspace(0, 350, grid_sie, endpoint=True)
    deform = np.linspace(0, 0.2, grid_sie, endpoint=True)
    area_um_grid, deform_grid = np.meshgrid(area_um, deform, indexing="ij")

    emod = emodulus.get_emodulus(area_um=area_um_grid,
                                 deform=deform_grid,
                                 medium=6.0,
                                 channel_width=20,
                                 flow_rate=0.04,
                                 px_um=0,
                                 temperature=None,
                                 visc_model=None,
                                 lut_data=lut_name)

    levels = [0.5, 0.75, 1.0, 1.25, 1.5, 2.0, 3.0, 6.0]
    for level in levels:
        conts = skimage.measure.find_contours(emod, level=level)
        if not conts:
            continue
        # get the longest one
        idx = np.argmax([len(cc) for cc in conts])
        cc = conts[idx]
        # remove nan values
        cc = cc[~np.isnan(np.sum(cc, axis=1))]
        # scale isoelastics back
        cc_sc = np.copy(cc)
```

(continues on next page)

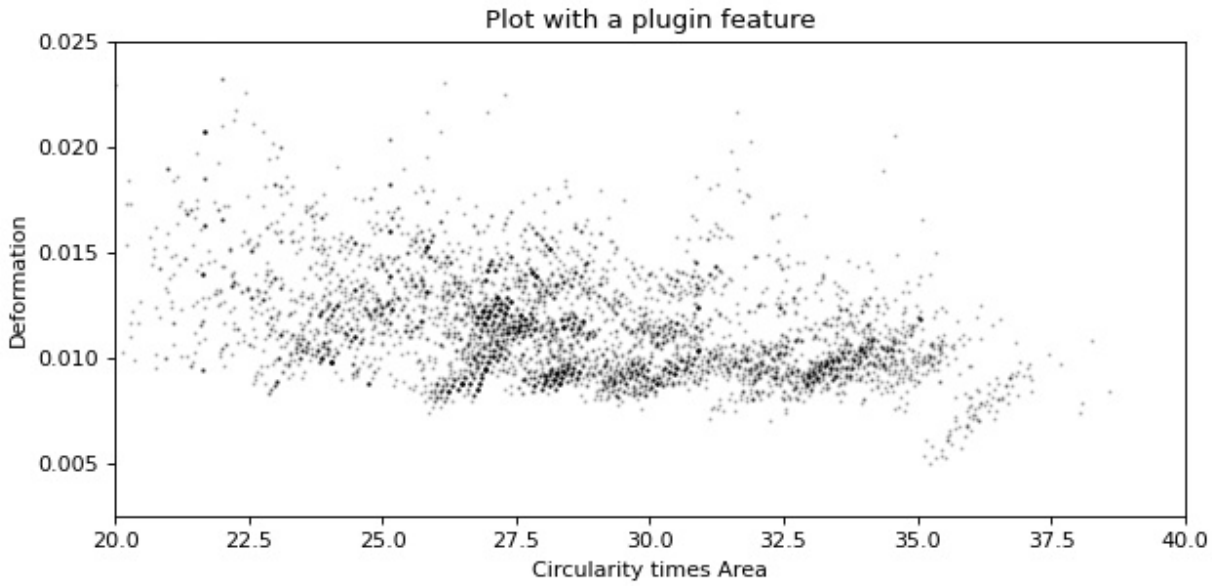
(continued from previous page)

```
        cc_sc[:, 0] = cc[:, 0] / grid_sie * 350
        cc_sc[:, 1] = cc[:, 1] / grid_sie * 0.2
        plt.plot(cc_sc[:, 0], cc_sc[:, 1],
                 color=colors[ii],
                 ls=linestyles[ii],
                 label=lut_name if level == levels[0] else None)

ax.set_ylim(-0.005, 0.1)
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
plt.legend()
plt.tight_layout()
plt.show()
```

3.10 Working with plugin features

This example shows how to load a user-defined plugin feature recipe in dclab and use it in a scatter plot.

Please also download the `plugin_example.py` file for this example.



`plugin_usage.py`

```
import pathlib

import matplotlib.pyplot as plt

import dclab


plugin_path = pathlib.Path(__file__).parent
```

(continues on next page)

(continued from previous page)

```

# load a single plugin feature
dclab.load_plugin_feature(plugin_path / "plugin_example.py")

# load some data from DCOR
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0")

# access the features
circ_per_area = ds["circ_per_area"]
circ_times_area = ds["circ_times_area"]

# create a plot with a plugin feature
plt.figure(figsize=(8, 4))
xlabel = dclab.dfn.get_feature_label("circ_times_area")
ylabel = dclab.dfn.get_feature_label("deform")

ax1 = plt.subplot(title="Plot with a plugin feature")
ax1.plot(ds["circ_times_area"], ds["deform"],
         "o", color="k", alpha=.2, ms=1)
ax1.set_xlabel(xlabel)
ax1.set_ylabel(ylabel)
ax1.set_xlim(20, 40)
ax1.set_ylim(0.0025, 0.025)

plt.tight_layout()
plt.show()
```

CHAPTER

FOUR

ADVANCED USAGE

This section motivates the design of dclab and highlights useful built-in functionalities.

4.1 Notation

When coding with dclab, you should be aware of the following definitions and design principles.

4.1.1 Events

An event comprises all data recorded for the detection of one object (e.g. cell or bead) in an RT-DC measurement.

4.1.2 Features

A feature is a measurement parameter of an RT-DC measurement. For instance, the feature "index" enumerates all recorded events, the feature "deform" contains the deformation values of all events. There are scalar features, i.e. features that assign a single number to an event, and non-scalar features, such as "image" and "contour". All features in a dataset are exposed as read-only to the user. The following features are supported by dclab:

Scalar features

scalar features	description [units]
area_cvx	Convex area [px]
area_msd	Measured area [px]
area_ratio	Porosity (convex to measured area ratio)
area_um	Area [µm^2]
area_um_raw	Area [µm^2] of raw contour
aspect	Aspect ratio of bounding box
basinmap0	Basin mapping 0
basinmap1	Basin mapping 1
basinmap2	Basin mapping 2
basinmap3	Basin mapping 3
basinmap4	Basin mapping 4
basinmap5	Basin mapping 5
basinmap6	Basin mapping 6
basinmap7	Basin mapping 7
basinmap8	Basin mapping 8

continues on next page

Table 4.1 – continued from previous page

scalar features	description [units]
basinmap9	Basin mapping 9
bg_med	Median frame background brightness [a.u.]
bright_avg	Brightness average [a.u.]
bright_bc_avg	Brightness average (bgc) [a.u.]
bright_bc_sd	Brightness SD (bgc) [a.u.]
bright_perc_10	10th Percentile of brightness (bgc)
bright_perc_90	90th Percentile of brightness (bgc)
bright_sd	Brightness SD [a.u.]
circ	Circularity
deform	Deformation
deform_raw	Deformation of raw contour
eccentr_prnc	Eccentricity of raw contour
emodulus	Young's modulus [kPa]
fl1_area	FL-1 area of peak [a.u.]
fl1_dist	FL-1 distance between two first peaks [µs]
fl1_max	FL-1 maximum [a.u.]
fl1_max_ctc	FL-1 maximum, crosstalk-corrected [a.u.]
fl1_npeaks	FL-1 number of peaks
fl1_pos	FL-1 position of peak [µs]
fl1_width	FL-1 width [µs]
fl2_area	FL-2 area of peak [a.u.]
fl2_dist	FL-2 distance between two first peaks [µs]
fl2_max	FL-2 maximum [a.u.]
fl2_max_ctc	FL-2 maximum, crosstalk-corrected [a.u.]
fl2_npeaks	FL-2 number of peaks
fl2_pos	FL-2 position of peak [µs]
fl2_width	FL-2 width [µs]
fl3_area	FL-3 area of peak [a.u.]
fl3_dist	FL-3 distance between two first peaks [µs]
fl3_max	FL-3 maximum [a.u.]
fl3_max_ctc	FL-3 maximum, crosstalk-corrected [a.u.]
fl3_npeaks	FL-3 number of peaks
fl3_pos	FL-3 position of peak [µs]
fl3_width	FL-3 width [µs]
flow_rate	Flow rate [µLs^1]
frame	Video frame number
g_force	Gravitational force in multiples of g
index	Index (Dataset)
index_online	Index (Online)
inert_ratio_cvx	Inertia ratio of convex contour
inert_ratio_prnc	Principal inertia ratio of raw contour
inert_ratio_raw	Inertia ratio of raw contour
ml_class	Most probable ML class
nevents	Number of events in the same image
pc1	Principal component 1
pc2	Principal component 2
per_ratio	Inverse Convexity (raw to convex perimeter ratio)
per_um_raw	Perimeter [µm] of raw contour
pos_x	Position along channel axis [µm]
pos_y	Position lateral in channel [µm]

continues on next page

Table 4.1 – continued from previous page

scalar features	description [units]
pressure	Pressure [mPa]
qpi_dm_avg	Dry mass (average) [pg]
qpi_dm_sd	Dry mass (SD) [pg]
qpi_focus	Computed focus distance [µm]
qpi_pha_int	Integrated phase [rad]
qpi_ri_avg	Refractive index (average)
qpi_ri_sd	Refractive index (SD)
size_x	Bounding box size x [µm]
size_y	Bounding box size y [µm]
sym_x	Symmetry ratio left-right
sym_y	Symmetry ratio top-bottom
temp	Chip temperature [°C]
temp_amb	Ambient temperature [°C]
tex_asm_avg	Texture angular second moment (avg)
tex_asm_ptp	Texture angular second moment (ptp)
tex_con_avg	Texture contrast (avg)
tex_con_ptp	Texture contrast (ptp)
tex_cor_avg	Texture correlation (avg)
tex_cor_ptp	Texture correlation (ptp)
tex_den_avg	Texture difference entropy (avg)
tex_den_ptp	Texture difference entropy (ptp)
tex_ent_avg	Texture entropy (avg)
tex_ent_ptp	Texture entropy (ptp)
tex_f12_avg	Texture First measure of correlation (avg)
tex_f12_ptp	Texture First measure of correlation (ptp)
tex_f13_avg	Texture Second measure of correlation (avg)
tex_f13_ptp	Texture Second measure of correlation (ptp)
tex_idm_avg	Texture inverse difference moment (avg)
tex_idm_ptp	Texture inverse difference moment (ptp)
tex_sen_avg	Texture sum entropy (avg)
tex_sen_ptp	Texture sum entropy (ptp)
tex_sva_avg	Texture sum variance (avg)
tex_sva_ptp	Texture sum variance (ptp)
tex_var_avg	Texture variance (avg)
tex_var_ptp	Texture variance (ptp)
tilt	Absolute tilt of raw contour
time	Time [s]
userdef0	User-defined 0
userdef1	User-defined 1
userdef2	User-defined 2
userdef3	User-defined 3
userdef4	User-defined 4
userdef5	User-defined 5
userdef6	User-defined 6
userdef7	User-defined 7
userdef8	User-defined 8
userdef9	User-defined 9
volume	Volume [μm^3]

In addition to these scalar features, it is possible to define a large number of features dedicated to machine-learning, the "ml_score_???" features: The "?" can be a digit or a lower-case letter of the alphabet, e.g. "ml_score_rbc" or

"ml_score_3a3". If "ml_score_???" features are defined, then the ancillary "ml_class" feature, which identifies the most-probable feature for each event, becomes available.

Non-scalar features

non-scalar features	description [units]
contour	Event contour
image	Gray scale event image
image_bg	Gray scale event background image
mask	Binary mask labeling the event in the image
qpi_amp	Hologram amplitude image
qpi_oah	Off-axis hologram
qpi_oah_bg	Off-axis hologram background
qpi_pha	Hologram phase image [rad]
trace	Dictionary of fluorescence traces

Examples

deformation vs. area plot

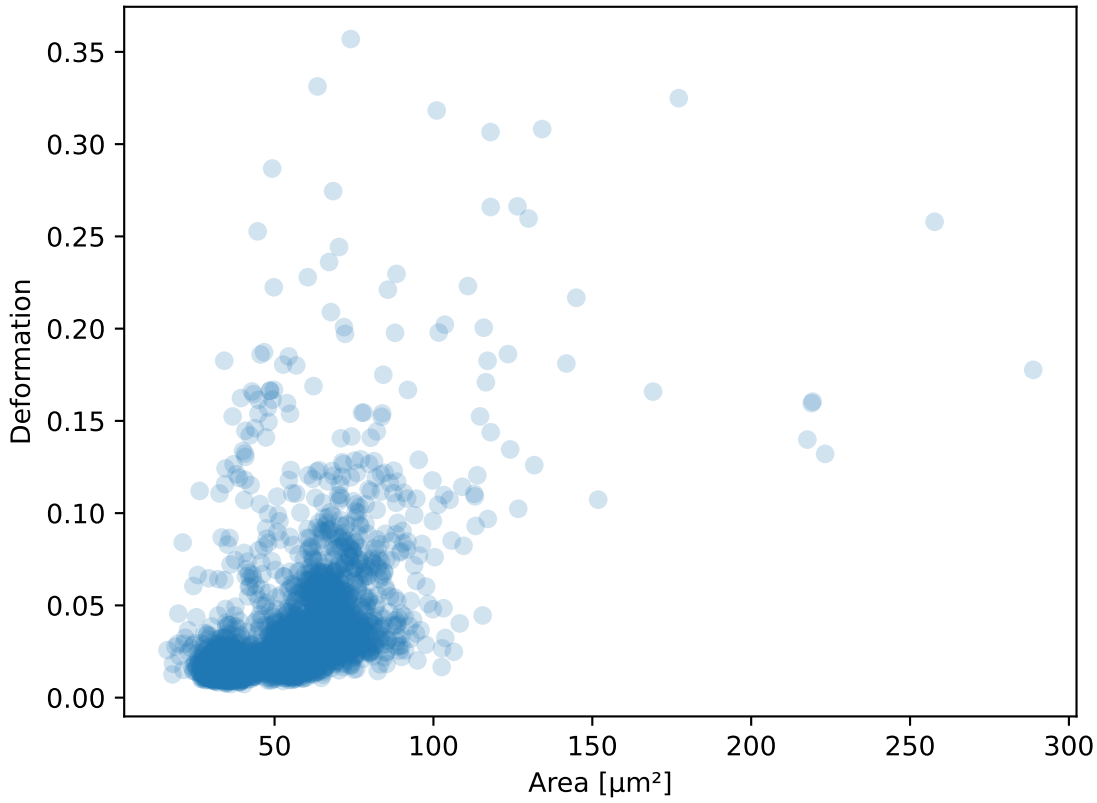
```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
ax = plt.subplot(111)
ax.plot(ds["area_um"], ds["deform"], "o", alpha=.2)
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
plt.show()
```

event image plot

```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example_video.rtdc")
ax1 = plt.subplot(211, title="image")
ax2 = plt.subplot(212, title="mask")
ax1.imshow(ds["image"][6], cmap="gray")
ax2.imshow(ds["mask"][6])
```

4.1.3 Ancillary features

Not all features available in dclab are recorded online during the acquisition of the experimental dataset. Some of the features are computed offline by dclab, such as "volume", "emodulus", or scores from imported machine learning models ("ml_score_xxx"). These ancillary features are computed on-the-fly and are made available seamlessly through the same interface.


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Deformation
50
100
150
200
250
300
Area [µm²]

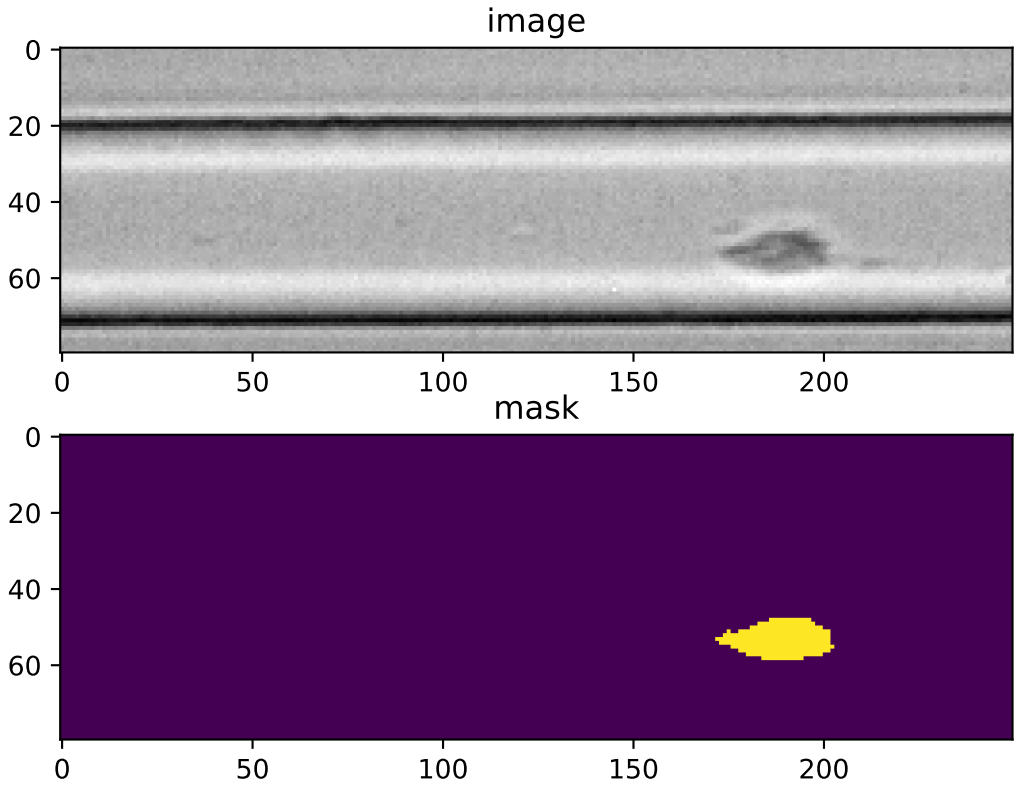


image
0
20
40
60
0
50
100
150
200
mask
0
20
40
60
0
50
100
150
200

4.1.4 Filters

A filter can be used to gate events using features. There are min/max filters and 2D *polygon filters*. The following table defines the main filtering parameters:

filtering	parsed	description [units]
enable filters	*{f}*	Enable filtering
hierarchy parent	`str`	Hierarchy parent of the dataset
limit events	*{f}*	Upper limit for number of filtered events
polygon filters	*{f}*	Polygon filter indices
remove invalid events	*{f}*	Remove events with inf/nan values

Min/max filters are also defined in the *filters* section:

filtering	explanation
area_um min	Exclude events with area [µm^2] below this value
area_um max	Exclude events with area [µm^2] above this value
aspect max	Exclude events with an aspect ratio above this value
…	…

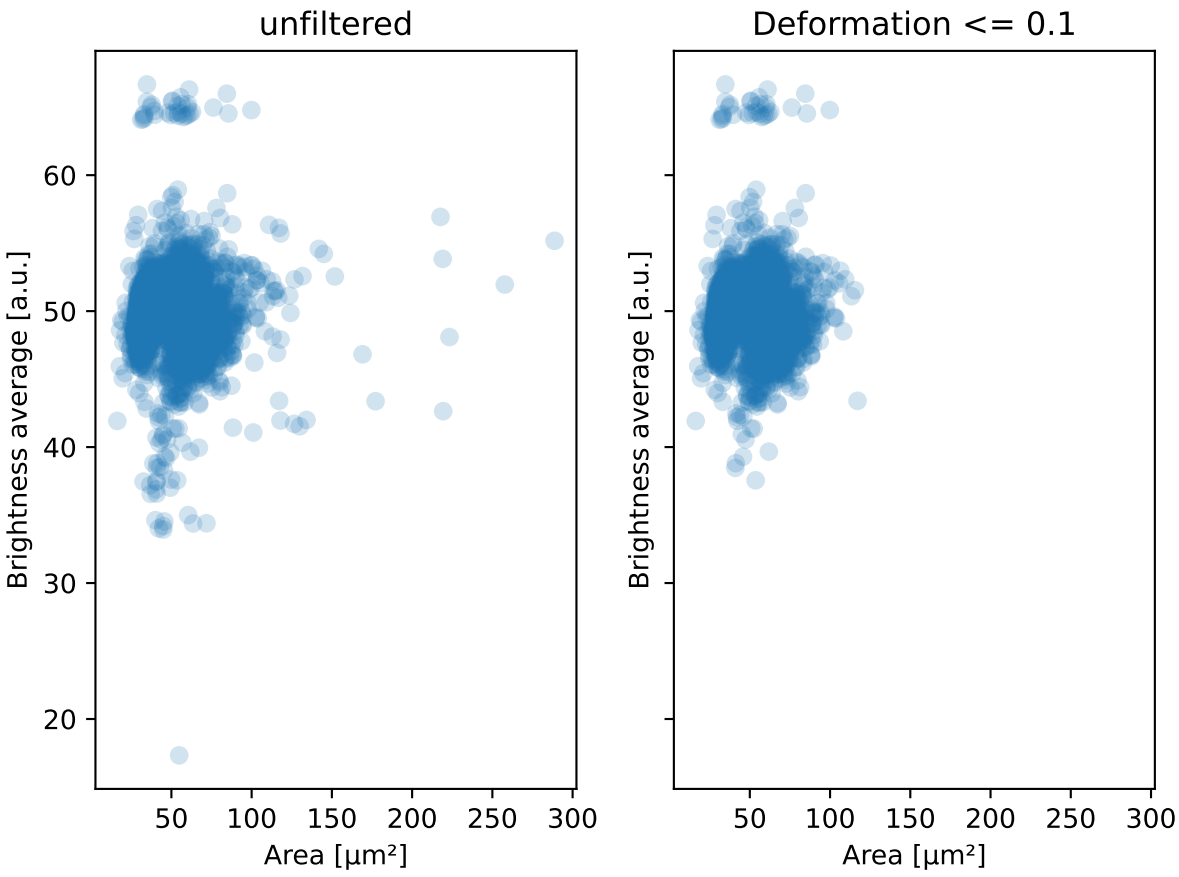
Examples

excluding events with large deformation

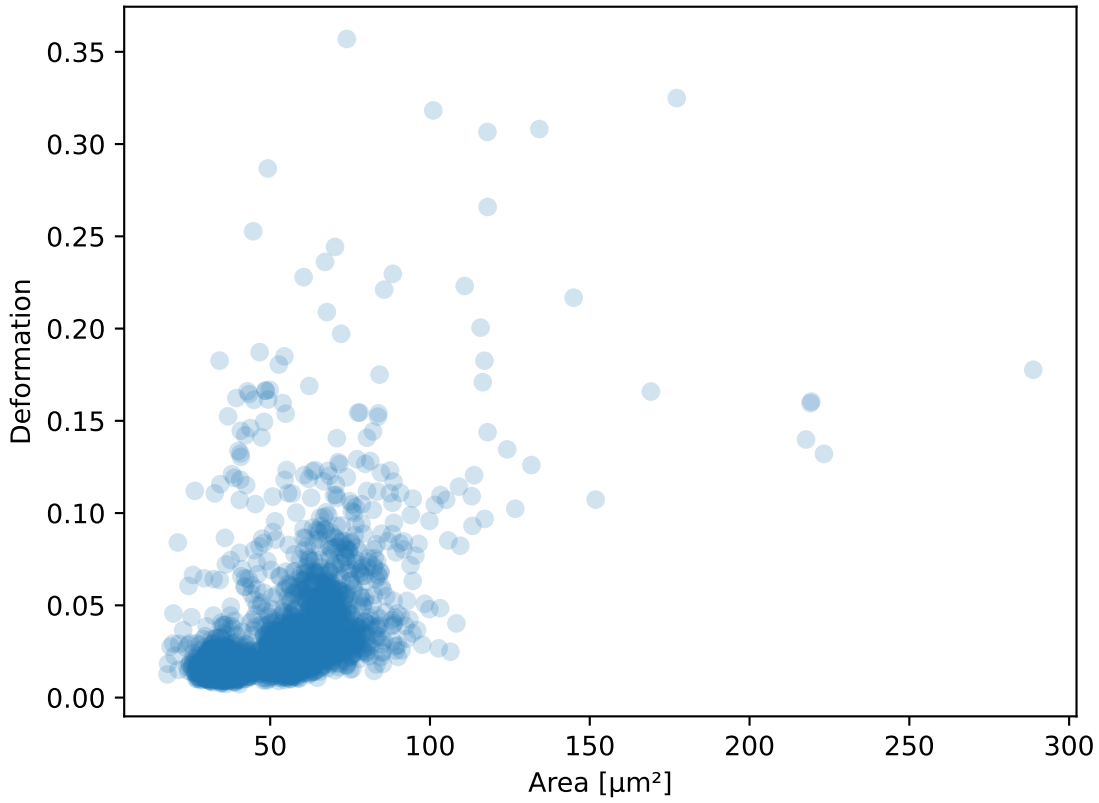
```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")

ds.config["filtering"]["deform min"] = 0
ds.config["filtering"]["deform max"] = .1
ds.apply_filter()
dif = ds.filter.all

f, axes = plt.subplots(1, 2, sharex=True, sharey=True)
axes[0].plot(ds["area_um"], ds["bright_avg"], "o", alpha=.2)
axes[0].set_title("unfiltered")
axes[1].plot(ds["area_um"][dif], ds["bright_avg"][dif], "o", alpha=.2)
axes[1].set_title("Deformation <= 0.1")

for ax in axes:
    ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
    ax.set_ylabel(dclab.dfn.get_feature_label("bright_avg"))

plt.tight_layout()
plt.show()
```

excluding random events

This is useful if you need to have a (sub-)dataset of a specified size. The downsampling is reproducible (the same points are excluded).


unfiltered
Deformation <= 0.1
Brightness average [a.u.]
Area [µm²]
20
30
40
50
60
50
100
150
200
250
300

```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
ds.config["filtering"]["limit events"] = 4000
ds.apply_filter()
fid = ds.filter.all

ax = plt.subplot(111)
ax.plot(ds["area_um"][fid], ds["deform"][fid], "o", alpha=.2)
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
plt.show()
```



4.1.5 Experiment metadata

Every RT-DC measurement has metadata consisting of key-value-pairs. The following are supported:

experiment	parsed	description [units]
date	str	Date of measurement ('YYYY-MM-DD')
event count	{f}	Number of recorded events
run identifier	str	Unique measurement identifier
run index	{f}	Index of measurement run
sample	str	Measured sample or user-defined reference
time	str	Start time of measurement ('HH:MM:SS[.S]')

fluorescence	parsed	description [units]
baseline 1 offset	{f}	Baseline offset channel 1
baseline 2 offset	{f}	Baseline offset channel 2
baseline 3 offset	{f}	Baseline offset channel 3
bit depth	{f}	Trace bit depth
channel 1 name	str	FL1 description
channel 2 name	str	FL2 description
channel 3 name	str	FL3 description
channel count	{f}	Number of active channels
channels installed	{f}	Number of available channels
laser 1 lambda	float	Laser 1 wavelength [nm]
laser 1 power	float	Laser 1 output power [%]
laser 2 lambda	float	Laser 2 wavelength [nm]
laser 2 power	float	Laser 2 output power [%]
laser 3 lambda	float	Laser 3 wavelength [nm]
laser 3 power	float	Laser 3 output power [%]
laser count	{f}	Number of active lasers
lasers installed	{f}	Number of available lasers
sample rate	{f}	Trace sample rate [Hz]
samples per event	{f}	Samples per event
signal max	float	Upper voltage detection limit [V]
signal min	float	Lower voltage detection limit [V]
trace median	{f}	Rolling median filter size for traces

fmt_tdms	parsed	description [units]
video frame offset	{f}	Missing events at beginning of video

imaging	parsed	description [units]
flash device	str	Light source device type
flash duration	float	Light source flash duration [µs]
frame rate	float	Imaging frame rate [Hz]
pixel size	float	Pixel size [µm]
roi position x	{f}	Image x coordinate on sensor [px]
roi position y	{f}	Image y coordinate on sensor [px]
roi size x	{f}	Image width [px]
roi size y	{f}	Image height [px]

online_contour	parsed	description [units]
bg empty	*{f}*	Background correction from empty frames only
bin area min	*{f}*	Minium pixel area of binary image event
bin kernel	*{f}*	Disk size for binary closing of mask image
bin threshold	*{f}*	Threshold for mask from bg-corrected image
image blur	*{f}*	Odd sigma for Gaussian blur (21x21 kernel)
no absdiff	*{f}*	Do not use OpenCV 'absdiff' for bg-correction

online_filter	parsed	description [units]
target duration	`float`	Target measurement duration [min]
target event count	*{f}*	Target event count for online gating

pipeline	parsed	description [units]
dcnum background	`str`	Background ID
dcnum data	`str`	Data ID
dcnum feature	`str`	Feature extractor ID
dcnum gate	`str`	Gating ID
dcnum generation	`str`	Generation ID
dcnum hash	`str`	Hash
dcnum segmenter	`str`	Segmenter ID
dcnum yield	*{f}*	Event yield

qpi	parsed	description [units]
amp border loc	str	Border location specifier for amplitude
amp border px	{f}	Width of border for amplitude [pix]
amp fit offset	str	Amplitude offset correction
amp fit profile	str	Amplitude profile correction
bg method	str	Background computation method
filter name	str	Fourier filter used
filter size	float	Fourier filter size [1/pix]
focus interval	{f}	Focus interval to search [µm]
focus kernel	str	Propagation kernel
focus metric	str	Metric used to calculate focus
focus minimizer	str	Minimizer used to calculate focus
focus padding	{f}	Level of padding for refocus
invert phase	{f}	Invert the phase data
medium index	float	Refractive index of medium
padding	{f}	Level of padding
pha border loc	str	Border location specifier for phase
pha border px	{f}	Width of border for phase [pix]
pha fit offset	str	Phase offset correction
pha fit profile	str	Phase profile correction
pixel size proc	float	QPI pixel size [µm].
pixel size raw	float	Hologram pixel size [µm].
scale to filter	{f}	Scale QPI data to filter size
sideband freq	{f}	Sideband coordinates [1/pix]
software version	str	Software version(s)
subtract mean	{f}	Subtract mean before processing
wavelength	float	Imaging wavelength [nm]

setup	parsed	description [units]
channel width	float	Width of microfluidic channel [µm]
chip identifier	{f}	Unique identifier of the chip used
chip region	{f}	Imaged chip region (channel or reservoir)
flow rate	float	Flow rate in channel [µL/s]
flow rate sample	float	Sample flow rate [µL/s]
flow rate sheath	float	Sheath flow rate [µL/s]
identifier	str	Unique setup identifier
medium	str	Medium used
module composition	str	Comma-separated list of modules used
software version	str	Acquisition software with version
temperature	float	Mean chip temperature [°C]

Example: date and time of a measurement

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example.rtdc")

In [3]: ds.config["experiment"]["date"], ds.config["experiment"]["time"]
Out[3]: ('2017-07-16', '19:01:36')
```

4.1.6 Analysis metadata

In addition to inherent (defined during data acquisition) metadata, dclab also supports additional metadata that are relevant for certain data analysis pipelines, such as Young's modulus computation or fluorescence crosstalk correction.

calculation	parsed	description [units]
crosstalk fl12	`float`	Fluorescence crosstalk, channel 1 to 2
crosstalk fl13	`float`	Fluorescence crosstalk, channel 1 to 3
crosstalk fl21	`float`	Fluorescence crosstalk, channel 2 to 1
crosstalk fl23	`float`	Fluorescence crosstalk, channel 2 to 3
crosstalk fl31	`float`	Fluorescence crosstalk, channel 3 to 1
crosstalk fl32	`float`	Fluorescence crosstalk, channel 3 to 2
emodulus lut	`str`	Look-up table identifier
emodulus medium	`str`	Medium used (e.g. '0.49% MC-PBS')
emodulus temperature	`float`	Chip temperature [°C]
emodulus viscosity	`float`	Viscosity [Pa*s] if 'medium' unknown
emodulus viscosity model	`str`	Viscosity model for known media

4.1.7 User-defined metadata

In addition to the registered metadata keys listed above, you may also define custom metadata in the "user" section. This section will be saved alongside the other metadata when a dataset is exported as an .rtdc (HDF5) file.

Note: It is recommended to use the following data types for the value of each key: `str`, `bool`, `float` and `int`. Other data types may not render nicely in ShapeOut2 or DCOR.

To edit the "user" section in dclab, simply modify the *config* property of a loaded dataset. The changes made are *not* written to the underlying file.

Example: Setting custom "user" metadata in dclab

```
In [4]: import dclab

In [5]: ds = dclab.new_dataset("data/example.rtdc")

In [6]: my_metadata = {"inlet": True, "n_channels": 4}

In [7]: ds.config["user"] = my_metadata

In [8]: other_metadata = {"outlet": False, "RBC": True}

# we can also add metadata with the `update` method
In [9]: ds.config["user"].update(other_metadata)

# or
In [10]: ds.config.update({"user": other_metadata})

In [11]: print(ds.config["user"])
{'inlet': True, 'n_channels': 4, 'outlet': False, 'RBC': True}
```

(continues on next page)

(continued from previous page)

```
# we can clear the "user" section like so:
In [12]: ds.config["user"].clear()
```

If you are implementing a custom data acquisition pipeline, you may alternatively add user-defined meta data (permanently) to an .rtdc file in a post-measurement step like so.

Example: Setting custom "user" metadata permanently

```
import h5py
with h5py.File("/path/to/your/dataset.rtdc") as h5:
    h5.attrs["user:inlet"] = True
    h5.attrs["user:n_channels"] = 4
    h5.attrs["user:outlet"] = False
    h5.attrs["user:RBC"] = True
    h5.attrs["user:project"] = "strangelove"
```

User-defined metadata can also be used with user-defined *plugin features*. This allows you to design plugin features which utilize your pipeline-specific metadata.

4.1.8 Basins

Since dclab 0.51.0, you can define so-called *basins* in .rtdc files. Basins are files or remote locations that contain additional *features* that are not part of the file you opened initially.

For instance, you might want to compute some additional features for a measurement, but you want to avoid editing the original file `data/example.rtdc`, and you also need to have access to the features of the original file when working with the new file `test.rtdc`.

```
In [13]: import dclab

# Create the smaller file with the basin defined.
In [14]: with dclab.new_dataset("data/example.rtdc") as dso, dclab.RTDCWriter(
↪"test.rtdc", mode="reset") as hw:
   ....:     # copy metadata
   ....:     meta = dict(dso.config)
   ....:     meta.pop("filtering")
   ....:     hw.store_metadata(meta)
   ....:     # store a feature from the original dataset
   ....:     hw.store_feature("deform", dso["deform"])
   ....:     # store a user-defined featurr
   ....:     hw.store_feature("userdef1", 2.5*dso["deform"])
   ....:     # store the basin information
   ....:     hw.store_basin(basin_name="mytest",
   ....:                    basin_type="file",
   ....:                    basin_format="hdf5",
   ....:                    basin_locs=["data/example.rtdc"])
   ....:

In [15]: ds2 = dclab.new_dataset("test.rtdc")

# the basin in "test.rtdc" gives you access to features stored in "data/
↪example.rtdc"
```

(continues on next page)

(continued from previous page)

```
In [16]: print(ds2.features)
['area_cvx', 'area_msd', 'area_ratio', 'area_um', 'aspect', 'bright_avg',
↪'bright_sd', 'circ', 'circ_times_area', 'deform', 'frame', 'index', 'inert_
↪ratio_cvx', 'inert_ratio_raw', 'nevents', 'pos_x', 'pos_y', 'size_x', 'size_y
↪', 'time', 'userdef1']
```

For more information, please take a look at the documentation of *Basin* and its subclasses.

4.2 Using DC data

Knowing and understanding the *RT-DC dataset classes* is an important prerequisite when working with dclab. They are all derived from *RTDCBase* which allows read-only access to all features with a dictionary-like interface, facilitates data export or filtering, and comes with several convenience methods that are useful for data visualization. RT-DC datasets can be based on a data file format (*RTDC_TDMS* and *RTDC_HDF5*), accessed from an online repository (*RTDC_DCOR* or *RTDC_S3*), created from user-defined dictionaries (*RTDC_Dict*), or derived from other RT-DC datasets (*RTDC_Hierarchy*).

4.2.1 Opening a file

The convenience function *dclab.new_dataset()* takes care of determining the data format and returns the corresponding derived class.

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example.rtdc")

In [3]: ds.__class__.__name__
Out[3]: 'RTDC_HDF5'
```

4.2.2 Creating an in-memory dataset

It is also possible to load other data into dclab from a dictionary.

```
In [4]: data = dict(deform=np.random.rand(100),
   ...:             area_um=np.random.rand(100))
   ...:

In [5]: ds_dict = dclab.new_dataset(data)

In [6]: ds_dict.__class__.__name__
Out[6]: 'RTDC_Dict'
```

4.2.3 Filtering (Gating)

Filters are used to mask e.g. debris or doublets from a dataset.

```
# Restrict the deformation to 0.15
In [7]: ds.config["filtering"]["deform min"] = 0

In [8]: ds.config["filtering"]["deform max"] = .15

# Manually excluding events using array indices is also possible:
# The writeable `ds.filter.manual` is a 1D boolean array of size `len(ds)`
# where `False` can manually set, implying that these events are excluded.
# The following line sets the four events located at indices
# 0, 345, 400, and 1000 to False, so that they are added to ds.filter.all
# when `ds.apply_filter()` is called.
In [9]: ds.filter.manual[[0, 400, 345, 1000]] = False

In [10]: ds.apply_filter()

In [11]: assert not ds.filter.all[345]

# The read-only boolean array `ds.filter.all` represents the applied filter
# and can be used for indexing.
In [12]: ds["deform"][:].mean(), ds["deform"][ds.filter.all].mean()
Out[12]: (0.0287258, 0.026486598)
```

Note that `ds.apply_filter()` must be called, otherwise `ds.filter.all` is not updated.

4.2.4 Creating hierarchies

When applying filtering operations, it is sometimes helpful to use hierarchies for keeping track of the individual filtering steps.

```
In [13]: child = dclab.new_dataset(ds)

In [14]: child.config["filtering"]["area_um min"] = 0

In [15]: child.config["filtering"]["area_um max"] = 80

In [16]: grandchild = dclab.new_dataset(child)

In [17]: grandchild.rejuvenate()

In [18]: len(ds), len(child), len(grandchild)
Out[18]: (5000, 4933, 4778)

In [19]: ds.filter.all.sum(), child.filter.all.sum(), grandchild.filter.all.sum()
Out[19]: (4933, 4778, 4778)
```

Note that calling `grandchild.rejuvenate()` automatically calls `child.rejuvenate()` and `ds.apply_filter()`. Also note that, as expected, the size of each hierarchy child is identical to the sum of the boolean filtering array from its hierarchy parent.

Always make sure to call *rejuvenate* to the youngest members of your hierarchy (here *grandchild*), when you changed

a filter in the hierarchy or when you modified an ancillary feature or the dataset metadata/configuration. Otherwise you cannot be sure that all information properly propagated through your hierarchy (Your grandchild might be an orphan).

4.2.5 Computing feature statistics

The *statistics* module comes with a predefined set of methods to compute simple feature statistics.

```
In [20]: import dclab

In [21]: ds = dclab.new_dataset("data/example.rtdc")

In [22]: stats = dclab.statistics.get_statistics(ds,
   ....:                                         features=["deform", "aspect"],
   ....:                                         methods=["Mode", "Mean", "SD"])
   ....:

In [23]: dict(zip(*stats))
Out[23]:
{'Mode Deformation': 0.016635261,
 'Mean Deformation': 0.0287258,
 'SD Deformation': 0.028740086,
 'Mode Aspect ratio of bounding box': 1.1091421916433233,
 'Mean Aspect ratio of bounding box': 1.2719607587337494,
 'SD Aspect ratio of bounding box': 0.2523385371130096}
```

Note that the statistics take into account the applied filters:

```
In [24]: ds.config["filtering"]["deform min"] = 0

In [25]: ds.config["filtering"]["deform max"] = .1

In [26]: ds.apply_filter()

In [27]: stats2 = dclab.statistics.get_statistics(ds,
   ....:                                          features=["deform", "aspect"],
   ....:                                          methods=["Mode", "Mean", "SD"])
   ....:

In [28]: dict(zip(*stats2))
Out[28]:
{'Mode Deformation': 0.017006295,
 'Mean Deformation': 0.02476519,
 'SD Deformation': 0.015638638,
 'Mode Aspect ratio of bounding box': 1.1232223188589807,
 'Mean Aspect ratio of bounding box': 1.240720618624576,
 'SD Aspect ratio of bounding box': 0.15993707940243287}
```

These are the available statistics methods:

```
In [29]: dclab.statistics.Statistics.available_methods.keys()
Out[29]: dict_keys(['Mean', 'Median', 'Mode', 'SD', 'Events', '%-gated', 'Flow rate'])
```

4.2.6 Commonly used scripting examples

Here are a few useful functionalities for scripting with dclab.

```
# unique identifier of the RTDCBase instance (not reproducible)
In [30]: ds.identifier
Out[30]: 'mm-hdf5_c5812ae'

# reproducible hash of the dataset
In [31]: ds.hash
Out[31]: 'e9726c7be19cde2cc415a56a69272a19'

# dataset format
In [32]: ds.format
Out[32]: 'hdf5'

# all available features
In [33]: ds.features
Out[33]:
['area_cvx',
 'area_msd',
 'area_ratio',
 'area_um',
 'aspect',
 'bright_avg',
 'bright_sd',
 'circ',
 'deform',
 'frame',
 'index',
 'inert_ratio_cvx',
 'inert_ratio_raw',
 'nevents',
 'pos_x',
 'pos_y',
 'size_x',
 'size_y',
 'time']

# scalar (one number per event) features
In [34]: ds.features_scalar
Out[34]:
['area_cvx',
 'area_msd',
 'area_ratio',
 'area_um',
 'aspect',
 'bright_avg',
 'bright_sd',
 'circ',
 'deform',
 'frame',
 'index',
```

(continues on next page)

(continued from previous page)

```
 'inert_ratio_cvx',
 'inert_ratio_raw',
 'nevents',
 'pos_x',
 'pos_y',
 'size_x',
 'size_y',
 'time']

# innate (present in the underlying data file) features
In [35]: ds.features_innate
Out[35]:
['area_cvx',
 'area_msd',
 'bright_avg',
 'bright_sd',
 'circ',
 'frame',
 'inert_ratio_cvx',
 'inert_ratio_raw',
 'nevents',
 'pos_x',
 'pos_y',
 'size_x',
 'size_y']

# loaded (innate and computed ancillaries) features
In [36]: ds.features_loaded
Out[36]:
['area_cvx',
 'area_msd',
 'area_ratio',
 'area_um',
 'aspect',
 'bright_avg',
 'bright_sd',
 'circ',
 'deform',
 'frame',
 'index',
 'inert_ratio_cvx',
 'inert_ratio_raw',
 'nevents',
 'pos_x',
 'pos_y',
 'size_x',
 'size_y',
 'time']

# test feature availability (success)
In [37]: "area_um" in ds
Out[37]: True
```

(continues on next page)

(continued from previous page)

```
# test feature availability (failure)
In [38]: "image" in ds
Out[38]: False

# accessing a feature and computing its mean
In [39]: ds["area_um"][:].mean()
Out[39]: 49.728645

# accessing the measurement configuration
In [40]: ds.config.keys()
Out[40]: KeysView({'filtering': {'remove invalid events': False, 'enable filters': True,
 ↪'limit events': 0, 'polygon filters': [], 'hierarchy parent': 'none', 'deform min': 0,
 ↪'deform max': 0.1}, 'experiment': {'date': '2017-07-16', 'event count': 5000, 'run␣
 ↪index': 1, 'sample': 'docs-data', 'time': '19:01:36'}, 'imaging': {'flash device':
 ↪'LED (undefined)', 'flash duration': 2.0, 'frame rate': 2000.0, 'pixel size': 0.34,
 ↪'roi position x': 504, 'roi position y': 472, 'roi size x': 256, 'roi size y': 96},
 ↪'online_contour': {'bin area min': 50, 'bin kernel': 5, 'bin threshold': 6, 'image blur
 ↪': 0, 'no absdiff': True}, 'setup': {'channel width': 20.0, 'chip region': 'channel',
 ↪'flow rate': 0.06, 'flow rate sample': 0.041999999999999996, 'flow rate sheath': 0.018,
 ↪ 'identifier': 'ZMDD-AcC-8ecba5-cd57e2', 'medium': 'CellCarrier', 'module composition
 ↪': 'Cell_Flow_2 + Fluor', 'software version': 'dclab 0.5.2.dev13'}})

In [41]: ds.config["experiment"]
Out[41]: {'date': '2017-07-16', 'event count': 5000, 'run index': 1, 'sample': 'docs-data
 ↪', 'time': '19:01:36'}

# determine the identifier of the hierarchy parent
In [42]: child.config["filtering"]["hierarchy parent"]
Out[42]: 'mm-hdf5_9969d57'
```

4.3 DC data I/O

When working with DC data, you will inevitably run into the situation where you would like to write some part of dataset (be it an .rtdc file or data on DCOR) to a new file. Depending on the situation, one or more of the following subsection will probably cover what you need.

4.3.1 Exporting data

The *RTDCBase* class has the attribute *RTDCBase.export* which allows to export event data to several data file formats. See *Export* for more information.

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example.rtdc")

# Restrict the deformation to 0.15
In [3]: ds.config["filtering"]["deform min"] = 0
```

(continues on next page)

(continued from previous page)

```
In [4]: ds.config["filtering"]["deform max"] = .15

In [5]: ds.apply_filter()

In [6]: print("Deformation mean before filtering:", ds["deform"][:].mean())
Deformation mean before filtering: 0.0287258

In [7]: print("Deformation mean after filtering:", ds["deform"][ds.filter.all].mean())
Deformation mean after filtering: 0.026480934

# Export to .tsv
In [8]: ds.export.tsv(path="export_example.tsv",
   ...:               features=["area_um", "deform"],
   ...:               filtered=True,
   ...:               override=True)
   ...:

# Export to .rtdc
In [9]: ds.export.hdf5(path="export_example.rtdc",
   ...:                features=["area_um", "aspect", "deform"],
   ...:                filtered=True,
   ...:                override=True)
   ...:
```

Note that data exported as HDF5 files can be loaded with dclab (reproducing the previously computed statistics - without filters).

```
In [10]: ds2 = dclab.new_dataset("export_example.rtdc")

In [11]: ds2["deform"][:].mean()
Out[11]: 0.026480934
```

4.3.2 Writing to an .rtdc file

If you would like to create your own .rtdc files, you can make use of the *RTDCWriter* class.

```
In [12]: with dclab.RTDCWriter("my-data.rtdc", mode="reset") as hw:
   ....:     hw.store_metadata({"experiment": {"sample": "my sample",
   ....:                                       "run index": 1}})
   ....:     hw.store_feature("deform", np.random.rand(100))
   ....:     hw.store_feature("area_um", np.random.rand(100))
   ....:

In [13]: ds_custom = dclab.new_dataset("my-data.rtdc")

In [14]: print(ds_custom.features)
['area_um', 'deform', 'index']

In [15]: print(ds_custom.config["experiment"])
{'event count': 100, 'run index': 1, 'sample': 'my sample'}
```

4.3.3 Copying (parts of) a dataset

In some situations, you would only like to copy an entire feature column from one dataset to a new file without modification. The *copier* submodule enables this on a low-level.

- Use the *rtdc_copy()* method to create a compressed version of a DC dataset opened as an HDF5 file (*RTDC_HDF5* or *RTDC_S3*).
- Use the *h5ds_copy()* method to copy parts of an HDF5 dataset to another HDF5 file, with the option to enforce compression (if the source `h5py.Dataset` is not compressed properly already).

4.4 Basins

4.4.1 Motivation

Basins are a powerful concept that allow you to both save time, disk space, and network usage when processing DC data. The basic idea behind basins is that you avoid duplicating feature data, by not copying all of the features from an input file to an output file, but by *linking* from the output file to the input file.

For instance, let's say your pipeline is designed to compute a new feature `userdef1` and you would like to open the output file in Shape-Out, visualizing this feature in combination with other features defined in the input file (e.g. `deform`). What you *could* do is to write the `userdef1` feature directly to the input file or to create a copy of your input file and write `userdef1` to that copy. However, this might make you feel uneasy, because you would like to avoid editing your original raw data (possible data loss) and copying an entire file seems like unnecessary overhead (redundant data, high disk usage). With basin features, you can create an empty output file, write your new feature `userdef1` to it, define a basin that links to your input file, and you are done. Without ever modifying your input file. And it is even possible to create an output file that is gated (containing a subset of events from the input file).

4.4.2 Defining Basins

Basins may have different properties depending on the use case. Let's dive into an example and afterwards explain what was done and how it could have been done differently.

```
import dclab

with (dclab.new_dataset("input.rtdc") as ds,
      dclab.RTDCWriter("output.rtdc") as hw):

    # First of all, we have to copy the metadata from the input file
    # to the output file. If we forget to do this, then dclab will
    # not be able to open the output file.
    hw.store_metadata(ds.config.as_dict(pop_filtering=True))

    # Next, we can compute and write the new feature to the output file.
    hw.store_feature("userdef1", np.random.random(len(ds)))

    # Finally, we write the basin information to the output file.
    hw.store_basin(
        basin_name="raw data",
        basin_type="file",
        basin_format="hdf5",
        basin_locs=["input.rtdc"],
```

(continues on next page)

(continued from previous page)

```
    )

# You can now open the output file and verify that everything worked.
with dclab.new_dataset("output.rtdc") as ds_out:
    assert "userdef1" in ds_out, "check that the feature we wrote is there"
    assert "image" in ds_out, "check that we can access basin features"
    # You could also be more specific:
    assert "userdef1" in ds_out.features_innate
    assert "image" in ds_out.features_basin
```

What happened? We created an `output.rtdc` file that contains the metadata from the `input.rtdc` file, a feature `userdef1` filled with random data, and the basin information referencing **all** features from the `input.rtdc` file. We opened the output file and verified that it transparently gives us access to the features stored in the (basin) input file.

What could we have done differently? Take a look at the options that *RTDCWriter.store_basin* allows you to set. You will notice that there are three major properties that define the nature of a basin:

- **type** of a basin: A basin can be a local *file* (including files on a network share), or a *remote* which means that it is accessible via a networking protocol (e.g. HTTP).
- **format** of a basin: This is the subclass of *RTDCBase* that defines how the basin is accessed. For *file*-type basins, this is "hdf5" and for *remote*-type basins, this is "dcor", "http", or "s3".
- **mapping** of a basin: If the output file has the same number of events in the same order as the input file, then we call the mapping "same". If the output file only contains a subset of the events from the input file, then we call the basin a **mapped basin**. These mapped basins allow you to minimize data redundancy for analysis pipelines that produce output files with a subset of the events from the input file. To be able to map from the input file to the output file, dclab stores the mapping information as integer indices in dedicated features enumerated `basinmap0`, `basinmap1`, etc.
- **features** defined by a basin: You may define basins and explicitly state the features this basin provides. In combination with mapping, you could e.g. realize your own event segmentation pipeline, storing only the `mask` feature and extracted scalar features in you output file, while you define the `image` feature via the input file basin. If you combine this approach with the dcor basin format, you can distribute all of your data (raw and processed) in a very efficient and transparent manner.

4.4.3 Examples

Mapped basin via RTDCWriter

You can explicitly define a mapped basin via the *RTDCWriter.store_basin* method (see also the example after this one).

```
import dclab
import numpy as np

with (dclab.new_dataset("input.rtdc") as ds,
      dclab.RTDCWriter("output.rtdc") as hw):

    # metadata
    hw.store_metadata(ds.config.as_dict(pop_filtering=True))

    # take every second event from the input file
    event_mapping = np.arange(len(ds), None, 2, dtype=np.uint64)
```

(continues on next page)

(continued from previous page)

```
    # write the basin
    hw.store_basin(
        basin_name="raw data",
        basin_type="file",
        basin_format="hdf5",
        basin_locs=["input.rtdc"],
        basin_map=event_mapping,
    )

# verify that this worked
with (dclab.new_dataset("input.rtdc") as ds_in,
      dclab.new_dataset("output.rtdc") as ds_out):
    assert np.allclose(ds_in["deform"][::2], ds_out["deform"])
```

Implicitly mapped basin via HDF5 export

It is also possible to implicitly write basin information to an exported file, achieving the same result as above (a very small output file).

```
import dclab
import numpy as np

with dclab.new_dataset("input.rtdc") as ds:
    # remove every second event
    ds.filter.manual[1::2] = False
    ds.apply_filter()
    # export the dataset with the mapped basin
    ds.export.hdf5(path="output.rtdc",
                   features=[],
                   filtered=True,
                   basins=True)

# verify that this worked
with (dclab.new_dataset("input.rtdc") as ds_in,
      dclab.new_dataset("output.rtdc") as ds_out):
    assert np.allclose(ds_in["deform"][::2], ds_out["deform"])
```

4.4.4 Basin internals

Storing the basin information

In the `output.rtdc` file, the basin is stored as a json-encoded string in an HDF5 dataset in the `"/basins"` group. For the HDF5 export example above, the json data looks like this:

```
{
  "description": "Exported with dclab 0.58.0",
  "format": "hdf5",
  "name": "Exported data",
  "type": "file",
```

(continues on next page)

(continued from previous page)

```
  "features": null,
  "mapping": "basinmap0",
  "paths": [
    "/absolute/path/to/input.rtdc",
    "input.rtdc"
  ]
}
```

The description and name are filled automatically by dclab here. As expected, the type of the basin is *file* and the format of the basin is *hdf5*. There are a few things to notice:

- The features are set to `null` which means `None`, i.e. **all** features from the input file are allowed.
- The *mapping* key reads *basinmap0*. This is the name of the feature in which to find the mapping information from the input file to the output file. The information can be found in the HDF5 dataset `/events/basinmap0` in the output file. Note that the fact that this mapping information is stored *as a feature* means that it is also properly gated when you define basins iteratively.
- There are two *paths* defined, an absolute path (from the root of the file system) and a relative path (relative to the directory of the output file). This relative path makes it possible to copy-paste these two files *together* to other locations. You will always be able to open the output file and see the basin features defined in the input file. Internally, dclab also checks the `measurement identifier` of the output file against that of the input file to avoid loading basin features from the wrong file.

For the sake of completeness, let's see how the basin information looks like when you derive the output file from a DCOR resource:

```
import dclab
import numpy as np

with dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0") as ds:
    ds.filter.manual[1::2] = False
    ds.apply_filter()
    ds.export.hdf5(path="output.rtdc",
                   features=[],
                   filtered=True,
                   basins=True)
```

The corresponding json data:

```
{
  "description": "Exported with dclab 0.58.0",
  "format": "dcor",
  "name": "Exported data",
  "type": "remote",
  "features": null,
  "mapping": "basinmap0",
  "urls": [
    "https://dcor.mpl.mpg.de/api/3/action/dcserv?id=fb719fb2-bd9f-817a-7d70-f4002af916f0"
  ]
}
```

As you can see, *paths* is replaced by *urls* and the *format* and *type* keys changed. The rest remains the same. This also works with private DCOR resources, given that you have globally set your API token as described in the *DCOR section*.

Basin loading procedure

When dclab opens a dataset the defines a basin, the basin features are retrieved only when they are needed (i.e. when the user tries to access them and they are not defined as innate features). Internally, dclab instantiates an *RTDCBase* subclass as defined by the *format* key. For mapped basins, dclab additionally creates a hierarchy child from the original dataset by filling the manual filtering array with the mapping information. To see which features are defined in basins, you can check the *RTDCBase.features_basin* property. The basins are directly accessible via *RTDCBase.basins* (and the basin datasets via `RTDCBase.basins[index].ds`).

4.5 Plugin features

For specialized applications, the features defined internally in dclab might not be enough to describe certain aspects of your data. Plugin features allow you to define a recipe for computing a new feature. This new feature is then available *automatically* for *every* dataset loaded in dclab.

Note: The advantages of plugin features over *temporary features* are that plugin features are reproducible, shareable, versionable, and generally more transparent. You should only use temporary features if absolutely necessary.

4.5.1 Using plugin feature recipes

If a colleague sent you a plugin feature recipe (a .py file), you just have to load it in dclab to use it.

```
In [1]: import dclab

In [2]: import numpy as np

# load a plugin feature (makes `circ_times_area` available)
In [3]: dclab.load_plugin_feature("data/example_plugin.py")
Out[3]: [<PlugInFeature 'circ_times_area' (id 70254...) with priority 0 at␣
↪0x7fd4654dd190>]

# load some data
In [4]: ds = dclab.new_dataset("data/example.rtdc")

# access the new feature
In [5]: circ_per_area = ds["circ_times_area"]

# do some filtering
In [6]: ds.config["filtering"]["circ_times_area min"] = 23

In [7]: ds.config["filtering"]["circ_times_area max"] = 29

In [8]: ds.apply_filter()

In [9]: print("Removed {} out of {} events!".format(np.sum(~ds.filter.all), len(ds)))
Removed 4828 out of 5000 events!
```

Please also have a look at the *plugin usage example*.

4.5.2 Auto-loading multiple plugin feature recipes

If you have several plugins and would like to load them all at once, you can do the following at the beginning of your scripts:

```
for plugin_path in pathlib.Path("my_plugin_directory").rglob("*.py"):
    dclab.load_plugin_feature(plugin_path)
```

4.5.3 Writing a plugin feature recipe

A plugin feature recipe is defined in a Python script (e.g. *my_dclab_plugin.py*). A plugin feature recipe contains a function and an `info` dictionary. The function calculates the desired feature and can even calculate several features, while the dictionary defines any extra (meta-)information of the calculated feature (or features). Both, "method" (the function) and "feature names", must be included in the `info` dictionary. Note that many of the items in the dictionary must be lists! Also note that in case a feature recipe contains *multiple* features, there must be only one function for their calculation and only one `info` dictionary. Below are three examples of creating and using plugin features.

Note: Plugin features are based on *ancillary features* (*code reference*).

Simple plugin feature recipe

In this `basic example`, the function `compute_my_feature()` defines the basic feature *"circ_times_area"*.

```
def compute_my_feature(rtdc_ds):
    """Compute circularity times area"""
    circ_times_area = rtdc_ds["circ"] * rtdc_ds["area_um"]
    return {"circ_times_area": circ_times_area}


info = {
    "method": compute_my_feature,
    "description": "Compute area times circularity",
    "feature names": ["circ_times_area"],
    "features required": ["circ", "area_um"],
    "version": "0.1.0",
}
```

Advanced plugin feature recipe

In `this example`, the function `compute_some_new_features()` defines two basic features: *"circ_per_area"* and *"circ_times_area"*. Notice that both features are computed in one function and that there is only one `info` dictionary:

```
"""Exemplary plugin feature

You can import the features defined in this file into dclab
with ``dclab.load_plugin_feature("/path/to/plugin_example.py")``.
"""
```

(continues on next page)

(continued from previous page)

```
def compute_some_new_features(rtdc_ds):
    """The function that does the heavy-lifting"""
    circ_per_area = rtdc_ds["circ"] / rtdc_ds["area_um"]
    circ_times_area = rtdc_ds["circ"] * rtdc_ds["area_um"]
    # returns a dictionary-like object
    return {"circ_per_area": circ_per_area, "circ_times_area": circ_times_area}


info = {
    "method": compute_some_new_features,
    "description": "This plugin will compute some features",
    "long description": "Even longer description that "
                        "can span multiple lines",
    "feature names": ["circ_per_area", "circ_times_area"],
    "feature labels": ["Circularity per Area", "Circularity times Area"],
    "features required": ["circ", "area_um"],
    "config required": [],
    "method check required": lambda x: True,
    "scalar feature": [True, True],
    "version": "0.1.0",
}
```

Here, all possible keys in the *info* dictionary are shown (but not all are used). The keys are additional keyword arguments to the *AncillaryFeature* class:

- `features required` corresponds to `req_features`
- `config required` corresponds to `req_config`
- `method check required` corresponds to `req_func`

The `scalar feature` is a list of boolean values that defines whether a feature is scalar or not (defaults to True).

Plugin feature recipe with user-defined metadata

In this `example`, the function `compute_area_exponent()` defines the basic feature *area_exp*, which is calculated using *user-defined metadata*.

```
def compute_area_exponent(rtdc_ds):
    """Compute area^exp depending on the given user-defined metadata"""
    area_exp = rtdc_ds["area_um"] ** rtdc_ds.config["user"]["exp"]
    return {"area_exp": area_exp}


info = {
    "method": compute_area_exponent,
    "description": "Compute area to the power of exp",
    "feature names": ["area_exp"],
    "features required": ["area_um"],
    "config required": [["user", ["exp"]]],
    "version": "0.1.0",
}
```

The above plugin uses the "exp" key in the "user" configuration section to set the exponent value (notice the `"config required"` key in the `info` dict). Therefore, the feature *area_exp* is only available, when `rtdc_ds.config["user"]["exp"]` is set.

```
In [10]: import dclab

In [11]: dclab.load_plugin_feature("data/example_plugin_metadata.py")
Out[11]: [<PlugInFeature 'area_exp' (id 5f03f...) with priority 0 at␣
↪0x7fd4654efd90>]

In [12]: ds = dclab.new_dataset("data/example.rtdc")

# The plugin feature is not yet available, because "user:exp" is missing
In [13]: "area_exp" in ds
Out[13]: False

# Set user-defined metadata
In [14]: my_metadata = {"inlet": True, "n_channels": 4, "exp": 3}

In [15]: ds.config["user"] = my_metadata

# The plugin feature is now available
In [16]: "area_exp" in ds
Out[16]: True

# Now the plugin feature can be accessed like any regular feature
In [17]: area_exp = ds["area_exp"]
```

4.5.4 Reloading plugin features stored in data files

It is also possible to store plugin features within datasets on disk. This may be useful if the speed of calculation of your plugin feature is slow, and you don't want to recalculate each time you open your dataset. The process for storing plugin feature data is similar to that *described for temporary features*. If you would like to access those feature data at a later time point, you still have to load the plugin feature recipe first:

```
dclab.load_plugin_feature("/path/to/plugin.py")
ds = dclab.new_dataset("/path/to/data_with_new_plugin_feature.rtdc")
circ_per_area = ds["circ_per_area"]
```

And this works as well (loading plugin after instantiation):

```
ds = dclab.new_dataset("/path/to/data_with_new_plugin_feature.rtdc")
dclab.load_plugin_feature("/path/to/plugin.py")
circ_per_area = ds["circ_per_area"]
```

Note: After storing and reloading, this feature is now an *innate* feature. You could in principle also access it by registering it as a temporary feature (e.g. if you don't have the recipe lying around).

See the *code reference on plugin features* for more information.

4.6 Temporary features

If *plugin features* are not suitable for your task, either because your feature data cannot be obtained automatically or because you are just testing things, you are in the right place.

Let's say you are interested in the mean overall fluorescence signal of each event in channel 1 and you would like to filter the dataset according to that information[1]. You can define a temporary feature in your dataset without modifying any files on disk.

Note: Temporary features are not supported by Shape-Out, DCKit, or DCOR/DCOR-Aid. They are only really helpful if you quickly need to test things. If possible, it is recommended to work with *plugin features*.

4.6.1 Setting a temporary feature in a dataset

For this example, you can register the temporary feature *fl1_mean* and manually set a corresponding filter for your dataset.

```
In [1]: import dclab

In [2]: import numpy as np

In [3]: ds = dclab.new_dataset("data/example_traces.rtdc")

# register a temporary feature
In [4]: dclab.register_temporary_feature(feature="fl1_mean")

# compute the temporary feature
In [5]: fl1_mean = np.array([np.mean(ds["trace"]["fl1_raw"][ii]) for ii in␣
↪range(len(ds))])

# set the temporary feature
In [6]: dclab.set_temporary_feature(rtdc_ds=ds, feature="fl1_mean", data=fl1_mean)

# do some filtering
In [7]: ds.config["filtering"]["fl1_mean min"] = 4

In [8]: ds.config["filtering"]["fl1_mean max"] = 200

In [9]: ds.apply_filter()

In [10]: print("Removed {} out of {} events!".format(np.sum(~ds.filter.all), len(ds)))
Removed 32 out of 47 events!
```

[1] You could, in principle, of course create a plugin feature for that.

4.6.2 Accessing temporary features stored in data files

It is also possible to store temporary features within datasets on disk. At a later time point, you can then load this data file from disk with access to those temporary features[2].

There are two ways of adding temporary features to an .rtdc data file.

- 1. With h5py:

```
import dclab
import h5py
import numpy as np

# extract the feature data from the dataset
with dclab.new_dataset("/path/to/data.rtdc") as ds:
    fl1_mean = np.array([np.mean(ds["trace"]["fl1_raw"][ii]) for ii in␣
↪range(len(ds))])

# write the feature to the HDF5 file
with h5py.File("/path/to/data.rtdc", "a") as h5:
    h5["events"]["fl1_mean"] = fl1_mean
```

- 2. Via *RTDCBase.export.hdf5*:

```
import dclab
import h5py
import numpy as np

# register temporary feature
dclab.register_temporary_feature(feature="fl1_mean")

with dclab.new_dataset("/path/to/data.rtdc") as ds:
    # extract the feature information from the dataset
    fl1_mean = np.array([np.mean(ds["trace"]["fl1_raw"][ii]) for ii in␣
↪range(len(ds))])
    # set the data
    dclab.set_temporary_feature(rtdc_ds=ds, feature="fl1_mean", data=fl1_
↪mean)
    # export the data to a new file
    ds.export.hdf5("/path/to/data_with_fl1_mean.rtdc",
                   features=ds.features_innate + ["fl1_mean"])
```

If you wish to load the data at a later time point, you have to make sure that you register the temporary feature before trying to access it. This will not work:

```
ds = dclab.new_dataset("/path/to/data_with_fl1_mean.rtdc")
fl1_mean = ds["fl1_mean"]
```

But this works:

```
dclab.register_temporary_feature(feature="fl1_mean")
ds = dclab.new_dataset("/path/to/data_with_fl1_mean.rtdc")
fl1_mean = ds["fl1_mean"]
```

[2] I know, storing *temporary* features on disk sounds like a counter-intuitive concept, but this is a very convenient extension of temporary features which came with almost no overhead. In a sense, it's still temporary, because you always have to register the feature before you can access it.

And this works as well (registering after instantiation):

```
ds = dclab.new_dataset("/path/to/data_with_fl1_mean.rtdc")
dclab.register_temporary_feature(feature="fl1_mean")
fl1_mean = ds["fl1_mean"]
```

Please read the *code reference on temporary features* for more information.

4.7 Scatter plots

For data visualization, dclab comes with predefined *kernel density estimators (KDEs)* and an *event downsampling* module. The functionalities of both modules are made available directly via the `RTDCBase` class.

4.7.1 KDE scatter plot

The KDE of the events in a 2D scatter plot can be used to colorize events according to event density using the `RTDCBase.get_kde_scatter` function.

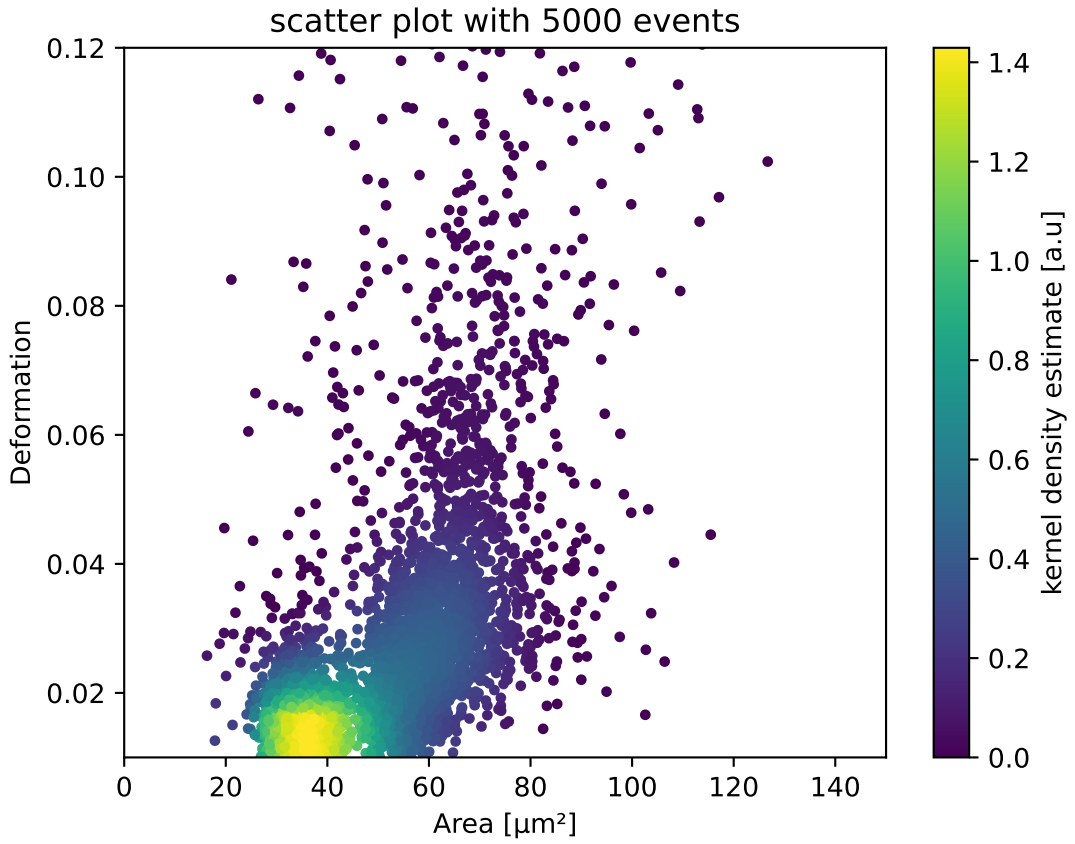
```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
kde = ds.get_kde_scatter(xax="area_um", yax="deform")

ax = plt.subplot(111, title="scatter plot with {} events".format(len(kde)))
sc = ax.scatter(ds["area_um"], ds["deform"], c=kde, marker=".")
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
ax.set_xlim(0, 150)
ax.set_ylim(0.01, 0.12)
plt.colorbar(sc, label="kernel density estimate [a.u]")
plt.show()
```

4.7.2 KDE scatter plot with event-density-based downsampling

To reduce the complexity of the plot (e.g. when exporting to scalable vector graphics (.svg)), the plotted events can be downsampled by removing events from high-event-density regions. The number of events plotted is reduced but the resulting visualization is almost indistinguishable from the one above.

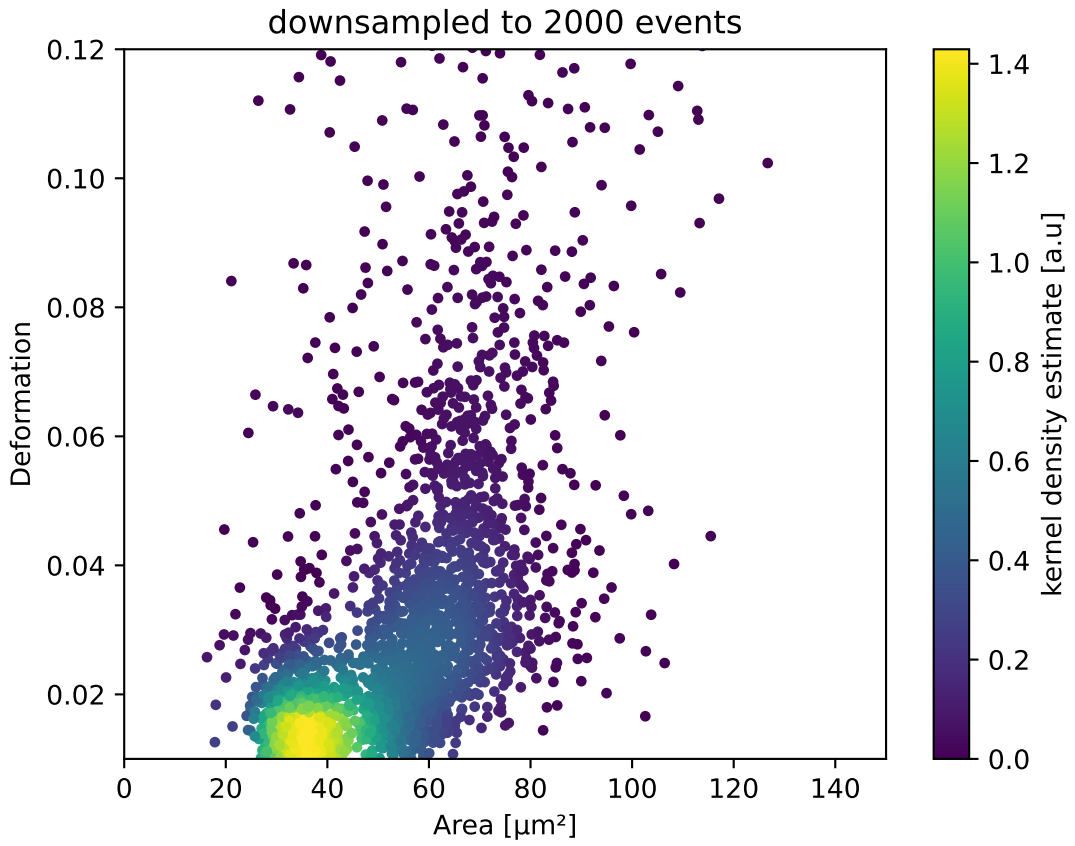
```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
xsamp, ysamp = ds.get_downsampled_scatter(xax="area_um", yax="deform", downsample=2000)
kde = ds.get_kde_scatter(xax="area_um", yax="deform", positions=(xsamp, ysamp))

ax = plt.subplot(111, title="downsampled to {} events".format(len(kde)))
sc = ax.scatter(xsamp, ysamp, c=kde, marker=".")
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
ax.set_xlim(0, 150)
ax.set_ylim(0.01, 0.12)
```

(continues on next page)


scatter plot with 5000 events
Deformation
0.12
0.10
0.08
0.06
0.04
0.02
0
20
40
60
80
100
120
140
Area [µm²]
kernel density estimate [a.u]
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0

(continued from previous page)

```
plt.colorbar(sc, label="kernel density estimate [a.u]")
plt.show()
```



4.7.3 KDE estimate on a log-scale

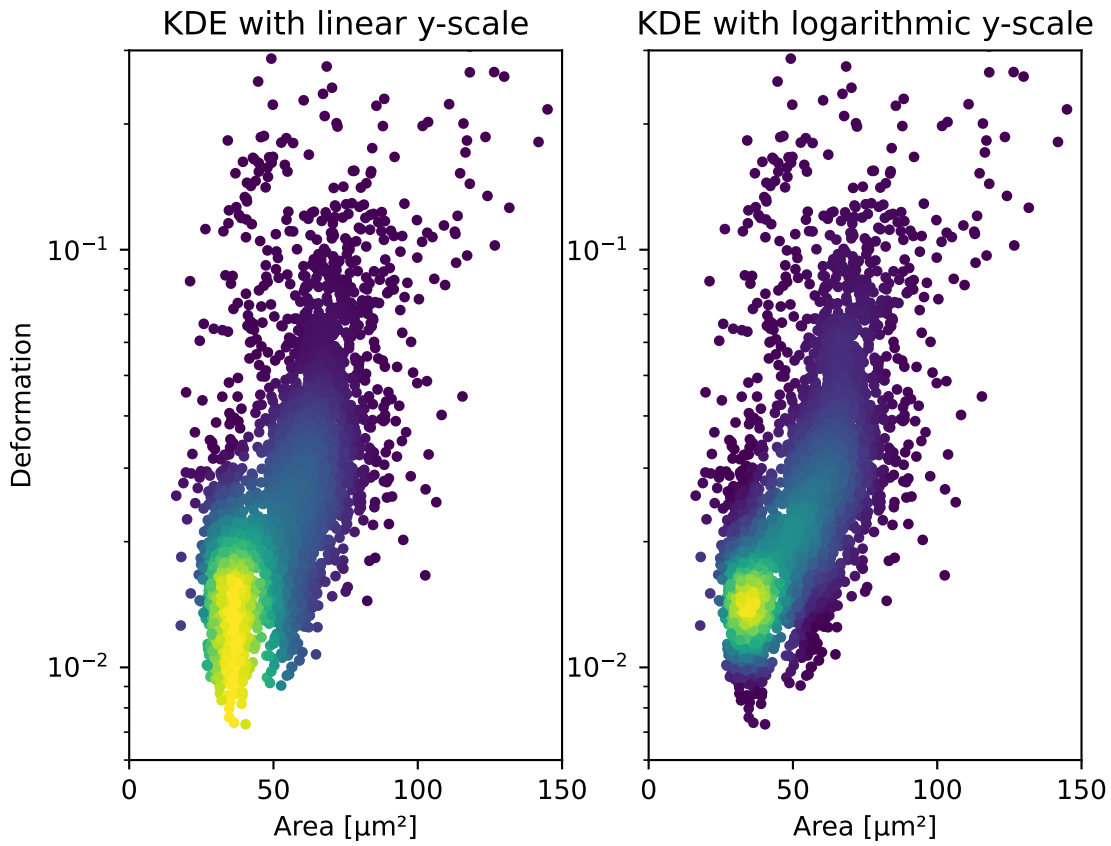
Frequently, data is visualized on logarithmic scales. If the KDE is computed on a linear scale, then the result will look unaesthetic when plotted on a logarithmic scale. Therefore, the methods *get_downsampled_scatter*, *get_kde_contour*, and *get_kde_scatter* offer the keyword arguments `xscale` and `yscale` which can be set to "log" for prettier plots.

```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
kde_lin = ds.get_kde_scatter(xax="area_um", yax="deform", yscale="linear")
kde_log = ds.get_kde_scatter(xax="area_um", yax="deform", yscale="log")

ax1 = plt.subplot(121, title="KDE with linear y-scale")
sc1 = ax1.scatter(ds["area_um"], ds["deform"], c=kde_lin, marker=".")

ax2 = plt.subplot(122, title="KDE with logarithmic y-scale")
sc2 = ax2.scatter(ds["area_um"], ds["deform"], c=kde_log, marker=".")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel(dclab.dfn.get_feature_label("deform"))
for ax in [ax1, ax2]:
    ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
    ax.set_xlim(0, 150)
    ax.set_ylim(6e-3, 3e-1)
    ax.set_yscale("log")

plt.show()
```



4.7.4 Isoelasticity lines

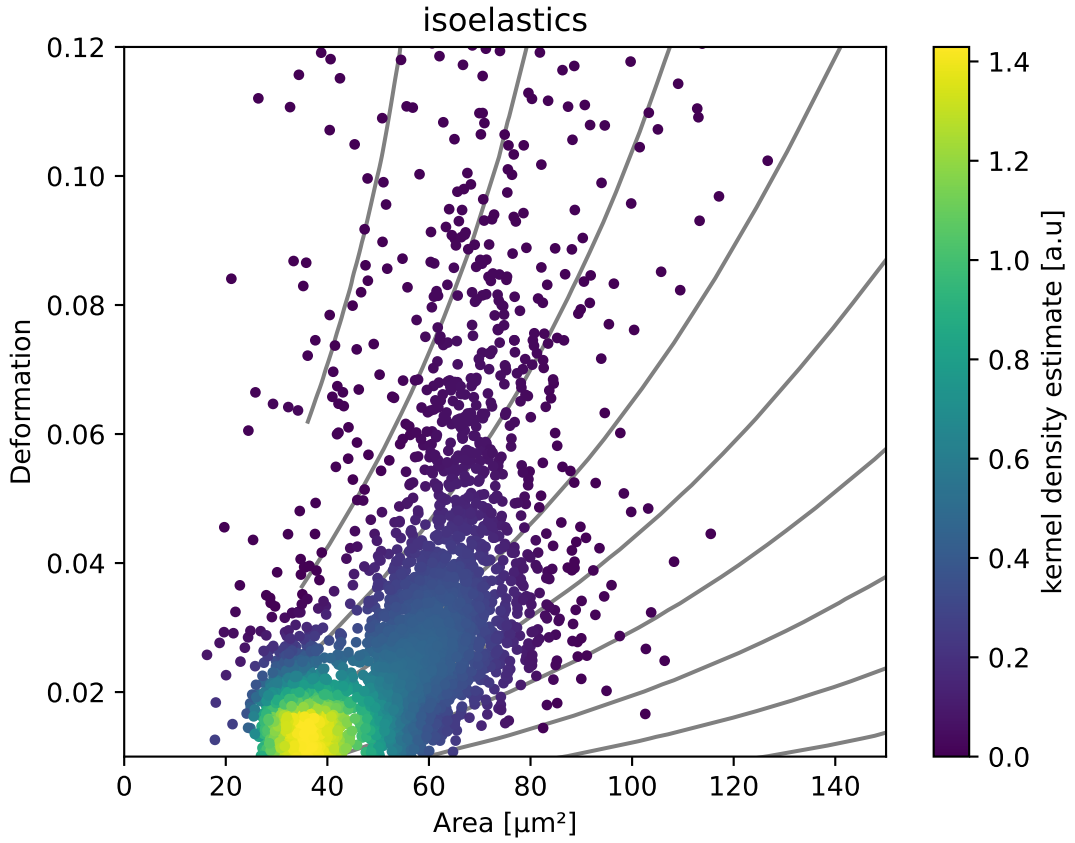
In addition, dclab comes with predefined isoelasticity lines that are commonly used to identify events with similar elastic moduli. Isoelasticity lines are available via the *isoelastics* module.

```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
kde = ds.get_kde_scatter(xax="area_um", yax="deform")

isodef = dclab.isoelastics.get_default()
```

(continues on next page)

(continued from previous page)

```
iso = isodef.get_with_rtdcbase(method="numerical",
                               col1="area_um",
                               col2="deform",
                               dataset=ds)

ax = plt.subplot(111, title="isoelastics")
for ss in iso:
    ax.plot(ss[:, 0], ss[:, 1], color="gray", zorder=1)
sc = ax.scatter(ds["area_um"], ds["deform"], c=kde, marker=".", zorder=2)
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
ax.set_xlim(0, 150)
ax.set_ylim(0.01, 0.12)
plt.colorbar(sc, label="kernel density estimate [a.u]")
plt.show()
```



4.7.5 Contour plot with percentiles

Contour plots are commonly used to compare the kernel density between measurements. Kernel density estimates (on a grid) for contour plots can be computed with the function *RTDCBase.get_kde_contour*. In addition, it is possible to compute contours at data percentiles using *dclab.kde_contours.get_quantile_levels()*.

```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
X, Y, Z = ds.get_kde_contour(xax="area_um", yax="deform")
Z /= Z.max()
quantiles = [.1, .5, .75]
levels = dclab.kde_contours.get_quantile_levels(density=Z,
                                                x=X,
                                                y=Y,
                                                xp=ds["area_um"],
                                                yp=ds["deform"],
                                                q=quantiles,
                                                )

ax = plt.subplot(111, title="contour lines")
sc = ax.scatter(ds["area_um"], ds["deform"], c="lightgray", marker=".", zorder=1)
cn = ax.contour(X, Y, Z,
                levels=levels,
                linestyles=["--", "-", "-"],
                colors=["blue", "blue", "darkblue"],
                linewidths=[2, 2, 3],
                zorder=2)

ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
ax.set_xlim(0, 150)
ax.set_ylim(0.01, 0.12)
# label contour lines with percentiles
fmt = {}
for l, q in zip(levels, quantiles):
    fmt[l] = "{:.0f}th".format(q*100)
plt.clabel(cn, fmt=fmt)
plt.show()
```

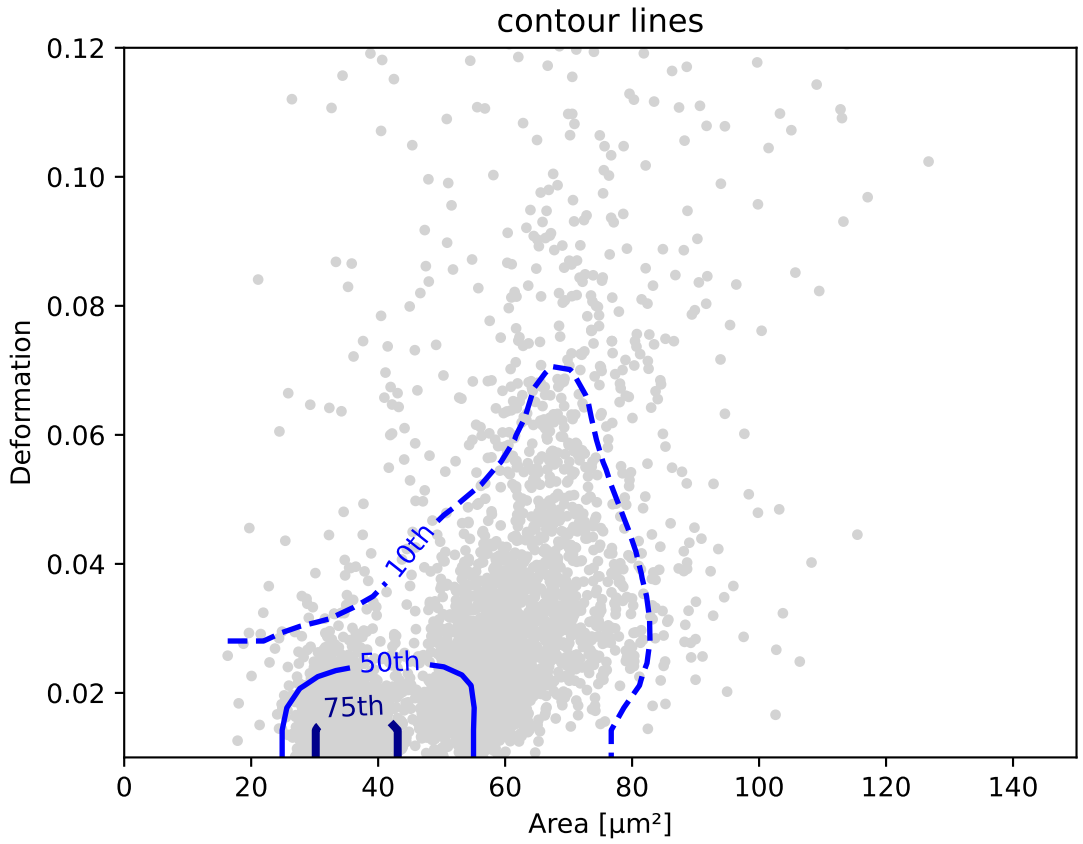
Note that you may compute (and plot) the contour lines directly yourself using the function *dclab.kde_contours.find_contours_level()*.

4.7.6 Polygon filters / Shape-Out

Keep in mind that you can combine your dclab analysis pipeline with Shape-Out. For instance, you can create and export *polygon filters* in Shape-Out and then import them in dclab.

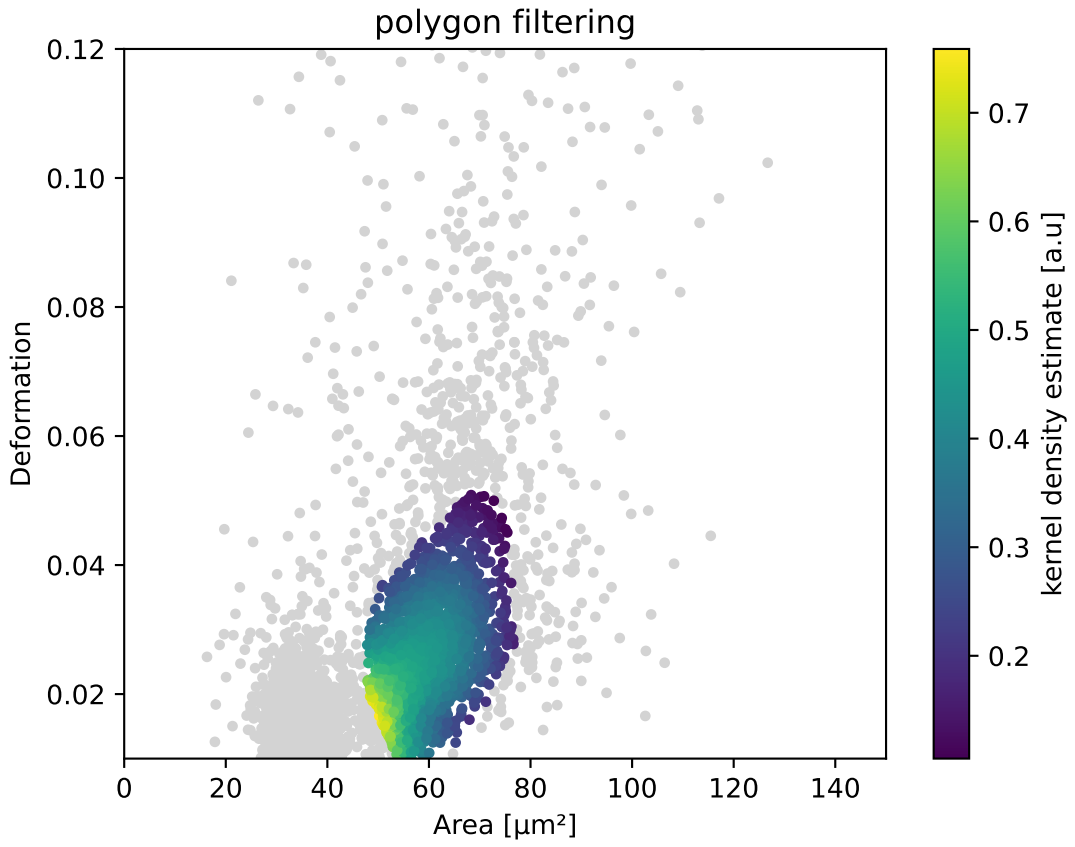
```
import matplotlib.pylab as plt
import dclab
ds = dclab.new_dataset("data/example.rtdc")
kde = ds.get_kde_scatter(xax="area_um", yax="deform")
# load and apply polygon filter from file
```

(continues on next page)


contour lines
0.12
0.10
0.08
0.06
0.04
0.02
Deformation
10th
50th
75th
0
20
40
60
80
100
120
140
Area [µm²]

(continued from previous page)

```
pf = dclab.PolygonFilter(filename="data/example.poly")
ds.polygon_filter_add(pf)
ds.apply_filter()
# valid events
val = ds.filter.all

ax = plt.subplot(111, title="polygon filtering")
ax.scatter(ds["area_um"][~val], ds["deform"][~val], c="lightgray", marker=".")
sc = ax.scatter(ds["area_um"][val], ds["deform"][val], c=kde[val], marker=".")
ax.set_xlabel(dclab.dfn.get_feature_label("area_um"))
ax.set_ylabel(dclab.dfn.get_feature_label("deform"))
ax.set_xlim(0, 150)
ax.set_ylim(0.01, 0.12)
plt.colorbar(sc, label="kernel density estimate [a.u]")
plt.show()
```



4.8 Fluorescence traces

In RT-FDC, fluorescence data are stored alongside the regular image and scalar features. The fluorescence data consist of the trace data (fluorescence signal over time) and several scalar features (maximum, peak position, peak width, etc.) for each fluorescence channel. The trace data are stored as *raw* and *median-filtered* traces, where *median-filtered* means that the *raw* data is filtered with a rolling median filter.

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example_traces.rtdc")

# list the available traces in the dataset
In [3]: sorted(ds["trace"].keys())
Out[3]: ['fl1_median', 'fl1_raw', 'fl2_median', 'fl2_raw', 'fl3_median', 'fl3_raw']

# show fluorescence meta data
In [4]: ds.config["fluorescence"]
Out[4]: {'bit depth': 16, 'channel 1 name': '525/50', 'channel 2 name': '593/46',
 ↪'channel 3 name': '700/75', 'channel count': 3, 'channels installed': 3, 'laser 1␣
 ↪lambda': 488.0, 'laser 1 power': 8.0, 'laser 3 lambda': 640.0, 'laser 3 power': 100.0,
 ↪'laser count': 2, 'lasers installed': 3, 'sample rate': 312500, 'samples per event':␣
 ↪177, 'signal max': 1.0, 'signal min': -1.0, 'trace median': 0}
```

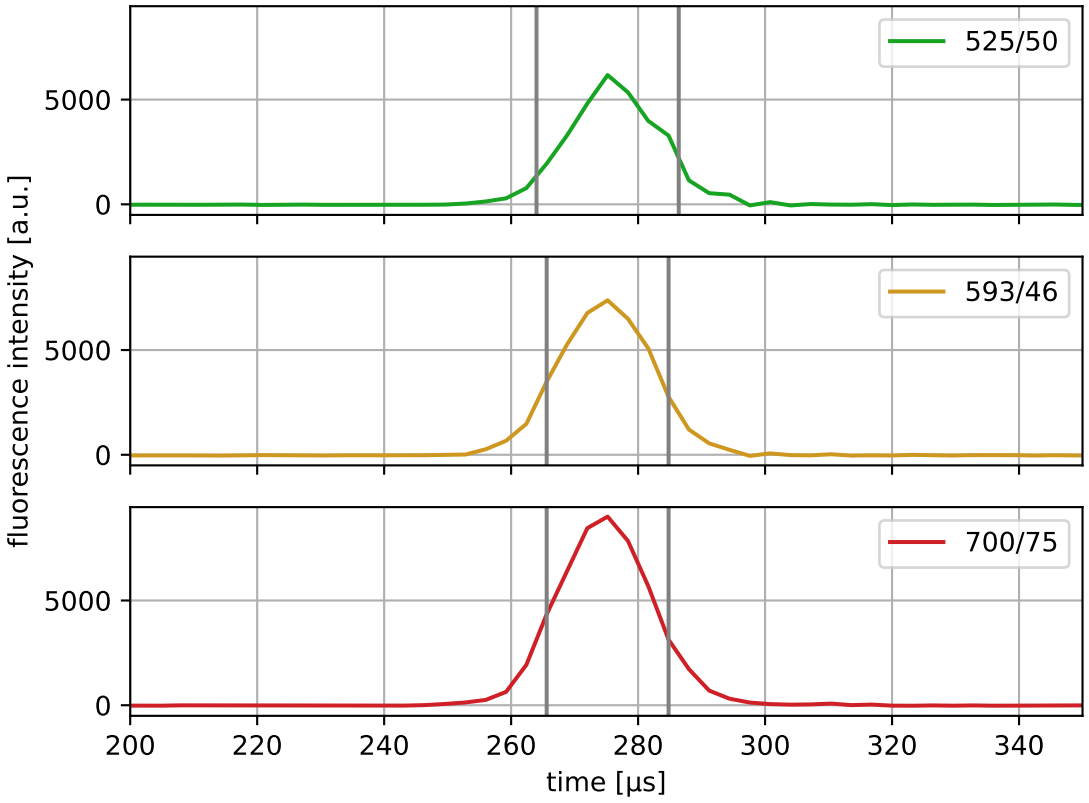
Please note that the value of `trace median` is zero (no median filter applied), which tells us that the values of the *raw* and *median* trace data are identical. The example dataset is an excerpt from the calibration beads dataset, with a total of three fluorescence channels used.

```
import matplotlib.pylab as plt
import dclab

ds = dclab.new_dataset("data/example_traces.rtdc")
# event index to plot
idx = 8
# measuring time
samples = ds.config["fluorescence"]["samples per event"]
sample_rate = ds.config["fluorescence"]["sample rate"]
t = np.arange(samples) / sample_rate * 1e6

fig, axes = plt.subplots(nrows=3, sharex=True, sharey=True)

# fluorescence traces (colors manually chosen to represent filter set)
axes[0].plot(t, ds["trace"]["fl1_median"][idx], color="#16A422",
             label=ds.config["fluorescence"]["channel 1 name"])
axes[1].plot(t, ds["trace"]["fl2_median"][idx], color="#CE9720",
             label=ds.config["fluorescence"]["channel 2 name"])
axes[2].plot(t, ds["trace"]["fl3_median"][idx], color="#CE2026",
             label=ds.config["fluorescence"]["channel 3 name"])

# detected peak widths
axes[0].axvline(ds["fl1_pos"][idx] + ds["fl1_width"][idx]/2, color="gray")
axes[0].axvline(ds["fl1_pos"][idx] - ds["fl1_width"][idx]/2, color="gray")
axes[1].axvline(ds["fl2_pos"][idx] + ds["fl2_width"][idx]/2, color="gray")
axes[1].axvline(ds["fl2_pos"][idx] - ds["fl2_width"][idx]/2, color="gray")
```

(continues on next page)

(continued from previous page)

```
axes[2].axvline(ds["fl3_pos"][idx] + ds["fl3_width"][idx]/2, color="gray")
axes[2].axvline(ds["fl3_pos"][idx] - ds["fl3_width"][idx]/2, color="gray")

# axes labels
axes[1].set_ylabel("fluorescence intensity [a.u.]")
axes[2].set_xlabel("time [µs]")

for ax in axes:
    ax.set_xlim(200, 350)
    ax.grid()
    ax.legend()

plt.show()
```



Please note that the fluorescence traces are stored as integer values and have to be converted to µs using the meta data stored in `ds.config["fluorescence"]`. Also, notice how the scalar features are used for plotting the peak width.

4.9 Young's modulus computation

4.9.1 Background

The computation of the Young's modulus makes use of look-up tables (LUTs) which are discussed in detail further below. All LUTs are treated identically with respect to the following correction terms:

- **scaling laws:** The original LUT was computed for a specific channel width L, flow rate Q, and viscosity η. If the experimental values of these parameters differ from those in the simulation, then they must be scaled before interpolating the Young's modulus. The scale conversion rules can be derived from the characteristic length L and stress $\sigma = \eta \cdot Q / L^3$ [MOG+15]. For instance, the event area scales with $(L_{\mathrm{exp}}/L_{\mathrm{LUT}})^2$, the Young's modulus scales with $\sigma_{\mathrm{exp}}/\sigma_{\mathrm{LUT}}$, and the deformation is not scaled as it has no units. Please note that the scaling laws were derived for linear elastic materials and may not be accurate for other materials (e.g. hyperelastic). The scaling laws are implemented in the submodule `dclab.features.emodulus.scale_linear`.
- **pixelation effects**: All features (including deformation and area) are computed from a pixelated contour. This has the effect that deformation is overestimated and area is underestimated (compared to features computed from a "smooth" contour). While a slight change in area does not have a significant effect on the interpolated Young's modulus, a systematic error in deformation may lead to a strong underestimation of the Young's modulus. A deeper analysis is visualized in the plot pixelation_correction.png which was created with pixelation_correction.py. Thus, before interpolation, the measured deformation must be corrected using a hard-coded correction function [Her17]. The pixelation correction is implemented in the submodule `dclab.features.emodulus.pxcorr`.
- **shear-thinning and temperature-dependence**: The viscosity of a medium usually is a function of temperature. In addition, complex media, such as 0.6% methyl cellulose (CellCarrier B), may also exhibit shear-thinning. The viscosity of such media decreases with increasing flow rates. Since the viscosity is required to apply the scaling laws (above), it must be corrected which is done using hard-coded correction functions as described in the next section.

4.9.2 Viscosity

The computation of the viscosity is implemented in the `corresponding submodule`. For regular RT-DC measurements, the medium used is methyl cellulose (MC) dissolved in phosphate-buffered saline (PBS). For the most common MC concentrations, dclab comes with hard-coded models that compute the corresponding medium viscosity. These models are the original *herold-2017* [Her17] model and the more recent *buyukurganci-2022* [BBN+23, RB23] model.

The MC-PBS solutions show a shear thinning behavior, which can be described by the viscosity η following a power law at sufficienty high shear rates $\dot{\gamma}$:

$$\eta = K \cdot \left(\frac{\dot{\gamma}}{\dot{\gamma}_0} \right)^{n-1},$$

where $\dot{\gamma}$ is the shear rate, K is the flow consistency index and n is the flow behavior index. The shear rate inside a square microchannel cannot be described as a single number for a shear thinning liquid. The shear rate is best described by the shear rate at the channel walls as proposed by Herold [Her17] and can be calculated as follows:

$$\dot{\gamma} = \frac{8Q}{L^3} \left(0.6671 + \frac{0.2121}{n} \right).$$

These considerations are the foundation for the viscosity calculations in the *herold-2017* [Her17] and *buyukurganci-2022* [BBN+23] models.

Note: As discussed in [RB23], the *herold-2017* model inaccurately models the temperature dependency of the MC-PBS viscosity. The temperature dependency was measured using a falling ball viscometer where the change in shear rate

could not be controlled. For the *buyukurganci-2022* model, the temperature dependency was measured as a function of shear rate. Take a look at the *example script that compares these models* to gain more insight.

Warning: Never compare the Young's moduli computed from different viscosity models. Up until dclab 0.47.8, all values of the Young's modulus were computed using the old *herold-2017* model. For new data analysis pipelines, you should use the more accurate *buyukurganci-2022* model.

Büyükurgancı 2022

Büyükurgancı et al. characterized the viscosity curves of three MC-PBS solutions (0.49 w% MC-PBS, 0.59 w% MC-PBS, 0.83 w% MC-PBS) in a temperature range of 22-37 °C. As mentioned above, the viscosity follows a power law behavior for large shear rates.

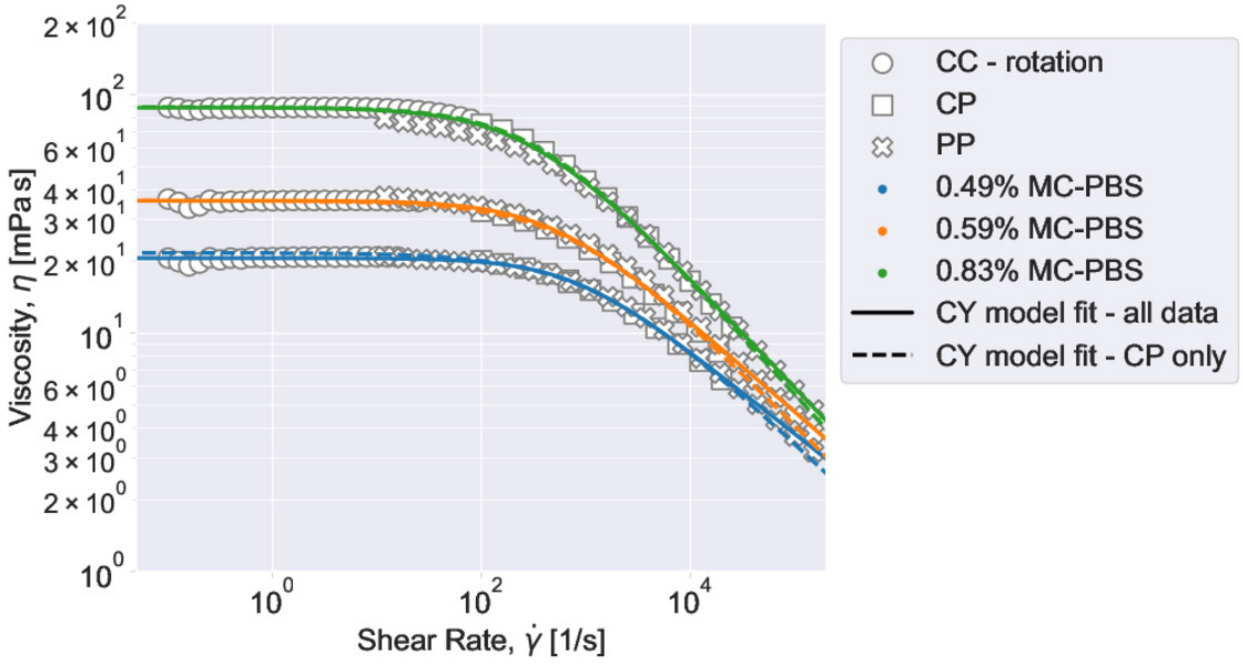


Fig. 4.1: The viscosity of MC-PBS changes from a viscosity plateau at lower shear rates into a power law behavior at higher shear rates, which can be considered fully developed above 5000 1/s. Shear thinning starts at lower shear rates for higher concentrations of MC-PBS, which is typical for polymer solutions. The viscosity was measured using three viscometer designs: Concentric cylinders (CC), cone plate (CP), and parallel disks (PP). See [BBN+23] for details.

The power law parameters K and n were found to be temperature dependent. The temperature dependency can be described as follows:

$$n = \alpha \cdot T + \beta$$
$$K = A \cdot e^{\lambda / T}$$

It was found that α and λ are not dependent on the MC concentration and can be considered material constants of MC dissolved in PBS [BBN+23]. As a result, a global model, valid for the three measured concentrations of MC-PBS was proposed [RB23] and implemented here in `get_viscosity_mc_pbs_buyukurganci_2022()`.

4.9.3 LUT selection

When computing the Young's modulus, the user has to select a LUT via a keyword argument (see next section). The LUT initially implemented in dclab has the identifier "LE-2D-FEM-19".

LE-2D-FEM-19

This LUT was derived from simulations based on the finite elements method (FEM) [MMM+17] and the analytical solution [MOG+15]. The LUT was generated with an incompressible (Poisson's ratio of 0.5) linear elastic sphere model (an artificial viscosity was added to avoid division-by-zero errors) in an axis-symmetric channel (2D). Although the simulations were carried out in this cylindrical symmetry, they can be mapped onto a square cross-sectional channel by adjusting the channel radius to approximately match the desired flow profile. This was done with the spatial scaling factor 1.094 (see also supplement S3 in [MOG+15]). The original data used to generate the LUT are available on figshare [WMM+20].

HE-2D-FEM-22 and HE-3D-FEM-22

These LUTs are based on a hyperelastic neo-Hookean material model for cells with a shear-thinning non-Newtonian fluid (e.g. 0.6% MC-PBS). The simulations were done in cylindrical (2D, with same scaling factor 1.094 as for LE-2D-FEM-19) and square channel (3D) geometries as discussed in [WRM+22]. The original data used to generate these LUTs are available on figshare [WMR+22].

external LUT

If you are generating LUTs yourself, you may register them in dclab using the function *dclab.features.emodulus.load.register_lut()*:

```
import dclab
dclab.features.emodulus.register_lut("/path/to/lut.txt")
```

Please make sure that you adhere to the file format. An example can be found here.

4.9.4 Usage

Since the Young's modulus is model-dependent, it is not made available right away as an *ancillary feature* (in contrast to e.g. event volume or average event brightness).

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example.rtdc")

# "False", because we have not set any additional information.
In [3]: "emodulus" in ds
Out[3]: False
```

Additional information is required. There are three scenarios:

A) The viscosity/Young's modulus is computed individually from the chip temperature for **each** event:

- The *temp* feature which holds the chip temperature of each event
- The configuration key [calculation]: 'emodulus lut'

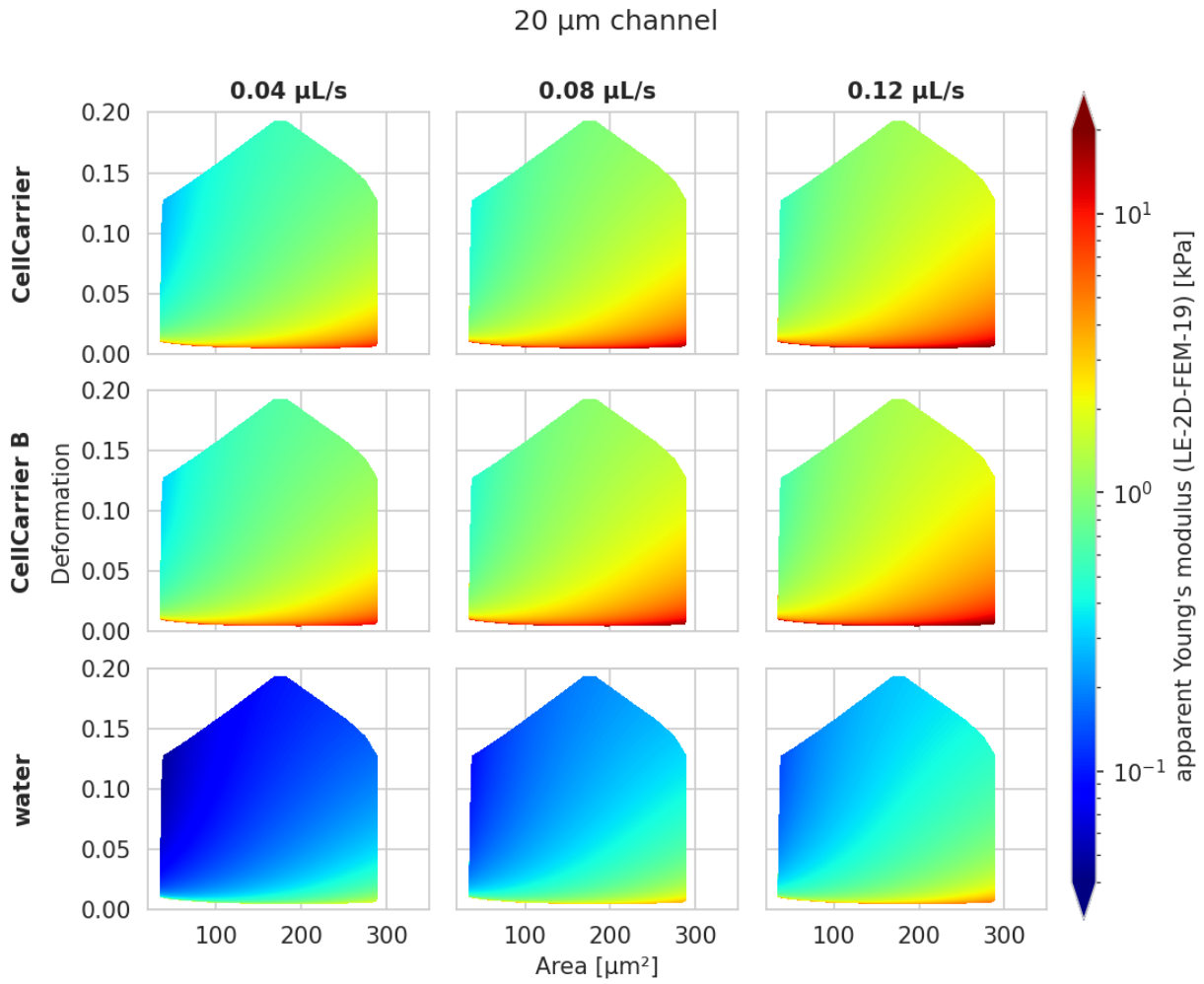


Fig. 4.2: Visualizations of the support and the values of the look-up table (LUT) 'LE-2D-FEM-19' used for determining the Young's modulus from deformation and cell area. The values of the Young's moduli in the regions shown depend on the channel size, the flow rate, the temperature, and the viscosity of the medium [MOG+15]. Here, they are computed for a 20 µm wide channel at 23°C using the viscosity model *buyukurganci-2022* with an effective pixel size of 0.34 µm. The data are corrected for pixelation effects according to [Her17].

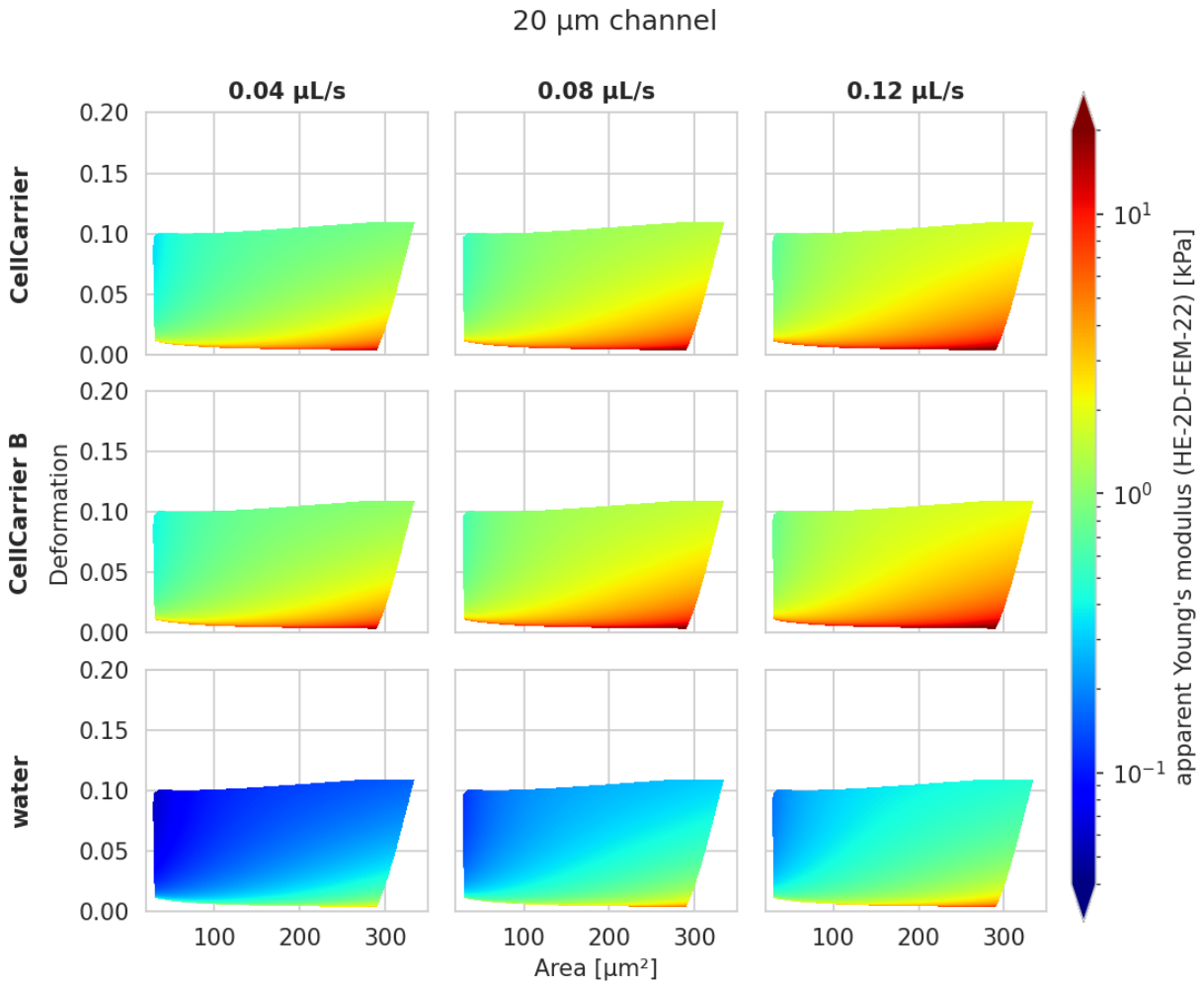


Fig. 4.3: Visualizations of the support and the values of the look-up table (LUT) 'HE-2D-FEM-22' [WRM+22] for a 20 µm wide channel at 23°C (*buyukurganci-2022* model) with an effective pixel size of 0.34 µm. The data are corrected for pixelation effects according to [Her17].

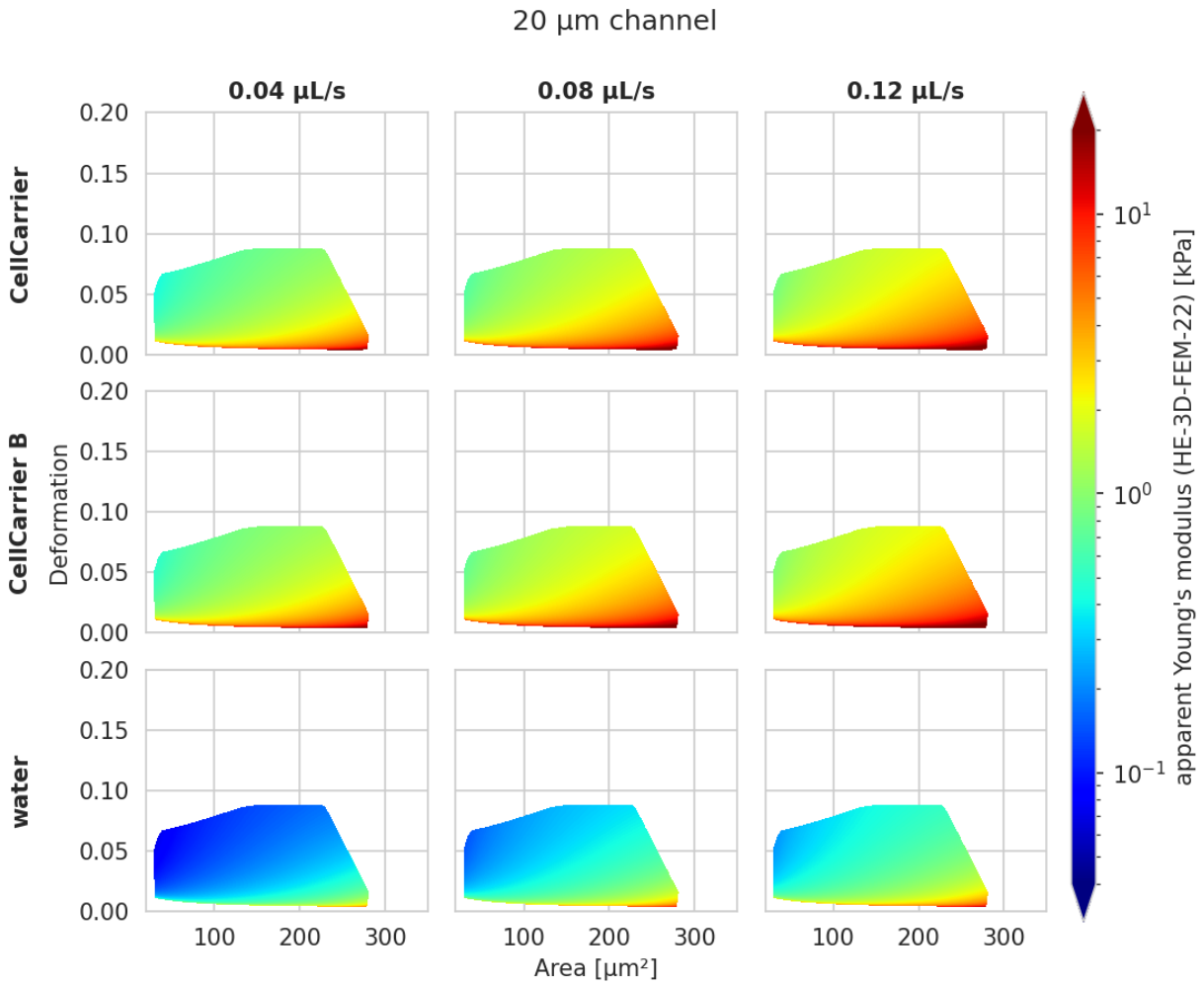


Fig. 4.4: Visualizations of the support and the values of the look-up table (LUT) 'HE-3D-FEM-22' [WRM+22] for a 20 µm wide channel at 23°C (*buyukurganci-2022* model) with an effective pixel size of 0.34 µm. The data are corrected for pixelation effects according to [Her17].

- The configuration key [calculation]: 'emodulus medium'
- The configuration key [calculation]: 'emodulus viscosity model'

B) Set a global viscosity in [mPa·s]. Use this if you have measured the viscosity of your medium (and know all there is to know about shear thinning [Her17] and temperature dependence):

- The configuration key [calculation]: 'emodulus lut'
- The configuration key [calculation]: 'emodulus viscosity'

C) Compute the Young's modulus using the viscosities of known media for a fixed temperature:

  - The configuration key [calculation]: 'emodulus lut'
  - The configuration key [calculation]: 'emodulus medium'
  - The configuration key [calculation]: 'emodulus temperature'
  - The configuration key [calculation]: 'emodulus viscosity model'

  Note that if 'emodulus temperature' is given, then this temperature is used, even if the *temp* feature exists (scenario A).

Description of the configuration keywords:

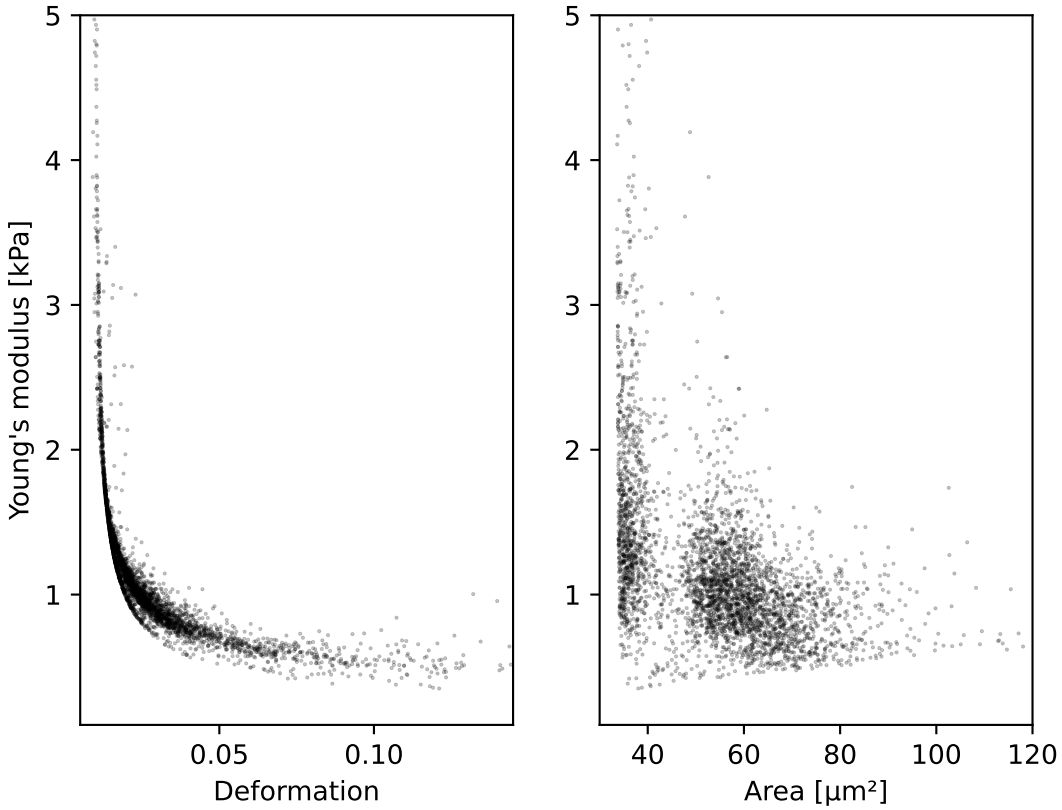
- 'emodulus lut': This is the LUT identifier (see previous section).
- 'emodulus medium': This must be one of the supported media defined in *dclab.features.emodulus.viscosity.KNOWN_MEDIA* and can be taken from the configuration key *[setup]: 'medium'*.
- 'emodulus temperature': is the mean chip temperature and could possibly be available in [setup]: 'temperature'.
- 'emodulus viscosity model': This is the viscosity model key to use (see *Viscosity* above). This key was introduced in dclab 0.48.0.

```
import matplotlib.pylab as plt

import dclab

ds = dclab.new_dataset("../data/example.rtdc")

# Add additional information. We cannot go for (A), because this example
# does not have the temperature feature (`"temp" not in ds`). We go for
# (C), because the beads were measured in a known medium.
ds.config["calculation"]["emodulus lut"] = "LE-2D-FEM-19"
ds.config["calculation"]["emodulus medium"] = ds.config["setup"]["medium"]
ds.config["calculation"]["emodulus temperature"] = 23.0  # a guess
ds.config["calculation"]["emodulus viscosity model"] = 'buyukurganci-2022'

# Plot a few features
ax1 = plt.subplot(121)
ax1.plot(ds["deform"], ds["emodulus"], ".", color="k", markersize=1, alpha=.3)
ax1.set_ylim(0.1, 5)
ax1.set_xlim(0.005, 0.145)
ax1.set_xlabel(dclab.dfn.get_feature_label("deform"))
ax1.set_ylabel(dclab.dfn.get_feature_label("emodulus"))

ax2 = plt.subplot(122)
ax2.plot(ds["area_um"], ds["emodulus"], ".", color="k", markersize=1, alpha=.3)
ax2.set_ylim(0.1, 5)
```

(continues on next page)

(continued from previous page)

```
ax2.set_xlim(30, 120)
ax2.set_xlabel(dclab.dfn.get_feature_label("area_um"))

plt.show()
```



4.10 Linear mixed-effects models

It is not straightforward to define a p-Value for RT-DC data (e.g. change in deformation for a treatment vs. its control). This is somewhat counter-intuitive, because one could assume that the large number of events in a single dataset should be enough to compare two datasets. However, Focus changes, chip-to-chip variations, etc. may generate systematic offsets which make a direct comparison (e.g. t-Test) impossible. Linear mixed effect models (LMM) allow to assign a significance to a treatment measurement compared to a control measuerement (fixed effect) while considering the systematic bias in-between the measurement repetitions (random effect).

dclab offers LMM analysis as described in [HMMO18]. The LMM analysis is performed using the lme4 R package.

4.10.1 Computing p-values with lme4 in dclab

dclab exposes two models from lme4:

- **linear mixed-effects models** ("lmer"): This is basically the simplest way of determining whether or not a treatment has an effect.
- **generalized linear mixed-effects models with a log-link function** ("glmer+loglink"): This model makes use of lme4's generalized linear effects model (GLMM) `glmer` function with a log-link function (`family=Gamma(link='log')`). This is used for data that is log-normally distributed. Log-normal behaviour is quite common, especially in biology. When a physical parameter has a lower limit, and the measured values are close to that limit, the resulting distribution will be skewed, resembling a log-normal distribution. In case of RT-DC this is specially (but not only) true for deformation. Another example is area, which also has a lower limit of zero and may therefore have a skewed distribution. While GLMMs are designed to handle skewed data, it was shown that LMMs already deliver robust results, even for highly skewed data [GH06].

Warning: The decision whether to use LMM or GLMM is not particularly important. Ideally, both LMM and GLMM are consistent. However, never perform both analyses only to then pick the one with the lowest p-value. This is p-hacking! The analysis routine should be defined beforehand. If in doubt, stick to LMM.

An LMM analysis is straight-forward in dclab:

```
import dclab
from dclab import lme4

# Load the data
ds_rep1_ctl = dclab.new_dataset(...)  # control measurement, 1st repetition
ds_rep1_trt = dclab.new_dataset(...)  # treatment measurement, 1st repetition
ds_rep2_ctl = dclab.new_dataset(...)  # control measurement, 2nd repetition
ds_rep2_trt = dclab.new_dataset(...)  # treatment measurement, 2nd repetition

# Instantiate Rlme4
rlme4 = lme4.Rlme4(model="lmer", feature="deform")

# Add the datasets
rlme4.add_dataset(ds=ds_rep1_ctl, group="control", repetition=1)
rlme4.add_dataset(ds=ds_rep1_trt, group="treatment", repetition=1)
rlme4.add_dataset(ds=ds_rep2_ctl, group="control", repetition=2)
rlme4.add_dataset(ds=ds_rep2_ctl, group="treatment", repetition=2)

# Perform the analysis
result = rlme4.fit()
print("p-value:", result["anova p-value"])
print("fixed effect:", result["fixed effects treatment"])
print("model converged:", result["model converged"])
```

The `fit()` function returns the most important results and also exposes some of the underlying R objects (see *dclab.lme4.wrapr.Rlme4.fit()*). An LMM example is also given in the *example section*.

Note: If a treatment and a control share the same repetition number, it is implied that they are paired. For those measurements, lme4 will perform a paired test. In your experimental design you determine which measurements are paired, before doing any experiments. Pairing can be done e.g. for measurements done on the same day or on the same chip. In cases where you perform the control measurements on one day and the treatment measurements on another day,

you could still pair them. Just keep in mind that this could introduce systematic errors, if the measurement conditions (temperature, illumination, etc.) were not identical. Under no circumstances, choose a pairing that yields the lowest p-value (p-hacking).

Alternatively, you can also run an unpaired test by just giving each measurement a different repetition number. For example for 3x control and 3x treatment measurements, you could enumerate the repetition number from 1 to 6.

4.10.2 Differential feature analysis with reservoir data

The (G)LMM analysis is only applicable if the feature chosen is not pronounced visibly in the reservoir measurements. For instance, if a treatment results in a significant change in deformation already in the reservoir, then the p-value determined for the channel data might be underestimated (too many stars). In this case, the information of the reservoir measurement must be included by means of differential deformation [HMMO18]. The idea of differential deformation is to subtract the reservoir from the channel deformation. Since it is not possible to assign the events in the reservoir to the events in the channel (two different measurements), bootstrapping is employed which generates statistical representations of the two measurements that can then be subtracted from one another. Then, for the actual LMM analysis, only the differential deformation is used.

To perform a differential feature analysis, simply add the reservoir measurements to the *dclab.lme4.wrapr.Rlme4* class (they are recognized as reservoir measurements via their meta data).

```
# Load the data
ds_rep1_ctl = dclab.new_dataset(...)  # control measurement, 1st repetition (channel)
ds_rep1_ctl_res = dclab.new_dataset(...)  # control measurement, 1st repetition␣
↪(reservoir)
[...]

# Instantiate Rlme4
rlme4 = lme4.Rlme4(model="lmer", feature="deform")

# Add the datasets
rlme4.add_dataset(ds=ds_rep1_ctl, group="control", repetition=1)
rlme4.add_dataset(ds=ds_rep1_ctl_res, group="control", repetition=1)
[...]

# Perform the analysis
result = rlme4.fit()
assert results["is differential"]  # adding "reservoir" data forces differential analysis
```

Keep in mind that the analysis is now performed using the differential features and not the actual features (`result["is differential"]`). For more information, please see *dclab.lme4.wrapr.Rlme4.get_differential_dataset()* and *dclab.lme4.wrapr.bootstrapped_median_distributions()*. A full example, including GLMM and differential deformation, is given in the *example section*.

4.11 DCOR access

The deformability cytometry open repository (DCOR) allows you to upload and access RT-DC datasets online (internet connection required). The advantage is that you can access parts of the dataset (e.g. just two features) without downloading the entire data file (which includes image, contour, and traces information).

4.11.1 Public data

When you would previously download an entire dataset and do

```
import dclab
ds = dclab.new_dataset("/path/to/Downloads/calibration_beads.rtdc")
```

you can now skip the download and use the identifier (id) of a DCOR resource like so:

```
import dclab
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0")
```

To determine the DCOR resource id, go to https://dcor.mpl.mpg.de, find the resource you are interested in, scroll down to the bottom, and copy the value from the **id** (**not** *package id* or *revision id*) field in (*Additional Information*). The DCOR format is documented in *DCOR (online) format*.

4.11.2 Private data

If you want to access private data, you need to pass a personal API Token.

```
import dclab
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0",
                       api_key="XXXX-YYYY-ZZZZ")
```

Alternatively, you can also set an API Token globally using

```
import dclab
from dclab.rtdc_dataset.fmt_dcor.api import APIHandler
APIHandler.add_api_key("XXXX-YYYY-ZZZZ")
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0")
```

4.11.3 Managing API Tokens

You can manage your API Tokens on your profile page when logged in at https://dcor.mpl.mpg.de.

- Deleting a token:

  Click on the tab "API Tokens" to view all currently existing tokens and the date they were last accessed. By clicking on the red "X" you can delete a token. It cannot be restored, so be careful when deleting tokens!

- Creating a new token:

  To create a new token, insert a name in the field at the top and click "Create API Token". The newly generated token will be shown at the top of the page. Make sure you copy it directly, because you will not be able to recall it again!

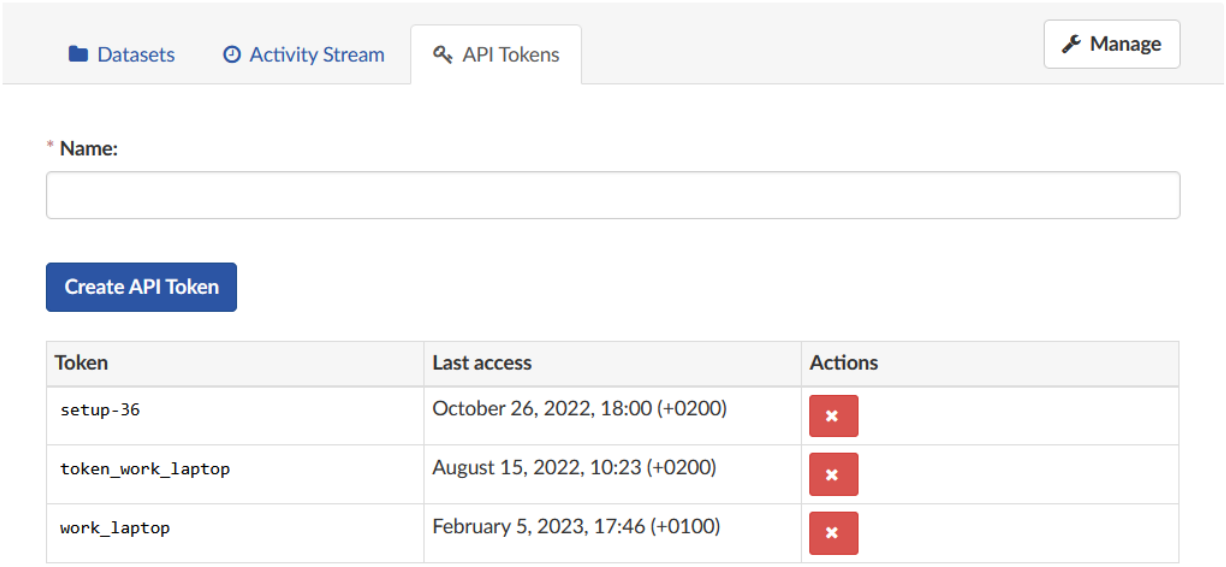


Fig. 4.5: Managing API Tokens on DCOR.

4.11.4 Accessing data on a different DCOR instance

To access data on a different DCOR instance, you have to pass the respective URL when opening the dataset via the keyword `host`. The procedure to retrieve the DCOR resource id is the same as for the default DCOR.

```
import dclab
ds = dclab.new_dataset("fb719fb2-bd9f-817a-7d70-f4002af916f0",
                       host="dcor-dev.mpl.mpg.de")
```

4.11.5 Bypassing DCOR and using S3 directly

The *DCOR format* connects to the dcserv API on on the DCOR server side. Internally, DCOR uses an S3-compatible object store to manage all resources. In some scenarios you might want to bypass this API and access individual DCOR resources directly.

Advantages:

- potentially faster access to HDF5 data using the *S3 format* or other software, since the `dcserv` wrapper is bypassed
- you don't have to depend on dclab in your code

Disadvantages:

- no direct access to private resources: You either need to use the `dcserv` API to obtain a presigned S3 URL (which also has an expiry date) or you need to own S3 credentials for the object store.
- no direct access to features from the condensed file: DCOR automatically computes a condensed file upon upload. This file contains only (but more) scalar features. The `dcserv` API transparently combines features from the original and the condensed file.

Resources are stored in the following pattern by DCOR:

```
https://{endpoint_domain}/{instance-specific-prefix}{circle-id}/resource/{resource-id}
```

For instance, the calibration beads dataset, has this S3 URL:

```
https://objectstore.hpccloud.mpcdf.mpg.de/circle-5a7a053d-55fb-4f99-960c-f478d0bd418f/
↪resource/fb7/19f/b2-bd9f-817a-7d70-f4002af916f0
```

You can access condensed resources by replacing `resource` with `condensed` in the above URL:

```
https://objectstore.hpccloud.mpcdf.mpg.de/circle-5a7a053d-55fb-4f99-960c-f478d0bd418f/
↪condensed/fb7/19f/b2-bd9f-817a-7d70-f4002af916f0
```

4.12 S3 access

Since DC datasets can become quite large, it often makes sense to put them somewhere centrally, such as a shared network drive or *DCOR*. You may also choose to upload your files directly to an S3-compatible object store, which dclab supports as well (It is actually in integral part of the DCOR format).

4.12.1 Public data

Opening public datasets on S3 is straight forward. To get started, you only need to know the URL of the object:

```
import dclab
s3_url = "https://objectstore.hpccloud.mpcdf.mpg.de/circle-5a7a053d-55fb-4f99-960c-
↪f478d0bd418f/resource/fb7/19f/b2-bd9f-817a-7d70-f4002af916f0"
ds = dclab.new_dataset(s3_url)
print(ds.config)
```

4.12.2 Private data

Accessing private data requires you to pass the key ID and the access secret like so:

```
import dclab
s3_url = "..."
ds = dclab.new_dataset(s3_url,
                       secret_id="YOUR-S3-KEY-ID",
                       secret_key="YOUR-S3-ACCESS-SECRET")
```

4.13 Machine learning

To simplify machine-learning (ML) tasks in the context of RT-DC, dclab offers a few convenience methods. This section describes the recommended way of implementing and distributing ML models based on RT-DC data. Please make sure that you have installed dclab with the *ml* extra (`pip install dclab[ml]`).

4.13.1 Using models in dclab

For RT-DC analysis, the most common task for ML is to determine the probability for a specific event (e.g. a cell) to belong to a specific class (e.g. red blood cell). Since RT-DC data always has a very specific format, it is worthwile to standardize this regression/classification process.

In dclab, you are not directly using the *bare* models that you would e.g. get from tensorflow/keras. Instead, models are wrapped via a specific *dclab.rtdc_dataset.feat_anc_ml.ml_model.BaseModel* class that holds additional information about the features from which and to which a model maps. For instance, a model might have the inputs `deform` and `area_um` and make predictions regarding a defined output feature, e.g. `ml_score_rbc`. Output features for machine learning are always of the form `ml_score_xxx` where `x` can be any alphanumeric character (you are free to choose).

```
from dclab.rtdc_dataset.feat_anc_ml import hook_tensorflow
import tensorflow as tf

# do your magic
bare_model = tf.keras.Sequential(...)
bare_model.compile(...)
bare_model.fit(...)

# create a dclab model
dc_model = hook_tensorflow.TensorflowModel(
    bare_model=bare_model,
    inputs=["deform", "area_um"],
    outputs=["ml_score_rbc"],
    info={
        "description": "RBC identification",
        "output labels": ["Red Blood Cells"]
        }
    )

# once you get here, you can use your model directly for inference
ds = dclab.new_dataset("path/to/a/dataset")
# `prediction` is a dictionary with the key "ml_score_rbc" mapping
# to a 1D ndarray of length `len(ds)`, holding the probability data.
prediction = dc_model.predict(ds)["ml_score_rbc"]
```

For user convenience, a model can also be registered with dclab as an *ancillary feature*.

```
dclab.MachineLearningFeature(feature_name="ml_score_rbc",
                             dc_model=dc_model)
prediction = ds["ml_score_rbc"]  # same result as above
```

Please have a look at *this example* to see dclab models in action.

4.13.2 The .modc file format

The .modc file format is not a reinvention of the wheel. It is merely a wrapper around other ML file formats and describes which input features (e.g. `deform`, `area_um`, `image`, etc.) a machine learning method maps onto which output features (e.g. `ml_score_rbc`). A .modc file is just a .zip file containing an index.json file that lists all models. A model may be stored in multiple file formats (e.g. as a tensorflow SavedModel **and** as a Frozen Graph). Alongside the models, the .modc file format also contains human-readable versions of the output features, SHA256 checksums, and the creation date:

```
example.modc (ZIP file contents)
├── index.json
├── model_0
│       ├── another-format
│       │       └── another_formats_file.suffix
│       └── tensorflow-SavedModel.tf
│           ├── assets
│           ├── saved_model.pb
│           └── variables
│               ├── variables.data-00000-of-00001
│               └── variables.index
└── model_1
    └── tensorflow-SavedModel.tf
        ├── assets
        ├── saved_model.pb
        └── variables
            ├── variables.data-00000-of-00001
            └── variables.index
```

The corresponding index.json file could look like this:

```
{
  "model count": 2,
  "models": [
    {
      "date": "2020-11-03 17:01",
      "description": "Determine sensitivity X",
      "formats": {
        "tensorflow-SavedModel": "tensorflow-SavedModel.tf",
        "library-OtherFormat": "another-format"
      },
      "index": 0,
      "input features": [
        "deform"
      ],
      "output features": [
        "ml_score_low",
        "ml_score_hig"
      ],
      "output labels": [
        "Low",
        "High"
      ],
      "path": "model_0",
      "sha256": "ec11c73ae870da4551d9fa9cc73271566b8f2356f284d4c2cb02057ecb5bf6ce"
```

(continues on next page)

(continued from previous page)


```
    },
    {
      "date": "2020-11-03 17:02",
      "description": "Find RBCs and sad cells",
      "formats": {
        "tensorflow-SavedModel": "tensorflow-SavedModel.tf"
      },
      "index": 1,
      "input features": [
        "area_um",
        "image"
      ],
      "output features": [
        "ml_score_rbc",
        "ml_score_sad"
      ],
      "output labels": [
        "red blood cells",
        "sad cells"
      ],
      "path": "model_1",
      "sha256": "ac43c73ae870da4551d9fa9cc73271566b8f2356f284d4c2cb02057ecb5ba812"
    }
  ]
}
```


The great advantage of such a file format is that users can transparently exchange machine learning methods and apply them in a reproducible manner to any RT-DC dataset using dclab or Shape-Out.

To save a machine learning model to a .modc file, you can use the *dclab.save_modc* function:

```
dclab.save_modc("path/to/file.modc", dc_model)
```

Conversely, you can load such a model at any time and use it for inference using the *dclab.load_modc*. To directly load the model as an ancillary feature, use `dclab.load_ml_feature`:

```
dclab.load_ml_feature("path/to/file.modc")
prediction = ds["ml_score_rbc"]  # same result as above
```

The methods for saving and loading .modc files are described in the *code reference*.

4.13.3 Helper functions

If you are working with tensorflow, you might find the functions in the *dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow* submodule helpful. Please also have a look at the *machine-learning examples*.

CHAPTER

FIVE

CODE REFERENCE

5.1 Module-level methods

dclab.**new_dataset**(*data*, *identifier=None*, ***kwargs*)

Initialize a new RT-DC dataset

Parameters

- **data** – can be one of the following:
  - dict
  - .tdms file
  - .rtdc file
  - subclass of *RTDCBase* (will create a hierarchy child)
  - DCOR resource URL
  - URL to file in S3-compatible object store
- **identifier** (*str*) – A unique identifier for this dataset. If set to *None* an identifier is generated.
- **kwargs** – Additional parameters passed to the RTDCBase subclass

Returns

dataset – A new dataset instance

Return type

subclass of *dclab.rtdc_dataset.RTDCBase*

5.2 Global definitions

These definitionas are used throughout the dclab/Shape-In/Shape-Out ecosystem.

5.2.1 Metadata

Valid configuration sections and keys are described in: *Analysis metadata* and *Experiment metadata*. You should use the following methods instead of accessing the static metadata constants.

dclab.definitions.**config_key_exists**(*section*, *key*)
Return *True* if the configuration key exists

dclab.definitions.**get_config_value_descr**(*section*, *key*)
Return the description of a config value

Returns *key* if not defined anywhere

dclab.definitions.**get_config_value_func**(*section*, *key*)
Return configuration type converter function

dclab.definitions.**get_config_value_type**(*section*, *key*)
Return the expected type of a config value

Returns *None* if no type is defined

These constants are also available in the dclab.definitions module.

dclab.definitions.meta_const.**CFG_ANALYSIS**
All configuration keywords editable by the user

dclab.definitions.meta_const.**CFG_METADATA**
All read-only configuration keywords for a measurement

dclab.definitions.meta_const.**config_keys**
dict with section as keys and config parameter names as values

5.2.2 Metadata parsers

dclab.definitions.meta_parse.**f1dfloatduple**(*value*)
Tuple of two floats (duple)

dclab.definitions.meta_parse.**f2dfloatarray**(*value*)
numpy floating point array

dclab.definitions.meta_parse.**fbool**(*value*)
boolean

dclab.definitions.meta_parse.**fboolorfloat**(*value*)
Bool or float

dclab.definitions.meta_parse.**fint**(*value*)
integer

dclab.definitions.meta_parse.**fintlist**(*alist*)
A list of integers

dclab.definitions.meta_parse.**lcstr**(*astr*)
lower-case string

dclab.definitions.meta_parse.**func_types = {<class 'float'>: <class 'numbers.Number'>, <function f1dfloatduple>: (<class 'tuple'>, <class 'numpy.ndarray'>), <function f2dfloatarray>: <class 'numpy.ndarray'>, <function fbool>: (<class 'bool'>, <class 'numpy.bool_'>), <function fboolorfloat>: (<class 'bool'>, <class 'numpy.bool_'>, <class 'float'>), <function fint>: <class 'numbers.Integral'>, <function fintlist>: <class 'list'>, <function lcstr>: <class 'str'>}**

maps functions to their expected output types

5.2.3 Features

Features are discussed in more detail in *Features*.

dclab.definitions.**check_feature_shape**(*name*, *data*)

Check if (non)-scalar feature matches with its data's dimensionality

Parameters

- **name** (*str*) – name of the feature
- **data** (*array-like*) – data whose dimensionality will be checked

Raises

ValueError – If the data's shape does not match its scalar description

dclab.definitions.**feature_exists**(*name*, *scalar_only=False*)

Return True if *name* is a valid feature name

This function not only checks whether *name* is in `feature_names`, but also validates against the machine learning scores *ml_score_???* (where *?* can be a digit or a lower-case letter in the English alphabet).

Parameters

- **name** (*str*) – name of a feature
- **scalar_only** (*bool*) – Specify whether the check should only search in scalar features

Returns

valid – True if name is a valid feature, False otherwise.

Return type

bool

See also:

scalar_feature_exists

Wraps *feature_exists* with *scalar_only=True*

dclab.definitions.**get_feature_label**(*name*, *rtdc_ds=None*, *with_unit=True*)

Return the label corresponding to a feature name

This function not only checks `feature_name2label`, but also supports registered *ml_score_???* features.

Parameters

- **name** (*str*) – name of a feature
- **with_unit** (*bool*) – set to False to remove units in square brackets

Returns

label – feature label corresponding to the feature name

Return type
str

Notes

TODO: extract feature label from ancillary information when an rtdc_ds is given.

dclab.definitions.**scalar_feature_exists**(*name*)

Convenience method wrapping *feature_exists(..., scalar_only=True)*

These constants are also available in the `dclab.definitions` module.

dclab.definitions.feat_const.**FEATURES_NON_SCALAR**

list of non-scalar features

dclab.definitions.feat_const.**feature_names**

list of feature names

dclab.definitions.feat_const.**feature_labels**

list of feature labels (same order as *feature_names*

dclab.definitions.feat_const.**feature_name2label**

dict for converting feature names to labels

dclab.definitions.feat_const.**scalar_feature_names**

list of scalar feature names

5.3 RT-DC dataset manipulation

5.3.1 Base class

class dclab.rtdc_dataset.**RTDCBase**(*identifier=None*, *enable_basins=True*)

RT-DC measurement base class

Notes

Besides the filter arrays for each data feature, there is a manual boolean filter array `RTDCBase.filter.manual` that can be edited by the user - a boolean value of `False` means that the event is excluded from all computations.

apply_filter(*force=None*)

Compute the filters for the dataset

basins_get_dicts()

Return the list of dictionaries describing the dataset's basins

basins_retrieve()

Load all basins available

Added in version 0.54.0.

In dclab 0.51.0, we introduced basins, a simple way of combining HDF5-based datasets (including the `HDF5_S3` format). The idea is to be able to store parts of the dataset (e.g. images) in a separate file that could then be located someplace else (e.g. an S3 object store).

If an RT-DC file has "basins" defined, then these are sought out and made available via the *features_basin* property.

Changed in version 0.57.5: "file"-type basins are only available for subclasses that set the *_local_basins_allowed* attribute to True.

`close()`

Close any open files or connections, including basins

If implemented in a subclass, the subclass must call this method via *super*, otherwise basins are not closed. The subclass is responsible for closing its specific file handles.

`get_downsampled_scatter`(*xax='area_um'*, *yax='deform'*, *downsample=0*, *xscale='linear'*, *yscale='linear'*, *remove_invalid=False*, *ret_mask=False*)

Downsampling by removing points at dense locations

Parameters

- **xax** (*str*) – Identifier for x axis (e.g. "area_um", "aspect", "deform")
- **yax** (*str*) – Identifier for y axis
- **downsample** (*int*) – Number of points to draw in the down-sampled plot. This number is either
  - **>=1: exactly downsample to this number by randomly adding** or removing points
  - 0 : do not perform downsampling
- **xscale** (*str*) – If set to "log", take the logarithm of the x-values before performing downsampling. This is useful when data are are displayed on a log-scale. Defaults to "linear".
- **yscale** (*str*) – See *xscale*.
- **remove_invalid** (*bool*) – Remove nan and inf values before downsampling; if set to *True*, the actual number of samples returned might be smaller than *downsample* due to infinite or nan values (e.g. due to logarithmic scaling).
- **ret_mask** (*bool*) – If set to *True*, returns a boolean array of length *len(self)* where *True* values identify the filtered data.

Returns

- **xnew, xnew** (1d ndarray of lenght *N*) – Filtered data; *N* is either identical to *downsample* or smaller (if *remove_invalid==True*)
- **mask** (1d boolean array of length *len(RTDCBase)*) – Array for identifying the downsampled data points

`get_kde_contour`(*xax='area_um'*, *yax='deform'*, *xacc=None*, *yacc=None*, *kde_type='histogram'*, *kde_kwargs=None*, *xscale='linear'*, *yscale='linear'*)

Evaluate the kernel density estimate for contour plots

Parameters

- **xax** (*str*) – Identifier for X axis (e.g. "area_um", "aspect", "deform")
- **yax** (*str*) – Identifier for Y axis
- **xacc** (*float*) – Contour accuracy in x direction
- **yacc** (*float*) – Contour accuracy in y direction
- **kde_type** (*str*) – The KDE method to use

- **kde_kwargs** (*dict*) – Additional keyword arguments to the KDE method
- **xscale** (*str*) – If set to “log”, take the logarithm of the x-values before computing the KDE. This is useful when data are displayed on a log-scale. Defaults to “linear”.
- **yscale** (*str*) – See *xscale*.

Returns
X, Y, Z – The kernel density Z evaluated on a rectangular grid (X,Y).

Return type
coordinates

get_kde_scatter(*xax='area_um'*, *yax='deform'*, *positions=None*, *kde_type='histogram'*, *kde_kwargs=None*, *xscale='linear'*, *yscale='linear'*)

Evaluate the kernel density estimate for scatter plots

Parameters

- **xax** (*str*) – Identifier for X axis (e.g. “area_um”, “aspect”, “deform”)
- **yax** (*str*) – Identifier for Y axis
- **positions** (*list of two 1d ndarrays or ndarray of shape (2, N)*) – The positions where the KDE will be computed. Note that the KDE estimate is computed from the points that are set in *self.filter.all*.
- **kde_type** (*str*) – The KDE method to use, see `kde_methods.methods`
- **kde_kwargs** (*dict*) – Additional keyword arguments to the KDE method
- **xscale** (*str*) – If set to “log”, take the logarithm of the x-values before computing the KDE. This is useful when data are are displayed on a log-scale. Defaults to “linear”.
- **yscale** (*str*) – See *xscale*.

Returns
density – The kernel density evaluated for the filtered data points.

Return type
1d ndarray

static get_kde_spacing(*a*, *scale='linear'*, *method=<function bin_width_doane>*, *method_kw=None*, *feat='undefined'*, *ret_scaled=False*)

Convenience function for computing the contour spacing

Parameters

- **a** (*ndarray*) – feature data
- **scale** (*str*) – how the data should be scaled (“log” or “linear”)
- **method** (*callable*) – KDE method to use (see *kde_methods* submodule)
- **method_kw** (*dict*) – keyword arguments to *method*
- **feat** (*str*) – feature name for debugging
- **ret_scaled** (*bool*) – whether to return the scaled array of *a*

get_measurement_identifier()

Return a unique measurement identifier

Return the [experiment]:”run identifier” configuration feat, if it exists. Otherwise, return the MD5 sum computed from the measurement time, date, and setup identifier.

Returns *None* if no identifier could be found or computed.

Added in version 0.51.0.

polygon_filter_add(*filt*)

Associate a Polygon Filter with this instance

Parameters
filt (int or instance of *PolygonFilter*) – The polygon filter to add

polygon_filter_rm(*filt*)

Remove a polygon filter from this instance

Parameters
filt (int or instance of *PolygonFilter*) – The polygon filter to remove

reset_filter()

Reset the current filter

property basins

Basins containing upstream features from other datasets

config

Configuration of the measurement

export

Export functionalities; instance of *dclab.rtdc_dataset.export.Export*.

property features

All available features

property features_ancillary

All available ancillary features

This includes all ancillary features, excluding the features that are already in *self.features_innate*. This means that there may be overlap between *features_ancillary* and e.g. *self.features_basin*.

Added in version 0.58.0.

property features_basin

All features accessed via upstream basins from other locations

property features_innate

All features excluding ancillary, basin, or temporary features

property features_loaded

All features that have been computed

This includes ancillary features and temporary features.

Notes

Ancillary features that are computationally cheap to compute are always included. They are defined in `dclab.rtdc_dataset.feat_anc_core.FEATURES_RAPID`.

property features_scalar

All scalar features available

property filter

Filtering functionalities; instance of *Filter*

format

Dataset format (derived from class name)

abstract property hash

Reproducible dataset hash (defined by derived classes)

property identifier

Unique (unreproducible) identifier

logs

Dictionary of log files. Each log file is a list of strings (one string per line).

path

Path or DCOR identifier of the dataset (set to "none" for *RTDC_Dict*)

tables

Dictionary of tables. Each table is an indexable compound numpy array.

title

Title of the measurement

5.3.2 HDF5 (.rtdc) format

class dclab.rtdc_dataset.**RTDC_HDF5**(*h5path: str | Path | BinaryIO | IOBase*, *h5kwargs: Dict[str, Any] = None*, **args*, ***kwargs*)

HDF5 file format for RT-DC measurements

Parameters

- **h5path** (*str or pathlib.Path or file-like object*) – Path to an '.rtdc' measurement file or a file-like object
- **h5kwargs** (*dict*) – Additional keyword arguments given to `h5py.File`
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

path

Path to the experimental HDF5 (.rtdc) file

Type
pathlib.Path

basins_get_dicts()

Return list of dicts for all basins defined in *self.h5file*

static can_open(*h5path*)

Check whether a given file is in the .rtdc file format

close()

Close the underlying HDF5 file

static parse_config(*h5path*)

Parse the RT-DC configuration of an HDF5 file

h5path may be a h5py.File object or an actual path

property hash

Hash value based on file name and content

class dclab.rtdc_dataset.fmt_hdf5.basin.**HDF5Basin**(*location: str*, *name: str = None*, *description: str = None*, *features: List[str] = None*, *measurement_identifier: str = None*, *mapping: Literal['same', 'basinmap0', 'basinmap1', 'basinmap2', 'basinmap3', 'basinmap4', 'basinmap5', 'basinmap6', 'basinmap7', 'basinmap8', 'basinmap9'] = 'same'*, *mapping_referrer: Dict = None*, ***kwargs*)

Parameters

- **location** (*str*) – Location of the basin, this can be a path or a URL, depending on the implementation of the subclass
- **name** (*str*) – Human-readable name of the basin
- **description** (*str*) – Lengthy description of the basin
- **features** (*list of str*) – List of features this basin provides; This list is enforced, even if the basin actually contains more features.
- **measurement_identifier** (*str*) – A measurement identifier against which to check the basin. If this is set to None (default), there is no certainty that the downstream dataset is from the same measurement.
- **mapping** (*str*) – Which type of mapping to use. This can be either "same" when the event list of the basin is identical to that of the dataset defining the basin, or one of the "basinmap" features (e.g. "basinmap1") in cases where the dataset consists of a subset of the events of the basin dataset. In the latter case, the feature defined by *mapping* must be present in the dataset and consist of integer-valued indices (starting at 0) for the basin dataset.
- **mapping_referrer** (*dict-like*) – Dict-like object from which "basinmap" features can be obtained in situations where *mapping != "same"*. This can be a simple dictionary of numpy arrays or e.g. an instance of *RTDCBase*.
- **kwargs** – Additional keyword arguments passed to the *load_dataset* method of the *Basin* subclass.
- **versionchanged** (*..*) – Added the *mapping* keyword argument to support basins with a superset of events.

is_available()

Return True if the basin is available

dclab.rtdc_dataset.fmt_hdf5.**MIN_DCLAB_EXPORT_VERSION = '0.3.3.dev2'**

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

5.3.3 DCOR (online) format

class dclab.rtdc_dataset.**RTDC_DCOR**(*url*, *host='dcor.mpl.mpg.de'*, *api_key=''*, *use_ssl=None*, *cert_path=None*, *dcserv_api_version=2*, **args*, ***kwargs*)

Wrap around the DCOR API

Parameters

- **url** (*str*) – Full URL or resource identifier; valid values are
  - https://dcor.mpl.mpg.de/api/3/action/dcserv?id=b1404eb5-f661-4920-be79-5ff4e85915d5
  - dcor.mpl.mpg.de/api/3/action/dcserv?id=b1404eb5-f 661-4920-be79-5ff4e85915d5
  - b1404eb5-f661-4920-be79-5ff4e85915d5
- **host** (*str*) – The default host machine used if the host is not given in *url*
- **api_key** (*str*) – API key to access private resources
- **use_ssl** (*bool*) – Set this to False to disable SSL (should only be used for testing). Defaults to None (does not force SSL if the URL starts with "http://").
- **cert_path** (*pathlib.Path*) – The (optional) path to a server CA bundle; this should only be necessary for DCOR instances in the intranet with a custom CA or for certificate pinning.
- **dcserv_api_version** (*int*) – Version of the dcserv API to use. In version 0.13.2 of ckanext-dc_serve, version 2 was introduced which entails serving an S3-basin-only dataset.
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

path

Full URL to the DCOR resource

Type
str

basins_get_dicts()

Return list of dicts for all basins defined in *self.h5file*

static get_full_url(*url*, *use_ssl*, *host=None*)

Return the full URL to a DCOR resource

Parameters

- **url** (*str*) – Full URL or resource identifier; valid values are
  - https://dcor.mpl.mpg.de/api/3/action/dcserv?id=caab96f6- df12-4299-aa2e-089e390aafd5'

- dcor.mpl.mpg.de/api/3/action/dcserv?id=caab96f6-df12- 4299-aa2e-089e390aafd5
- caab96f6-df12-4299-aa2e-089e390aafd5

- **use_ssl** (*bool or None*) – Set this to False to disable SSL (should only be used for testing). Defaults to None (does not force SSL if the URL starts with "http://").
- **host** (*str*) – Use this host if it is not specified in *url*

property hash

Hash value based on file name and content

class dclab.rtdc_dataset.fmt_dcor.api.**APIHandler**(*url*, *api_key=''*, *cert_path=None*, *dcserv_api_version=2*)

Handles the DCOR api with caching for simple queries

Parameters

- **url** (*str*) – URL to DCOR API
- **api_key** (*str*) – DCOR API token
- **cert_path** (*pathlib.Path*) – the path to the server's CA bundle; by default this will use the default certificates (which depends on from where you obtained certifi/requests)

classmethod add_api_key(*api_key*)

Add an API Key/Token to the base class

When accessing the DCOR API, all available API Keys/Tokens are used to access a resource (trial and error).

api_key

DCOR API Token

api_keys = []

DCOR API Keys/Tokens in the current session

cache_queries = ['metadata', 'size', 'feature_list', 'valid']

these are cached to minimize network usage

dcserv_api_version

ckanext-dc_serve dcserv API version

session

create a session

url

DCOR API URL

verify

keyword argument to `requests.request()`

5.3.4 HTTP (online) file format

class dclab.rtdc_dataset.fmt_http.**HTTPBasin**(**args*, ***kwargs*)

Parameters

- **location** (*str*) – Location of the basin, this can be a path or a URL, depending on the implementation of the subclass
- **name** (*str*) – Human-readable name of the basin
- **description** (*str*) – Lengthy description of the basin
- **features** (*list of str*) – List of features this basin provides; This list is enforced, even if the basin actually contains more features.
- **measurement_identifier** (*str*) – A measurement identifier against which to check the basin. If this is set to None (default), there is no certainty that the downstream dataset is from the same measurement.
- **mapping** (*str*) – Which type of mapping to use. This can be either "same" when the event list of the basin is identical to that of the dataset defining the basin, or one of the "basinmap" features (e.g. "basinmap1") in cases where the dataset consists of a subset of the events of the basin dataset. In the latter case, the feature defined by *mapping* must be present in the dataset and consist of integer-valued indices (starting at 0) for the basin dataset.
- **mapping_referrer** (*dict-like*) – Dict-like object from which "basinmap" features can be obtained in situations where *mapping != "same"*. This can be a simple dictionary of numpy arrays or e.g. an instance of *RTDCBase*.
- **kwargs** – Additional keyword arguments passed to the *load_dataset* method of the *Basin* subclass.
- **versionchanged** (..) – Added the *mapping* keyword argument to support basins with a superset of events.

is_available()

Check for *requests* and object availability

Caching policy: Once this method returns True, it will always return True.

class dclab.rtdc_dataset.fmt_http.**RTDC_HTTP**(*url: str*, **args*, ***kwargs*)

Access RT-DC measurements via HTTP

This class allows you to open .rtdc files accessible via an HTTP URL, for instance files on an S3 object storage or figshare download links.

This is essentially just a wrapper around *RTDC_HDF5* with `HTTPFile` passing a file object to h5py.

Parameters

- **url** (*str*) – Full URL to an HDF5 file
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

path

The URL to the object

Type
str

Notes

Since this format still requires random access to the file online, i.e. not the entire file is downloaded, only parts of it, the web server must support range requests.

close()

Close the underlying HDF5 file

5.3.5 S3 (online) file format

class dclab.rtdc_dataset.fmt_s3.**RTDC_S3**(*url: str*, *endpoint_url: str = None*, *access_key_id: str = None*, *secret_access_key: str = None*, *use_ssl: bool = True*, **args*, ***kwargs*)

Access RT-DC measurements in an S3-compatible object store

This is essentially just a wrapper around *RTDC_HDF5* with `boto3` and `HTTPFile` passing a file object to h5py.

Parameters

- **url** (*str*) – URL to an object in an S3 instance; this can be either a full URL (including the endpoint), or just *bucket/key*
- **access_key_id** (*str*) – S3 access identifier
- **secret_access_key** (*str*) – Secret S3 access key
- **use_ssl** (*bool*) – Whether to enforce SSL (defaults to True)
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

path

The URL to the object

Type
str

close()

Close the underlying HDF5 file

class dclab.rtdc_dataset.fmt_s3.**S3Basin**(**args*, ***kwargs*)

Parameters

- **location** (*str*) – Location of the basin, this can be a path or a URL, depending on the implementation of the subclass
- **name** (*str*) – Human-readable name of the basin
- **description** (*str*) – Lengthy description of the basin
- **features** (*list of str*) – List of features this basin provides; This list is enforced, even if the basin actually contains more features.
- **measurement_identifier** (*str*) – A measurement identifier against which to check the basin. If this is set to None (default), there is no certainty that the downstream dataset is from the same measurement.

- **mapping** (*str*) – Which type of mapping to use. This can be either "same" when the event list of the basin is identical to that of the dataset defining the basin, or one of the "basinmap" features (e.g. "basinmap1") in cases where the dataset consists of a subset of the events of the basin dataset. In the latter case, the feature defined by *mapping* must be present in the dataset and consist of integer-valued indices (starting at 0) for the basin dataset.
- **mapping_referrer** (*dict-like*) – Dict-like object from which "basinmap" features can be obtained in situations where *mapping != "same"*. This can be a simple dictionary of numpy arrays or e.g. an instance of *RTDCBase*.
- **kwargs** – Additional keyword arguments passed to the *load_dataset* method of the *Basin* subclass.
- **versionchanged** (..) – Added the *mapping* keyword argument to support basins with a superset of events.

is_available()

Check for boto3 and object availability

Caching policy: Once this method returns True, it will always return True.

class dclab.rtdc_dataset.fmt_s3.**S3File**(*object_path: str*, *endpoint_url: str*, *access_key_id: str = ''*, *secret_access_key: str = ''*, *use_ssl: bool = True*, *verify_ssl: bool = True*)

Monkeypatched *HTTPFile* to support authenticated access to S3

Parameters

- **object_path** (*str*) – bucket/key path to object in the object store
- **endpoint_url** (*str*) – the explicit endpoint URL for accessing the object store
- **access_key_id** – S3 access key
- **secret_access_key** – secret S3 key mathcing *access_key_id*
- **use_ssl** (*bool*) – use SSL to connect to the endpoint, only disabled for testing
- **verify_ssl** (*bool*) – make sure the SSL certificate is sound, only used for testing

close()

Close the file

This closes the requests session and then calls *close* on the super class.

download_range(*start*, *stop*)

Download bytes given by the range (*start*, *stop*)

stop is not inclusive (In the HTTP range request it normally is).

dclab.rtdc_dataset.fmt_s3.**get_endpoint_url**(*url*)

Given a URL of an S3 object, return the endpoint URL

Return None if no endpoint URL can be extracted (e.g. because just *bucket_name/object_path* was passed).

dclab.rtdc_dataset.fmt_s3.**get_object_path**(*url*)

Given a URL of an S3 object, return the *bucket_name/object_path* part

Return object paths always without leading slash /.

dclab.rtdc_dataset.fmt_s3.**is_s3_object_available**(*url: str*, *access_key_id: str = None*, *secret_access_key: str = None*)

Check whether an S3 object is available

Parameters

- **url** (*str*) – full URL to the object
- **access_key_id** (*str*) – S3 access identifier
- **secret_access_key** (*str*) – Secret S3 access key

dclab.rtdc_dataset.fmt_s3.**is_s3_url**(*string*)

Check whether *string* is a valid S3 URL using regexp

dclab.rtdc_dataset.fmt_s3.**REGEXP_S3_URL = re.compile('^(https?:\\/\\/)([a-z0-9-\\.]*)(\\:[0-9]*)?\\/.+\\/.+')**

Regular expression for matching a DCOR resource URL

5.3.6 Dictionary format

class dclab.rtdc_dataset.**RTDC_Dict**(*ddict*, **args*, ***kwargs*)

Dictionary-based RT-DC dataset

Parameters

- **ddict** (*dict*) – Dictionary with features as keys (valid features like "area_cvx", "deform", "image" are defined by *dclab.definitions.feature_exists*) with which the class will be instantiated. The configuration is set to the default configuration of dclab.

  Changed in version 0.27.0: Scalar features are automatically converted to arrays.
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

property hash

Reproducible dataset hash (defined by derived classes)

5.3.7 Hierarchy format

class dclab.rtdc_dataset.**RTDC_Hierarchy**(*hparent*, *apply_filter=True*, **args*, ***kwargs*)

Hierarchy dataset (filtered from RTDCBase)

A few words on hierarchies: The idea is that a subclass of RTDCBase can use the filtered data of another subclass of RTDCBase and interpret these data as unfiltered events. This comes in handy e.g. when the percentage of different subpopulations need to be distinguished without the noise in the original data.

Children in hierarchies always update their data according to the filtered event data from their parent when *apply_filter* is called. This makes it easier to save and load hierarchy children with e.g. Shape-Out and it makes the handling of hierarchies more intuitive (when the parent changes, the child changes as well).

Parameters

- **hparent** (*instance of* RTDCBase) – The hierarchy parent
- **apply_filter** (*bool*) – Whether to apply the filter during instantiation; If set to *False*, *apply_filter* must be called manually.

- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

hparent

Hierarchy parent of this instance

Type
RTDCBase

apply_filter(**args*, ***kwargs*)

Overridden *apply_filter* to perform tasks for hierarchy child

get_root_parent()

Return the root parent of this dataset

rejuvenate()

Redraw the hierarchy tree, updating config and features

You should call this function whenever you change something in the hierarchy parent(s), be it filters or metadata for computing ancillary features.

property basins

Basins containing upstream features from other datasets

property features

All available features

property features_ancillary

All available ancillary features

This includes all ancillary features, excluding the features that are already in *self.features_innate*. This means that there may be overlap between *features_ancillary* and e.g. *self.features_basin*.

Added in version 0.58.0.

property features_basin

All features accessed via upstream basins from other locations

property features_innate

All features excluding ancillary, basin, or temporary features

property features_loaded

All features that have been computed

This includes ancillary features and temporary features.

Notes

Ancillary features that are computationally cheap to compute are always included. They are defined in `dclab.rtdc_dataset.feat_anc_core.FEATURES_RAPID`.

property features_scalar

All scalar features available

property hash

Hashes of a hierarchy child changes if the parent changes

hparent

hierarchy parent

property logs

property tables

5.3.8 TDMS format

class dclab.rtdc_dataset.**RTDC_TDMS**(*tdms_path*, **args*, ***kwargs*)

TDMS file format for RT-DC measurements

Parameters

- **tdms_path** (*str or pathlib.Path*) – Path to a '.tdms' measurement file.
- ***args** – Arguments for *RTDCBase*
- ****kwargs** – Keyword arguments for *RTDCBase*

path

Path to the experimental dataset (main .tdms file)

Type
pathlib.Path

dclab.rtdc_dataset.fmt_tdms.**get_project_name_from_path**(*path*, *append_mx=False*)

Get the project name from a path.

For a path "/home/peter/hans/HLC12398/online/M1_13.tdms" or For a path "/home/peter/hans/HLC12398/online/data/M1_13.tdms" or without the ".tdms" file, this will return always "HLC12398".

Parameters

- **path** (*str or pathlib.Path*) – path to tdms file
- **append_mx** (*bool*) – append measurement number, e.g. "M1"

dclab.rtdc_dataset.fmt_tdms.**get_tdms_files**(*directory*)

Recursively find projects based on '.tdms' file endings

Searches the *directory* recursively and return a sorted list of all found '.tdms' project files, except fluorescence data trace files which end with *_traces.tdms*.

5.3.9 Basin features

With basins, you can create analysis pipelines that result in output files which, when opened in dclab, can access features stored in the input file (without having to write those features to the output file).

exception dclab.rtdc_dataset.feat_basin.**BasinmapFeatureMissingError**

Used when one of the *basinmap* features is not defined

class dclab.rtdc_dataset.feat_basin.**Basin**(*location: str*, *name: str = None*, *description: str = None*, *features: List[str] = None*, *measurement_identifier: str = None*, *mapping: Literal['same', 'basinmap0', 'basinmap1', 'basinmap2', 'basinmap3', 'basinmap4', 'basinmap5', 'basinmap6', 'basinmap7', 'basinmap8', 'basinmap9'] = 'same'*, *mapping_referrer: Dict = None*, ***kwargs*)

A basin represents data from an external source

The external data must be a valid RT-DC dataset, subclasses should ensure that the corresponding API is available.

Parameters

- **location** (*str*) – Location of the basin, this can be a path or a URL, depending on the implementation of the subclass
- **name** (*str*) – Human-readable name of the basin
- **description** (*str*) – Lengthy description of the basin
- **features** (*list of str*) – List of features this basin provides; This list is enforced, even if the basin actually contains more features.
- **measurement_identifier** (*str*) – A measurement identifier against which to check the basin. If this is set to None (default), there is no certainty that the downstream dataset is from the same measurement.
- **mapping** (*str*) – Which type of mapping to use. This can be either "same" when the event list of the basin is identical to that of the dataset defining the basin, or one of the "basinmap" features (e.g. "basinmap1") in cases where the dataset consists of a subset of the events of the basin dataset. In the latter case, the feature defined by *mapping* must be present in the dataset and consist of integer-valued indices (starting at 0) for the basin dataset.
- **mapping_referrer** (*dict-like*) – Dict-like object from which "basinmap" features can be obtained in situations where *mapping != "same"*. This can be a simple dictionary of numpy arrays or e.g. an instance of *RTDCBase*.
- **kwargs** – Additional keyword arguments passed to the *load_dataset* method of the *Basin* subclass.
- **versionchanged** (..) – Added the *mapping* keyword argument to support basins with a superset of events.

as_dict()

Return basin kwargs for RTDCWriter.store_basin()

Note that each subclass of *RTDCBase* has its own implementation of RTDCBase.basins_get_dicts() which returns a list of basin dictionaries that are used to instantiate the basins in RTDCBase.basins_enable(). This method here is only intended for usage with RTDCWriter.store_basin().

close()

Close any open file handles or connections

get_feature_data(*feat*)

Return an object representing feature data of the basin

get_measurement_identifier()

Return the identifier of the basin dataset

abstract is_available()

Return True if the basin is available

load_dataset(*location*, ***kwargs*)

Return an instance of *RTDCBase* for this basin

If the basin mapping (*self.mapping*) is not the same as the referencing dataset

abstract property basin_format

Basin format (*RTDCBase* subclass), e.g. "hdf5" or "s3"

abstract property basin_type

Storage type to use (e.g. "file" or "remote")

property basinmap

Contains the indexing array in case of a mapped basin

description

lengthy description of the basin

property ds

The *RTDCBase* instance represented by the basin

property features

Features made available by the basin

kwargs

additional keyword arguments passed to the basin

location

location of the basin (e.g. path or URL)

mapping

Event mapping strategy. If this is "same", it means that the referring dataset and the basin dataset have identical event indices. If *mapping* is e.g. *basinmap1* then the mapping of the indices from the basin to the referring dataset is defined in *self.basinmap* (copied during initialization of this class from the array in the key *basinmap1* from the dict-like object *mapping_referrer*).

measurement_identifier

measurement identifier of the referencing dataset

name

user-defined name of the basin

class dclab.rtdc_dataset.feat_basin.**BasinAvailabilityChecker**(*basin*, **args*, ***kwargs*)

Helper thread for checking basin availability in the background

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is a list or tuple of arguments for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

dclab.rtdc_dataset.feat_basin.**basin_priority_sorted_key**(*bdict: Dict*)

Yield a sorting value for a given basin that can be used with *sorted*

Basins are normally stored in random order in a dataset. This method brings them into correct order, prioritizing:

- type: "file" over "remote"
- format: "HTTP" over "S3" over "dcor"
- mapping: "same" over anything else

5.3.10 Ancillaries

Computation of ancillary features

Ancillary features are computed on-the-fly in dclab if the required data are available. The features are registered here and are computed when *RTDCBase.__getitem__* is called with the respective feature name. When *RTDCBase.__contains__* is called with the feature name, then the feature is not yet computed, but the prerequisites are evaluated:

```
In [1]: import dclab

In [2]: ds = dclab.new_dataset("data/example.rtdc")

In [3]: ds.config["calculation"]["emodulus lut"] = "LE-2D-FEM-19"

In [4]: ds.config["calculation"]["emodulus medium"] = "CellCarrier"

In [5]: ds.config["calculation"]["emodulus temperature"] = 23.0

In [6]: ds.config["calculation"]["emodulus viscosity model"] = 'buyukurganci-2022'

In [7]: "emodulus" in ds  # nothing is computed yet
Out[7]: True

In [8]: ds["emodulus"]  # now data are computed and cached
Out[8]:
array([1.11112189, 0.98155247,        nan, ...,        nan,        nan,
       0.68137091])
```

Once the data has been computed, *RTDCBase* caches it in the *_ancillaries* property dict together with a hash that is computed with *AncillaryFeature.hash*. The hash is computed from the feature data *req_features* and the configuration metadata *req_config*.

exception dclab.rtdc_dataset.feat_anc_core.ancillary_feature.**BadFeatureSizeWarning**

class dclab.rtdc_dataset.feat_anc_core.ancillary_feature.**AncillaryFeature**(*feature_name*, *method*, *req_config=None*, *req_features=None*, *req_func=<function AncillaryFeature.<lambda>>*, *priority=0*, *data=None*, *identifier=None*)

A data feature that is computed from existing data

Parameters

- **feature_name** (*str*) – The name of the ancillary feature, e.g. "emodulus".
- **method** (*callable*) – The method that computes the feature. This method takes an instance of *RTDCBase* as argument.
- **req_config** (*list*) – Required configuration parameters to compute the feature, e.g. ["calculation", ["emodulus lut", "emodulus viscosity"]]
- **req_features** (*list*) – Required existing features in the dataset, e.g. ["area_cvx", "deform"]
- **req_func** (*callable*) – A function that takes an instance of *RTDCBase* as an argument and checks whether any other necessary criteria are met. By default, this is a lambda function that returns True. The function should return False if the necessary criteria are not met. This function may also return a hashable object (via `dclab.util.objstr()`) instead of True, if the criteria are subject to change. In this case, the return value is used for identifying the cached ancillary feature.

  Changed in version 0.27.0: Support non-boolean return values for caching purposes.
- **priority** (*int*) – The priority of the feature; if there are multiple AncillaryFeature defined for the same feature_name, then the priority of the features defines which feature returns True in *self.is_available*. A higher value means a higher priority.
- **data** (*object or* BaseModel) – Any other data relevant for the feature (e.g. the ML model for computing 'ml_score_xxx' features)
- **identifier** (*None or str*) – A unique identifier (e.g. MD5 hash) of the ancillary feature. For PluginFeatures or ML features, this should be computed at least from the input file and the feature name.

Notes

req_config and *req_features* are used to test whether the feature can be computed in *self.is_available*.

static available_features(*rtdc_ds*)

Determine available features for an RT-DC dataset

Parameters
rtdc_ds (*instance of* RTDCBase) – The dataset to check availability for

Returns
features – Dictionary with feature names as keys and instances of *AncillaryFeature* as values.

Return type
dict

static check_data_size(*rtdc_ds*, *data_dict*)

Check the feature data is the correct size. If it isn't, resize it.

Parameters

- **rtdc_ds** (*instance of* RTDCBase) – The dataset from which the features are computed
- **data_dict** (*dict*) – Dictionary with *AncillaryFeature.feature_name* as keys and the computed data features (to be resized) as values.

Returns
data_dict – Dictionary with *feature_name* as keys and the correctly resized data features as values.

Return type
dict

`compute`(*rtdc_ds*)

Compute the feature with self.method. All ancillary features that share the same method will also be populated automatically.

Parameters
rtdc_ds (*instance of* RTDCBase) – The dataset to compute the feature for

Returns
data_dict – Dictionary with *AncillaryFeature.feature_name* as keys and the computed data features (read-only) as values.

Return type
dict

`static get_instances`(*feature_name*)

Return all instances that compute *feature_name*

`hash`(*rtdc_ds*)

Used for identifying an ancillary computation

The required features, the used configuration keys/values, and the return value of the requirement function are hashed.

`is_available`(*rtdc_ds*, *verbose=False*)

Check whether the feature is available

Parameters
rtdc_ds (*instance of* RTDCBase) – The dataset to check availability for

Returns
available – *True*, if feature can be computed with *compute*

Return type
bool

Notes

This method returns *False* for a feature if there is a feature defined with the same name but with higher priority (even if the feature would be available otherwise).

`feature_names = ['time', 'index', 'area_ratio', 'area_um', 'aspect', 'deform', 'emodulus', 'emodulus', 'emodulus', 'emodulus', 'emodulus', 'fl1_max_ctc', 'fl2_max_ctc', 'fl3_max_ctc', 'fl1_max_ctc', 'fl2_max_ctc', 'fl1_max_ctc', 'fl3_max_ctc', 'fl2_max_ctc', 'fl3_max_ctc', 'contour', 'bright_avg', 'bright_sd', 'bright_bc_avg', 'bright_bc_sd', 'bright_perc_10', 'bright_perc_90', 'inert_ratio_cvx', 'inert_ratio_prnc', 'inert_ratio_raw', 'tilt', 'volume', 'ml_class', 'circ_times_area', 'area_exp']`

All feature names registered

features = [<AncillaryFeature 'time' (no ID) with priority 0>, <AncillaryFeature 'index' (no ID) with priority 0>, <AncillaryFeature 'area_ratio' (no ID) with priority 0>, <AncillaryFeature 'area_um' (no ID) with priority 0>, <AncillaryFeature 'aspect' (no ID) with priority 0>, <AncillaryFeature 'deform' (no ID) with priority 0>, <AncillaryFeature 'emodulus' (no ID) with priority 5>, <AncillaryFeature 'emodulus' (no ID) with priority 1>, <AncillaryFeature 'emodulus' (no ID) with priority 4>, <AncillaryFeature 'emodulus' (no ID) with priority 0>, <AncillaryFeature 'emodulus' (no ID) with priority 2>, <AncillaryFeature 'fl1_max_ctc' (no ID) with priority 1>, <AncillaryFeature 'fl2_max_ctc' (no ID) with priority 1>, <AncillaryFeature 'fl3_max_ctc' (no ID) with priority 1>, <AncillaryFeature 'fl1_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'fl2_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'fl1_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'fl3_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'fl2_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'fl3_max_ctc' (no ID) with priority 0>, <AncillaryFeature 'contour' (no ID) with priority 0>, <AncillaryFeature 'bright_avg' (no ID) with priority 0>, <AncillaryFeature 'bright_sd' (no ID) with priority 0>, <AncillaryFeature 'bright_bc_avg' (no ID) with priority 0>, <AncillaryFeature 'bright_bc_sd' (no ID) with priority 0>, <AncillaryFeature 'bright_perc_10' (no ID) with priority 0>, <AncillaryFeature 'bright_perc_90' (no ID) with priority 0>, <AncillaryFeature 'inert_ratio_cvx' (no ID) with priority 0>, <AncillaryFeature 'inert_ratio_prnc' (no ID) with priority 0>, <AncillaryFeature 'inert_ratio_raw' (no ID) with priority 0>, <AncillaryFeature 'tilt' (no ID) with priority 0>, <AncillaryFeature 'volume' (no ID) with priority 0>, <AncillaryFeature 'ml_class' (no ID) with priority 0>, <PlugInFeature 'circ_times_area' (id 70254...) with priority 0>, <PlugInFeature 'area_exp' (id 5f03f...) with priority 0>]

All ancillary features registered

5.3.11 Plugin features

Added in version 0.34.0.

exception dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**PluginImportError**

class dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**PlugInFeature**(*feature_name: str*, *info: dict*, *plugin_path: str | Path | None = None*)

A user-defined plugin feature

Parameters

- **feature_name** (*str*) – name of a feature that matches that defined in *info*
- **info** (*dict*) – Full plugin recipe (for all features) as given in the *info* dictionary in the plugin file. At least the following keys must be specified:
  - ”method”: callable function computing the plugin feature values (takes an :class`dclab.rtdc_dataset.core.RTDCBase` as argument)
  - ”feature names”: list of plugin feature names provided by the plugin

  The following features are optional:
  - ”description”: short (one-line) description of the plugin
  - ”long description”: long description of the plugin
  - ”feature labels”: feature labels used e.g. for plotting

- ”feature shapes”: list of tuples for each feature indicating the shape (this is required only for non-scalar features; for scalar features simply set this to `None` or `(1,)`).
- ”scalar feature”: list of boolean values indicating whether the features are scalar
- ”config required”: configuration keys required to compute the plugin features (see the *req_config* parameter for *AncillaryFeature*)
- ”features required”: list of feature names required to compute the plugin features (see the *req_features* parameter for *AncillaryFeature*)
- ”method check required”: additional method that checks whether the features can be computed (see the *req_func* parameter for *AncillaryFeature*)
- ”version”: version of this plugin (please use semantic verioning)

plugin_path (*str or pathlib.Path, optional*) – path which was used to load the *PlugInFeature* with *load_plugin_feature()*.

Notes

PluginFeature inherits from *AncillaryFeature*. Please read the advanced section on *PluginFeatures* in the dclab docs.

feature_name

Plugin feature name

plugin_feature_info

Dictionary containing all information relevant for this particular plugin feature instance

plugin_path

Path to the original plugin file

dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**import_plugin_feature_script**(*plugin_path: str | Path*) → dict

Import the user-defined recipe and return the info dictionary

Parameters
plugin_path (*str or Path*) – pathname to a valid dclab plugin script

Returns
info – Dictionary with the information required to instantiate one (or multiple) *PlugInFeature*.

Return type
dict

Raises
PluginImportError – If the plugin can not be found

Notes

One recipe may define multiple plugin features.

dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**load_plugin_feature**(*plugin_path: str | Path*) → List[*PlugInFeature*]

Find and load PlugInFeature(s) from a user-defined recipe

Parameters
plugin_path (*str or Path*) – pathname to a valid dclab plugin Python script

Returns
plugin_list – list of PlugInFeature instances loaded from *plugin_path*

Return type
list of *PlugInFeature*

Raises
ValueError – If the script dictionary "feature names" are not a list

Notes

One recipe may define multiple plugin features.

See also:

import_plugin_feature_script
function that imports the plugin script

PlugInFeature
class handling the plugin feature information

dclab.rtdc_dataset.feat_temp.register_temporary_feature
alternative method for creating user-defined features

dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**remove_all_plugin_features**()

Convenience function for removing all *PlugInFeature* instances

See also:

remove_plugin_feature
remove a single *PlugInFeature* instance

dclab.rtdc_dataset.feat_anc_plugin.plugin_feature.**remove_plugin_feature**(*plugin_instance:* PlugInFeature)

Convenience function for removing a *PlugInFeature* instance

Parameters
plugin_instance (PlugInFeature) – The *PlugInFeature* instance to be removed from dclab

Raises
TypeError – If the *plugin_instance* is not a *PlugInFeature* instance

5.3.12 Temporary features

Added in version 0.33.0.

dclab.rtdc_dataset.feat_temp.**deregister_all**()

Deregisters all temporary features

dclab.rtdc_dataset.feat_temp.**deregister_temporary_feature**(*feature: str*)

Convenience function for deregistering a temporary feature

This method is mostly used during testing. It does not remove the actual feature data from any dataset; the data will stay in memory but is not accessible anymore through the public methods of the RTDCBase user interface.

dclab.rtdc_dataset.feat_temp.**register_temporary_feature**(*feature: str*, *label: str | None = None*, *is_scalar: bool = True*)

Register a new temporary feature

Temporary features are custom features that can be defined ad hoc by the user. Temporary features are helpful when the integral features are not enough, e.g. for prototyping, testing, or collating with other data. Temporary features allow you to leverage the full functionality of RTDCBase with your custom features (no need to go for a custom *pandas.Dataframe*).

Parameters

- **feature** (*str*) – Feature name; allowed characters are lower-case letters, digits, and underscores
- **label** (*str*) – Feature label used e.g. for plotting
- **is_scalar** (*bool*) – Whether or not the feature is a scalar feature

dclab.rtdc_dataset.feat_temp.**set_temporary_feature**(*rtdc_ds: RTDCBase*, *feature: str*, *data: ndarray*)

Set temporary feature data for a dataset

Parameters

- **rtdc_ds** (*dclab.RTDCBase*) – Dataset for which to set the feature. Note that the length of the feature *data* must match the number of events in *rtdc_ds*. If the dataset is a hierarchy child, the data will also be set in the parent dataset, but only for those events that are part of the child. For all events in the parent dataset that are not part of the child dataset, the temporary feature is set to np.nan.
- **feature** (*str*) – Feature name
- **data** (*np.ndarray*) – The data

5.3.13 Config

class dclab.rtdc_dataset.config.**Configuration**(*files=None*, *cfg=None*, *disable_checks=False*)

Configuration class for RT-DC datasets

This class has a dictionary-like interface to access and set configuration values, e.g.

```
cfg = load_from_file("/path/to/config.txt")
# access the channel width
cfg["setup"]["channel width"]
# modify the channel width
cfg["setup"]["channel width"] = 30
```

Parameters

- **files** (*list of files*) – The config files with which to initialize the configuration
- **cfg** (*dict-like*) – The dictionary with which to initialize the configuration
- **disable_checks** (*bool*) – Set this to True if you want to avoid checking against section and key names defined in *dclab.definitions* using `verify_section_key()`. This avoids excess warning messages when loading data from configuration files not generated by dclab.

copy()

Return copy of current configuration

get(*key*, *other*)

Famous *dict.get* function

Added in version 0.29.1.

keys()

Return the configuration keys (sections)

save(*filename*)

Save the configuration to a file

tojson()

Convert the configuration to a JSON string

Note that the data type of some configuration options will likely be lost.

tostring(*sections=None*)

Convert the configuration to its string representation

The optional argument *sections* allows to export only specific sections of the configuration, i.e. *sections=dclab.dfn.CFG_METADATA* will only export configuration data from the original measurement and no filtering data.

update(*newcfg*)

Update current config with a dictionary

dclab.rtdc_dataset.config.**load_from_file**(*cfg_file*)

Load the configuration from a file

Parameters
cfg_file (*str*) – Path to configuration file

Returns
cfg – Dictionary with configuration parameters

Return type
ConfigurationDict

5.3.14 Export

exception dclab.rtdc_dataset.export.**LimitingExportSizeWarning**

class dclab.rtdc_dataset.export.**Export**(*rtdc_ds*)

Export functionalities for RT-DC datasets

avi(*path*, *filtered=True*, *override=False*)

Exports filtered event images to an avi file

Parameters

- **path** (*str*) – Path to a .avi file. The ending .avi is added automatically.
- **filtered** (*bool*) – If set to *True*, only the filtered data (index in ds.filter.all) are used.
- **override** (*bool*) – If set to *True*, an existing file `path` will be overridden. If set to *False*, raises *OSError* if `path` exists.

Notes

Raises OSError if current dataset does not contain image data

fcs(*path*, *features*, *meta_data=None*, *filtered=True*, *override=False*)

Export the data of an RT-DC dataset to an .fcs file

Parameters

- **path** (*str*) – Path to an .fcs file. The ending .fcs is added automatically.
- **features** (*list of str*) – The features in the resulting .fcs file. These are strings that are defined by *dclab.definitions.scalar_feature_exists*, e.g. "area_cvx", "deform", "frame", "fl1_max", "aspect".
- **meta_data** (*dict*) – User-defined, optional key-value pairs that are stored in the primary TEXT segment of the FCS file; the version of dclab is stored there by default
- **filtered** (*bool*) – If set to *True*, only the filtered data (index in ds.filter.all) are used.
- **override** (*bool*) – If set to *True*, an existing file `path` will be overridden. If set to *False*, raises *OSError* if `path` exists.

Notes

Due to incompatibility with the .fcs file format, all events with NaN-valued features are not exported.

hdf5(*path: str | Path*, *features: List[str] = None*, *filtered: bool = True*, *logs: bool = False*, *tables: bool = False*, *basins: bool = False*, *meta_prefix: str = 'src_'*, *override: bool = False*, *compression_kwargs: Dict = None*, *compression: str = 'deprecated'*, *skip_checks: bool = False*)

Export the data of the current instance to an HDF5 file

Parameters

- **path** (*str*) – Path to an .rtdc file. The ending .rtdc is added automatically.
- **features** (*list of str*) – The features in the resulting .rtdc file. These are strings that are defined by *dclab.definitions.feature_exists*, e.g. "area_cvx", "deform", "frame", "fl1_max", "image". Defaults to *self.rtdc_ds.features_innate*.
- **filtered** (*bool*) – If set to *True*, only the filtered data (index in ds.filter.all) are used.

- **logs** (*bool*) – Whether to store the logs of the original file prefixed with *source_* to the output file.
- **tables** (*bool*) – Whether to store the tables of the original file prefixed with *source_* to the output file.
- **basins** (*bool*) – Whether to export basins. If filtering is disabled, basins are copied directly to the output file. If filtering is enabled, then mapped basins are exported.
- **meta_prefix** (*str*) – Prefix for log and table names in the exported file
- **override** (*bool*) – If set to *True*, an existing file `path` will be overridden. If set to *False*, raises *OSError* if `path` exists.
- **compression_kwargs** (*dict*) – Dictionary with the keys "compression" and "compression_opts" which are passed to `h5py.H5File.create_dataset()`. The default is Zstandard compression with the lowest compression level *hdf5plugin.Zstd(clevel=1)*.
- **compression** (*str or None*) – Compression method used for data storage; one of [None, "lzf", "gzip", "szip"].

  Deprecated since version 0.43.0: Use *compression_kwargs* instead.
- **skip_checks** (*bool*) – Disable checking whether all features have the same length.
- **versionchanged:** (`..`) – 0.58.0: The `basins` keyword argument was added, and it is now possible to pass an empty list to `features`. This combination results in a very small file consisting of metadata and a mapped basin referring to the original dataset.

tsv(*path*, *features*, *meta_data=None*, *filtered=True*, *override=False*)

Export the data of the current instance to a .tsv file

Parameters

- **path** (*str*) – Path to a .tsv file. The ending .tsv is added automatically.
- **features** (*list of str*) – The features in the resulting .tsv file. These are strings that are defined by *dclab.definitions.scalar_feature_exists*, e.g. "area_cvx", "deform", "frame", "fl1_max", "aspect".
- **meta_data** (*dict*) – User-defined, optional key-value pairs that are stored at the beginning of the tsv file - one key-value pair is stored per line which starts with a hash. The version of dclab is stored there by default.
- **filtered** (*bool*) – If set to *True*, only the filtered data (index in ds.filter.all) are used.
- **override** (*bool*) – If set to *True*, an existing file `path` will be overridden. If set to *False*, raises *OSError* if `path` exists.

5.3.15 Filter

class dclab.rtdc_dataset.filter.**Filter**(*rtdc_ds*)

Boolean filter arrays for RT-DC measurements

Parameters

rtdc_ds (*instance of* RTDCBase) – The RT-DC dataset the filter applies to

reset()

Reset all filters

update(*rtdc_ds*, *force=None*)

Update the filters according to *rtdc_ds.config["filtering"]*

Parameters

- **rtdc_ds** (dclab.rtdc_dataset.core.RTDCBase) – The measurement to which the filter is applied
- **force** (*list*) – A list of feature names that must be refiltered with min/max values.

Notes

This function is called when *ds.apply_filter* is called.

property all

All filters combined (see *Filter.update()*)

Use this property to filter the features of *dclab.rtdc_dataset.RTDCBase* instances

property box

All box filters

property invalid

Invalid (nan/inf) events

property polygon

Polygon filters

5.4 Low-level functionalities

5.4.1 downsampling

Content-based downsampling of ndarrays

dclab.downsampling.**downsample_grid**(*a*, *b*, *samples*, *remove_invalid=False*, *ret_idx=False*)

Content-based downsampling for faster visualization

The arrays *a* and *b* make up a 2D scatter plot with high and low density values. This method takes out points at indices with high density.

Parameters

- **a** (*1d ndarrays*) – The input arrays to downsample
- **b** (*1d ndarrays*) – The input arrays to downsample
- **samples** (*int*) – The desired number of samples
- **remove_invalid** (*bool*) – Remove nan and inf values before downsampling; if set to *True*, the actual number of samples returned might be smaller than *samples* due to infinite or nan values.
- **ret_idx** (*bool*) – Also return a boolean array that corresponds to the downsampled indices in *a* and *b*.

Returns

- **dsa, dsb** (*1d ndarrays of shape (samples,)*) – The arrays *a* and *b* downsampled by evenly selecting points and pseudo-randomly adding or removing points to match *samples*.

- **idx** (1d boolean array with same shape as *a*) – Only returned if *ret_idx* is True. A boolean array such that *a[idx] == dsa*

dclab.downsampling.**downsample_rand**(*a*, *samples*, *remove_invalid=False*, *ret_idx=False*)

Downsampling by randomly removing points

Parameters

- **a** (*1d ndarray*) – The input array to downsample
- **samples** (*int*) – The desired number of samples
- **remove_invalid** (*bool*) – Remove nan and inf values before downsampling
- **ret_idx** (*bool*) – Also return a boolean array that corresponds to the downsampled indices in *a*.

Returns

- **dsa** (1d ndarray of size *samples*) – The pseudo-randomly downsampled array *a*
- **idx** (1d boolean array with same shape as *a*) – Only returned if *ret_idx* is True. A boolean array such that *a[idx] == dsa*

dclab.downsampling.**norm**(*a*)

Normalize *a* with its min/max values

dclab.downsampling.**populate_grid**(*x_discrete*, *y_discrete*, *keepd*, *toproc*)

dclab.downsampling.**valid**(*a*, *b*)

Check whether *a* and *b* are not inf or nan

5.4.2 features

image-based

dclab.features.contour.**get_contour**(*mask*)

Compute the image contour from a mask

The contour is computed in a very inefficient way using scikit-image and a conversion of float coordinates to pixel coordinates.

Parameters

mask (*binary ndarray of shape (M,N) or (K,M,N)*) – The mask outlining the pixel positions of the event. If a 3d array is given, then *K* indexes the individual contours.

Returns

cont – A 2D array that holds the contour of an event (in pixels) e.g. obtained using *mm.contour* where *mm* is an instance of *RTDCBase*. The first and second columns of *cont* correspond to the x- and y-coordinates of the contour.

Return type

ndarray or list of K ndarrays of shape (J,2)

dclab.features.bright.**get_bright**(*mask*, *image*, *ret_data='avg,sd'*)

Compute avg and/or std of the event brightness

The event brightness is defined by the gray-scale values of the image data within the event mask area.

Parameters

- **mask** (*ndarray or list of ndarrays of shape (M,N) and dtype bool*) – The mask values, True where the event is located in *image*.
- **image** (*ndarray or list of ndarrays of shape (M,N)*) – A 2D array that holds the image in form of grayscale values of an event.
- **ret_data** (*str*) – A comma-separated list of metrices to compute - "avg": compute the average - "sd": compute the standard deviation Selected metrics are returned in alphabetical order.

Returns

- **bright_avg** (*float or ndarray of size N*) – Average image data within the contour
- **bright_std** (*float or ndarray of size N*) – Standard deviation of image data within the contour

dclab.features.inert_ratio.**get_inert_ratio_cvx**(*cont*)

Compute the inertia ratio of the convex hull of a contour

The inertia ratio is computed from the central second order of moments along x (mu20) and y (mu02) via *sqrt(mu20/mu02)*.

Parameters

cont (*ndarray or list of ndarrays of shape (N,2)*) – A 2D array that holds the contour of an event (in pixels) e.g. obtained using *mm.contour* where *mm* is an instance of *RTDCBase*. The first and second columns of *cont* correspond to the x- and y-coordinates of the contour.

Returns

- **inert_ratio_cvx** (*float or ndarray of size N*) – The inertia ratio of the contour's convex hull
- .. *versionchanged:: 0.48.2* – For long channels, an integer overflow could occur in previous versions, leading invalid or nan values. See https://github.com/DC-analysis/dclab/issues/212

Notes

The contour moments mu20 and mu02 are computed the same way they are computed in OpenCV's *moments.cpp*.

See also:

get_inert_ratio_raw
Compute inertia ratio of a raw contour

References

- https://en.wikipedia.org/wiki/Image_moment#Central_moments
- https://github.com/opencv/opencv/blob/f81370232a651bdac5042efe907bcaa50a66c487/modules/imgproc/src/moments.cpp#L93

dclab.features.inert_ratio.**get_inert_ratio_raw**(*cont*)

Compute the inertia ratio of a contour

The inertia ratio is computed from the central second order of moments along x (mu20) and y (mu02) via *sqrt(mu20/mu02)*.

Parameters

cont (*ndarray or list of ndarrays of shape (N,2)*) – A 2D array that holds the contour of an event (in pixels) e.g. obtained using *mm.contour* where *mm* is an instance of *RTDCBase*. The first and second columns of *cont* correspond to the x- and y-coordinates of the contour.

Returns

- **inert_ratio_raw** (*float or ndarray of size N*) – The inertia ratio of the contour
- *.. versionchanged:: 0.48.2* – For long channels, an integer overflow could occur in previous versions, leading invalid or nan values. See https://github.com/DC-analysis/dclab/issues/212

Notes

The contour moments mu20 and mu02 are computed the same way they are computed in OpenCV's *moments.cpp*.

See also:

get_inert_ratio_cvx
Compute inertia ratio of the convex hull of a contour

References

- https://en.wikipedia.org/wiki/Image_moment#Central_moments
- https://github.com/opencv/opencv/blob/f81370232a651bdac5042efe907bcaa50a66c487/modules/imgproc/src/moments.cpp#L93

dclab.features.volume.**get_volume**(*cont*, *pos_x*, *pos_y*, *pix*, *fix_orientation=False*)

Calculate the volume of a polygon revolved around an axis

The volume estimation assumes rotational symmetry.

Parameters

- **cont** (*ndarray or list of ndarrays of shape (N,2)*) – A 2D array that holds the contour of an event [px] e.g. obtained using *mm.contour* where *mm* is an instance of *RTDCBase*. The first and second columns of *cont* correspond to the x- and y-coordinates of the contour.
- **pos_x** (*float or ndarray of length N*) – The x coordinate(s) of the centroid of the event(s) [µm] e.g. obtained using *mm.pos_x*
- **pos_y** (*float or ndarray of length N*) – The y coordinate(s) of the centroid of the event(s) [µm] e.g. obtained using *mm.pos_y*
- **pix** (*float*) – The detector pixel size in µm. e.g. obtained using: *mm.config["imaging"]["pixel size"]*
- **fix_orientation** (*bool*) – If set to True, make sure that the orientation of the contour is counter-clockwise in the r-z plane (see *vol_revolve()*). This is False by default, because (1) Shape-In always stores the contours in the correct orientation and (2) there may be events with high porosity where "fixing" the orientation makes things worse and a negative volume is returned.

Returns

volume – volume in um^3

Return type
float or ndarray

Notes

The computation of the volume is based on a full rotation of the upper and the lower halves of the contour from which the average is then used.

The volume is computed radially from the the center position given by (*pos_x*, *pos_y*). For sufficiently smooth contours, such as densely sampled ellipses, the center position does not play an important role. For contours that are given on a coarse grid, as is the case for RT-DC, the center position must be given.

References

- https://de.wikipedia.org/wiki/Kegelstumpf#Formeln
- Yields identical results to the Matlab script by Geoff Olynyk <https://de.mathworks.com/matlabcentral/fileexchange/36525-volrevolve>`_

dclab.features.volume.**counter_clockwise**(*cx*, *cy*)

Put contour coordinates into counter-clockwise order

Parameters

- **cx** (*1d ndarrays*) – The x- and y-coordinates of the contour
- **cy** (*1d ndarrays*) – The x- and y-coordinates of the contour

Returns
The x- and y-coordinates of the contour in counter-clockwise orientation.

Return type
cx_cc, cy_cc

Notes

The contour must be centered around (0, 0).

dclab.features.volume.**vol_revolve**(*r*, *z*, *point_scale=1.0*)

Calculate the volume of a polygon revolved around the Z-axis

This implementation yields the same results as the volRevolve Matlab function by Geoff Olynyk (from 2012-05-03) https://de.mathworks.com/matlabcentral/fileexchange/36525-volrevolve.

The difference here is that the volume is computed using (a much more approachable) implementation using the volume of a truncated cone (https://de.wikipedia.org/wiki/Kegelstumpf).

$$V = \frac{h \cdot \pi}{3} \cdot (R^2 + R \cdot r + r^2)$$

Where h is the height of the cone and r and R are the smaller and larger radii of the truncated cone.

Each line segment of the contour resembles one truncated cone. If the z-step is positive (counter-clockwise contour), then the truncated cone volume is added to the total volume. If the z-step is negative (e.g. inclusion), then the truncated cone volume is removed from the total volume.

Changed in version 0.37.0: The volume in previous versions was overestimated by on average 2µm^3.

Parameters

- **r** (*1d np.ndarray*) – radial coordinates (perpendicular to the z axis)
- **z** (*1d np.ndarray*) – coordinate along the axis of rotation
- **point_scale** (*float*) – point size in your preferred units; The volume is multiplied by a factor of *point_scale**3*.

Notes

The coordinates must be given in counter-clockwise order, otherwise the volume will be negative.

emodulus

Computation of apparent Young's modulus for RT-DC measurements

exception dclab.features.emodulus.**KnowWhatYouAreDoingWarning**

exception dclab.features.emodulus.**YoungsModulusLookupTableExceededWarning**

dclab.features.emodulus.**extrapolate_emodulus**(*lut*, *datax*, *deform*, *emod*, *deform_norm*, *deform_thresh=0.05*, *inplace=True*)

Use spline interpolation to fill in nan-values

When points (*datax*, *deform*) are outside the convex hull of the lut, then `scipy.interpolate.griddata()` returns nan-valules.

With this function, some of these nan-values are extrapolated using `scipy.interpolate.SmoothBivariateSpline`. The supported extrapolation values are currently limited to those where the deformation is above 0.05.

A warning will be issued, because this is not really recommended.

Parameters

- **lut** (*ndarray of shape (N, 3)*) – The normalized (!! see *normalize()*) LUT (first axis is points, second axis enumerates datax, deform, and emodulus)
- **datax** (*ndarray of size N*) – The normalized x data (corresponding to *lut[:, 0]*)
- **deform** (*ndarray of size N*) – The normalized deform (corresponding to *lut[:, 1]*)
- **emod** (*ndarray of size N*) – The emodulus (corresponding to *lut[:, 2]*); If *emod* does not contain nan-values, there is nothing to do here.
- **deform_norm** (*float*) – The normalization value used to normalize *lut[:, 1]* and *deform*.
- **deform_thresh** (*float*) – Not the entire LUT is used for bivariate spline interpolation. Only the points where *lut[:, 1]* > *deform_thresh/deform_norm* are used. This is necessary, because for small deformations, the LUT has an extreme slope that kills any meaningful spline interpolation.
- **inplace** (*bool*) – If True (default), replaces nan values in *emod* in-place. If False, *emod* is not modified.

dclab.features.emodulus.**get_emodulus**(*deform: float | np.array*, *area_um: float | np.array | None = None*, *volume: float | np.array | None = None*, *medium: float | str = '0.49% MC-PBS'*, *channel_width: float = 20.0*, *flow_rate: float = 0.16*, *px_um: float = 0.34*, *temperature: float | np.ndarray | None = 23.0*, *lut_data: str | pathlib.Path | np.ndarray = 'LE-2D-FEM-19'*, *visc_model: Literal['herold-2017', 'herold-2017-fallback', 'buyukurganci-2022', 'kestin-1978', None] = 'herold-2017-fallback'*, *extrapolate: bool = False*, *copy: bool = True*)

Compute apparent Young's modulus using a look-up table

Parameters

- **area_um** (*float or ndarray*) – Apparent (2D image) area [µm^2] of the event(s)
- **deform** (*float or ndarray*) – Deformation (1-circularity) of the event(s)
- **volume** (*float or ndarray*) – Apparent volume of the event(s). It is not possible to define *volume* and *area_um* at the same time (makes no sense).

  Added in version 0.25.0.
- **medium** (*str or float*) – The medium to compute the viscosity for. If a string is given, the viscosity is computed. If a float is given, this value is used as the viscosity in mPa*s (Note that *temperature* and *visc_model* must be set to None in this case).
- **channel_width** (*float*) – The channel width [µm]
- **flow_rate** (*float*) – Flow rate [µL/s]
- **px_um** (*float*) – The detector pixel size [µm] used for pixelation correction. Set to zero to disable.
- **temperature** (*float, ndarray, or None*) – Temperature [°C] of the event(s)
- **lut_data** (*path, str, or tuple of (np.ndarray of shape (N, 3), dict)*) – The LUT data to use. If it is a built-in identifier, then the respective LUT will be used. Otherwise, a path to a file on disk or a tuple (LUT array, metadata) is possible. The LUT metadata is used to check whether the given features (e.g. *area_um* and *deform*) are valid interpolation choices.

  Added in version 0.25.0.
- **visc_model** (*str*) – The viscosity model to use, see *dclab.features.emodulus.viscosity.get_viscosity()*
- **extrapolate** (*bool*) – Perform extrapolation using *extrapolate_emodulus()*. This is discouraged!
- **copy** (*bool*) – Copy input arrays. If set to false, input arrays are overridden.

Returns

elasticity – Apparent Young's modulus in kPa

Return type

float or ndarray

Notes

- The look-up table used was computed with finite elements methods according to [MMM+17] and complemented with analytical isoelastics from [MOG+15]. The original simulation results are available on figshare [WMM+20].
- The computation of the Young's modulus takes into account a correction for the viscosity (medium, channel width, flow rate, and temperature) [MOG+15] and a correction for pixelation for the deformation which were derived from a (pixelated) image [Her17].
- Note that while deformation is pixelation-corrected, area_um and volume are scaled to match the LUT data. This is somewhat fortunate, because we don't have to worry about the order of applying pixelation correction and scale conversion.
- By using external LUTs, it is possible to interpolate on the volume-deformation plane. This feature was added in version 0.25.0.

See also:

dclab.features.emodulus.viscosity.get_viscosity
compute viscosity for known media

dclab.features.emodulus.**normalize**(*data*, *dmax*)

Perform normalization in-place for interpolation

Note that `scipy.interpolate.griddata()` has a *rescale* option which rescales the data onto the unit cube. For some reason this does not work well with LUT data, so we just normalize it by dividing by the maximum value.

dclab.features.emodulus.**INACCURATE_SPLINE_EXTRAPOLATION = False**

Set this to True to globally enable spline extrapolation when the *area_um/deform* data are outside the LUT. This is discouraged and a *KnowWhatYouAreDoingWarning* warning will be issued.

dclab.features.emodulus.load.**get_internal_lut_names_dict**()

Return list of internal lut names

dclab.features.emodulus.load.**get_lut_path**(*path_or_id*)

Find the path to a LUT

path_or_id: str or pathlib.Path
Identifier of a LUT. This can be either an existing path (checked first), or an internal identifier (see *get_internal_lut_names_dict()*).

dclab.features.emodulus.load.**load_lut**(*lut_data: str | Path | ndarray = 'LE-2D-FEM-19'*)

Load LUT data from disk

Parameters
lut_data (*path, str, or tuple of (np.ndarray of shape (N, 3), dict)*) – The LUT data to use. If it is in *get_internal_lut_names_dict()*, then the respective LUT will be used. Otherwise, a path to a file on disk or a tuple (LUT array, metadata) is possible.

Returns

- **lut** (*np.ndarray of shape (N, 3)*) – The LUT data for interpolation
- **meta** (*dict*) – The LUT metadata

Notes

If lut_data is a tuple of (lut, meta), then nothing is actually done (this is implemented for user convenience).

dclab.features.emodulus.load.**load_mtext**(*path*)

Load column-based data from text files with metadata

This file format is used for isoelasticity lines and look-up table data in dclab.

The text file is loaded with *numpy.loadtxt*. The metadata are stored as a json string between the "BEGIN METADATA" and the "END METADATA" tags. The last comment (#) line before the actual data defines the features with units in square brackets and tab-separated. For instance:

> # [...] # # BEGIN METADATA # { # "authors": "A. Mietke, C. Herold, J. Guck", # "channel_width": 20.0, # "channel_width_unit": "um", # "date": "2018-01-30", # "dimensionality": "2Daxis", # "flow_rate": 0.04, # "flow_rate_unit": "uL/s", # "fluid_viscosity": 15.0, # "fluid_viscosity_unit": "mPa s", # "identifier": "LE-2D-ana-18", # "method": "analytical", # "model": "linear elastic", # "publication": "https://doi.org/10.1016/j.bpj.2015.09.006", # "software": "custom Matlab code", # "summary": "2D-axis-symmetric analytical solution" # } # END METADATA # # [...] # # area_um [um^2] deform emodulus [kPa] 3.75331e+00 5.14496e-03 9.30000e-01 4.90368e+00 6.72683e-03 9.30000e-01 6.05279e+00 8.30946e-03 9.30000e-01 7.20064e+00 9.89298e-03 9.30000e-01 [...]

dclab.features.emodulus.load.**register_lut**(*path*, *identifier=None*)

Register an external LUT file in dclab

This will add it to *EXTERNAL_LUTS*, which is required for emodulus computation as an ancillary feature.

Parameters

- **path** (*str or pathlib.Path*) – Path to the external LUT file
- **identifier** (*str or None*) – The identifier is used for ancillary emodulus computation via the [calculation]: "emodulus lut" key. It is also used as the key in *EXTERNAL_LUTS* during registration. If not specified, (default) then the identifier given as JSON metadata in *path* is used.

dclab.features.emodulus.load.**EXTERNAL_LUTS = {}**

Dictionary of look-up tables that the user added via *register_lut()*.

Pixelation correction definitions

dclab.features.emodulus.pxcorr.**corr_deform_with_area_um**(*area_um*, *px_um=0.34*)

Deformation correction for area_um-deform data

The contour in RT-DC measurements is computed on a pixelated grid. Due to sampling problems, the measured deformation is overestimated and must be corrected.

The correction formula is described in [Her17].

Parameters

- **area_um** (*float or ndarray*) – Apparent (2D image) area in µm^2 of the event(s)
- **px_um** (*float*) – The detector pixel size in µm.

Returns

deform_delta – Error of the deformation of the event(s) that must be subtracted from *deform*. deform_corr = deform - deform_delta

Return type

float or ndarray

dclab.features.emodulus.pxcorr.**corr_deform_with_volume**(*volume*, *px_um=0.34*)

Deformation correction for volume-deform data

The contour in RT-DC measurements is computed on a pixelated grid. Due to sampling problems, the measured deformation is overestimated and must be corrected.

The correction is derived in scripts/pixelation_correction.py.

Parameters

- **volume** (*float or ndarray*) – The "volume" feature (rotation of raw contour) [µm^3]
- **px_um** (*float*) – The detector pixel size in µm.

Returns

deform_delta – Error of the deformation of the event(s) that must be subtracted from *deform*. deform_corr = deform - deform_delta

Return type

float or ndarray

dclab.features.emodulus.pxcorr.**get_pixelation_delta**(*feat_corr*, *feat_absc*, *data_absc*, *px_um=0.34*)

Convenience function for obtaining pixelation correction

Parameters

- **feat_corr** (*str*) – Feature for which to compute the pixelation correction (e.g. "deform")
- **feat_absc** (*str*) – Feature with which to compute the correction (e.g. "area_um");
- **data_absc** (*ndarray or float*) – Corresponding data for *feat_absc*
- **px_um** (*float*) – Detector pixel size [µm]

dclab.features.emodulus.pxcorr.**get_pixelation_delta_pair**(*feat1*, *feat2*, *data1*, *data2*, *px_um=0.34*)

Convenience function that returns pixelation correction pair

Scale conversion applicable to a linear elastic model

dclab.features.emodulus.scale_linear.**convert**(*area_um*, *deform*, *channel_width_in*, *channel_width_out*, *emodulus=None*, *flow_rate_in=None*, *flow_rate_out=None*, *viscosity_in=None*, *viscosity_out=None*, *inplace=False*)

convert area-deformation-emodulus triplet

The conversion formula is described in [MOG+15].

Parameters

- **area_um** (*ndarray*) – Convex cell area [µm^2]
- **deform** (*ndarray*) – Deformation
- **channel_width_in** (*float*) – Original channel width [µm]
- **channel_width_out** (*float*) – Target channel width [µm]
- **emodulus** (*ndarray*) – Young's Modulus [kPa]
- **flow_rate_in** (*float*) – Original flow rate [µL/s]
- **flow_rate_out** (*float*) – Target flow rate [µL/s]
- **viscosity_in** (*float*) – Original viscosity [mPa*s]

- **viscosity_out** (*float or ndarray*) – Target viscosity [mPa*s]; This can be an array
- **inplace** (*bool*) – If True, override input arrays with corrected data

Returns

- **area_um_corr** (*ndarray*) – Corrected cell area [µm^2]
- **deform_corr** (*ndarray*) – Deformation (a copy if *inplace* is False)
- **emodulus_corr** (*ndarray*) – Corrected emodulus [kPa]; only returned if *emodulus* is given.

Notes

If only *area_um*, *deform*, *channel_width_in* and *channel_width_out* are given, then only the area is corrected and returned together with the original deform. If all other arguments are not set to None, the emodulus is corrected and returned as well.

dclab.features.emodulus.scale_linear.**scale_area_um**(*area_um*, *channel_width_in*, *channel_width_out*, *inplace=False*, ***kwargs*)

Perform scale conversion for area_um (linear elastic model)

The area scales with the characteristic length "channel radius" L according to (L'/L)^2.

The conversion formula is described in [MOG+15].

Parameters

- **area_um** (*ndarray*) – Convex area [µm^2]
- **channel_width_in** (*float*) – Original channel width [µm]
- **channel_width_out** (*float*) – Target channel width [µm]
- **inplace** (*bool*) – If True, override input arrays with corrected data
- **kwargs** – not used

Returns
area_um_corr – Scaled area [µm^2]

Return type
ndarray

dclab.features.emodulus.scale_linear.**scale_emodulus**(*emodulus*, *channel_width_in*, *channel_width_out*, *flow_rate_in*, *flow_rate_out*, *viscosity_in*, *viscosity_out*, *inplace=False*)

Perform scale conversion for area_um (linear elastic model)

The conversion formula is described in [MOG+15].

Parameters

- **emodulus** (*ndarray*) – Young's Modulus [kPa]
- **channel_width_in** (*float*) – Original channel width [µm]
- **channel_width_out** (*float*) – Target channel width [µm]
- **flow_rate_in** (*float*) – Original flow rate [µL/s]
- **flow_rate_out** (*float*) – Target flow rate [µL/s]
- **viscosity_in** (*float*) – Original viscosity [mPa*s]

- **viscosity_out** (*float or ndarray*) – Target viscosity [mPa*s]; This can be an array
- **inplace** (*bool*) – If True, override input arrays with corrected data

Returns
emodulus_corr – Scaled emodulus [kPa]

Return type
ndarray

dclab.features.emodulus.scale_linear.**scale_feature**(*feat*, *data*, *inplace=False*, ***scale_kw*)

Convenience function for scale conversions (linear elastic model)

This method wraps around all the other scale_* methods and also supports deform/circ.

Parameters

- **feat** (*str*) – Valid scalar feature name
- **data** (*float or ndarray*) – Feature data
- **inplace** (*bool*) – If True, override input arrays with corrected data
- ****scale_kw** – Scale keyword arguments for the wrapped methods

dclab.features.emodulus.scale_linear.**scale_volume**(*volume*, *channel_width_in*, *channel_width_out*, *inplace=False*, ***kwargs*)

Perform scale conversion for volume (linear elastic model)

The volume scales with the characteristic length "channel radius" L according to (L'/L)^3.

Parameters

- **volume** (*ndarray*) – Volume [µm^3]
- **channel_width_in** (*float*) – Original channel width [µm]
- **channel_width_out** (*float*) – Target channel width [µm]
- **inplace** (*bool*) – If True, override input arrays with corrected data
- **kwargs** – not used

Returns
volume_corr – Scaled volume [µm^3]

Return type
ndarray

Viscosity computation for various media

exception dclab.features.emodulus.viscosity.**TemperatureOutOfRangeWarning**

dclab.features.emodulus.viscosity.**check_temperature**(*model: str*, *temperature: float | np.array*, *tmin: float*, *tmax: float*)

Raise a TemperatureOutOfRangeWarning if applicable

dclab.features.emodulus.viscosity.**get_viscosity**(*medium: str = '0.49% MC-PBS'*, *channel_width: float = 20.0*, *flow_rate: float = 0.16*, *temperature: float | ndarray = 23.0*, *model: Literal['herold-2017', 'herold-2017-fallback', 'buyukurganci-2022', 'kestin-1978'] = 'herold-2017-fallback'*)

Returns the viscosity for RT-DC-specific media

Media that are not pure (e.g. ketchup or polymer solutions) often exhibit a non-linear relationship between shear rate (determined by the velocity profile) and shear stress (determined by pressure differences). If the shear stress grows non-linearly with the shear rate resulting in a slope in log-log space that is less than one, then we are talking about shear thinning. The viscosity is not a constant anymore (as it is e.g. for water). At higher flow rates, the viscosity becomes smaller, following a power law. Christoph Herold characterized shear thinning for the CellCarrier media [Her17]. The resulting formulae for computing the viscosities of these media at different channel widths, flow rates, and temperatures, are implemented here.

Parameters

- **medium** (*str*) – The medium to compute the viscosity for; Valid values are defined in *KNOWN_MEDIA*.
- **channel_width** (*float*) – The channel width in µm
- **flow_rate** (*float*) – Flow rate in µL/s
- **temperature** (*float or ndarray*) – Temperature in °C
- **model** (*str*) – The model name to use for computing the medium viscosity. For water, this value is ignored, as there is only the 'kestin-1978' model [KSW78]. For MC-PBS media, there are the 'herold-2017' model [Her17] and the 'buyukurganci-2022' model [BBN+23].

Returns

viscosity – Viscosity in mPa*s

Return type

float or ndarray

Notes

- CellCarrier (0.49% MC-PBS) and CellCarrier B (0.59% MC-PBS) are media designed for RT-DC experiments.
- A *TemperatureOutOfRangeWarning* is issued if the input temperature range exceeds the temperature ranges of the models.

dclab.features.emodulus.viscosity.**get_viscosity_mc_pbs_buyukurganci_2022**(*medium: Literal['0.49% MC-PBS', '0.59% MC-PBS', '0.83% MC-PBS'] = '0.49% MC-PBS'*, *channel_width: float = 20.0*, *flow_rate: float = 0.16*, *temperature: float = 23.0*)

Compute viscosity of MC-PBS according to [BBN+23]

This viscosity model was derived in [BBN+23] and adapted for RT-DC in [RB23].

dclab.features.emodulus.viscosity.**get_viscosity_mc_pbs_herold_2017**(*medium: Literal['0.49% MC-PBS', '0.59% MC-PBS'] = '0.49% MC-PBS'*, *channel_width: float = 20.0*, *flow_rate: float = 0.16*, *temperature: float = 23.0*)

Compute viscosity of MC-PBS according to [Her17]

Note that all the factors in equation 5.2 in [Her17] compute to 8, which is essentially what is implemented in *shear_rate_square_channel()*:

$$1.1856 \cdot 6 \cdot \frac{2}{3} \cdot \frac{1}{0.5928} = 8$$

dclab.features.emodulus.viscosity.**get_viscosity_water_kestin_1978**(*temperature: float = 23.0*)

Compute the viscosity of water according to [KSW78]

dclab.features.emodulus.viscosity.**shear_rate_square_channel**(*flow_rate*, *channel_width*, *flow_index*)

Returns The wall shear rate of a power law liquid in a squared channel.

Parameters

- **flow_rate** (*float*) – Flow rate in µL/s
- **channel_width** (*float*) – The channel width in µm
- **flow_index** (*float*) – Flow behavior index aka the power law exponent of the shear thinning

Returns

shear_rate – Shear rate in 1/s.

Return type

float

dclab.features.emodulus.viscosity.**ALIAS_MEDIA = {'0.49% MC-PBS': '0.49% MC-PBS', '0.49% mc-pbs': '0.49% MC-PBS', '0.5% MC-PBS': '0.49% MC-PBS', '0.5% mc-pbs': '0.49% MC-PBS', '0.50% MC-PBS': '0.49% MC-PBS', '0.50% mc-pbs': '0.49% MC-PBS', '0.59% MC-PBS': '0.59% MC-PBS', '0.59% mc-pbs': '0.59% MC-PBS', '0.6% MC-PBS': '0.59% MC-PBS', '0.6% mc-pbs': '0.59% MC-PBS', '0.60% MC-PBS': '0.59% MC-PBS', '0.60% mc-pbs': '0.59% MC-PBS', '0.8% MC-PBS': '0.83% MC-PBS', '0.8% mc-pbs': '0.83% MC-PBS', '0.80% MC-PBS': '0.83% MC-PBS', '0.80% mc-pbs': '0.83% MC-PBS', '0.83% MC-PBS': '0.83% MC-PBS', '0.83% mc-pbs': '0.83% MC-PBS', 'CellCarrier': '0.49% MC-PBS', 'CellCarrier B': '0.59% MC-PBS', 'CellCarrierB': '0.59% MC-PBS', 'cellcarrier': '0.49% MC-PBS', 'cellcarrier b': '0.59% MC-PBS', 'cellcarrierb': '0.59% MC-PBS', 'water': 'water'}**

Many media names are actually shorthand for one medium

dclab.features.emodulus.viscosity.**KNOWN_MEDIA = ['0.49% MC-PBS', '0.49% mc-pbs', '0.5% MC-PBS', '0.5% mc-pbs', '0.50% MC-PBS', '0.50% mc-pbs', '0.59% MC-PBS', '0.59% mc-pbs', '0.6% MC-PBS', '0.6% mc-pbs', '0.60% MC-PBS', '0.60% mc-pbs', '0.8% MC-PBS', '0.8% mc-pbs', '0.80% MC-PBS', '0.80% mc-pbs', '0.83% MC-PBS', '0.83% mc-pbs', 'CellCarrier', 'CellCarrier B', 'CellCarrierB', 'cellcarrier', 'cellcarrier b', 'cellcarrierb', 'water']**

Media for which computation of viscosity is defined (has duplicate entries)

dclab.features.emodulus.viscosity.**SAME_MEDIA = {'0.49% MC-PBS': ['0.49% MC-PBS', '0.5% MC-PBS', '0.50% MC-PBS', 'CellCarrier'], '0.59% MC-PBS': ['0.59% MC-PBS', '0.6% MC-PBS', '0.60% MC-PBS', 'CellCarrier B', 'CellCarrierB'], '0.83% MC-PBS': ['0.83% MC-PBS', '0.8% MC-PBS', '0.80% MC-PBS'], 'water': ['water']}**

Dictionary with different names for one medium

fluorescence

dclab.features.fl_crosstalk.**correct_crosstalk**(*fl1*, *fl2*, *fl3*, *fl_channel*, *ct21=0*, *ct31=0*, *ct12=0*, *ct32=0*, *ct13=0*, *ct23=0*)

Perform crosstalk correction

Parameters

- **fli** (*int, float, or np.ndarray*) – Measured fluorescence signals
- **fl_channel** (*int (1, 2, or 3)*) – The channel number for which the crosstalk-corrected signal should be computed
- **cij** (*float*) – Spill (crosstalk or bleed-through) from channel i to channel j This spill is computed from the fluorescence signal of e.g. single-stained positive control cells; It is defined by the ratio of the fluorescence signals of the two channels, i.e cij = flj / fli.

See also:

get_compensation_matrix
compute the inverse crosstalk matrix

Notes

If there are only two channels (e.g. fl1 and fl2), then the crosstalk to and from the other channel (ct31, ct32, ct13, ct23) should be set to zero.

dclab.features.fl_crosstalk.**get_compensation_matrix**(*ct21*, *ct31*, *ct12*, *ct32*, *ct13*, *ct23*)

Compute crosstalk inversion matrix

The spillover matrix is

| c11 c12 c13 |
| c21 c22 c23 |
| c31 c32 c33 |

The diagonal elements are set to 1, i.e.

ct11 = c22 = c33 = 1

Parameters
cij (*float*) – Spill from channel i to channel j

Returns
inv – Compensation matrix (inverted spillover matrix)

Return type
np.ndarray

5.4.3 isoelastics

Isoelastics management

exception dclab.isoelastics.**IsoelasticsEmodulusMeaninglessWarning**

class dclab.isoelastics.**AutoRecursiveDict**(*dict=None*, */*, ***kwargs*)

class dclab.isoelastics.**Isoelastics**(*paths=None*)

Isoelasticity line management

Parameters

- **paths** (*list of pathlib.Path or list of str*) – list of paths to files containing isoelasticity lines
- **versionchanged:** (..) – 0.24.0: The isoelasticity lines of the analytical model [MOG+15] and the linear-elastic numerical model [MMM+17] were recomputed with an equidistant spacing. The metadata section of the text file format was restructured.

add(*isoel*, *col1*, *col2*, *channel_width*, *flow_rate*, *viscosity*, *method=None*, *lut_identifier=None*)

Add isoelastics

Parameters

- **isoel** (*list of ndarrays*) – Each list item resembles one isoelastic line stored as an array of shape (N,3). The last column contains the emodulus data.
- **col1** (*str*) – Name of the first feature of all isoelastics (e.g. isoel[0][:,0])
- **col2** (*str*) – Name of the second feature of all isoelastics (e.g. isoel[0][:,1])
- **channel_width** (*float*) – Channel width in µm
- **flow_rate** (*float*) – Flow rate through the channel in µL/s
- **viscosity** (*float*) – Viscosity of the medium in mPa*s
- **method** (*str*) – The method used to compute the isoelastics DEPRECATED since 0.32.0. Please use *lut_identifier* instead.
- **lut_identifier** (*str:*) – Look-up table identifier used to identify which isoelasticity lines to show. The function *get_available_identifiers()* returns a list of available identifiers.

Notes

The following isoelastics are automatically added for user convenience:

- isoelastics with *col1* and *col2* interchanged
- isoelastics for circularity if deformation was given

static add_px_err(*isoel*, *col1*, *col2*, *px_um*, *inplace=False*)

Undo pixelation correction

Since isoelasticity lines are usually computed directly from the simulation data (e.g. the contour data are not discretized on a grid but are extracted from FEM simulations), they are not affected by pixelation effects as described in [Her17].

If the isoelasticity lines are displayed alongside experimental data (which are affected by pixelation effects), then the lines must be "un"-corrected, i.e. the pixelation error must be added to the lines to match the experimental data.

Parameters

- **isoel** (*list of 2d ndarrays of shape (N, 3)*) – Each item in the list corresponds to one isoelasticity line. The first column is defined by *col1*, the second by *col2*, and the third column is the emodulus.
- **col1** (*str*) – Define the fist two columns of each isoelasticity line.
- **col2** (*str*) – Define the fist two columns of each isoelasticity line.
- **px_um** (*float*) – Pixel size [µm]
- **inplace** (*bool*) – If True, do not create a copy of the data in *isoel*

static convert(*isoel*, *col1*, *col2*, *channel_width_in*, *channel_width_out*, *flow_rate_in*, *flow_rate_out*, *viscosity_in*, *viscosity_out*, *inplace=False*)

Perform isoelastics scale conversion

Parameters

- **isoel** (*list of 2d ndarrays of shape (N, 3)*) – Each item in the list corresponds to one isoelasticity line. The first column is defined by *col1*, the second by *col2*, and the third column is the emodulus.
- **col1** (*str*) – Define the fist to columns of each isoelasticity line. One of [“area_um”, “circ”, “deform”]
- **col2** (*str*) – Define the fist to columns of each isoelasticity line. One of [“area_um”, “circ”, “deform”]
- **channel_width_in** (*float*) – Original channel width [µm]
- **channel_width_out** (*float*) – Target channel width [µm]
- **flow_rate_in** (*float*) – Original flow rate [µL/s]
- **flow_rate_out** (*float*) – Target flow rate [µL/s]
- **viscosity_in** (*float*) – Original viscosity [mPa*s]
- **viscosity_out** (*float*) – Target viscosity [mPa*s]
- **inplace** (*bool*) – If True, do not create a copy of the data in *isoel*

Returns

isoel_scale – The scale-converted isoelasticity lines.

Return type

list of 2d ndarrays of shape (N, 3)

Notes

If only the positions of the isoelastics are of interest and not the value of the elastic modulus, then it is sufficient to supply values for the channel width and set the values for flow rate and viscosity to a constant (e.g. 1).

See also:

dclab.features.emodulus.scale_linear.scale_feature
scale conversion method used

get(*col1*, *col2*, *channel_width*, *method=None*, *lut_identifier=None*, *flow_rate=None*, *viscosity=None*, *add_px_err=False*, *px_um=None*)

Get isoelastics

Parameters

- **col1** (*str*) – Name of the first feature of all isoelastics (e.g. isoel[0][:,0])
- **col2** (*str*) – Name of the second feature of all isoelastics (e.g. isoel[0][:,1])
- **channel_width** (*float*) – Channel width in µm
- **method** (*str*) – The method used to compute the isoelastics DEPRECATED since 0.32.0. Please use *lut_identifier* instead.
- **lut_identifier** (*str:*) – Look-up table identifier used to identify which isoelasticity lines to show. The function *get_available_identifiers()* returns a list of available identifiers.
- **flow_rate** (float or *None*) – Flow rate through the channel in µL/s. If set to *None*, the flow rate of the imported data will be used (only do this if you do not need the correct values for elastic moduli).
- **viscosity** (float or *None*) – Viscosity of the medium in mPa*s. If set to *None*, the flow rate of the imported data will be used (only do this if you do not need the correct values for elastic moduli).
- **add_px_err** (*bool*) – If True, add pixelation errors according to C. Herold (2017), https://arxiv.org/abs/1704.00572 and scripts/pixelation_correction.py
- **px_um** (*float*) – Pixel size [µm], used for pixelation error computation

See also:

dclab.features.emodulus.scale_linear.scale_feature
scale conversion method used

dclab.features.emodulus.pxcorr.get_pixelation_delta
pixelation correction (applied to the feature data)

get_with_rtdcbase(*col1*, *col2*, *dataset*, *method=None*, *lut_identifier=None*, *viscosity=None*, *add_px_err=False*)

Convenience method that extracts the metadata from RTDCBase

Parameters

- **col1** (*str*) – Name of the first feature of all isoelastics (e.g. isoel[0][:,0])
- **col2** (*str*) – Name of the second feature of all isoelastics (e.g. isoel[0][:,1])
- **method** (*str*) – The method used to compute the isoelastics DEPRECATED since 0.32.0. Please use *lut_identifier* instead.
- **lut_identifier** (*str:*) – Look-up table identifier used to identify which isoelasticity lines to show. The function *get_available_identifiers()* returns a list of available identifiers.
- **dataset** (dclab.rtdc_dataset.RTDCBase) – The dataset from which to obtain the metadata.
- **viscosity** (float, *None*, or False) – Viscosity of the medium in mPa*s. If set to *None*, the viscosity is computed from the meta data (medium, flow rate, channel width,

temperature) in the [setup] config section. If this is not possible, the flow rate of the imported data is used and a warning will be issued.

- **add_px_err** (*bool*) – If True, add pixelation errors according to C. Herold (2017), https://arxiv.org/abs/1704.00572 and scripts/pixelation_correction.py

load_data(*path*)

Load isoelastics from a text file

Parameters

path (*str or pathlib.Path*) – Path to an isoelasticity lines text file

dclab.isoelastics.**check_lut_identifier**(*lut_identifier*, *method*)

Transitional function that can be removed once *method* is removed

dclab.isoelastics.**get_available_files**()

Return list of available isoelasticity line files in dclab

dclab.isoelastics.**get_available_identifiers**()

Return a list of available LUT identifiers

dclab.isoelastics.**get_default**()

Return default isoelasticity lines

5.4.4 kde_contours

dclab.kde_contours.**find_contours_level**(*density*, *x*, *y*, *level*, *closed=False*)

Find iso-valued density contours for a given level value

Parameters

- **density** (*2d ndarray of shape (M, N)*) – Kernel density estimate (KDE) for which to compute the contours
- **x** (*2d ndarray of shape (M, N) or 1d ndarray of size M*) – X-values corresponding to *density*
- **y** (*2d ndarray of shape (M, N) or 1d ndarray of size M*) – Y-values corresponding to *density*
- **level** (*float between 0 and 1*) – Value along which to find contours in *density* relative to its maximum
- **closed** (*bool*) – Whether to close contours at the KDE support boundaries

Returns

contours – Contours found for the given level value

Return type

list of ndarrays of shape (P, 2)

See also:

skimage.measure.find_contours

Contour finding algorithm used

dclab.kde_contours.**get_quantile_levels**(*density*, *x*, *y*, *xp*, *yp*, *q*, *normalize=True*)

Compute density levels for given quantiles by interpolation

For a given 2D density, compute the density levels at which the resulting contours contain the fraction *1-q* of all data points. E.g. for a measurement of 1000 events, all contours at the level corresponding to a quantile of *q=0.95* (95th percentile) contain 50 events (5%).

Parameters

- **density** (*2d ndarray of shape (M, N)*) – Kernel density estimate for which to compute the contours
- **x** (*2d ndarray of shape (M, N) or 1d ndarray of size M*) – X-values corresponding to *density*
- **y** (*2d ndarray of shape (M, N) or 1d ndarray of size M*) – Y-values corresponding to *density*
- **xp** (*1d ndarray of size D*) – Event x-data from which to compute the quantile
- **yp** (*1d ndarray of size D*) – Event y-data from which to compute the quantile
- **q** (*array_like or float between 0 and 1*) – Quantile along which to find contours in *density* relative to its maximum
- **normalize** (*bool*) – Whether output levels should be normalized to the maximum of *density*

Returns

level – Contours level(s) corresponding to the given quantile

Return type

np.ndarray or float

Notes

NaN-values events in *xp* and *yp* are ignored.

5.4.5 kde_methods

Kernel Density Estimation methods

dclab.kde_methods.**bin_num_doane**(*a*)

Compute number of bins based on Doane's formula

Notes

If the bin width cannot be determined, then a bin number of 5 is returned.

See also:

bin_width_doane
method used to compute the bin width

dclab.kde_methods.**bin_width_doane**(*a*)

Compute contour spacing based on Doane's formula

References

- https://en.wikipedia.org/wiki/Histogram#Number_of_bins_and_width
- https://stats.stackexchange.com/questions/55134/doanes-formula-for-histogram-binning

Notes

Doane's formula is actually designed for histograms. This function is kept here for backwards-compatibility reasons. It is highly recommended to use *bin_width_percentile()* instead.

dclab.kde_methods.**bin_width_percentile**(*a*)

Compute contour spacing based on data percentiles

The 10th and the 90th percentile of the input data are taken. The spacing then computes to the difference between those two percentiles divided by 23.

Notes

The Freedman–Diaconis rule uses the interquartile range and normalizes to the third root of len(a). Such things do not work very well for RT-DC data, because len(a) is huge. Here we use just the top and bottom 10th percentiles with a fixed normalization.

dclab.kde_methods.**get_bad_vals**(*x*, *y*)

dclab.kde_methods.**ignore_nan_inf**(*kde_method*)

Ignores nans and infs from the input data

Invalid positions in the resulting density are set to nan.

dclab.kde_methods.**kde_gauss**(*events_x*, *events_y*, *xout=None*, *yout=None*, **args*, ***kwargs*)

Gaussian Kernel Density Estimation

Parameters

- **events_x** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **events_y** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **xout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.
- **yout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.

Returns

density – The KDE for the points in (xout, yout)

Return type

ndarray, same shape as *xout*

See also:

None

Notes

This is a wrapped version that ignores nan and inf values.

dclab.kde_methods.**kde_histogram**(*events_x*, *events_y*, *xout=None*, *yout=None*, **args*, ***kwargs*)

Histogram-based Kernel Density Estimation

Parameters

- **events_x** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **events_y** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **xout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.
- **yout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.
- **bins** (*tuple (binsx, binsy)*) – The number of bins to use for the histogram.

Returns

density – The KDE for the points in (xout, yout)

Return type

ndarray, same shape as *xout*

See also:

None, None

Notes

This is a wrapped version that ignores nan and inf values.

dclab.kde_methods.**kde_multivariate**(*events_x*, *events_y*, *xout=None*, *yout=None*, **args*, ***kwargs*)

Multivariate Kernel Density Estimation

Parameters

- **events_x** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **events_y** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **bw** (*tuple (bwx, bwy) or None*) – The bandwith for kernel density estimation.
- **xout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.
- **yout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.

Returns

density – The KDE for the points in (xout, yout)

Return type

ndarray, same shape as *xout*

See also:

None

Notes

This is a wrapped version that ignores nan and inf values.

dclab.kde_methods.**kde_none**(*events_x*, *events_y*, *xout=None*, *yout=None*)

No Kernel Density Estimation

Parameters

- **events_x** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **events_y** (*1D ndarray*) – The input points for kernel density estimation. Input is flattened automatically.
- **xout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.
- **yout** (*ndarray*) – The coordinates at which the KDE should be computed. If set to none, input coordinates are used.

Returns

density – The KDE for the points in (xout, yout)

Return type

ndarray, same shape as *xout*

Notes

This method is a convenience method that always returns ones in the shape that the other methods in this module produce.

5.4.6 polygon_filter

exception dclab.polygon_filter.**FilterIdExistsWarning**

exception dclab.polygon_filter.**PolygonFilterError**

class dclab.polygon_filter.**PolygonFilter**(*axes=None*, *points=None*, *inverted=False*, *name=None*, *filename=None*, *fileid=0*, *unique_id=None*)

An object for filtering RTDC data based on a polygonial area

Parameters

- **axes** (*tuple of str or list of str*) – The axes/features on which the polygon is defined. The first axis is the x-axis. Example: (“area_um”, “deform”).
- **points** (*array-like object of shape (N,2)*) – The N coordinates (x,y) of the polygon. The exact order is important.
- **inverted** (*bool*) – Invert the polygon filter. This parameter is overridden if *filename* is given.
- **name** (*str*) – A name for the polygon (optional).

- **filename** (*str*) – A path to a .poly file as created by this classes' *save* method. If *filename* is given, all other parameters are ignored.
- **fileid** (*int*) – Which filter to import from the file (starting at 0).
- **unique_id** (*int*) – An integer defining the unique id of the new instance.

Notes

The minimal arguments to this class are either *filename* OR (*axes*, *points*). If *filename* is set, all parameters are taken from the given .poly file.

static clear_all_filters()

Remove all filters and reset instance counter

copy(*invert=False*)

Return a copy of the current instance

Parameters
invert (*bool*) – The copy will be inverted w.r.t. the original

filter(*datax*, *datay*)

Filter a set of datax and datay according to *self.points*

static get_instance_from_id(*unique_id*)

Get an instance of the *PolygonFilter* using a unique id

static import_all(*path*)

Import all polygons from a .poly file.

Returns a list of the imported polygon filters

static instace_exists(*unique_id*)

Determine whether an instance with this unique id exists

static point_in_poly(*p*, *poly*)

Determine whether a point is within a polygon area

Uses the ray casting algorithm.

Parameters

- **p** (*tuple of floats*) – Coordinates of the point
- **poly** (*array_like of shape (N, 2)*) – Polygon (*PolygonFilter.points*)

Returns
inside – *True*, if point is inside.

Return type
bool

Notes

If *p* lies on a side of the polygon, it is defined as

- "inside" if it is on the lower or left
- "outside" if it is on the top or right

Changed in version 0.24.1: The new version uses the cython implementation from scikit-image. In the old version, the inside/outside definition was the other way around. In favor of not having to modify upstram code, the scikit-image version was adapted.

static remove(*unique_id*)

Remove a polygon filter from *PolygonFilter.instances*

save(*polyfile*, *ret_fobj=False*)

Save all data to a text file (appends data if file exists).

Polyfile can be either a path to a file or a file object that was opened with the write "w" parameter. By using the file object, multiple instances of this class can write their data.

If *ret_fobj* is *True*, then the file object will not be closed and returned.

static save_all(*polyfile*)

Save all polygon filters

static unique_id_exists(*pid*)

Whether or not a filter with this unique id exists

property hash

Hash of *axes*, *points*, and *inverted*

instances = [<dclab.polygon_filter.PolygonFilter object>]

property points

dclab.polygon_filter.**get_polygon_filter_names**()

Get the names of all polygon filters in the order of creation

5.4.7 statistics

Statistics computation for RT-DC dataset instances

exception dclab.statistics.**BadMethodWarning**

class dclab.statistics.**Statistics**(*name*, *method*, *req_feature=False*)

A helper class for computing statistics

All statistical methods are registered in the dictionary *Statistics.available_methods*.

get_feature(*ds*, *feat*)

Return filtered feature data

The features are filtered according to the user-defined filters, using the information in *ds.filter.all*. In addition, all *nan* and *inf* values are purged.

Parameters

- **ds** (dclab.rtdc_dataset.RTDCBase) – The dataset containing the feature
- **feat** (*str*) – The name of the feature; must be a scalar feature

```
available_methods = {'%-gated': <dclab.statistics.Statistics object>, 'Events':
<dclab.statistics.Statistics object>, 'Flow rate': <dclab.statistics.Statistics
object>, 'Mean': <dclab.statistics.Statistics object>, 'Median':
<dclab.statistics.Statistics object>, 'Mode': <dclab.statistics.Statistics object>,
'SD': <dclab.statistics.Statistics object>}
```

dclab.statistics.**flow_rate**(*ds*)

Return the flow rate of an RT-DC dataset

dclab.statistics.**get_statistics**(*ds*, *methods=None*, *features=None*)

Compute statistics for an RT-DC dataset

Parameters

- **ds** (dclab.rtdc_dataset.RTDCBase) – The dataset for which to compute the statistics.
- **methods** (*list of str or None*) – The methods wih which to compute the statistics. The list of available methods is given with *dclab.statistics.Statistics.available_methods.keys()* If set to *None*, statistics for all methods are computed.
- **features** (*list of str*) – Feature name identifiers are defined by *dclab.definitions.feature_exists*. If set to *None*, statistics for all scalar features available are computed.

Returns

- **header** (*list of str*) – The header (feature + method names) of the computed statistics.
- **values** (*list of float*) – The computed statistics.

dclab.statistics.**mode**(*data*)

Compute an intelligent value for the mode

The most common value in experimental is not very useful if there are a lot of digits after the comma. This method approaches this issue by rounding to bin size that is determined by the Freedman–Diaconis rule.

Parameters

data (*1d ndarray*) – The data for which the mode should be computed.

Returns

mode – The mode computed with the Freedman-Diaconis rule.

Return type

float

5.5 HDF5 manipulation

Helper methods for copying .rtdc data

dclab.rtdc_dataset.copier.**h5ds_copy**(*src_loc*, *src_name*, *dst_loc*, *dst_name=None*, *ensure_compression=True*, *recursive=True*)

Copy an HDF5 Dataset from one group to another

Parameters

- **src_loc** (*h5py.H5Group*) – The source location
- **src_name** (*str*) – Name of the dataset in *src_loc*

- **dst_loc** (*h5py.H5Group*) – The destination location
- **dst_name** (*str*) – The name of the destination dataset, defaults to *src_name*
- **ensure_compression** (*bool*) – Whether to make sure that the data are compressed, If disabled, then all data from the source will be just copied and not compressed.
- **recursive** (*bool*) – Whether to recurse into HDF5 Groups (this is required e.g. for copying the "trace" feature)

Returns
dst – The dataset *dst_loc[dst_name]*

Return type
h5py.Dataset

Raises
ValueError: – If the named source is not a h5py.Dataset

dclab.rtdc_dataset.copier.**is_properly_compressed**(*h5obj*)

Check whether an HDF5 object is properly compressed

The compression check only returns True if the input file was compressed with the Zstandard compression using compression level 5 or higher.

dclab.rtdc_dataset.copier.**rtdc_copy**(*src_h5file: Group*, *dst_h5file: Group*, *features: List[str] | Literal['all', 'scalar', 'none'] = 'all'*, *include_basins: bool = True*, *include_logs: bool = True*, *include_tables: bool = True*, *meta_prefix: str = ''*)

Create a compressed copy of an RT-DC file

Parameters

- **src_h5file** (*h5py.Group*) – Input HDF5 file
- **dst_h5file** (*h5py.Group*) – Output HDF5 file
- **features** (*list of strings or one of ['all', 'scalar', 'none']*) – If this is a list then it specifies the features that are copied from *src_h5file* to *dst_h5file*. Alternatively, you may specify 'all' (copy all features), 'scalar' (copy only scalar features), or 'none' (don't copy any features).
- **include_basins** (*bool*) – Copy the basin information from *src_h5file* to *dst_h5file*.
- **include_logs** (*bool*) – Copy the logs from *src_h5file* to *dst_h5file*.
- **include_tables** (*bool*) – Copy the tables from *src_h5file* to *dst_h5file*.
- **meta_prefix** (*str*) – Add this prefix to the name of the logs and tables in *dst_h5file*.

Tools for linking HDF5 datasets across files

exception dclab.rtdc_dataset.linker.**ExternalDataForbiddenError**

Raised when a dataset contains external data

External data are a security risk, because they could be used to access data that are not supposed to be accessed. This is especially critical when the data are accessed within a web server process (e.g. in DCOR).

dclab.rtdc_dataset.linker.**assert_no_external**(*h5*)

Raise ExternalDataForbiddenError if *h5* refers to external data

dclab.rtdc_dataset.linker.**check_external**(*h5*)

Check recursively, whether an h5py object contains external data

External data includes binary data in external files, virtual datasets, and external links.

Returns a tuple of either

- *(True, path_ext)* if the object contains external data
- *(False, None)* if this is not the case

where *path_ext* is the path to the group or dataset in *h5*.

Added in version 0.51.0.

dclab.rtdc_dataset.linker.**combine_h5files**(*paths: list*, *external: Literal['follow', 'raise'] = 'follow'*) → BinaryIO

Create an in-memory file that combines multiple .rtdc files

The .rtdc files must have the same number of events. The in-memory file is populated with the "events" data from *paths* according to the order that *paths* are given in. Metadata, including logs, basins, and tables are only taken from the first path.

Added in version 0.51.0.

Parameters

- **paths** (*list of str or pathlib.Path*) – Paths of the input .rtdc files. The first input file is always used as a source for the metadata. The other files only complement the features.
- **external** (*str*) – Defines how external (links, binary, virtual) data in *paths* should be handled. The default is to "follow" external datasets or links to external data. In a zero-trust context, you can set this to "raise" which will cause an *ExternalDataForbiddenError* exception when external data are encountered.

Returns

fd – seekable, file-like object representing an HDF5 file opened in binary mode; This can be passed to *:class:h5py.File*

Return type

BinaryIO

5.6 Writing RT-DC files

class dclab.rtdc_dataset.writer.**RTDCWriter**(*path_or_h5file: str | Path | File*, *mode: Literal['append', 'replace', 'reset'] = 'append'*, *compression_kwargs: Dict | Mapping = None*, *compression: str = 'deprecated'*)

RT-DC data writer classe

Parameters

- **path_or_h5file** (*str or pathlib.Path or h5py.Group*) – Path to an HDF5 file or an HDF5 file opened in write mode
- **mode** (*str*) – Defines how the data are stored:
  - "append": append new feature data to existing h5py Datasets
  - "replace": replace existing h5py Datasets with new features (used for ancillary feature storage)
  - "reset": do not keep any previous data

- **compression_kwargs** (*dict-like*) – Dictionary with the keys "compression" and "compression_opts" which are passed to `h5py.H5File.create_dataset()`. The default is Zstandard compression with the lowest compression level *hdf5plugin.Zstd(clevel=1)*. To disable compression, use *{"compression": None}*.
- **compression** (*str or None*) – Compression method used for data storage; one of [None, "lzf", "gzip", "szip"].

  Deprecated since version 0.43.0: Use *compression_kwargs* instead.

`close()`

Close the underlying HDF5 file if a path was given during init

`static get_best_nd_chunks(item_shape, item_dtype=<class 'numpy.float64'>)`

Return best chunks for HDF5 datasets

Chunking has performance implications. It's recommended to keep the total size of dataset chunks between 10 KiB and 1 MiB. This number defines the maximum chunk size as well as half the maximum cache size for each dataset.

`rectify_metadata()`

Autocomplete the metadta of the RTDC-measurement

The following configuration keys are updated:

- experiment:event count
- fluorescence:samples per event
- imaging: roi size x (if image or mask is given)
- imaging: roi size y (if image or mask is given)

The following configuration keys are added if not present:

- fluorescence:channel count

`store_basin(basin_name: str, basin_type: Literal['file', 'remote'], basin_format: str, basin_locs: List[str | Path], basin_descr: str | None = None, basin_feats: List[str] = None, basin_map: ndarray | Tuple[str, ndarray] = None, verify: bool = True)`

Write basin information

Parameters

- **basin_name** (*str*) – basin name; Names do not have to be unique.
- **basin_type** (*str*) – basin type (file or remote); Files are paths accessible by the operating system (including e.g. network shares) whereas remote locations normally require an active internet connection.
- **basin_format** (*str*) – The basin format must match the `format` property of an *RTDCBase* subclass (e.g. "hdf5" or "dcor")
- **basin_locs** (*list*) – location of the basin as a string or (optionally) a `pathlib.Path`
- **basin_descr** (*str*) – optional string describing the basin
- **basin_feats** (*list of str*) – list of features this basin provides; You may use this to restrict access to features for a specific basin.
- **basin_map** (*np.ndarray or tuple of (str, np.ndarray)*) – If this is an integer numpy array, it defines the mapping of event indices from the basin dataset to the referring dataset (the dataset being written to disk). Normally, the basinmap feature

used for the storing the mapping information is inferred from the currently defined basinmap features. However, if you are incepting basins, then this might not be sufficient, and you have to specify explicitly which basinmap feature to use. In such a case, you may specify a tuple *(feature_name, mapping_array)* where *feature_name* is the explicit mapping name, e.g. *"basinmap3"*.

- **verify** (*bool*) – whether to verify the basin before storing it; You might have set this to False if you would like to write a basin that is e.g. temporarily not available

Returns

basin_hash – hash of the basin which serves as the name of the HDF5 dataset stored in the output file

Added in version 0.58.0.

Return type
str

store_feature(*feat*, *data*, *shape=None*)

Write feature data

Parameters

- **feat** (*str*) – feature name
- **data** (*np.ndarray or list or dict*) – feature data
- **shape** (*tuple of int*) – For non-scalar features, this is the shape of the feature for one event (e.g. *(90, 250)* for an "image". Usually, you do not have to specify this value, but you do need it in case of plugin features that don't have the "feature shape" set or in case of temporary features. If you don't specify it, then the shape is guessed based on the data you provide and a UserWarning will be issued.

store_log(*name*, *lines*)

Write log data

Parameters

- **name** (*str*) – name of the log entry
- **lines** (*list of str or str*) – the text lines of the log

store_metadata(*meta*)

Store RT-DC metadata

Parameters
meta (*dict-like*) – The metadata to store. Each key depicts a metadata section name whose data is given as a dictionary, e.g.:

```
meta = {"imaging": {"exposure time": 20,
                    "flash duration": 2,
                    ...
                    },
        "setup": {"channel width": 20,
                  "chip region": "channel",
                  ...
                  },
        ...
        }
```

Only section key names and key values therein registered in dclab are allowed and are converted to the pre-defined dtype. Only sections from the `dclab.definitions.CFG_METADATA` dictionary are stored. If you have custom metadata, you can use the "user" section.

store_table(*name*, *cmp_array*)

Store a compound array table

Tables are semi-metadata. They may contain information collected during a measurement (but with a lower temporal resolution) or other tabular data relevant for a dataset. Tables have named columns. Therefore, they can be represented as a numy recarray, and they should be stored as such in an HDF5 file (compund dataset).

Parameters

- **name** (*str*) – Name of the table
- **cmp_array** (*np.recarray, h5py.Dataset, or dict*) – If a np.recarray or h5py.Dataset are provided, then they are written as-is to the file. If a dictionary is provided, then the dictionary is converted into a numpy recarray.

version_brand(*old_version=None*, *write_attribute=True*)

Perform version branding

Append a " | dclab X.Y.Z" to the "setup:software version" attribute.

Parameters

- **old_version** (*str or None*) – By default, the version string is taken from the HDF5 file. If set to a string, then this version is used instead.
- **write_attribute** (*bool*) – If True (default), write the version string to the "setup:software version" attribute

write_image_float32(*group*, *name*, *data*)

Write 32bit floating point image array

This function wraps *RTDCWriter.write_ndarray()* and adds image attributes to the HDF5 file so HDFView can display the images properly.

Parameters

- **group** (*h5py.Group*) – parent group
- **name** (*str*) – name of the dataset containing the text
- **data** (*np.ndarray or list of np.ndarray*) – image data

write_image_grayscale(*group*, *name*, *data*, *is_boolean*)

Write grayscale image data to and HDF5 dataset

This function wraps *RTDCWriter.write_ndarray()* and adds image attributes to the HDF5 file so HDFView can display the images properly.

Parameters

- **group** (*h5py.Group*) – parent group
- **name** (*str*) – name of the dataset containing the text
- **data** (*np.ndarray or list of np.ndarray*) – image data
- **is_boolean** (*bool*) – whether the input data is of boolean nature (e.g. mask data) - if so, data are converted to uint8

write_ndarray(*group*, *name*, *data*, *dtype=None*)

Write n-dimensional array data to an HDF5 dataset

It is assumed that the shape of the array data is correct, i.e. that the shape of *data* is (number_events, feat_shape_1, ..., feat_shape_n).

Parameters

- **group** (*h5py.Group*) – parent group
- **name** (*str*) – name of the dataset containing the text
- **data** (*np.ndarray*) – data
- **dtype** (*dtype*) – the dtype to use for storing the data (defaults to *data.dtype*)

write_ragged(*group*, *name*, *data*)

Write ragged data (i.e. list of arrays of different lenghts)

Ragged array data (e.g. contour data) are stored in a separate group and each entry becomes an HDF5 dataset.

Parameters

- **group** (*h5py.Group*) – parent group
- **name** (*str*) – name of the dataset containing the text
- **data** (*list of np.ndarray or np.ndarray*) – the data in a list

write_text(*group*, *name*, *lines*)

Write text to an HDF5 dataset

Text data are written as a fixed-length string dataset.

Parameters

- **group** (*h5py.Group*) – parent group
- **name** (*str*) – name of the dataset containing the text
- **lines** (*list of str or str*) – the text, line by line

dclab.rtdc_dataset.writer.**CHUNK_SIZE = 100**

DEPRECATED (use *CHUNK_SIZE_BYTES* instead)

dclab.rtdc_dataset.writer.**CHUNK_SIZE_BYTES = 1048576**

Chunks size in bytes for storing HDF5 datasets

5.7 Command-line interface methods

command line interface

dclab.cli.**compress**(*path_in: str | Path = None*, *path_out: str | Path = None*, *force: bool = False*, *check_suffix: bool = True*, *ret_path: bool = False*)

Create a new dataset with all features compressed lossless

Parameters

- **path_in** (*str or pathlib.Path*) – file to compress
- **path_out** (*str or pathlib*) – output file path

- **force** (*bool*) – DEPRECATED
- **check_suffix** (*bool*) – check suffixes for input and output files
- **ret_path** (*bool*) – whether to return the output path

Returns
path_out – output path (with possibly corrected suffix)

Return type
pathlib.Path (optional)

dclab.cli.**condense**(*path_in: str | Path = None*, *path_out: str | Path = None*, *ancillaries: bool = None*, *store_ancillary_features: bool = True*, *store_basin_features: bool = True*, *check_suffix: bool = True*, *ret_path: bool = False*)

Create a new dataset with all available scalar-only features

Besides the innate scalar features, this also includes all fast-to-compute ancillary and all basin features (*features_loaded*).

Parameters

- **path_in** (*str or pathlib.Path*) – file to compress
- **path_out** (*str or pathlib*) – output file path
- **ancillaries** (*bool*) – DEPRECATED, use *store_ancillary_features* instead
- **store_ancillary_features** (*bool*) – compute and store ancillary features in the output file
- **store_basin_features** (*bool*) – copy basin features from the input path to the output file
- **check_suffix** (*bool*) – check suffixes for input and output files
- **ret_path** (*bool*) – whether to return the output path

Returns
path_out – output path (with possibly corrected suffix)

Return type
pathlib.Path (optional)

dclab.cli.**condense_dataset**(*ds: RTDCBase*, *h5_cond: File*, *ancillaries: bool = None*, *store_ancillary_features: bool = True*, *store_basin_features: bool = True*, *warnings_list: List = None*)

Condense a dataset using low-level HDF5 methods

For ancillary and basin features, high-level dclab methods are used.

dclab.cli.**get_command_log**(*paths*, *custom_dict=None*)

Return a json dump of system parameters

Parameters

- **paths** (*list of pathlib.Path or str*) – paths of related measurement files; up to 5MB of each of them is md5-hashed and included in the "files" key
- **custom_dict** (*dict*) – additional user-defined entries; must contain simple Python objects (json.dumps must still work)

dclab.cli.**get_job_info**()

Return dictionary with current job information

Returns
info – Job information including details about time, system, python version, and libraries used.

Return type
dict of dicts

dclab.cli.**join**(*paths_in: List[str | Path] = None*, *path_out: str | Path = None*, *metadata: Dict = None*, *ret_path: bool = False*)

Join multiple RT-DC measurements into a single .rtdc file

Parameters

- **paths_in** (*list of paths*) – input paths to join
- **path_out** (*str or pathlib.Path*) – output path
- **metadata** (*dict*) – optional metadata dictionary (configuration dict) to store in the output file
- **ret_path** (*bool*) – whether to return the output path

Returns
path_out – output path (with corrected path suffix if applicable)

Return type
pathlib.Path (optional)

Notes

The first input file defines the metadata written to the output file. Only features that are present in all input files are written to the output file.

dclab.cli.**repack**(*path_in: str | Path = None*, *path_out: str | Path = None*, *strip_basins: bool = False*, *strip_logs: bool = False*, *check_suffix: bool = True*, *ret_path: bool = False*)

Repack/recreate an .rtdc file, optionally stripping the logs

Parameters

- **path_in** (*str or pathlib.Path*) – file to compress
- **path_out** (*str or pathlib*) – output file path
- **strip_basins** (*bool*) – do not write basin information to the output file
- **strip_logs** (*bool*) – do not write logs to the output file
- **check_suffix** (*bool*) – check suffixes for input and output files
- **ret_path** (*bool*) – whether to return the output path

Returns
path_out – output path (with possibly corrected suffix)

Return type
pathlib.Path

dclab.cli.**split**(*path_in: str | Path = None*, *path_out: str | Path = None*, *split_events: int = 10000*, *skip_initial_empty_image: bool = True*, *skip_final_empty_image: bool = True*, *ret_out_paths: bool = False*, *verbose: bool = False*)

Split a measurement file

Parameters

- **path_in** (*str or pathlib.Path*) – path of input measurement file
- **path_out** (*str or pathlib.Path*) – path to output directory (optional)
- **split_events** (*int*) – maximum number of events in each output file
- **skip_initial_empty_image** (*bool*) – remove the first event of the dataset if the image is zero
- **skip_final_empty_image** (*bool*) – remove the final event of the dataset if the image is zero
- **ret_out_paths** – if True, return the list of output file paths
- **verbose** (*bool*) – if True, print messages to stdout

Returns

[out_paths] – List of generated files (only if *ret_out_paths* is specified)

Return type

list of pathlib.Path

dclab.cli.**tdms2rtdc**(*path_tdms=None*, *path_rtdc=None*, *compute_features=False*, *skip_initial_empty_image=True*, *skip_final_empty_image=True*, *verbose=False*)

Convert .tdms datasets to the hdf5-based .rtdc file format

Parameters

- **path_tdms** (*str or pathlib.Path*) – Path to input .tdms file
- **path_rtdc** (*str or pathlib.Path*) – Path to output .rtdc file
- **compute_features** (*bool*) – If *True*, compute all ancillary features and store them in the output file
- **skip_initial_empty_image** (*bool*) – In old versions of Shape-In, the first image was sometimes not stored in the resulting .avi file. In dclab, such images are represented as zero-valued images. If *True* (default), this first image is not included in the resulting .rtdc file.
- **skip_final_empty_image** (*bool*) – In other versions of Shape-In, the final image is sometimes also not stored in the .avi file. If *True* (default), this final image is not included in the resulting .rtdc file.
- **verbose** (*bool*) – If *True*, print messages to stdout

dclab.cli.**verify_dataset**(*path_in=None*)

Perform checks on experimental datasets

5.8 R and lme4

exception dclab.lme4.rlibs.**ROutdatedError**

exception dclab.lme4.rlibs.**RPY2ImportError**

exception dclab.lme4.rlibs.**RPY2OutdatedError**

exception dclab.lme4.rlibs.**RPY2UnavailableError**

exception dclab.lme4.rlibs.**RUnavailableError**

class dclab.lme4.rlibs.**MockRPackage**(*exception*)

dclab.lme4.rlibs.**import_r_submodules**()

dclab.lme4.rlibs.**RPY2_MIN_VERSION = '2.9.4'**

Minimum rpy2 version

dclab.lme4.rlibs.**R_MIN_VERSION = '3.6.0'**

Minimum R version This is actually a dependency for rpy2, because the API changed then (ffi.error: symbol 'R_tryCatchError' not found in library).

exception dclab.lme4.rsetup.**RNotFoundError**

class dclab.lme4.rsetup.**AutoRConsole**

Helper class for catching R console output

By default, this console always returns "yes" when asked a question. If you need something different, you can subclass and override *consoleread* fucntion. The console stream is recorded in *self.stream*.

close()

Remove the rpy2 monkeypatches

consoleread(*prompt*)

Read user input, returns "yes" by default

consolewrite_print(*s*)

consolewrite_warnerror(*s*)

get_prints()

get_warnerrors()

write_to_stream(*topic*, *s*)

lock = False

perform_lock = True

dclab.lme4.rsetup.**check_r**()

Make sure R is installed an R HOME is set

dclab.lme4.rsetup.**get_r_path**()

Get the path of the R executable/binary from rpy2

dclab.lme4.rsetup.**get_r_version**()

dclab.lme4.rsetup.**has_lme4**()

Return True if the lme4 package is installed

dclab.lme4.rsetup.**has_r**()

Return True if R is available

dclab.lme4.rsetup.**import_lme4**()

dclab.lme4.rsetup.**install_lme4**()

Install the lme4 package (if not already installed)

The packages are installed to the user data directory given in `lib_path`.

dclab.lme4.rsetup.**set_r_path**(*r_path*)

Set the path of the R executable/binary for rpy2

R lme4 wrapper

exception dclab.lme4.wrapr.**Lme4InstallWarning**

class dclab.lme4.wrapr.**Rlme4**(*model='lmer'*, *feature='deform'*)

Perform an R-lme4 analysis with RT-DC data

Parameters

- **model** (*str*) – One of:
  - "lmer": linear mixed model using lme4's `lmer`
  - "glmer+loglink": generalized linear mixed model using lme4's `glmer` with an additional a log-link function via the `family=Gamma(link='log'))` keyword.
- **feature** (*str*) – Dclab feature for which to compute the model

add_dataset(*ds*, *group*, *repetition*)

Add a dataset to the analysis list

Parameters

- **ds** (RTDCBase) – Dataset
- **group** (*str*) – The group the measurement belongs to ("control" or "treatment")
- **repetition** (*int*) – Repetition of the measurement

Notes

- For each repetition, there must be a "treatment" and a "control" `group`.
- If you would like to perform a differential feature analysis, then you need to pass at least a reservoir and a channel dataset (with same parameters for *group* and *repetition*).

check_data()

Perform sanity checks on `self.data`

fit(*model=None*, *feature=None*)

Perform (generalized) linear mixed-effects model fit

The response variable is modeled using two linear mixed effect models:

- model *Rlme4.r_func_model* (random intercept + random slope model)
- the null model *Rlme4.r_func_nullmodel* (without the fixed effect introduced by the "treatment" group).

Both models are compared in R using "anova" (from the R-package "stats" [Eve92]) which performs a likelihood ratio test to obtain the p-Value for the significance of the fixed effect (treatment).

If the input datasets contain data from the "reservoir" region, then the analysis is performed for the differential feature.

Parameters

- **model** (*str (optional)*) – One of:
  - "lmer": linear mixed model using lme4's `lmer`
  - "glmer+loglink": generalized linear mixed model using lme4's `glmer` with an additional log-link function via `family=Gamma(link='log'))` [BMBW15]
- **feature** (*str (optional)*) – dclab feature for which to compute the model

Returns

results – Dictionary with the results of the fitting process:

- "anova p-value": Anova likelyhood ratio test (significance)
- "feature": name of the feature used for the analysis `self.feature`
- "fixed effects intercept": Mean of `self.feature` for all controls; In the case of the "glmer+loglink" model, the intercept is already backtransformed from log space.
- "fixed effects treatment": The fixed effect size between the mean of the controls and the mean of the treatments relative to "fixed effects intercept"; In the case of the "glmer+loglink" model, the fixed effect is already backtransformed from log space.
- "fixed effects repetitions": The effects (intercept and treatment) for each repetition. The first axis defines intercept/treatment; the second axis enumerates the repetitions; thus the shape is (2, number of repetitions) and `np.mean(results["fixed effects repetitions"], axis=1)` is equivalent to the tuple (`results["fixed effects intercept"]`, `results["fixed effects treatment"]`) for the "lmer" model. This does not hold for the "glmer+loglink" model, because of the non-linear inverse transform back from log space.
- "is differential": Boolean indicating whether or not the analysis was performed for the differential (bootstrapped and subtracted reservoir from channel data) feature
- "model": model name used for the analysis `self.model`
- "model converged": boolean indicating whether the model converged
- "r anova": Anova model (exposed from R)
- "r model summary": Summary of the model (exposed from R)
- "r model coefficients": Model coefficient table (exposed from R)
- "r stderr": errors and warnings from R
- "r stdout": standard output from R

Return type

dict

`get_differential_dataset()`

Return the differential dataset for channel/reservoir data

The most famous use case is differential deformation. The idea is that you cannot tell what the difference in deformation from channel to reservoir is, because you never measure the same object in the reservoir and the channel. You usually just have two distributions. Comparing distributions is possible via bootstrapping. And then, instead of running the lme4 analysis with the channel deformation data, it is run with the differential deformation (subtraction of the bootstrapped deformation distributions for channel and reservoir).

get_feature_data(*group*, *repetition*, *region='channel'*)

Return array containing feature data

Parameters

- **group** (*str*) – Measurement group (“control” or “treatment”)
- **repetition** (*int*) – Measurement repetition
- **region** (*str*) – Either “channel” or “reservoir”

Returns
fdata – Feature data (Nans and Infs removed)

Return type
1d ndarray

is_differential()

Return True if the differential feature is computed for analysis

This effectively just checks the regions of the datasets and returns True if any one of the regions is “reservoir”.

See also:

get_differential_features
for an explanation

set_options(*model=None*, *feature=None*)

Set analysis options

data

list of [RTDCBase, column, repetition, chip_region]

feature

dclab feature for which to perform the analysis

model

modeling method to use (e.g. “lmer”)

r_func_model

model function

r_func_nullmodel

null model function

dclab.lme4.wrapr.**bootstrapped_median_distributions**(*a*, *b*, *bs_iter=1000*, *rs=117*)

Compute the bootstrapped distributions for two arrays.

Parameters

- **a** (*1d ndarray of length N*) – Input data
- **b** (*1d ndarray of length N*) – Input data
- **bs_iter** (*int*) – Number of bootstrapping iterations to perform (outtput size).
- **rs** (*int*) – Random state seed for random number generator

Returns
median_dist_a, median_dist_b – Boostrap distribution of medians for a and b.

Return type
1d arrays of length bs_iter

See also:

https://en.wikipedia.org/wiki/Bootstrapping_(statistics)

Notes

From a programmatical point of view, it would have been better to implement this method for just one input array (because of redundant code). However, due to historical reasons (testing and comparability to Shape-Out 1), bootstrapping is done interleaved for the two arrays.

5.9 Machine learning

Added in version 0.38.0.

class dclab.rtdc_dataset.feat_anc_ml.ml_feature.**MachineLearningFeature**(*feature_name*, *dc_model*, *modc_path=None*)

A user-defined machine-learning feature

Parameters

- **feature_name** (*str*) – name of the ML feature score (starts with *ml_score_*)
- **dc_model** (dclab.rtdc_dataset.feat_anc_ml.ml_model.BaseModel) – ML model to register
- **modc_path** (*str or Path*) – path to the original .modc file (if applicable)

Notes

MachineLearningFeature inherits from *AncillaryFeature*.

hash(*rtdc_ds*)

Used for identifying an ancillary ML computation

The required features, the used configuration keys/values, and the return value of the requirement function are hashed. In addition, the ML model's info dictionary is hashed.

dclab.rtdc_dataset.feat_anc_ml.ml_feature.**load_ml_feature**(*modc_path*)

Find and load MachineLearningFeature(s) from a .modc file

Parameters
modc_path (*str or Path*) – pathname to a .modc file

Returns
ml_list – list of MachineLearningFeature instances loaded from *modc_path*

Return type
list of *MachineLearningFeature*

See also:

MachineLearningFeature
class handling the plugin feature information

dclab.rtdc_dataset.feat_anc_ml.ml_feature.**remove_all_ml_features**()

Convenience function for removing all *MachineLearningFeature* instances

See also:

remove_ml_feature
remove a single *MachineLearningFeature* instance

dclab.rtdc_dataset.feat_anc_ml.ml_feature.**remove_ml_feature**(*ml_instance*)

Convenience function for removing a *MachineLearningFeature* instance

Parameters
ml_instance (MachineLearningFeature) – The *MachineLearningFeature* instance to be removed from dclab

Raises
TypeError – If the *ml_instance* is not a *MachineLearningFeature* instance

dclab.rtdc_dataset.feat_anc_ml.ml_libs.**import_or_mock_package**(*name*, *min_version*)

Reading and writing trained machine learning models for dclab

exception dclab.rtdc_dataset.feat_anc_ml.modc.**ModelFormatExportFailedWarning**

dclab.rtdc_dataset.feat_anc_ml.modc.**export_model**(*path*, *model*, *enforce_formats=None*)

Export an ML model to all possible formats

The model must be exportable with at least one method listed by `BaseModel.all_formats()`.

Parameters

- **path** (*str or pathlib.Path*) – Directory where the model is stored to. For each supported model, a new subdirectory or file is created.
- **model** (*An instance of an ML model, NOT dclab.cfeat_anc_ml.models.BaseModel*) – Trained model instance
- **enforce_formats** (*list of str*) – Enforced file formats for export. If the export for one of these file formats fails, a ValueError is raised.

dclab.rtdc_dataset.feat_anc_ml.modc.**hash_path**(*path*)

Create a SHA256 hash of a file or all files in a directory

The files are sorted before hashing for reproducibility.

dclab.rtdc_dataset.feat_anc_ml.modc.**load_modc**(*path*, *from_format=None*)

Load models from a .modc file for inference

Parameters

- **path** (*str or path-like*) – Path to a .modc file
- **from_format** (*str*) – If set to None, the first available format in `BaseModel.all_formats()` is used. If set to a key in `BaseModel.all_formats()`, then this format will take precedence and an error will be raised if loading with this format fails.

Returns
model – Models that can be used for inference via *model.predict*

Return type
list of *dclab.rtdc_dataset.feat_anc_ml.ml_model.BaseModel*

dclab.rtdc_dataset.feat_anc_ml.modc.**save_modc**(*path*, *dc_models*)

Save ML models to a .modc file

Parameters

- **path** (*str, pathlib.Path*) – Output .modc path
- **dc_models** (*list of/or dclab.rtdc_dataset.feat_anc_ml.models.BaseModel*) – Models to save

Returns

meta – Dictionary written to index.json in the .modc file

Return type

dict

class dclab.rtdc_dataset.feat_anc_ml.ml_model.**BaseModel**(*bare_model*, *inputs*, *outputs*, *info=None*)

Parameters

- **bare_model** – Underlying ML model
- **inputs** (*list of str*) – List of model input features, e.g. `["deform", "area_um"]`
- **outputs** (*list of str*) – List of output features the model provides in that order, e.g. `["ml_score_rbc", "ml_score_rt1", "ml_score_tfe"]`
- **info** (*dict*) – Dictionary with model metadata

static all_formats()

Dict of dictionaries containing all model formats in dclab

Returns

fmt_dict – All file formats with names as keys. Each item contains the keys "name" (format name), "suffix" (saved file suffix), "requires" (Python dependencies).

Return type

dict

See also:

supported_formats

class-specific file formats

get_dataset_features(*ds*, *dtype=<class 'numpy.float32'>*)

Return the dataset features used for inference

Parameters

- **ds** (dclab.rtdc_dataset.RTDCBase) – Dataset from which to retrieve the feature data
- **dtype** (*dtype*) – All features are cast to this dtype

Returns

fdata – 2D array of shape (len(ds), len(self.inputs)); i.e. to access the array containing the first feature, for all events, you would do *fdata[:, 0]*.

Return type

2d ndarray

abstract static load_bare_model(*path*)

Load an implementation-specific model from a file

This will set the *self.model* attribute. Make sure that the other attributes are set properly as well.

abstract predict(*ds*)

Return the probabilities of *self.outputs* for *ds*

Parameters
ds (dclab.rtdc_dataset.RTDCBase) – Dataset to apply the model to

Returns
ofdict – Output feature dictionary with features as keys and 1d ndarrays as values.

Return type
dict

Notes

This function calls *BaseModel.get_dataset_features()* to obtain the input feature matrix.

abstract static save_bare_model(*path*, *bare_model*, *save_format=None*)

Save an implementation-specific model to a file

Parameters

- **path** (*str or path-like*) – Path to store model to
- **bare_model** (*object*) – The implementation-specific bare model
- **save_format** (*str*) – Must be in *supported_formats*

abstract static supported_formats()

List of dictionaries containing model formats

Returns
fmts – Each item contains the keys "name" (format name), "suffix" (saved file suffix), "requires" (Python dependencies).

Return type
list

Notes

The return value is automatically added to the return value of *BaseModel.all_formats()*.

tensorflow helper functions for RT-DC data

dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow.tf_dataset.**assemble_tf_dataset_scalars**(*dc_data*, *feature_inputs*, *labels=None*, *split=0.0*, *shuffle=True*, *batch_size=32*, *dtype=<class 'numpy.float32'>*)

Assemble a *tensorflow.data.Dataset* for scalar features

Scalar feature data are loaded directly into memory.

Parameters

- **dc_data** (*list of pathlib.Path, str, or dclab.rtdc_dataset.RTDCBase*) – List of source datasets (can be anything *dclab.new_dataset()* accepts).
- **feature_inputs** (*list of str*) – List of scalar feature names to extract from *paths*.
- **labels** (*list*) – Labels (e.g. an integer that classifies each element of *path*) used for training. Defaults to None (no labels).
- **split** (*float*) – If set to zero, only one dataset is returned; If set to a float between 0 and 1, a train and test dataset is returned. Please set *shuffle=True*.
- **shuffle** (*bool*) – If True (default), shuffle the dataset (A hard-coded seed is used for reproducibility).
- **batch_size** (*int*) – Batch size for training. The function *tf.data.Dataset.batch* is called with *batch_size* as its argument.
- **dtype** (*numpy.dtype*) – Desired dtype of the output data

Returns

train [,test] – Dataset that can be used for training with tensorflow

Return type

tensorflow.data.Dataset

dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow.tf_dataset.**get_dataset_event_feature**(*dc_data*, *feature*, *tf_dataset_indices=None*, *dc_data_indices=None*, *split_index=0*, *split=0.0*, *shuffle=True*)

Return RT-DC features for tensorflow Dataset indices

The functions *assemble_tf_dataset_** return a `tensorflow.data.Dataset` instance with all input data shuffled (or split). This function retrieves features using the *Dataset* indices, given the same parameters (*paths*, *split*, *shuffle*).

Parameters

- **dc_data** (*list of pathlib.Path, str, or dclab.rtdc_dataset.RTDCBase*) – List of source datasets (Must match the path list used to create the *tf.data.Dataset*).
- **feature** (*str*) – Name of the feature to retrieve
- **tf_dataset_indices** (*list-like*) – *tf.data.Dataset* indices corresponding to the events of interest. If None, all indices are used.
- **dc_data_indices** (*list of int*) – List with indices that correspond to the only items in *dc_data* for which the features should be returned.
- **split_index** (*int*) – The split index; 0 for the first part, 1 for the second part.
- **split** (*float*) – Splitting fraction (Must match the path list used to create the *tf.data.Dataset*)

- **shuffle** (*bool*) – Shuffling (Must match the path list used to create the *tf.data.Dataset*)

Returns
data – Feature list with elements corresponding to the events given by *dataset_indices*.

Return type
list

dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow.tf_dataset.**shuffle_array**(*arr*, *seed=42*)

Shuffle a numpy array in-place reproducibly with a fixed seed

The shuffled array is also returned.

class dclab.rtdc_dataset.feat_anc_ml.hook_tensorflow.tf_model.**TensorflowModel**(*bare_model*, *inputs*, *outputs*, *info=None*)

Handle tensorflow models

Parameters

- **bare_model** – Underlying ML model
- **inputs** (*list of str*) – List of model input features, e.g. `["deform", "area_um"]`
- **outputs** (*list of str*) – List of output features the model provides in that order, e.g. `["ml_score_rbc", "ml_score_rt1", "ml_score_tfe"]`
- **info** (*dict*) – Dictionary with model metadata

has_sigmoid_activation(*layer_config=None*)

Return True if final layer has "sigmoid" activation function

has_softmax_layer(*layer_config=None*)

Return True if final layer is a Softmax layer

static load_bare_model(*path*)

Load a tensorflow model

predict(*ds*, *batch_size=32*)

Return the probabilities of *self.outputs* for *ds*

Parameters

- **ds** (dclab.rtdc_dataset.RTDCBase) – Dataset to apply the model to
- **batch_size** (*int*) – Batch size for inference with tensorflow

Returns
ofdict – Output feature dictionary with features as keys and 1d ndarrays as values.

Return type
dict

Notes

Before prediction, this method asserts that the outputs of the model are converted to probabilities. If the final layer is one-dimensional and does not have a sigmoid activation, then a sigmoid activation layer is added (binary classification) `tf.keras.layers.Activation("sigmoid")`. If the final layer has more dimensions and is not a `tf.keras.layers.Softmax()` layer, then a softmax layer is added.

static save_bare_model(*path*, *bare_model*, *save_format='tensorflow'*)

Save a tensorflow model

static supported_formats()

List of dictionaries containing model formats

Returns

fmts – Each item contains the keys "name" (format name), "suffix" (saved file suffix), "requires" (Python dependencies).

Return type

list

Notes

The return value is automatically added to the return value of `BaseModel.all_formats()`.

CHAPTER

SIX

CHANGELOG

List of changes in-between dclab releases.

6.1 version 0.58.6

- enh: for access to private S3 data, introduce the environment variables *DCLAB_S3_ENDPOINT_URL*, *DCLAB_S3_ACCESS_KEY_ID*, and *DCLAB_S3_SECRET_ACCESS_KEY*
- ref: reorganize the *fmt_s3* module

6.2 version 0.58.5

- fix: CLI methods returned exit status 1 since paths were returned

6.3 version 0.58.4

- fix: support relative paths in basins with both nt \ and posix / path separators
- enh: store original metadata in .tsv export (#255)

6.4 version 0.58.3

- fix: keep only 1000 contours in LazyContourList, because keeping all contours causes the memory to fill up for large input files

6.5 version 0.58.2

- fix: KeyError when iteration over DCOR logs or tables
- fix: remove hack that populates feature names for basins unreliably
- enh: lazily load logs and tables in *fmt_hierarchy*

6.6 version 0.58.1

- fix: exporting with basins from a hierarchy child did not write a basin derived from the hierarchy root parent
- docs: add example for *map_indices_child2parent* in *fmt_hierarchy*
- ref: explicitly define ancillary features in *RTDC_Hierarchy*

6.7 version 0.58.0

- feat: implement "mapped basins" that have a superset of events
- feat: allow to export basin-only HDF5 files
- fix: support input files without "events" in *rtdc_copy*
- fix: CLI methods now store basin information in output paths (#239): compress, condense, and repack preserve the original input basins; split writes mapped basins (using export.hdf5); join does not write basins at all
- fix: enable nested basins for the HDF5, S3, and HTTP basins, because we have the *_local_basins_allowed* property since 0.57.6
- fix: when joining datasets, the name of the log holding the configuration of the source datasets should be src-#{i}_cfg instead of cfg_src-#{i}
- enh: introduce *RTDCBase.features_ancillary*, a list of all ancillary features (computed on-the-fly) for a dataset
- enh: add *include_basins* keyword argument for *rtdc_copy*
- enh: *RTDCWriter.store_basin* now returns the basin hash/identifier
- enh: CLI methods now return the output path
- enh: the repack CLI method now allows to strip basins
- docs: add an advanced usage section on basins
- docs: update doc strings for CLI methods
- setup: pin nptdms<1.9, because tests started to fail
- ref: modify basin *load_dataset* methods to support mapped basins
- ref: put input and output path in order for CLI methods
- tests: make sure basin features are not written automatically in CLI methods (#246)
- ci: install lme4 from archive

6.8 version 0.57.7

- fix: raise a ValueError in HTTPFile when the server does not return the status code 200
- enh: properly handle missing ETag in HTTPFile
- enh: compute fallback identifier using hash of first chunk in RTDC_HTTP
- ref: drop the old dcserve API version 1 and use version 2 by default

6.9 version 0.57.6

- enh: allow non-existent paths in *common.get_command_log*
- ref: factor out *cli.task_condense.condense_dataset* for condensing an open dataset without the knowledge of its path
- ref: when condensing a dataset, instead of the MD5-5M, the hash of its configuration is used as an suffix for previous condense logs

6.10 version 0.57.5

- fix: do not allow local basins for network-based subclasses of *RTDCBase*

6.11 version 0.57.4

- fix: catch zero-valued contour accuracy for KDE plot (#249)

6.12 version 0.57.3

- setup: the s3 extra does not require a specific version of urllib3
- ci: use main branch of GH Actions actions to reduce maintenance burden

6.13 version 0.57.2

- fix: speed-up S3 availability check (#245)
- enh: allow to specify list of features for data copier
- ref: migrate from s3fs to boto3 for fmt_s3
- test: data copier with *none* and invalid features
- docs: remove pin for sphinx
- build: make musllinux builds for e.g. docker images

6.14 version 0.57.1

- fix: RTDCWriter.rectify_metadata fails when image feature is empty
- fix: handle empty write requests in export.hdf5 and RTDCWriter (#242)
- tests: add test for writing all-nan data (#242)
- docs: improve documentation of hierarchy child mapper

6.15 version 0.57.0

- fix: integer overflow in downsample_grid
- fix: removed unnecessary computation of hierarchy filter instance
- enh: cythonize downsample_grid (~10x speed-up)
- enh: got rid of for-loop in map_indices_parent2child (~100x speed-up)
- enh: slight improvement of managing manual indices in hierarchy children
- enh: added dtype properties for contour and trace events
- enh: ensure all feature data objects have the dtype property
- enh: globally use chunk sizes of ~1MiB when writing HDF5 data (#217) (minor speedup since previously a chunk size of 100 events was used for images and scalar features were written in one big chunk)
- enh: favor array operations in *yield_filtered_array_stacks* (~4x speed-up)
- enh: increase verbosity for unreachable remote basin features
- ref: implement *__array__* methods for hierarchy event classes
- ref: implement *__array__* method for *H5MaskEvent*
- ref: new submodule for hierarchy format

6.16 version 0.56.3

- fix: regression missing check for basin availability

6.17 version 0.56.2

- enh: perform basin availability checks in daemon thread

6.18 version 0.56.1

- enh: priority-based basin sorting (file over remote, http over dcor)

6.19 version 0.56.0

- feat: allow nested basins
- feat: implement DCOR basins
- fix: make sure basins are always closed on context exit (#238)
- fix: failed to load basin data when port was specified in location
- enh: requests session pooling for fmt_http and fmt_dcor
- enh: implement context manager for RTDCBase
- enh: be forwards-compatible and ignore unsupported basin formats

- ref: new http_utils submodule for managing HTTP connections

6.20 version 0.55.7

- test: tested rtdc_dataset.config.Configuration is pickable (#190)
- enh: add dcnum pipeline identifier metadata constants

6.21 version 0.55.6

- fix: default *host* was overriding URL host in fmt_dcor

6.22 version 0.55.5

- enh: register new features area_um_raw, deform_raw, eccentr_prnc, per_ratio, per_um_raw, sym_x, sym_y
- ref: simplify/cleanup computation of tilt feature

6.23 version 0.55.4

- enh: improve timeout and availability check for fmt_http

6.24 version 0.55.3

- enh: migrate to purely requests-based HTTP file for fmt_http

6.25 version 0.55.2

- fix: fmt_dcor did not properly retry failed connections
- enh: minor optimizations for fmt_http and fmt_s3

6.26 version 0.55.1

- fix: disable instance cache fsspec.HTTPFileSystem (fmt_http)

6.27 version 0.55.0

- feat: implement HDF5-based HTTP file format and basin
- fix: replace lru_cache on instance methods to fix memory leak (#214)
- fix: remove lru_cache in DCOR format (#226)
- fix: implement *__contains__* for DCOR logs and tables
- enh: improve parsing of DCOR URLs (scheme in host)
- enh: speed-up of initialization of DCOR data
- enh: add requests timeout for DCOR data
- enh: more caching of event size and shape for HDF5 format
- enh: faster computation of contour length for DCOR format
- enh: use dcserv version 2 in DCOR format (fast S3 access)
- enh: change measurement identifier upon filtered export
- setup: pin s3fs>=2023.10.0
- setup: pin upper bounds of dependencies

6.28 version 0.54.2

- fix: availability checks for basins

6.29 version 0.54.1

- fix: properly check for basin availability before getting features

6.30 version 0.54.0

- fix: partial workaround for #238
- fix: hash alias for HDF5 Datasets in files not in the file system
- enh: delayed computation of measurement identifiers for basins
- enh: allow to specify which features are served by a basin
- enh: improve speed of S3 file format by increasing chunk size
- enh: do not immediately raise an exception when metadata are missing
- enh: lazy initialization of .RTDCBase.filters
- enh: lazy initialization of .RTDCBase.basins
- enh: lazy loading of feature list for HDF5 data
- setup: migrate from pkg_resources to importlib_resources
- ref: remove the dclab.isoelastics.ISOFILES constant

- ref: remove the dclab.features.emodulus.INTERNAL_LUTS constant

6.31 version 0.53.3

- fix: bad hdf5 basin check

6.32 version 0.53.2

- fix: catch OSError when accessing invalid hdf5 basin
- enh: implement retrieving basins in fmt_dcor

6.33 version 0.53.1

- fix: catch OSError when accessing invalid S3 URL
- enh: implement *Configuration.as_dict*
- enh: implement *Basin.as_dict*

6.34 version 0.53.0

- feat: introduce remote type basin format s3
- feat: implement *store_basin* in *RTDCWriter*
- docs: add information about basins
- ref: expand *fmt_dcor* submodule laterally
- ref: deprecate the *feat_anc_ml* submodule
- enh: improve URL parsing for s3 format
- enh: extract dataset size from metadata before resorting to features
- enh: support remote basins

6.35 version 0.52.6

- maintenance release

6.36 version 0.52.5

- maintenance release

6.37 version 0.52.4

  - ref: HDF5Basin gets its own submodule
  - setup: oldest-supported-numpy is not required anymore, since numpy is backwards-compatible now

0.52.3 - maintenance release

6.38 version 0.52.2

- maintenance release

6.39 version 0.52.1

- docs: document the copier submodule (#218)
- docs: simplify headings in advanced section
- docs: refactor writing DC data into its own subsection
- docs: add advanced section on how to work with S3 data (#228)
- docs: add information about S3 object storage in DCOR section
- setup: force requests >= 2.31.0 (CVE-2023-32681)
- setup: bump scipy to 1.10.0 (CVE-2023-25399)
- tests: add test tables and logs tests for linker
- tests: add tests for S3 file format (#228)

6.40 version 0.52.0

- feat: qpi features and related metadata added to definitions (#192)

6.41 version 0.51.4

- setup: wrong package setup

6.42 version 0.51.3

- setup: wrong package setup

6.43 version 0.51.2

- enh: support dictionary of arrays in RTDCWriter.store_table
- tests: added missing test files

6.44 version 0.51.1

- fix: workaround for segmentation fault caused by HDF5

6.45 version 0.51.0

- feat: introduce read-support for local basins (upstream features)
- fix: minor bug lead to issue with latest rpy2 versions (invisible)
- fix: support numpy integers and booleans for JSON serializer
- enh: introduce optional configuration key [experiment]:"run identifier"
- docs: restructuring and introduction of new features
- ref: restructured fmt_hdf5 submodule
- ref: remove deprecated *write* method for creating .rtdc files
- ref: introduce more general *RTDCBase._finalize_init*
- tests: tests passing on Python 3.11

6.46 version 0.50.4

- setup: cleanup pyproject.toml

6.47 version 0.50.3

- fix: do not allow editing reserved filter properties (#210)
- fix: do not allow editing temporary features (#202)
- ref: read-only features and proper subclassing in *RTDC_Dict*
- ref: subclass from *collections.UserDict* and not from *dict*
- docs: add more information about manual filters (#173)

6.48 version 0.50.2

- enh: support reading 'tables' from DCOR datasets (#221)

6.49 version 0.50.1

- enh: support reading 'logs' from DCOR datasets
- enh: hierarchy parents inherit their parent's logs and tables

6.50 version 0.50.0

- feat: support opening objects on S3-compatible storage (#213)
- feat: support opening file-like objects as HDF5 data
- enh: use regexp to identify valid DCOR URLs
- enh: allow to pass keyword arguments to h5py.File for fmt_hdf5

6.51 version 0.49.1

- fix: do not recompute inert_ratio_cvx and inert_ratio_raw computed with Shape-In >= 2.0.5 (#224)
- tests: add additional test for inertia ratio (#223)

6.52 version 0.49.0

- feat: implement *is_properly_compressed*, *rtdc_copy*, and *h5ds_copy* in *dclab.rtdc_dataset.copier* (#216)
- feat: support "tables" (compound arrays) in HDF5 files
- fix: explicitly use 64bit ints and floats when instantiating numpy arrays, because numpy on Windows defaults to 32bit ints which resulted in overflows when computing inertia ratio
- enh: unify names of exported logs
- enh: CLI use *rtdc_copy* in dclab-compress, dclab-condense, and dclab-repack (#216)
- enh: CLI compute the command log as early as possible
- enh: ignore "def" feature when performing an unknown-feature check
- enh: minor improvements in error message verbosity here and there
- ref: remove deprecated method dclab.load.check_dataset

6.53 version 0.48.8

- enh: CLI new –no-ancillary-features keyword argument for dclab-condense to skip the computation of expensive features

6.54 version 0.48.7

- maintenance release

6.55 version 0.48.6

- maintenance release

6.56 version 0.48.5

- fix: CLI dclab-verify-datasets now complains about missing input data
- enh: feed kwargs to sp.check_output to avoid pop-up command prompts
- ref: migrate from “tensorflow-SavedModel” to “tf” in ml submodule
- build: migrate to pyproject.toml (except for cython extensions)
- build: add manylinux wheels using cibuildwheel

6.57 version 0.48.4

- fix: do not cache all feature names, only defective feature names, because some features can be (de)registered dynamically

6.58 version 0.48.3

- fix: always treat ancillary time as float64
- enh: add defective feature check for inertia ratio features inert_ratio_cvx, inert_ratio_prnc, inert_ratio_raw, and tilt (#212)
- enh: cache available feature names in the HDF5 format

6.59 version 0.48.2

- fix: invalid computation of inertia ratio features (inert_ratio_cvx, inert_ratio_prnc, inert_ratio_raw, tilt) for events with large horizontal extent due to an integer overflow (#212)
- enh: many-fold increase in HDF5 dataset read speed via caching (#189)
- enh: detect all-zero temperature in dclab-verify-dataset (#183)

6.60 version 0.48.1

- fix: emodulus values were not correctly cached

6.61 version 0.48.0

- BREAKING CHANGE: remove deprecated "emodulus model" key from dataset configuration
- feat: new 'buyukurganci-2022' model to compute MC-PBS viscosity (#197)
- enh: use the 'buyukurganci-2022' model for isoelastics by default
- enh: increase verbosity when opening an .rtdc file failed
- enh: support shorthand for known media for viscosity computation
- docs: update part about DCOR (#182)

6.62 version 0.47.8

- fix: implement cleaner solution for time computation (#207)

6.63 version 0.47.7

- fix: include time offset when computing defective time feature (#207)
- ref: make the h5py.File object public in RTDC_HDF5

6.64 version 0.47.6

- fix: register float32 time from ShapeIn as defective feature
- docs: make "edit on GitHub" link lead to actual source file again

6.65 version 0.47.5

- enh: allow temporary features for hierarchy children (#123)
- enh: more type annotations (partially #198)
- build: add Python 3.11 CI pipeline

6.66 version 0.47.4

- enh: allow pathlib.Path objects in cli.get_command_log custom dicts

6.67 version 0.47.3

- fix: np.histrogram2d does not expect broken normed argument
- docs: fixed GH Actions badge
- enh: implement RTDC_HDF5.__len__

6.68 version 0.47.2

- enh: add –version flag to CLI

6.69 version 0.47.1

- fix: matplotlib.plot hierarchy child scalar data not working

6.70 version 0.47.0

- feat: lazy-loading and improved caching strategy for hierarchy child datasets via *RTDC_Hierarchy* (the call to *apply_filter* is now mandatory for all changes to propagate to the children)
- feat: introduce *RTDC_Hierarchy.rejuvenate*, an alias for *apply_filter* which more accurately describes what happens there
- feat: added file_monitoring_lru_cache convenience decorator
- enh: new feature bg_med: Median frame background brightness
- enh: add min/max/mean values as scalar dataset attributes (#175)

6.71 version 0.46.6

- fix: tdms format config parser not working when para.ini missing

6.72 version 0.46.5

- enh: new staticmethod RTDC_TDMS.extract_tdms_config for extracting metadata from tdms-based datasets; This method optionally returns a list of paths used for extracting metadata
- ref: cleanup in RTDC_TDMS with dict.setdefault

6.73 version 0.46.4

- enh: allow to export logs alongside HDF5 data (#180)
- enh: add features flow_rate and pressure

6.74 version 0.46.3

- docs: make compilation run on Windows
- docs: update information on plugin features (#151)
- ref: remove Zellmechanik Dresden branding

6.75 version 0.46.2

- fix: support passing instances of *Configuration* and *ConfigurationDict* to *dclab.util.obj2bytes*
- enh: return sorted json representation in *Configuration.tojson*
- ref: use *functools.update_wrapper* in *Cache* wrapper
- docs: make sure *dclab.downsampling.downsample_grid* is in code ref

6.76 version 0.46.1

- fix: ConfigurationDict should not subclass from dict (#174)
- ref: use f-strings and print correct feature for temporary feature error message (#193)

6.77 version 0.46.0

- BREAKING CHANGE: The LUT data for LE-2D-FEM-19 had to be revised, because the simulation support was sparse at high deformations (#191)
- feat: added new LUT and isoelastics for HE-2D-FEM-22 and HE-3D-FEM-22 (#188)
- docs: new LUT data version 10.6084/m9.figshare.12155064.v4

6.78 version 0.45.0

- feat: introduce 10th and 90th percentile brightness features
- enh: added definitions for the Haralick texture features (computed using the mahotas package from bg-corrected event mask)

6.79 version 0.44.0

- feat: introduce background-corrected brightness features
- enh: detect confusion in plugin feature names (#179)
- enh: cache innate feature data for hierarchy children
- enh: raise more verbose error message when the user tries to access non-existent features in hierarchy children (#92)

6.80 version 0.43.1

- fix: script fem2rtdc.py did not work with latest openCV (#176)
- fix: properly implement random pixel offsets in fem2rtdc.py and and pixelation_correction.py (#178)
- enh: improved test for compression (require Zstd with level >= 5)
- enh: added *close* method for *RTDCWriter*
- ref: added tests and cleaned up fem2rtdc.py

6.81 version 0.43.0

- feat: introduce Zstandard compression via hdf5plugin
- enh: speed-up HDF5 data export if *filltarr* are all-True
- ref: deprecate “compression” keyword argument for HDF5 export

6.82 version 0.42.3

- enh: support hierarhcy feature dtype property for image and mask event
- docs: fix regression in builds due to new H5ScalarEvent

6.83 version 0.42.2

- fix: lazy-load scalar feature data for HDF5 data
- fix: fix repr string of AncillaryFeature when identifier is None
- ref: compute identifiers from HDF5 data using file system path and HDF5 path

6.84 version 0.42.1

- fix: support "image_bg" feature for DCOR data
- ref: replace formats with f-strings in check.py

6.85 version 0.42.0

- enh: reduce computation time of file hashes in the cli submodule (and thus all other software downstream) by switching from a full SHA256 hash to a 5MB md5 hash; this speeds up operations on slow network shares
- ref: minor change when installing R packages (rlme tests are broken on Windows currently)
- ref: use np.float32 for principal inertia ratio computation (compatibility with OpenCV)

6.86 version 0.41.0

- feat: pull DCOR access token management from downstream DCOR-Aid
- feat: allow to set alternate DCOR server certificate bundles by appending paths *dclab.rtdc_dataset.fmt_dcor.DCOR_CERTS_SEARCH_PATHS*
- enh: allow to skip checks and by default use the innate features during export of an RTDCBase
- setup: remove appveyor build pipeline
- ref: cleanup fmt_dcor

6.87 version 0.40.0

- setup: bump numpy from 1.17.0 to 1.21.0
- setup: bump scipy from 0.14.0 to 1.8.0
- setup: drop support for Python 3.7

6.88 version 0.39.18

- fix: regression since 0.39.0; contours with wrong index where not repaired
- ref: Python distutils library is deprecated
- ref: some scipy namespaces are deprecated

6.89 version 0.39.17

- enh: allow to specify unique_id of PolygonFilter when loading from file

6.90 version 0.39.16

- fix: not all statistics exported if no feature was specified

6.91 version 0.39.15

- fix: fix accessing inherited non-scalar temporary or plugin features for RTDC_Hierarchy class (#165)
- fix: properly handle user-defined (temporary or plugin) features when exporting to HDF5 (#166)
- enh: add "shape" keyword argument in RTDCWrite.store_feature to allow the user to pass the correct data shape; this reduces warning messages when using the RTDCBase.export.hdf5 functionality
- ref: simplify data export to HDF5 with generator function
- tests: use correct tdms-dataset for tdms tests

6.92 version 0.39.14

- fix: enable loading of non-scalar plugin data as h5py.Dataset (#162)
- build: minor fixes in CI and testing pipeline

6.93 version 0.39.13

- fix: recompute the "ml_class" feature if the ml_score features change (either the underlying model or the number of features)

6.94 version 0.39.12

- fix: support nan values for the computation of the "ml_class" feature (#161)
- fix: cache of util.hashfile used access times of files (and thus might not have been of any use in some cases)

6.95 version 0.39.11

- fix: misleading error message when loading plugin features with Python modules that are not available (#160)
- docs: add information on how to call CLI functions from within Python (#145)

6.96 version 0.39.10

- fix: make absolutely sure HDF5 files are closed if RTDCWriter is used as a context manager
- docs: properly document the shapes option of plugin features (#146)

6.97 version 0.39.9

- fix: spelling of Young's modulus (#159)
- ref: define minimal R version 3.6.0 in lme4 submodule and propagate any errors with possible solutions to the user
- ref: extract exact version string for R

6.98 version 0.39.8

- enh: cache computed file hashes with additional os.stat info
- tests: fix tests on Windows

6.99 version 0.39.7

- fix: handle empty trace feature in RTDCWriter.rectify_metadata

6.100 version 0.39.6

- fix: do not close H5File object passed to RTDCWriter.__init__

6.101 version 0.39.5

- fix: write FeatureSetNotIdenticalJoinWarnings to logs in dclab-join

6.102 version 0.39.4

- fix: KeyError object 'index_online' doesn't exist when joining two datasets with dclab-join (#158)
- fix: generalize online_filter configuration keys (#156)
- fix: relax validation of online_filter configuration key types
- fix: make sure that only common features of the input files for dclab-join are written to the output file (#157)
- enh: add *with_unit* keyword to *get_feature_label*
- ref: slightly modified online_filter configuration descriptions

6.103 version 0.39.3

- ref: deprecate these lists/dicts in dclab.definitions: config_descr, config_funcs, config_types, feature_names, feature_labels, feature_name2label (#135)
- ref: use more intuitive trace feature in fmt_dcor

6.104 version 0.39.2

- ref: renamed mask and contour labels
- ref: increase verbosity in error messages
- tests: add .modc test model and data

6.105 version 0.39.1

- fix: don't create __pycache__ folders for plugin features
- fix: add "date" to machine-learning feature info dict

6.106 version 0.39.0

- enh: implement .shape and .__len__ for non-scalar features and all file formats (#117)
- ref: adjust ancillary machine-learning feature API to match that of plugin features (this is not a breaking change, because nobody is using machine-learning features yet)
- ref: minor refactoring of tdms file format code might lead to faster loading of trace data

6.107 version 0.38.2

- fix: reduce memory usage when writing "mask" feature data

6.108 version 0.38.1

- maintenance release

6.109 version 0.38.0

- enh: add identifier to AncillaryFeature class
- enh: introduce MachineLearningFeature to simplify the use of machine learning models in ancillary features
- fix: load_modc should return a list of models
- ref: rename *rtdc_dataset.ancillary* submodule to *anc_feat_core*
- ref: rename *rtdc_dataset.plugins* submodule to *anc_feat_plugin*
- ref: move the *ml* submodule to *rtdc_dataset.anc_feat_ml*
- ref: move tensorflow-related code into its own submodule
- setup: bump minimum Python version to 3.7

6.110 version 0.37.3

- fix: ignore defective feature "volume" in pre 0.37.0 .rtdc files
- fix: always perform version branding when using RTDCWriter context manager
- ref: silence numpy deprecation warning in RTDCWriter

6.111 version 0.37.2

- enh: speed-up contour computation by a factor of three (volume is computed by factor of two faster)

6.112 version 0.37.1

- enh: make IntegrityChecker class available on top level module

6.113 version 0.37.0

- BREAKING CHANGE: The volume feature in all previous versions was overestimated by about 2µm^3. Please re-run your analysis pipeline.
- fix: volume was computed incorrectly (#141)

6.114 version 0.36.1

- setup: bump h5py version to 3.0 (issue with integer chunk argument)

6.115 version 0.36.0

- feat: introduce new RTDCWriter class (#144)
- fix: in some rare cases, an empty .rtdc file could be generated when splitting an .rtdc file via the CLI dclab-split
- fix: user-defined metadata were not copied with CLI repack
- fix: support exporting non-scalar plugin features to HDF5 (#143)
- enh: implement consistent *.shape* property for contour data (partially #117)
- enh: improve export speed of contour data
- enh: improve export speed if data filtering is disabled
- ref: disable *time_offset* of internal *export.hdf5_append* function
- ref: deprecate write_hdf5.write function
- ref: deprecate export.hdf5_autocomplete_config function
- ref: deprecate export.hdf5_append function

6.116 version 0.35.8

- fix: write logs to fixed-length strings in HDF5 files (HDF5 never supported variable length string datasets with fletcher32 or compression filters)
- fix: non-scalar features that are not defined in dclab were silently ignored upon export; temporarily raise a ValueError instead (#143)
- setup: release pinning of h5py

6.117 version 0.35.7

- fix: more Shape-In versions are affected by the medium-write bug addressed in 0.33.1
- enh: improve error handling when loading rpy2
- enh: add checks for negative or zero-valued metadata
- tests: fix broken example dataset with zero-valued metadata (#137)

6.118 version 0.35.6

- fix: config converters were not applied to raw metadata (#139)

6.119 version 0.35.5

- enh: add sanity_check method to IntegrityChecker

6.120 version 0.35.4

- enh: implement function for obtaining the description of a configuration key get_config_value_descr

6.121 version 0.35.3

- enh: implement dumping of Configuration to json

6.122 version 0.35.2

- enh: allow to compress, condense, and repack without checking for the input file suffix
- docs: fixed example data (sphinx encountered unexpected warnings)

6.123 version 0.35.1

- fix: ignore empty logs (raised IndexError before)
- tests: increase coverage (#136)

6.124 version 0.35.0

- enh: support new configuration keys for online polygon filtering in the form of "[online_filter]: area_um,deform polygon points" (#134)
- enh: new helper functions in dclab.definitions: get_config_value_type, get_config_value_func, and config_key_exists
- ref: issue a warning when the data type of a configuration key is not correct

6.125 version 0.34.6

- fix: make get_job_info and get_command_log available in cli submodule
- tests: add documentation for test data (#119)

6.126 version 0.34.5

- fix: cast data to float when filling with nan values when checking the size of ancillary features
- fix: dclab-split did not work when nptdms is not installed
- tests: skip tests that require nptdms, requests, or rpy2 if those
- enh: add check to warn users from setting None-type config values packaages are not installed
- tests: add test for non-scalar plugin feature
- ci: run tests with different setup.py-extras installed

6.127 version 0.34.4

- fix: use temporary file names during CLI operations

6.128 version 0.34.3

- fix: workaround for when rpy2 finds R_HOME, but R doesn't

6.129 version 0.34.2

- fix: slicing of non-scalar features for hierarchy children (#128)
- fix: passing a directory to the CLI tdms2rtdc did not work
- enh: support slicing of non-scalar features for DCOR data (#132)
- enh: support slicing of the contour feature for HDF5 data (#132)
- enh: support slicing of LazyContour for contour AncillaryFeature (#132)
- enh: minor optimizations of the DCOR format
- ref: raise NotImplementedError for futile attempts to slice the image, contour, or mask features for .tdms data (#132)
- ref: cleanup CLI code
- tests: increase coverage of CLI (#116)

6.130 version 0.34.1

- enh: introduce user-defined "user" configuration section (#125)
- scripts: refactor fem2lutiso_std.py and fem2iso_volume.py so that model-specific post-processing is done in hooks named after the FEM identifier, e.g. fem_hooks/LE-2D-FEM-19.py (#90)
- docs: new LUT data version 10.6084/m9.figshare.12155064.v3
- ref: minor code cleanup

6.131 version 0.34.0

- feat: introduce user-defined plugin features (#105)
- fix: dclab-verify-dataset now also prints other errors it encounters
- fix: installing the "lmer" R package failed, because "statmod" was not installed
- fix: correct data types for "fluorescence:sample rate", "imaging: roi position x", and "imaging: roi position y" (#124)
- enh: support new .rtdc attribute "online_contour:bg empty" which is True when the online background image is computed only from frames that do not contain any events (#124)
- enh: *AncillaryFeature* now populates other ancillary features when they share the same method (#104)
- enh: dclab-verify-dataset now returns a non-zero exit code if there were errors, alerts, or violations (#120)
- ref: streamlined dataset check function for missing meta data keys

6.132 version 0.33.3

- fix: add "chip identifier" to "setup" configuration section and make it optional during dataset checks (#109)
- fix: ignore empty metadata strings (partly #109); removed the *check_metadata_empty_string* check function because it does not apply anymore

6.133 version 0.33.2

- fix: some datasets with unknown feature names could not be opened (AssertionError regression in 0.33.1)
- fix: workaround for sporadic JSONDecodeError when accessing DCOR
- ref: cleanup cli.py
- ref: cleanup util.py and deprecate *hash_class* argument in *hashfile*

6.134 version 0.33.1

- fix: add dataset check for wrong medium written to .rtdc file by Shape-In
- fix: filters were ignored when exporting trace data to hdf5 (#112)
- enh: allow to set ICue identifier for integrity checking
- ref: code cleanup in export.py

6.135 version 0.33.0

- feat: introduce user-defined temporary features (point 2 in #98)
- fix: catch errors for integrity checks on non-raw datasets (#102)
- fix: add check for negative fluorescence values (#101)
- enh: add metadata keys for baseline offset (#107)
- setup: remove deprecated setup.py test

6.136 version 0.32.5

- fix: add check for zero-flow rate in dclab-verify-dataset
- setup: added new file CREDITS for docs and only use maintainer in setup.py
- docs: add autodoc to constant variables (#94)
- docs: add "features_innate" and "features_loaded" to scripting goodies section
- ref: cleanup of RTDCBase class
- ref: int/bool deprecation warnings in numpy 1.20.0 (#93)
- tests: test for area_cvx as a float, consistent with output from Shape-In (#96)

6.137 version 0.32.4

- fix: TypeError when registering emodulus LUT (#91)
- ref: minor cleanup

6.138 version 0.32.3

- build: use oldest-supported-numpy in pyproject.toml

6.139 version 0.32.2

- fix: export trace data in chunks to avoid out-of-memory errors when compressing large files
- ref: introduce CHUNK_SIZE in write_hdf5.py and use it when exporting to .rtdc

6.140 version 0.32.1

- enh: dclab-compress now by default does not compress any input files that are already fully compressed (fully compressed means that all HDF5 datasets are compressed somehow); to get the old behavior back (compress in any case, use the "force" keyword argument)

6.141 version 0.32.0

- feat: allow to register external look-up tables for Young's modulus computation (#88)
- ref: restructure look-up table file handling
- ref: deprecated [calculation]: "emodulus model" metadata key in favor of the more descriptive "emodulus lut" key.
- ref: the "method" argument in the context of isoelasticity lines is deprecated in favor of the "lut_identifier" argument

6.142 version 0.31.5

- fix: writing "filtering" and "calculation" metadata sections to .rtdc files should not be allowed

6.143 version 0.31.4

- ci: fix rtd builds
- ci: fix PyPI releases

6.144 version 0.31.3

- ci: migrate to GitHub Actions

6.145 version 0.31.2

- enh: add soft type check (assertion) for "emodulus medium" key in ancillary features (#86)
- fix: make sure that strings are not written as bytes in hdf5 files

6.146 version 0.31.1

- enh: add boolean "model converged" key to return dictionary of *Rlme4.fit* (#85)

6.147 version 0.31.0

- feat: implement (generalized) linear mixed-effects models via a wrapper around R/lme4 using rpy2 (install with extra "lme4")

6.148 version 0.30.1

- fix: *new_dataset* attempts to load DCOR dataset when given a non-existent path as a string (#81)

6.149 version 0.30.0

- BREAKING CHANGE: drop support for Python 2 (#34)
- feat: new machine learning (ML) submodule *dclab.ml*
- feat: implement ML model file format .modc (#78)
- feat: add tensorflow helper functions for RT-DC data
- setup: bump numpy>=1.17.0
- ref: minor improvements of code readability
- tests: set global temp dir and remove it after testing

6.150 version 0.29.1

- enh: lift restrictions on valid options for [setup]:medium (can now be any arbitrary string)

6.151 version 0.29.0

- feat: support the "image_bg" feature which contains the rolling mean background image computed by Shape-In

6.152 version 0.28.0

- feat: new CLI command dclab-split to split a large dataset into multiple smaller datasets

6.153 version 0.27.11

- fix: do not cache hierarchy child feature values; this might lead to intransparent situations where a child has different features than its parent (you cannot always rely on the user to call *apply_filter*)
- fix: hierarchy child configuration section "calculation" was not updated with the hierarchy parent values
- docs: add example for loading data from DCOR and computing the Young's modulus

6.154 version 0.27.10

- fix: support unicode characters when writing HDF5 in Python2

6.155 version 0.27.9

- docs: add artwork
- fix: support unicode characters in sample names in Python2

6.156 version 0.27.8

- docs: add more information on emodulus computation
- docs: add script for visualizing emodulus LUTs

6.157 version 0.27.7

- ref: replace deprecated .tostring() with .tobytes()

6.158 version 0.27.6

- fix: video seek issue workaround also for the first 100 frames
- cli: also skip the final event in tdms2rtdc if the image is empty
- cli: renamed kwarg *–include-initial-empty-image* to *include-empty-boundary-images*
- enh: improve detection and recovery of missing images for fmt_tdms

6.159 version 0.27.5

- maintenance build

6.160 version 0.27.4

- maintenance build

6.161 version 0.27.3

- fix: ignore ResourceWarning due to unknown _io.BufferedReader in third-party software when converting .tdms to .rtdc

6.162 version 0.27.2

- maintenance build

6.163 version 0.27.1

- setup: bump imageio to 2.8.0 for Python>=3.4
- ref: removed NoImageWarning during export (warning is already issued by fmt_tdms.event_image)

6.164 version 0.27.0

- feat: introduce new feature names *ml_score_???* where *?* can be a digit or a lower-case letter of the alphabet (#77)
- feat: introduce new functions *dclab.definitions.feature_exists* and *dclab.definitions.scalar_feature_exists* for checking the existence of features (including the *ml_score_???* features which are not in *dclab.definitions.feature_names*)
- feat: introduce ancillary feature *ml_class* which is defined by the *ml_score_???* features
- enh: fmt_dict automatically converts scalar features to arrays
- ref: replace check for *dclab.definitions.feature_names* by *dclab.definitions.feature_exists* where applicable
- ref: replace access of *dclab.definitions.feature_name2label* by *dclab.definitions.get_feature_label* where applicable
- ref: do not automatically fill up all the box filtering ranges in *RTDCBase.config["filtering"]* with zeros; raise ValueError if user forgets to set both ranges
- docs: major revision (promote Shape-Out 2 and DCOR)

6.165 version 0.26.2

- fix: *kde_methods.bin_num_doane* now uses 5 as default if it ecounters nan or zero-division
- docs: updates related to Young's modulus computation

6.166 version 0.26.1

- enh: cache more online data in fmt_dcor
- enh: add *dclab.warn.PipelineWarning* which is used as a parent class for warnings that a user might be interested in
- fix: temperature warnings during emodulus computation revealed only the lower temperature limit of the data

6.167 version 0.26.0

- feat: implement volume-deformation isoelasticity lines (#70)
- fix: specifying an external LUT as ndarray did not work
- scripts: finish 'fem2iso_volume.py' for extracting volume- deformation isoelasticity lines
- scripts: add 'pixelation_correction.py' for visualizing pixelation effects on area_um, volume, and emodulus
- ref: renamed isoelasticity line text files

6.168 version 0.25.0

- fix: appending data to an hdf5 file results in a broken "index" feature (re-enumeration from 0), if the given dataset contains the "index_online" feature
- enh: allow to set external LUT files or LUT data when computing the Young's modulus with the *lut_data* keyword argument in *dclab.features.emodulus.get_emodulus*.
- ref: refactored *features.emodulus*: New submodules *pxcorr* and *scale_linear*; *convert* is deprecated in favor of *scale_feature*

6.169 version 0.24.8

- setup: include Python 3.8 builds and remove Python<=3.5 builds
- scripts: renamed 'extract_lut_and_iso.py' to 'fem2lutiso_std.py'

6.170 version 0.24.7

- fix: *ConfigurationDict.update* did not take into account invalid keys (everything is now done with (__setitem__`)

6.171 version 0.24.6

- maintenance release

6.172 version 0.24.5

- maintenance release

6.173 version 0.24.4

- maintenance release

6.174 version 0.24.3

- fix: *ConfigurationDict.update* did not actually perform the requested update (does not affect *Configuration.update*)
- enh: also use points_in_polygon from scikit-image to determine contour levels

6.175 version 0.24.2

- build: import new skimage submodules so that PyInstaller will find and use them

6.176 version 0.24.1

- enh: improve polygon filter speed by roughly two orders of magnitude with a cython version taken from scikit-image; there are only minor differences to the old implementation (top right point included vs. lower left point included), so this is not a breaking change (#23)

6.177 version 0.24.0

- data: refurbished LUT for linear elastic spheres provided by Dominic Mokbel and Lucas Wittwer (based on the FEM simulation results from https://doi.org/10.6084/m9.figshare.12155064.v2); compared to the old LUT, there is a relative error in Young's modulus below 0.1 %, which should not cause any breaking changes
- data: updated isoelasticity lines (better spacing): analytical data was made available by Christoph Herold, numerical data was interpolated from the new LUT
- scripts: added 'scripts/extract_lut_and_iso.py' for extracting Young's modulus LUT and isoelastics from FEM simulation data provided by Lucas Wittwer; this is now the default method for extracting new LUTs and isoelastics
- scripts: added 'scripts/fem2rtdc.py' for generating in-silico .rtdc datasets from FEM simulation data provided by Lucas Wittwer
- fix: dclab-verify-dataset failed when the "logs" group was not present in HDF5 files
- fix: use predefined chunks when writing HDF5 data to avoid exploding file sizes when writing one event at a time
- fix: create a deep copy of the metadata dictionary when writing HDF5 data because it leaked to subsequent calls
- ref: changed the way isoelasticity lines and emodulus LUTs are stored and loaded (e.g. json metadata and a few more sanity checks)

6.178 version 0.23.0

- feat: enable emodulus extrapolation for *area_um*/*deform* values outside of the given LUT.

6.179 version 0.22.7

- enh: dclab-verify-dataset now also checks whether the sheath and sample flow rates add up to the channel flow rate
- ref: Configuration does not anymore load unknown meta data keyword arguments, but ignores them. This implies that dclab-verify-dataset will not anymore check them actively. Instead, any warning issued when opening a file is added to the list of cues.
- setup: bump nptdms to 0.23.0

6.180 version 0.22.6

- fix: data export to HDF5 did not work when the "frame rate" is not given in the configuration

6.181 version 0.22.5

- enh: add checks for valid keys in the Configuration dictionary of a dataset *RTDCBase().config*; unknown keys will issue an UnknownConfigurationKeyWarning (#58)
- ref: moved *rtdc_dataset.fmt_hdf5.UnknownKeyWarning* to *rtdc_dataset.config.UnknownConfigurationKeyWarning*
- ref: renamed *rtdc_dataset.config.CaseInsensitiveDict* to *rtdc_dataset.config.ConfigurationDict* and added option to check new keys

6.182 version 0.22.4

- fix: disable computation of Young's modulus for reservoir measurements (#75)
- enh: new keyword argument *req_func* for *AncillaryFeature* to define additional logic for checking whether a feature is available for a given instance of *RTDCBase*.

6.183 version 0.22.3

- enh: add *data* property to ICues (and use it when checking for compression)

6.184 version 0.22.2

- fix: when computing the contour from the mask image, always use the longest contour - critical when the mask image contains artefacts
- fix: minor issue with dclab-verify-dataset when nptdms was not installed and an exception occured
- enh: dclab-verify-dataset shows some info on data compression

6.185 version 0.22.1

- enh: remember working API Key
- docs: document DCOR format

6.186 version 0.22.0

- feat: implement DCOR client
- enh: improved .rtdc file format detection (with wrong extension)

6.187 version 0.21.2

- enh: dclab-verify-dataset now also checks HDF5 "mask" feature attributes
- setup: bump h5py to 2.10.0 (need *<object>.attrs.get_id*)

6.188 version 0.21.1

- fix: correct type of HDF5 image attributes for "mask" feature

6.189 version 0.21.0

- feat: implement new CLI dclab-repack
- fix: don't write "logs" group to HDF5 files if there aren't any
- fix: support HDF5 files that have no "logs" group
- docs: fix docstring of dclab-join

6.190 version 0.20.8

- fix: regression where old .tmds data could not be opened if they did not contain the "area_msd" feature
- fix: convert bytes logs to string in fmt_hdf5
- enh: support len(ds.logs) for fmt_hdf5
- enh: replace "info" by "build" in CLI job info

6.191 version 0.20.7

- fix: ensure file extension is .rtdc in dclab-join
- fix: correct "frame" and "index_online" features when exporting to hdf5
- enh: allow to set metadata dictionary in dclab.cli.join

6.192 version 0.20.6

- fix: typo in contour check resulted in small tolerance

6.193 version 0.20.5

- fix: be more trustful when it comes to contour data in the tdms file format; instead of raising errors, issue warnings (#72)

6.194 version 0.20.4

- ref: move integrity checks to new class check.IntegrityChecker
- docs: document remaining dictionaries in dclab.dfn

6.195 version 0.20.3

- docs: fix bad anchors

6.196 version 0.20.2

- ref: using temperature values outside the range for viscosity computation now issues a warning instead of raising an error; warnings were added for the CellCarrier buffers
- fix: handle number detection correctly in get_emodulus

6.197 version 0.20.1

- fix: always return an array when computing the KDE
- ref: make accessible static function RTDCBase.get_kde_spacing

6.198 version 0.20.0

- feat: compute elastic modulus from "temp" feature (#51)
- enh: computing isoelastics from datasets can use [setup]: "temperature" to compute the viscosity/emodulus (#51)
- enh: define new meta data key [setup]: "temperature"
- docs: add an advanced section on Young's modulus computation (#51)

6.199 version 0.19.1

- fix: hierarchy children did not pass *force* argument to hierarchy parent when *apply_filter* is called
- fix: revert histogram2d "density" argument to "normed" to support numpy 1.10 (Shape-Out 1)
- fix: implement unambiguous *RTDCBase.__repr__*

6.200 version 0.19.0

- feat: added better contour spacing computation based on percentiles (dclab.kde_methods.bin_width_percentile)
- feat: add feature "index_online" which may be missing values (#71)
- feat: implement __getstate__ and __setstate__ for polygon filters
- fix: write UTF-8 BOM when exporting to .tsv
- enh: add check whether unique_id exists in PolygonFilter

6.201 version 0.18.0

- fix: correctly handle filtering when features are removed from a dataset
- ref: move dclab.rtdc_dataset.util to dclab.util
- ref: minor cleanup in computation of viscosity (support lower-case *medum* values, add *dclab.features.emodulus_viscosity.KNOWN_MEDIA*)
- ref: cleanup dclab.rtdc_dataset.filter (use logical operators, correctly display nan-warning messages, keep track of polygon filters, add consistency checks, improve readability)

6.202 version 0.17.1

- maintenance release

6.203 version 0.17.0

- feat: add command line script for compressing HDF5 (.rtdc) data "dclab-compress"
- enh: record warnings under "/log" for all command line scripts
- enh: set gzip data compression for all command line scripts

6.204 version 0.16.1

- fix: circumvent UnicodeDecodeErrors which occured in frozen macOS binaries that use dclab
- enh: support subsecond accuracy in the the configuration key [experiment] time (e.g. "HH:MM:SS.SSS" instead of "HH:MM:SS")
- enh: store the correct, relative measurement time in dclab-join (#63)

6.205 version 0.16.0

- fix: RTDCBase.downsample_scatter with ret_mask=True did not return boolean array of len(RTDCBase) as indicated in the docs
- ref: remove RTDCBase._plot_filter, which was confusing anyway
- ref: deprecate usage of RTDCBase._filter

6.206 version 0.15.0

- feat: add method RTDCBase.reset_filter
- feat: implement RTDCBase.features_loaded
- feat: allow to instantiate RTDC_Hierarchy without initially applying the filter
- fix: non-scalar columns of RTDC_Hierarchy did not implement len()
- docs: add an example script dedicated to data plotting
- ref: remove circular references between Filter and RTDCBase

6.207 version 0.14.8

- fix: Ignore feature "trace" when the trace folder exists but is empty (HDF5 format)
- fix: If no contour can be found, raise an error before other ancillary features produce cryptic errors

6.208 version 0.14.7

- enh: allow to add meta data when exporting to .fcs or .tsv (dclab version is saved by default)
- setup: bump fcswrite from 0.4.1 to 0.5.0

6.209 version 0.14.6

- fix: improved handling of tdms trace data (split trace with fixed samples per event to avoid ValueError when exporting to hdf5)
- fix: transposed roi size x/y config value when exporting to hdf5

6.210 version 0.14.5

- cli: write warning messages to logs in tdms2rtdc
- ref: increase verbosity of warning messages

6.211 version 0.14.4

- fix: discard trace data when "samples per event" has multiple values for tdms data
- fix: prefer image shape over config keywords when determining the shape of the event mask and check the shape in dclab-verify-dataset
- fix: avoid ContourIndexingError by also searching the neighboring (+2/-2) events when the contour frame number does not match (#67)

6.212 version 0.14.3

- enh: explicitly check contour data when testing whether to include the first event in tdms2rtdc

6.213 version 0.14.2

- ref: convert said ValueError to ContourIndexingError

6.214 version 0.14.1

- fix: ValueError when verifying contour frame index due to comparison of float with int

6.215 version 0.14.0

- feat: new command line script for creating a scalar-feature-only dataset with all available ancillary features "dclab-condense"
- enh: enable scalar feature compression for hdf5 export
- docs: fix doc string for dclab-tdms2rtdc (*–include-initial-empty-image* falsely shown as "enabled by default")

6.216 version 0.13.0

- feat: allow to obtain a mask representing the filtered data with the *ret_mask* kwarg in *RTDCBase.get_downsampled_scatter*
- feat: allow to force-exclude invalid (inf/nan) events when downsampling using the *remove_invalid* keyword argument
- feat: exclude empty initial images in dclab-tdms2rtdc; they may optionally be included with "–include-initial-empty-image"
- feat: new property *RTDCBase.features_innate* (measuerd feature)
- enh: log which ancillary features were computed in dclab-tdms2rtdc (#65)
- enh: improved tdms meta data import (also affects dclab-tdms2rtdc)
- enh: update channel count and samples per event when writing hdf5 data
- enh: dclab-verify-dataset now recognizes invalid tdms data
- enh: several other improvements when reading tdms data
- enh: group meta data in log files (dclab-tdms2rtdc and dclab-join)
- fix: correctly handle hdf5 export when the image or contour columns have incorrect sizes (affects dclab-tdms2rtdc)
- fix: ignore empty configuration values when loading tdms data
- fix: image/contour files were searched recursively instead of only in the directory of the tdms file
- fix: check for precence of "time" feature before using it to correct measurement date and time
- fix: ancillary feature computation for brightness had wrong dependency coded (contour instead of mask)
- fix: ancillary feature computation when contour data is involved lead to error, because *LazyContourList* did not implement *identifier* (see #61)
- ref: remove NoContourDataWarning for tdms file format
- tests: improve dataset checks (#64)

6.217 version 0.12.0

- feat: add command line script for joining measurements "dclab-join" (#57)
- feat: make log files available as *RTDCBase.logs*
- feat: include log data in "dclab-join" and "dclab-tdms2rtdc"
- fix: *features* property for tdms file format falsely contains the keys "contour", "image", "mask", and "trace" when they are actually not available.
- enh: support loading TDMS data using the 'with' statement
- docs: add example for joining measurements
- docs: other minor improvements
- setup: add Python 3.7 wheels for Windows (#62)
- setup: remove Python 2 wheels for macOS

6.218 version 0.11.1

- docs: add example for fluorescence trace visualization
- docs: restructure advanced usage section
- ref: make dclab in principle compatible with imageio>=2.5.0; Dependencies are pinned due to segfaults during testing
- setup: make tdms format support and data export dependency optional; To get the previous behavior, use *pip install dclab[all]*

6.219 version 0.11.0

- feat: compute contours lazily (#61)

6.220 version 0.10.5

- setup: migrate to PEP 517 (pyproject.toml)

6.221 version 0.10.4

- enh: ignore defective feature "aspect" from Shape-In 2.0.6 and 2.0.7
- enh: support loading HDF5 data using the 'with' statement (e.g. *with dclab.new_dataset(rtdc_path) as ds:*)

6.222 version 0.10.3

- fix: add numpy build dependency (setup_requires)

6.223 version 0.10.2

- fix: HDF5-export did not re-enumerate "index" feature

6.224 version 0.10.1

- fix: support nan-valued events when computing quantile levels in submodule *kde_contours*

6.225 version 0.10.0

- BREAKING CHANGE: Change np.meshgrid indexing in *RTDCBase.get_kde_contour* from "xy" to "ij"
- feat: new submodule *kde_contours* for computing kernel density contour lines at specific data percentiles (#60)
- fix: range for contour KDE computation did not always contain end value (*RTDCBase.get_kde_contour*)
- fix: *positions* keyword argument in *RTDCBase.get_kde_scatter* was not correctly scaled in the logarithmic case
- ref: cleanup and document PolygonFilter.point_in_poly
- ref: move skimage code to separate submodule "external"
- ref: drop dependency on statsmodels and move relevant code to submodule "external"

6.226 version 0.9.1

- fix: all-zero features were treated as non-existent due to relic from pre-0.3.3 era
- fix: correct extraction of start time from tdms format (1h offset from local time and measurement duration offset)
- fix: correct extraction of module composition from tdms format (replace "+" with ",")
- enh: add configuration key mapping for tdms format to simplify conversion to hdf5 format (see `fmt_tdms.naming`)
- enh: do not add laser info for unused lasers for tdms format
- enh: dclab-verify-dataset checks for image attribute dtype
- enh: include original software version when exporting to rtdc format

6.227 version 0.9.0

- feat: add new feature: gravitational force, temperature, and ambient temperature
- ref: remove unused *has_key* function in *rtdc_dataset.config.CaseInsensitiveDict*
- setup: require numpy>=1.10.0 because of *equal_nan* argument in *allclose*

6.228 version 0.8.0

- fix: usage of "xor" (^) instead of "or" (|) in statistics
- feat: support *remove_invalid=False* in downsampling.downsample_rand (#27)
- feat: add keyword arguments *xscale* and *yscale* to improve data visualization in *RTDCBase.get_downsampled_scatter*, *RTDCBase.get_kde_contour*, and *RTDCBase.get_kde_scatter* (#55)
- enh: make downsampling code more transparent
- BREAKING CHANGE: low-level downsampling methods refactored
  - downsampling.downsample_grid: removed keyword argument *remove_invalid*, because setting it to *False* makes no sense in this context

  - downsampling.downsample_rand: changed default value of *remove_invalid* to *False*, because this is more objective
  - rename keyword argument *retidx* to *ret_idx*
  - these changes do not affect any other higher level functionalities in *dclab.rtdc_dataset* or in Shape-Out

6.229 version 0.7.0

- feat: add new ancillary feature: principal inertia ratio (#46)
- feat: add new ancillary feature: absolute tilt (#53)
- feat: add computation of viscosity for water (#52)

6.230 version 0.6.3

- fix: channel width not correctly identified for old tdms files

6.231 version 0.6.2

- ci: automate release to PyPI with appveyor and travis-ci

6.232 version 0.6.0

- fix: image export as .avi did not have option to use unfiltered data
- fix: avoid a few unicode gotchas
- feat: use Doane's formula for kernel density estimator defaults (#42)
- docs: usage examples, advanced scripting, and code reference update (#49)

6.233 version 0.5.2

- Migrate from os.path to pathlib (#50)
- fmt_hdf5: Add run index to title

6.234 version 0.5.1

- Setup: add dependencies for statsmodels
- Tests: filter known warnings
- fmt_hdf5: import unknown keys such that "dclab-verify-dataset" can complain about them

6.235 version 0.5.0

- BREAKING CHANGES:
  - definitions.feature_names now contains non-scalar features (inlcuding “image”, “contour”, “mask”, and “trace”). To test for scalar features, use definitions.scalar_feature_names.
  - features bright_* are computed from mask instead of from contour
- Bugfixes:
  - write correct event count in exported hdf5 data files
  - improve implementation of video file handling in fmt_tdms
- add new non-scalar feature “mask” (#48)
- removed configuration key [online_contour]: “bin margin” (#47)
- minor improvements for the tdms file format

6.236 version 0.4.0

- Bugfix: CLI “dclab-tdms2rtdc” did not work for single tdms files (#45)
- update configuration keys:
  - added new keys for [fluorescence]
  - added [setup]: “identifier”
  - removed [imaging]: “exposure time”, “flash current”
  - removed [setup]: “temperature”, “viscosity”
- renamed feature “ncells” to “nevents”

6.237 version 0.3.3

- ref: do not import missing features as zeros in fmt_tdms
- CLI:
  - add tdms-to-rtdc converter “dclab-tdms2rtdc” (#36)
  - improve “dclab-verify-dataset” user experience
- Bugfixes:
  - “limit events” filtering must be integer not boolean (#41)
  - Support opening tdms files with capitalized “userDef” column names
  - OSError when trying to open files from repository root

6.238 version 0.3.2

- CLI: add rudimentary dataset checker "dclab-verify-dataset" (#37)
- Add logic to compute parent/root/child event indices of RTDC_Hierarchy
  - Hierarchy children now support contour, image, and traces
  - Hierarchy children now support and remember manual filters (#22)
- Update emodulus look-up table with larger values for deformation
- Implement pixel size correction for emodulus computation
- Allow to add pixelation error to isoelastics (*add_px_err=True*) (#28)
- Bugfixes:
  - Pixel size not read from tdms-based measurements
  - Young's modulus computation wrong due to faulty FEM simulations (#39)

6.239 version 0.3.1

- Remove all-zero dummy columns from dict format
- Implement hdf5-based RT-DC data reader (#32)
- Implement hdf5-based RT-DC data writer (#33)
- Bugfixes:
  - Automatically fix inverted box filters
  - RTDC_TDMS trace data contained empty arrays when no trace data was present (trace key should not have been accessible)
  - Not possible to get isoelastics for circularity

6.240 version 0.3.0

- New fluorescence crosstalk correction feature recipe (#35)
- New ancillary features "fl1_max_ctc", "fl2_max_ctc", "fl3_max_ctc" (#35)
- Add priority for multiple ancillary features with same name
- Bugfixes:
  - Configuration key values were not hashed for ancillary features
- Code cleanup:
  - Refactoring: Put ancillary columns into a new folder module
  - Refactoring: Use the term "feature" consistently
  - Unify trace handling in dclab (#30)
  - Add functions to convert input config data

6.241 version 0.2.9

- Bugfixes:
  - Regression when loading configuration strings containing quotes
  - Parameters missing when loading ShapeIn 2.0.1 tdms data

6.242 version 0.2.8

- Refactor configuration class to support new format (#26)

6.243 version 0.2.7

- New submodule and classes for managing isoelastics
- New ancillary columns “inert_ratio_raw” and “inert_ratio_cvx”
- Bugfixes:
  - Typo when finding contour data files (tdms file format)
- Rrefactoring:
  - “features” submodule with basic methods for ancillary columns

6.244 version 0.2.6

- Return event images as gray scale (#17)
- Bugfixes:
  - Shrink ancillary column size if it exceeds dataset size
  - Generate random RTDCBase.identifier (do not use RTDCBase.hash) to fix problem with identical identifiers for hierarchy children
  - Correctly determine contour data files (tdms file format)
  - Allow contour data indices larger than uint8

6.245 version 0.2.5

- Add ancillary columns “bright_avg” and “bright_sd” (#18, #19)
- Standardize attributes of RTDCBase subclasses (#12)
- Refactoring:
  - New column names and removal of redundant column identifiers (#16)
  - Minor improvements towards PEP8 (e.g. #15)
  - New class for handling filters (#13)
- Bugfixes:

  - Hierarchy child computed all ancillary columns of parent upon checking availability of a column

6.246 version 0.2.4

- Replace OpenCV with imageio
- Add (ancilliary) computation of volume (#11)
- Add convenience methods for *Configuration*
- Refactoring (#8):
  - Separate classes for .tdms, dict-based, and hierarchy datasets
  - Introduce "_events" attribute for stored data
  - Data columns (including image, trace, contour) are accessed via keys instead of attributes.
  - Make space for new hdf5-based file format
  - Introduce ancilliary columns that are computed on-the-fly (new "_ancilliaries" attribute and "ancilliary_columns.py")

6.247 version 0.2.3

- Add look-up table for elastic modulus (#7)
- Add filtering option "remove invalid events" to remove nan/inf
- Support nan and inf in data analysis
- Improve downsampling performance
- Refactor downsampling methods (#6)

6.248 version 0.2.2

- Add new histogram-based kernel density estimator (#2)
- Refactoring:
  - Configuration fully handled by RTDC_DataSet module (#5)
  - Simplify video export function (#4)
  - Removed "Plotting" configuration key
  - Removed .cfg configuration files

6.249 version 0.2.1

- Support npTDMS 0.9.0
- Add AVI-Export function
- Add lazy submodule for event trace data and rename *RTDC_DataSet.traces* to *RTDC_DataSet.trace*
- Add "Event index" column

6.250 version 0.2.0

- Compute sensible default configuration parameters for KDE estimation and contour plotting
- Speed-up handling of contour text files
- Add support for "User Defined" column in tdms files

6.251 version 0.1.9

- Implement hierarchical instantiation of RTDC_DataSet
- Bugfix: Prevent instances of PolygonFilter that have same id
- Load InertiaRatio and InertiaRatioRaw from tdms files

6.252 version 0.1.8

- Allow to instantiate RTDC_DataSet without a tdms file
- Add statistics submodule
- Bugfixes:
  - Faulty hashing strategy in *RTDC_DataSet.GetDownSampledScatter*
- Code cleanup (renamed methods, cleaned structure)
- Corrections/additions in definitions (fRT-DC)

6.253 version 0.1.7

- Added channel: distance between to first fl. peaks
- Added fluorescence channels: peak position, peak area, number of peaks
- Allow to disable KDE computation
- Add filter array for manual (user-defined) filtering
- Add config parameters for log axis scaling
- Add channels: bounding box x- and y-size
- Bugfixes:

  - cached.py did not handle *None*
  - Limiting number of events caused integer/bool error

6.254 version 0.1.6

- Added *RTDC_DataSet.ExportTSV* for data export
- Bugfixes:
  - Correct determination of video file in RTDCDataSet
  - Fix multivariate KDE computation
  - Contour accuracy for Defo overridden by that of Circ

6.255 version 0.1.5

- Fix regressions with filtering. https://github.com/ZELLMECHANIK-DRESDEN/ShapeOut/issues/43
- Ignore empty columns in .tdms files (#1)
- Moved RTDC_DataSet and PolygonFilter classes to separate files
- Introduce more transparent caching - improves speed in some cases

6.256 version 0.1.4

- Added support for 3-channel fluorescence data (FL-1..3 max/width)

6.257 version 0.1.3

- Fixed minor polygon filter problems.
- Fix a couple of Shape-Out-related issues:
  - https://github.com/ZELLMECHANIK-DRESDEN/ShapeOut/issues/17
  - https://github.com/ZELLMECHANIK-DRESDEN/ShapeOut/issues/20
  - https://github.com/ZELLMECHANIK-DRESDEN/ShapeOut/issues/37
  - https://github.com/ZELLMECHANIK-DRESDEN/ShapeOut/issues/38

6.258 version 0.1.2

- Add support for limiting amount of data points analyzed with the configuration keyword “Limit Events”
- Comments refer to “events” instead of “points” from now on

CHAPTER

SEVEN

BILBLIOGRAPHY

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[BMBW15] D Bates, M Maechler, B Bolker, and S Walker. Lme4: linear mixed-effects models using eigen and s4. 2015. URL: https://CRAN.R-project.org/package=lme4.

[BBN+23] Beyza Büyükurgancı, Santanu Kumar Basu, Markus Neuner, Jochen Guck, Andreas Wierschem, and Felix Reichel. Shear rheology of methyl cellulose based solutions for cell mechanical measurements at high shear rates. *Soft Matter*, pages –, 2023. doi:10.1039/D2SM01515C.

[Eve92] Brian Everitt. Book reviews : chambers JM, hastie TJ eds 1992: statisti cal models in s. california: wadsworth and brooks/cole. ISBN 0 534 16765-9. *Statistical Methods in Medical Research*, 1(2):220–221, aug 1992. doi:10.1177/096228029200100208.

[GH06] Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press, 2006. doi:10.1017/CBO9780511790942.

[HMMO18] M. Herbig, A. Mietke, P. Müller, and O. Otto. Statistics for real-time deformability cytometry: Clustering, dimensionality reduction, and significance testing. *Biomicrofluidics*, 12(4):042214, 2018. doi:10.1063/1.5027197.

[Her17] Christoph Herold. Mapping of Deformation to Apparent Young's Modulus in Real-Time Deformability Cytometry. *ArXiv e-prints 1704.00572 [cond-mat.soft]*, 2017. arXiv:1704.00572v1.

[KSW78] Joseph Kestin, Mordechai Sokolov, and William A. Wakeham. Viscosity of liquid water in the range -8\hspace 0.167em°C to 150\hspace 0.167em°C. *Journal of Physical and Chemical Reference Data*, 7(3):941–948, jul 1978. doi:10.1063/1.555581.

[MOG+15] Alexander Mietke, Oliver Otto, Salvatore Girardo, Philipp Rosendahl, Anna Taubenberger, Stefan Golfier, Elke Ulbricht, Sebastian Aland, Jochen Guck, and Elisabeth Fischer-Friedrich. Extracting Cell Stiffness from Real-Time Deformability Cytometry: Theory and Experiment. *Biophysical Journal*, 109(10):2023–2036, nov 2015. doi:10.1016/j.bpj.2015.09.006.

[MMM+17] M. Mokbel, D. Mokbel, A. Mietke, N. Träber, S. Girardo, O. Otto, J. Guck, and S. Aland. Numerical Simulation of Real-Time Deformability Cytometry To Extract Cell Mechanical Properties. *ACS Biomaterials Science & Engineering*, 3(11):2962–2973, jan 2017. doi:10.1021/acsbiomaterials.6b00558.

[RB23] Felix Reichel and Beyza Büyükurgancı. Effective viscosity of methyl cellulose solutions in phosphate buffered saline in real-time deformability cytometry. *ArXiv e-prints 2302.01664 [cond-mat.soft]*, 2023. arXiv:2302.01664v3, doi:10.48550/arXiv.2302.01664.

[RHMG19] Philipp Rosendahl, Christoph Herold, Paul Müller, and Jochen Guck. Real-time deformability cytometry reference data. Feb 2019. doi:10.6084/m9.figshare.7771184.v2.

[WMM+20] Lucas D. Wittwer, Paul Müller, Dominic Mokbel, Marcel Mokbel, Jochen Guck, and Sebastian Aland. Finite element simulation data for the computation of the young's modulus in real-time deformability cytometry. Apr 2020. doi:10.6084/m9.figshare.12155064.v4.

[WMR+22] Lucas D. Wittwer, Paul Müller, Felix Reichel, Sebastian Aland, and Jochen Guck. Hyperelastic lookup table for real-time deformability cytometry (rt-dc). Auf 2022. doi:10.6084/m9.figshare.20630940.v1.

[WRM+22] Lucas D. Wittwer, Felix Reichel, Paul Müller, Jochen Guck, and Sebastian Aland. Mapping of Deformation to Apparent Young's Modulus in Real-Time Deformability Cytometry. *ArXiv e-prints 2208.12552 [q-bio.QM]*, 2022. arXiv:2208.12552v1.

[XRM+20] Miguel Xavier, Philipp Rosendahl, Paul Müller, Maik Herbig, and Jochen Guck. Real-time deformability cytometry data of primary human skeletal stem cells and the human osteosarcoma cell line mg-63. Feb 2020. doi:10.6084/m9.figshare.11662773.v2.

PYTHON MODULE INDEX

d

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W

Y