nanite Documentation

Release 3.5.4

Paul Müller **Shada Abuhattum**

Feb 20, 2023

CONTENTS

Nanite is a Python library for loading, fitting, and rating AFM force-distance data of cells and tissues. This is the documentation of nanite version 3.5.4.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

To install nanite, use one of the following methods (the package dependencies will be installed automatically):

- **from PyPI:** `pip install nanite[CLI]`
- **from sources:** `pip install -e .[CLI]`

The appendix `[CLI]` makes sure that all dependencies for the *command line interface* are installed. If you are only using nanite as a Python module, you may safely omit it.

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 nanite extensions. If this process fails, please request a binary wheel for your platform (e.g. Windows 64bit) and Python version (e.g. 3.6) by creating a new issue.

1.2 What is nanite?

The development of nanite was motivated by a unique problem that arises in AFM force-distance data analysis, particularly for biological samples: The data quality varies a lot due to biological variation and due to experimental complexities that have to be dealt with when measuring biological samples. To address this problem, nanite makes use of machine-learning (á la scikit-learn), which allows to automatically determine the quality of a force-distance curve based on a user-defined rating scheme (see *Rating workflow* for more information). But nanite is much more than just that. It comes with an extensive set of tools for AFM force-distance data analysis.

1.3 Supported file formats

Nanite relies on the afmformats package. A list of supported file formats can be found here.

1.4 Use cases

If you are a frequent AFM user, you might have run into several problems involving data analysis, ranging from simple data fitting to the visualization of quantitative force-distance maps. Here are a few usage examples of nanite:

- You would like to automate your data analysis pipeline from loading force-distance data to displaying a fit to the approach part with a Hertz model for a spherical indenter. You can do so with nanite, either via scripting or via the command-line interface that comes with nanite. For more information, see *Fitting guide*.

- You would like to automatically analyze and visualize maps of force-distance data. This is possible with the *nanite.QMap* class.
- You would like to sort force-distance data according to data quality using your own training set (not the one shipped with nanite). Nanite allows you to create your own training set from your own experimental data, locally. Besides that, you can make use of multiple regressors and visualize the rating e.g. of force-distance maps. For an overview, see *Rating workflow*.

1.5 Basic usage

If you are not interested in scripting, please have a look at the *fitting guide*.

In a Python script, you may use nanite as follows:

```
In [1]: import nanite

In [2]: group = nanite.load_group("data/force-save-example.jpk-force")

In [3]: idnt = group[0]  # This group actually as only one indentation curve.

In [4]: idnt.apply_preprocessing(["compute_tip_position",
   ...:                           "correct_force_offset",
   ...:                           "correct_tip_offset"])
   ...:

In [5]: idnt.fit_model(model_key="sneddon_spher")

In [6]: idnt.rate_quality()  # 0 means bad, 10 means good quality
Out[6]: 9.060746150910978
```

You can find more examples in the *examples* section.

1.6 How to cite

If you use nanite in a scientific publication, please cite Müller et al., *BMC Bioinformatics* (2019) [MAM+19].

CHAPTER

TWO

COMMAND-LINE INTERFACE

The nanite command-line interface (CLI) simplifies several functionalities of nanite, making fitting, rating, and the generation of training sets accessible to the user.

2.1 nanite-setup-profile

Set up a profile for fitting and rating. The profile is stored in the user's default configuration directory. Setting up a profile is required prior to running *nanite-fit* and *nanite-rate*.

```
usage: nanite-setup-profile [-h]
```

2.2 nanite-fit

Fit AFM force-distance data. Statistics (.tsv file) and visualizations of the fits (multi-page .tif file) are stored in the results directory.

```
usage: nanite-fit [-h] data_path out_dir
```

positional arguments	
data_path	input folder containing AFM force-distance data
out_dir	results directory

2.3 nanite-rate

Manually rate (the fit to) AFM force-distance data. A graphical user interface allows to rate and comment on each force-distance curve. The fits and the raw data are stored in a rating container that can then be passed to *nanite-generate-training-set*.

```
usage: nanite-rate [-h] data_path rating_path
```

positional arguments	
data_path	input folder containing AFM force-distance data
rating_path	path to the output rating container (will be created if it does not already exist)

2.4 nanite-generate-training-set

Create a training set for usage in nanite from rating containers (.h5 files manually created with *nanite-rate*).

```
usage: nanite-generate-training-set [-h] data_path out_dir
```

positional arguments	
data_path	path to a rating container or a folder containing rating containers
out_dir	directory where the training set will be stored

CHAPTER

THREE

FITTING GUIDE

This is a summary of the methods used by nanite for fitting force-distance data. Examples are given below.

3.1 Preprocessors

Prior to data analysis, a force-distance curve has to be preprocessed. One of the most important preprocessing steps is to perform a tip-sample separation which computes the correct tip position from the recorded piezo height and the cantilever deflection. Other preprocessing steps correct for offsets or smoothen the data:

preprocessor key	description	details
compute_tip_position	tip-sample separation	*code reference*
correct_force_offset	baseline correction	*code reference*
correct_tip_offset	contact point estimation	*code reference*
correct_split_approach_retract	segment discovery	*code reference*
smooth_height	monotonic height data	*code reference*

Several methods for estimating the point of contact (POC) are implemented in nanite:

POC method	description	details
deviation_from_baseline	Deviation from baseline	*code reference*
fit_constant_line	Piecewise fit with constant and line	*code reference*
fit_constant_polynomial	Piecewise fit with constant and polynomial	*code reference*
fit_line_polynomial	Piecewise fit with line and polynomial	*code reference*
frechet_direct_path	Fréchet distance to direct path	*code reference*
gradient_zero_crossing	Gradient zero-crossing of indentation part	*code reference*

3.2 Models

Nanite comes with a predefined set of model functions that are identified (in scripting as well as in the command line interface) via their model keys.

model key	description	details
hertz_cone	conical indenter (Hertz)	*code reference*
hertz_para	parabolic indenter (Hertz)	*code reference*
hertz_pyr3s	pyramidal indenter, three-sided (Hertz)	*code reference*
sneddon_spher	spherical indenter (Sneddon)	*code reference*
sneddon_spher_approx	spherical indenter (Sneddon, truncated power series)	*code reference*

These model functions can be used to fit experimental force-distance data that have been preprocessed as described above.

3.3 Parameters

Besides the modeling parameters (e.g. Young's modulus or contact point), nanite allows to define an extensive set of fitting options, that are described in more detail in *nanite.fit.IndentationFitter*.

parameter	description
model_key	Key of the model function used
optimal_fit_edelta	Plateau search for Young's modulus
optimal_fit_num_samples	Number of points for plateau search
params_initial	Initial parameters
preprocessing	List of preprocessor keys
range_type	'absolute' for static range, 'relative cp' for dynamic range
range_x	Fitting range (min/max)
segment	Which segment to fit ('approach' or 'retract')
weight_cp	Suppression of residuals near contact point
x_axis	X-data used for fitting (defaults to 'top position')
y_axis	Y-data used for fitting (defaults to 'force')
method	Minimizer method for lmfit.minimize
method_kws	Additional arguments (*fit_kws*) for the underlying scipy minimizer function

3.4 Geometrical correction factor

The basic models implemented in nanite are all *single-contact* models, which means that they assume there is *only one* indentation taking place during a measurement. In an AFM experiment, this holds true for e.g. measuring a flat hydrogel with a spherical AFM tip. However, many experiments require a *two-contact* model. A prominent example is the indentation of an elastic sphere between two parallel plates (e.g. a round cell on a glass cover slip indented by a wedged cantilever). Here, the top *and* bottom indentation of the sphere contribute to the overall indentation. However, the forces required to indent either side of the sphere are identical to the force in the *single-contact* version of the problem (where the elastic sphere *is* the cantilever). For instance, you need twice the force to squeeze a ball between your hands compared to when you squeeze it against a wall, but the overall indentation stays the same (Newton's third law). Thus, when you use a *single-contact* model in a *two-contact* problem, you have to be aware of the fact that the actual indentation may be larger. For the simple example of parallel-plate compression, the actual indentation is doubled. Thus, to be able to apply the single-contact model fit, we have to multiply the measured indentation by a factor of $k = 0.5$.

Let's take a look at the more general geometric problem (still neglecting adhesion forces and gravity). Let's assume whe have three spheres with Young's modul E_1, E_2, E_3 and radii R_1, R_2, R_3 (see figure Fig. 3.1). This is a two-contact problem. For each of the contact areas we can write down the *Hertz model* for the single-contact problem. The overall indentation is $\delta = \delta_{12} + \delta_{23}$

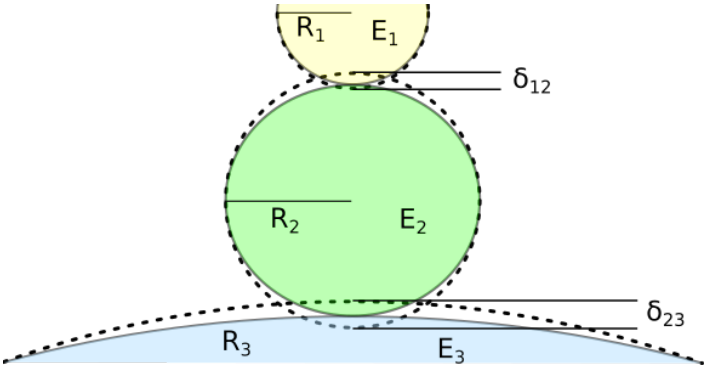


Fig. 3.1: Two-contact geometry for three elastic spheres.

with

$$\delta_{12} = \left(\frac{3F}{4E_{12}} \cdot \frac{1}{\sqrt{R_{12}}} \right)^{2/3}$$
$$\delta_{23} = \left(\frac{3F}{4E_{23}} \cdot \frac{1}{\sqrt{R_{23}}} \right)^{2/3}$$

and

$$\frac{1}{R_{ij}} = \frac{1}{R_i} + \frac{1}{R_j}$$
$$\frac{1}{E_{ij}} = \frac{1-\nu_i^2}{E_i} + \frac{1-\nu_j^2}{E_j}.$$

From here, we can start simplifying. Let's say the indenter and the substrate are comparatively stiff ($E_1 = E_3 >> E_2$) and the substrate is flat ($R_3 \to \inf$). Then we get

$$\delta_{12} = \left(\frac{3F(1-\nu_2^2)}{4E_2} \cdot \frac{1}{\sqrt{R_{12}}} \right)^{2/3}$$
$$\text{and } \delta_{23} = \left(\frac{3F(1-\nu_2^2)}{4E_2} \cdot \frac{1}{\sqrt{R_2}} \right)^{2/3}.$$

Thus, the overall indentation becomes

$$\delta = \left(\frac{3F(1-\nu_2^2)}{4E_2} \right)^{2/3} \left(\frac{1}{R_{12}^{1/3}} + \frac{1}{R_2^{1/3}} \right).$$

Finally, we arrive at

$$\delta = \left(\frac{3F(1-\nu_2^2)}{4E_2} \frac{1}{\sqrt{R_{12}}} \right)^{2/3} \cdot \frac{1}{k}$$
$$\text{with } k = \frac{R_2^{1/3}}{R_2^{1/3} + R_{12}^{1/3}}.$$

The parameter k is the geometrical correction factor. For an indenter with $R_1 = 2.5\,\mu\text{m}$ and a cell with $R_2 = 7.5\,\mu\text{m}$, the geometrical correction factor computes to $k = 0.6135$. Note that during fitting with the single-contact model, you now have to set the radius to the effective radius $R_{12} = 1.875\,\mu\text{m}$.

For a more general description of this problem, please have a look at [GMP+14].

3.5 Workflow

There are two ways to fit force-distance curves with nanite: via the *command line interface (CLI)* or via Python scripting. The CLI does not require programming knowledge while Python-scripting allows fine-tuning and straight-forward automation.

3.5.1 Command-line usage

First, set up a fitting profile by running (e.g. in a command prompt on Windows).

```
nanite-setup-profile
```

This program will ask you to specify preprocessors, model parameters, and other fitting parameters. Simply enter the values via the keyboard and hit enter to let them be acknowledged. If you want to use the default values, simply hit enter without typing anything. A typical output will look like this:

```
Define preprocessing:
  1: compute_tip_position
  2: correct_force_offset
  3: correct_split_approach_retract
  4: correct_tip_offset
  5: smooth_height
(currently '1,2,4'):

Select model number:
  1: hertz_cone
  2: hertz_para
  3: hertz_pyr3s
  4: sneddon_spher
  5: sneddon_spher_approx
(currently '5'):

Set fit parameters:
- initial value for E [Pa] (currently '3000.0'): 50
  vary E (currently 'True'):
- initial value for R [m] (currently '1e-5'): 18.64e-06
  vary R (currently 'False'):
- initial value for nu (currently '0.5'):
  vary nu (currently 'False'):
- initial value for contact_point [m] (currently '0.0'):
  vary contact_point (currently 'True'):
- initial value for baseline [N] (currently '0.0'):
  vary baseline (currently 'False'):

Select range type (absolute or relative):
(currently 'absolute'):

Select fitting interval:
left [µm] (currently '0.0'):
right [µm] (currently '0.0'):

Suppress residuals near contact point:
size [µm] (currently '0.5'): 2

Select training set:
training set (path or name) (currently 'zef18'):

Select rating regressor:
  1: AdaBoost
  2: Decision Tree
```

(continues on next page)

(continued from previous page)

```
  3: Extra Trees
  4: Gradient Tree Boosting
  5: Random Forest
  6: SVR (RBF kernel)
  7: SVR (linear kernel)
(currently '3'):

Done. You may edit all parameters in '/home/user/.config/nanite/cli_profile.cfg'.
```

In this example, the only modifications of the default values are the initial value of the Young's modulus (50 Pa), the value for the tip radius (18.64 µm), and the suppression of residuals near the contact point with a ±2 µm interval. When `nanite-setup-profile` is run again, it will use the values from the previous run as default values. The training set and rating regressor options are discussed in the *rating workflow*.

Finally, to perform the actual fitting, use the command-line script

```
nanite-fit data_path output_path
```

This command will recursively search the input folder `data_path` for data files, fit the data with the parameters in the profile, and write the statistics (*statistics.tsv*) and visualizations of the fits (multi-page TIFF file *plots.tif*, open with Fiji or the Windows Photo Viewer) to the directory `output_path`.

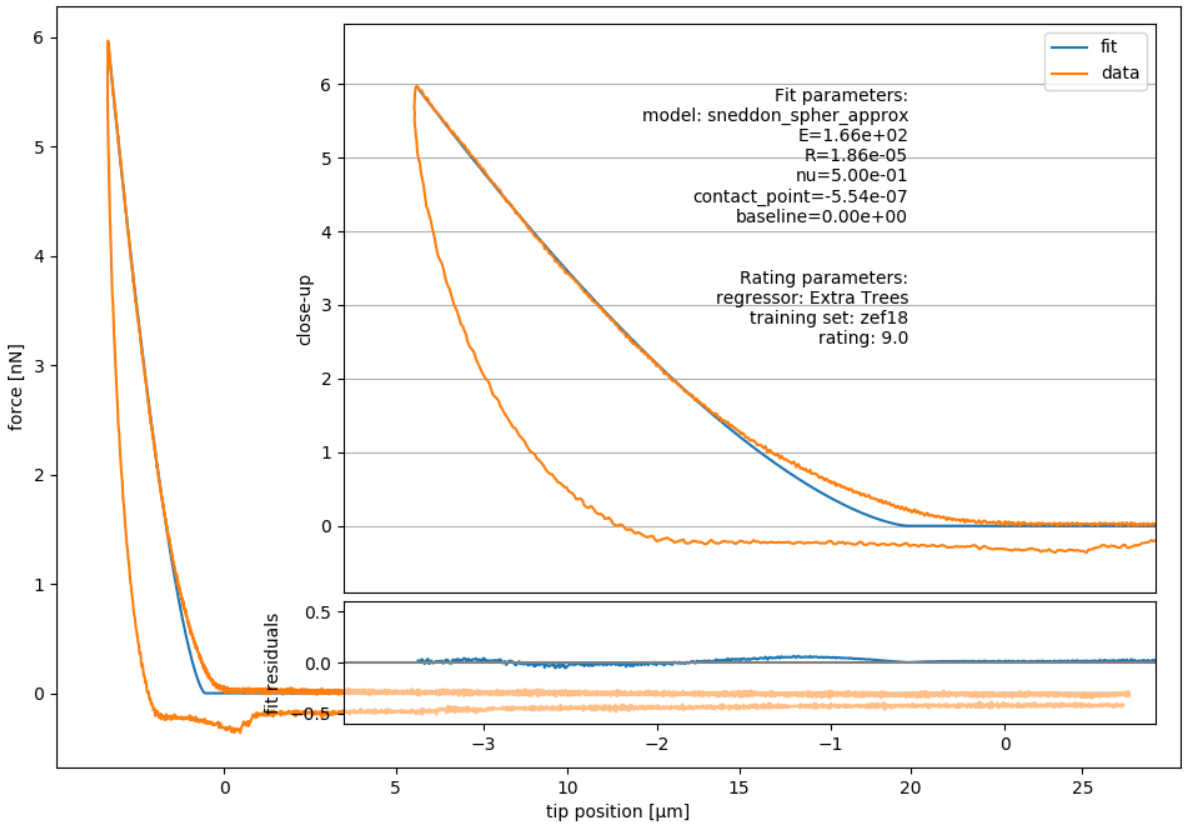


Fig. 3.2: Example image generated with `nanite-fit`. Note that the dataset is already rated with the default method "Extra Trees" and the default training set label "zef18". See *Rating workflow* for more information on rating.

3.5.2 Scripting usage

Using nanite in a Python script for data fitting is straight forward. First, load the data; `group` is an instance of *nanite.IndentationGroup*:

```
In [1]: import nanite

In [2]: group = nanite.load_group("data/force-save-example.jpk-force")
```

Second, obtain the first *nanite.Indentation* instance and apply the preprocessing:

```
In [3]: idnt = group[0]

In [4]: idnt.apply_preprocessing(["compute_tip_position",
   ...:                           "correct_force_offset",
   ...:                           "correct_tip_offset"])
   ...:
```

Now, setup the model parameters:

```
In [5]: idnt.fit_properties["model_key"] = "sneddon_spher"

In [6]: params = idnt.get_initial_fit_parameters()

In [7]: params["E"].value = 50

In [8]: params["R"].value = 18.64e-06

In [9]: params.pretty_print()
Name              Value      Min      Max   Stderr     Vary     Expr Brute_Step
E                    50        0      inf     None     True     None     None
R              1.864e-05        0      inf     None    False     None     None
baseline               0     -inf      inf     None     True     None     None
contact_point          0     -inf      inf     None     True     None     None
nu                   0.5        0      0.5     None    False     None     None
```

Finally, fit the model:

```
In [10]: idnt.fit_model(model_key="sneddon_spher", params_initial=params, weight_cp=2e-6)

In [11]: idnt.fit_properties["params_fitted"].pretty_print()
Name              Value      Min      Max   Stderr     Vary     Expr Brute_Step
E                 165.8        0      inf   0.1802     True     None     None
R              1.864e-05        0      inf        0    False     None     None
baseline       -6.083e-13     -inf      inf 2.318e-13     True     None     None
contact_point  -5.54e-07     -inf      inf 1.621e-09     True     None     None
nu                   0.5        0      0.5        0    False     None     None
```

The fitting results are identical to those shown in figure 3.2 above.

Note that, amongst other things, preprocessing can also be specified directly in the *fit_model* function.

CHAPTER

FOUR

RATING WORKFLOW

One of the main aims of nanite is to simplify data analysis by sorting out bad curves automatically based on a user defined rating scheme. Nanite allows to automate the rating process using machine learning, based on scikit-learn. In short, an estimator is trained with a sample dataset that was manually rated by a user. This estimator is then applied to new data and, in an optimal scenario, reproduces the rating scheme that the user intended when he rated the training dataset. For a more detailed analysis, please refer to [MAM+19].

Nanite already comes with a default training set that is based on AFM data recorded for zebrafish spinal cord sections, called *zef18*. The original zef18 dataset is available online [MMG18]. Download links:[1]

- https://ndownloader.figshare.com/files/13481393
- https://zenodo.org/record/1551200/files/zef18.h5
- https://b2share.eudat.eu/api/files/bf481c9b-14ff-47b1-baf5-e569d0199be6/zef18.h5

With nanite, you can also create your own training set. The required steps to do so are described in the following.

4.1 Rating experimental data manually

In the rating step, experimental data are fitted and manually rated by the user. The raw data, the preprocessed data, the fit, all parameters, and the manual rating are then stored in a rating container (an HDF5 file).

First, set up a fitting profile using *nanite-setup-profile* if you have not already done so in the *fitting guide*. You can run the command `nanite-setup-profile` again to verify that all settings are correct.

To start manual rating, use the command *nanite-rate*. The first argument is a folder containing experimental force-distance curves and the second argument is a path to a rating container (`nameXY.h5`). If the rating container already exists, new data will be appended (nothing is overridden).

```
nanite-rate path/to/data/directory path/to/nameXY.h5
```

This will open a graphical user interface that displays the preprocessed and fitted experimental data:

For the subsequent steps, it is irrelevant whether you create many small rating containers or one global rating container. Many small containers have the advantage that the effect of individual rating sessions could be analyzed separately, while a global rating container keeps all data in one place.

[1] The SHA256 checksum of *zef18.h5* is 63d89a8aa911a255fb4597b2c1801e30ea14810feef1bb42c11ef10f02a1d055.

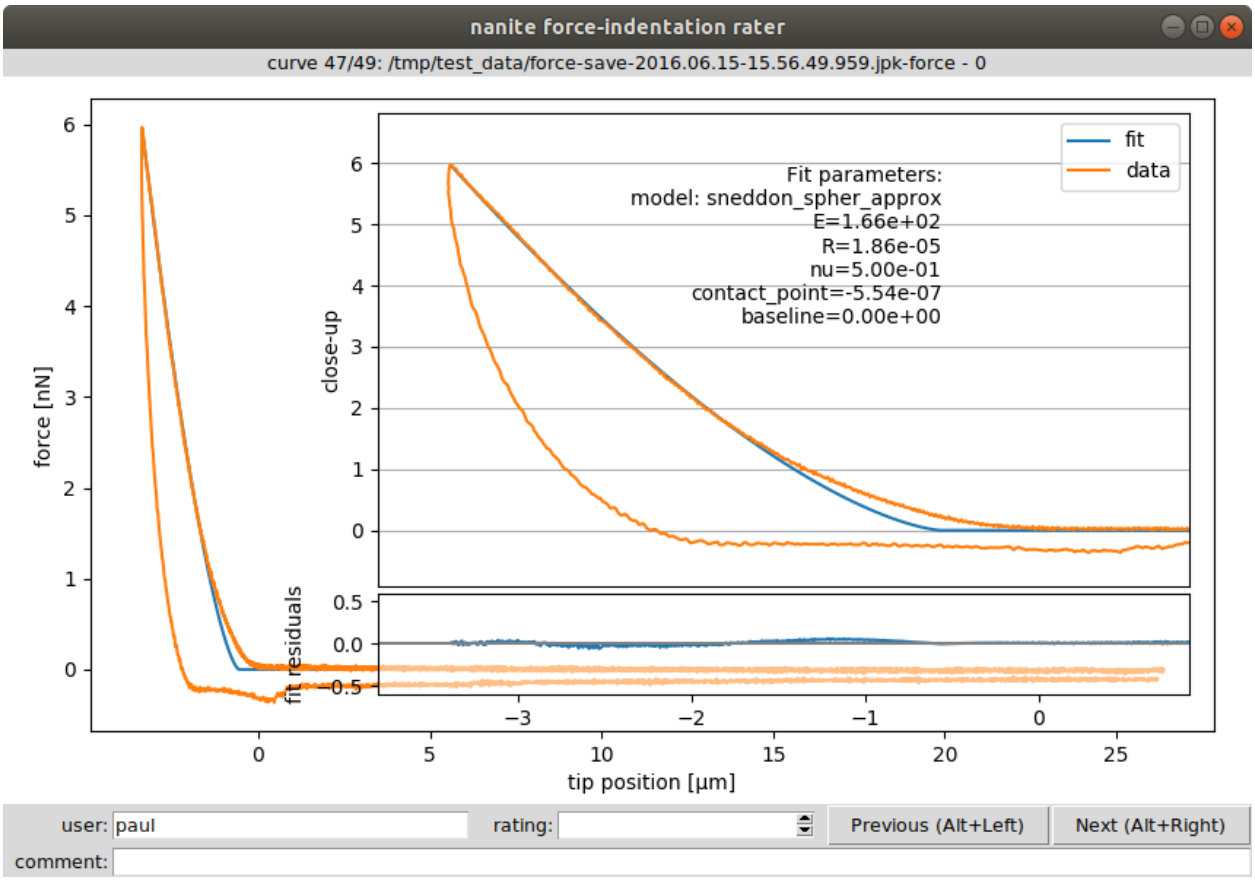


Fig. 4.1: Graphical user interface (GUI) for rating. The inset shows a close-up of the indentation part and the fitted parameters. The user name (defaults to login name) is used to assign a rating to a user (not mandatory). The rating (integer from 0/bad to 10/good or -1/invalid) and a comment can be defined for each curve. The shortcuts `ALT+Left` and `ALT+Right` can be used to navigate within the dataset while keeping the cursor focused in the *rating* field. While navigating, the data are stored in the rating container and the GUI can be closed without data loss.

4.2 Generating the training set

The training set consists only of the samples (features of each force-distance curve) and the manual ratings. It is stored as a set of small text files on disk. As described earlier, nanite comes with the predefined *zef18* training set. In this step, a user-defined training set will be generated for use with nanite.

Use the command *nanite-generate-trainining-set* to convert the rating container(s) to a training set:

```
nanite-generate-trainining-set path/to/nameXY.h5 path/to/training_set/
```

This will create the folder `path/to/training_set/ts_nameXY` containing several text files, one for each feature and one for the manual rating.

4.3 Applying the training set

To apply the training set when rating curves with `nanite-fit`, you will have to update the profile using `nanite-setup-profile` again (see *fitting guide*). The relevant program output will look like this:

```
[...]

Select training set:
training set (path or name) (currently 'zef18'): path/to/training_set/ts_nameXY

Select rating regressor:
  1: AdaBoost
  2: Decision Tree
  3: Extra Trees
  4: Gradient Tree Boosting
  5: Random Forest
  6: SVR (RBF kernel)
  7: SVR (linear kernel)
(currently '3'):

Done. You may edit all parameters in '/home/user/.config/nanite/cli_profile.cfg'.
```

When running `nanite-fit data_path output_path` now, the new training set is used for rating. The new ratings are stored in `output_path/statistics.tsv` and can be used for further analysis, e.g. quality assessment or sorting.

If you would like to employ a user-defined training set in a Python script, you may do so by specifying the training set path as an argument to *nanite.Indentation.rate_quality*.

CHAPTER

FIVE

SCRIPTING EXAMPLES

5.1 Approximating the Hertzian model with a spherical indenter

There is no closed form for the Hertzian model with a spherical indenter. The force F does not directly depend on the indentation depth δ, but has an indirect dependency via the radius of the circular contact area between indenter and sample a [Sne65]:

$$F = \frac{E}{1-\nu^2}\left(\frac{R^2+a^2}{2}\ln\left(\frac{R+a}{R-a}\right) - aR\right)$$
$$\delta = \frac{a}{2}\ln\left(\frac{R+a}{R-a}\right)$$

Here, E is the Young's modulus, R is the radius of the indenter, and ν is the Poisson's ratio of the probed material.

Because of this indirect dependency, fitting this model to experimental data can be time-consuming. Therefore, it is beneficial to approximate this model with a polynomial function around small values of δ/R using the Hertz model for a parabolic indenter as a starting point [Dob18]:

$$F = \frac{4}{3}\frac{E}{1-\nu^2}\sqrt{R}\delta^{3/2}\left(1 - \frac{1}{10}\frac{\delta}{R} - \frac{1}{840}\left(\frac{\delta}{R}\right)^2 + \frac{11}{15120}\left(\frac{\delta}{R}\right)^3 + \frac{1357}{6652800}\left(\frac{\delta}{R}\right)^4\right)$$

This example illustrates the error made with this approach. In nanite, the model for a spherical indenter has the identifier *"sneddon_spher"* and the approximate model has the identifier *"sneddon_spher_approx"*.

The plot shows the error for the parabolic indenter model *"hertz_para"* and for the approximation to the spherical indenter model. The maximum indentation depth is set to R. The error made by the approximation of the spherical indenter is more than four magnitudes lower than the maximum force during indentation.

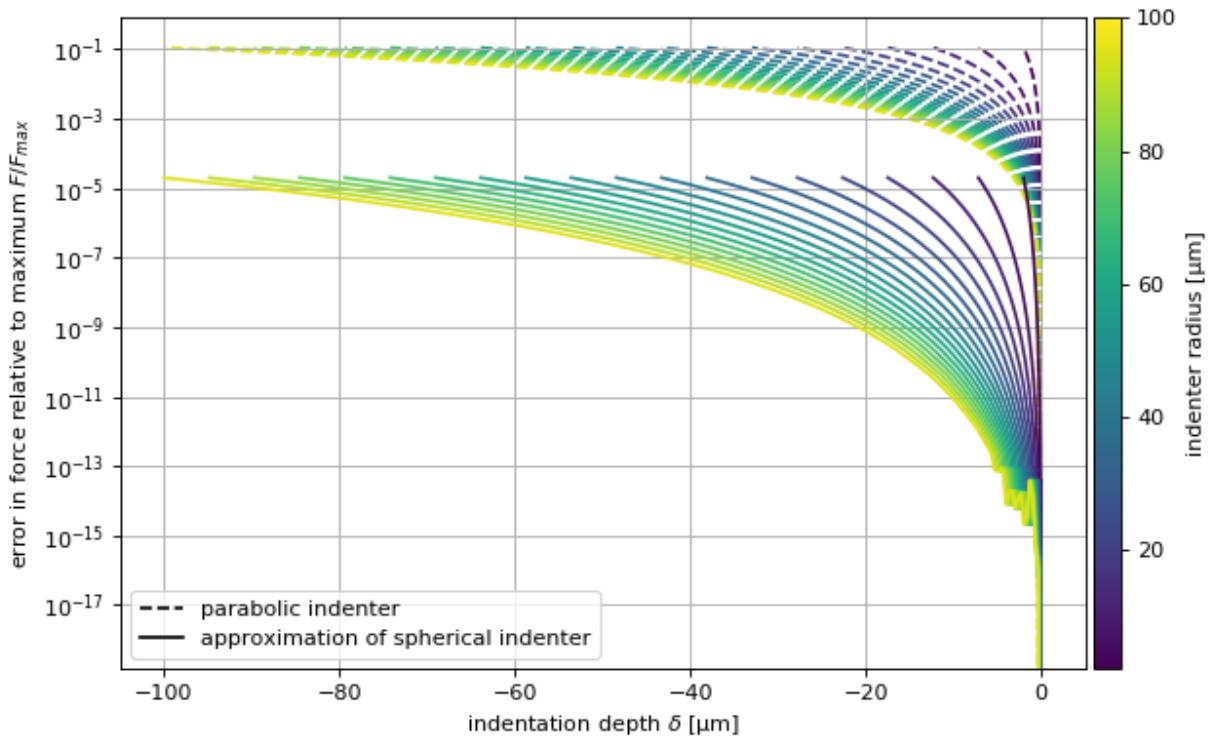
model_spherical_indenter.py

```
import matplotlib.pylab as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.lines import Line2D
import matplotlib as mpl
import numpy as np

from nanite.model import models_available

# models
exact = models_available["sneddon_spher"]
approx = models_available["sneddon_spher_approx"]
para = models_available["hertz_para"]
```

(continues on next page)



(continued from previous page)

```
# parameters
params = exact.get_parameter_defaults()
params["E"].value = 1000

# radii
radii = np.linspace(2e-6, 100e-6, 20)

# plot results
plt.figure(figsize=(8, 5))

# overview plot
ax = plt.subplot()
for ii, rad in enumerate(radii):
    params["R"].value = rad
    # indentation range
    x = np.linspace(0, -rad, 300)
    yex = exact.model(params, x)
    yap = approx.model(params, x)
    ypa = para.model(params, x)
    ax.plot(x*1e6, np.abs(yex - yap)/yex.max(),
            color=mpl.cm.get_cmap("viridis")(ii/radii.size),
            zorder=2)
    ax.plot(x*1e6, np.abs(yex - ypa)/yex.max(), ls="--",
            color=mpl.cm.get_cmap("viridis")(ii/radii.size),
            zorder=1)

```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r"indentation depth $\delta$ [µm]")
ax.set_ylabel("error in force relative to maximum $F/F_{max}$")
ax.set_yscale("log")
ax.grid()

# legend
custom_lines = [Line2D([0], [0], color="k", ls="--"),
                Line2D([0], [0], color="k", ls="-"),
                ]
ax.legend(custom_lines, ['parabolic indenter',
                         'approximation of spherical indenter'])

divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="3%", pad=0.05)

norm = mpl.colors.Normalize(vmin=radii[0]*1e6, vmax=radii[-1]*1e6)
mpl.colorbar.ColorbarBase(ax=cax,
                          cmap=mpl.cm.viridis,
                          norm=norm,
                          orientation='vertical',
                          label="indenter radius [µm]"
                          )

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

5.2 Fitting and rating

This example uses a force-distance curve of a zebrafish spinal cord section to illustrate basic data fitting and rating with nanite. The dataset is part of a study on spinal cord stiffness in zebrafish [MKH+19].

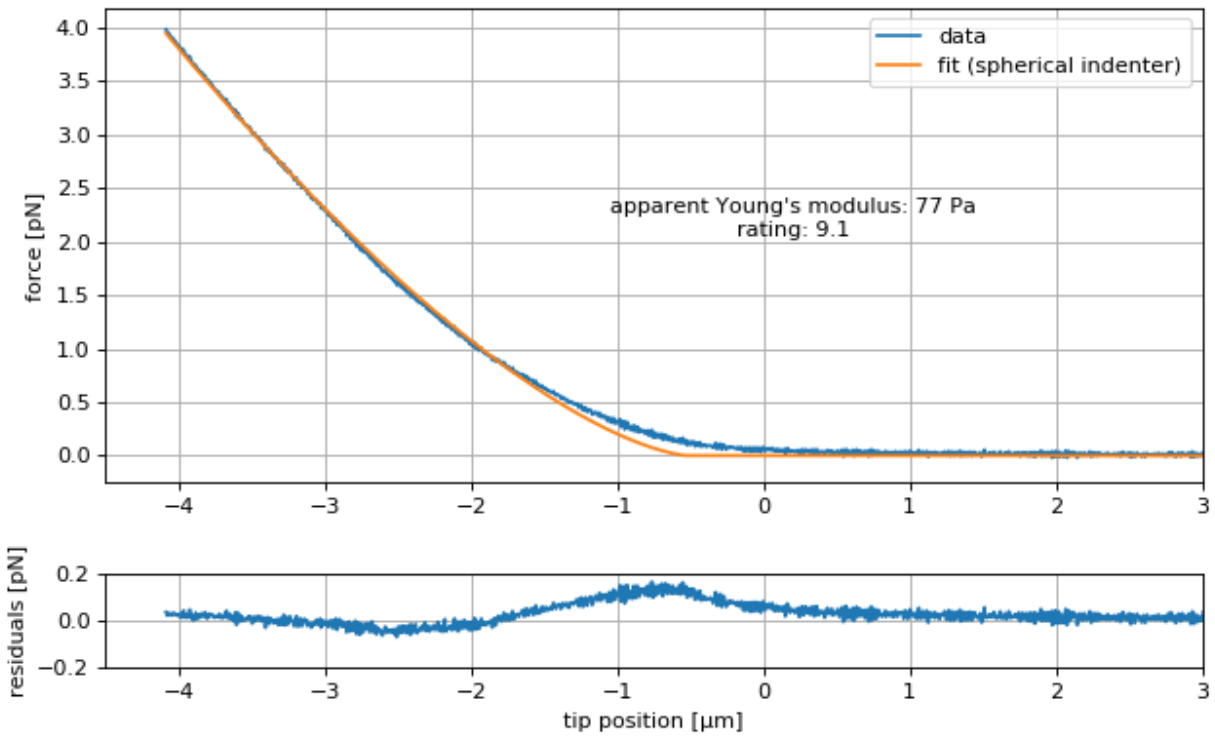
fit_and_rate.py

```
import matplotlib.gridspec as gridspec
import matplotlib.pylab as plt

import nanite

# load the data
group = nanite.load_group("data/zebrafish-head-section-gray-matter.jpk-force")
idnt = group[0]  # this is an instance of `nanite.Indentation`
# apply preprocessing
idnt.apply_preprocessing(["compute_tip_position",
                          "correct_force_offset",
                          "correct_tip_offset"])
# set the fit model ("sneddon_spher_approx" is faster than "sneddon_spher"
# and sufficiently accurate)
idnt.fit_properties["model_key"] = "sneddon_spher_approx"
# get the initial fit parameters
params = idnt.get_initial_fit_parameters()
```

(continues on next page)



(continued from previous page)

```
# set the correct indenter radius
params["R"].value = 18.64e-06
# perform the fit with the edited parameters
idnt.fit_model(params_initial=params)
# obtain the Young's modulus
emod = idnt.fit_properties["params_fitted"]["E"].value
# obtain a rating for the dataset
# (using default regressor and training set)
rate = idnt.rate_quality()

# overview plot
plt.figure(figsize=(8, 5))
gs = gridspec.GridSpec(2, 1, height_ratios=[5, 1])

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])

# only plot the approach part (`1` would be retract)
where_approach = idnt["segment"] == 0

# plot force-distance data (nanite uses SI units)
ax1.plot(idnt["tip position"][where_approach] * 1e6,
         idnt["force"][where_approach] * 1e9,
         label="data")
ax1.plot(idnt["tip position"][where_approach] * 1e6,
         idnt["fit"][where_approach] * 1e9,
```

(continues on next page)

(continued from previous page)

```
         label="fit (spherical indenter)")
ax1.text(.2, 2.05,
         "apparent Young's modulus: {:.0f} Pa\n".format(emod)
         + "rating: {:.1f}".format(rate),
         ha="center")
ax1.legend()
# plot resiudals
ax2.plot(idnt["tip position"][where_approach] * 1e6,
         (idnt["force"] - idnt["fit"])[where_approach] * 1e9)

# update plot parameters
ax1.set_xlim(-4.5, 3)
ax1.set_ylabel("force [pN]")
ax1.grid()
ax2.set_xlim(-4.5, 3)
ax2.set_ylim(-.2, .2)
ax2.set_ylabel("residuals [pN]")
ax2.set_xlabel("tip position [µm]")
ax2.grid()

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

CHAPTER

SIX

DEVELOPER GUIDE

6.1 How to contribute

Contributions via pull requests are very welcome. Just fork the "master" branch, make your changes, and create a pull request back to "master" with a descriptive title and an explanation of what you have done. If you decide to contribute code, please

1. properly document your code (in-line comments as well as doc strings),
2. ensure code quality with flake8 and autopep8,
3. write test functions for pytest (aim for 100% code coverage),
4. update the changelog (for new features, increment to the next minor release; for small changes or bug fixes, increment the patch number)

6.2 Updating the documentation

The documentation is stored in the `docs` directory of the repository and is built using sphinx.

To build the documentation, first install the build requirements by running this in the `docs` directory:

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

You can now build the documentation with

```
sphinx-build . _build
```

Open the file `_build/index.html` in your web browser to view the result.

6.3 Writing model functions

You are here because you would like to write a new model function for nanite. Note that all model functions implemented in nanite are consequently available in PyJibe as well.

6.3.1 Getting started

First, create a Python file `model_unique_name.py` which will be the home of your new model (make sure the name starts with `model_`). You have three options (**1, 2 or 3**) to make your model available in nanite:

1. Place the file anywhere in your file system (e.g. `/home/peter/model_unique_name.py`) and run:

   ```
   from nanite.model import load_model_from_file

   load_model_from_file("/home/peter/model_unique_name.py", register=True)
   ```

   This is probably the most convenient method when prototyping. Note that you can also import model scritps in PyJibe (via the Preferences menu).

2. Place the file in the following location: `nanite/model/model_unique_name.py`. Once you have created this file, you have to register it in nanite by adding the line

   ```
   from . import model_unique_name  # noqa: F401
   ```

   at the top in the file `nanite/model/__init__.py`. This is the procedure when you create a pull request.

3. Or place the file in another location from where you can import it. This can be a submodule in a different package, or just the script in your `PATH`. The only thing you need is to import the script and register it.

   ```
   import model_unique_name
   from nanite.model import register_model

   register_model(model_unique_name)
   ```

Your file should at least contain the following:

```
import lmfit
import numpy as np


def get_parameter_defaults():
    """Return the default model parameters"""
    # The order of the parameters must match the order
    # of ´parameter_names´ and ´parameter_keys´.
    params = lmfit.Parameters()
    params.add("E", value=3e3, min=0)
    params.add("contact_point", value=0)
    params.add("baseline", value=0)
    return params


def your_model_name(delta, E, contact_point=0, baseline=0):
    r"""A brief model description

    A more elaborate model description with a formula.

    .. math::

        F = \frac{4}{3}
            E
```

(continues on next page)

(continued from previous page)

```
            \delta^{3/2}

    Parameters
    ----------
    delta: 1d ndarray
        Indentation [m]
    E: float
        Young's modulus [N/m^2]
    contact_point: float
        Indentation offset [m]
    baseline: float
        Force offset [N]

    Returns
    -------
    F: float
        Force [N]

    Notes
    -----
    Here you can add more information about the model.

    References
    ----------
    Please give proper references for your model (e.g. publications or
    arXiv manuscripts. You can do so by editing the "docs/nanite.bib"
    file and cite it like so:
    Sneddon (1965) :cite:`Sneddon1965`
    """
    # this is a convention to avoid computing the root of negative values
    root = contact_point - delta
    pos = root > 0
    # this is the model output
    out = np.zeros_like(delta)
    out[pos] = 4/3 * E * root[pos]**(3/2)
    # add the baseline
    return out + baseline


model_doc = your_model_name.__doc__
model_func = your_model_name
model_key = "unique_model_key"
model_name = "short model name"
parameter_keys = ["E", "contact_point", "baseline"]
parameter_names = ["Young's Modulus", "Contact Point", "Force Baseline"]
parameter_units = ["Pa", "m", "N"]
valid_axes_x = ["tip position"]
valid_axes_y = ["force"]
```

A few things should be noted:

- When designing your model parameters, always use SI units.
- Always include a model formula in the doc string. You can test whether it renders correctly by building the

documentation (see above) and checking whether your model shows up properly in the code reference.

- Fitting parameters should not contain spaces. Only use characters that are allowed in Python variable names.
- Since fitting is based on lmfit, you may define mathematical constraints in `get_parameter_defaults`. This includes algebraic constraints. However, if possible, try to solve your particular problem with ancillaries (see below), a concept that is easier to debug.
- If you would like to define "helper" parameters that should be hidden from users in PyJibe, you can prepend an underscore (_) to the parameter name.
- By default, nanite uses the method *nanite.model.residuals.residual()* to compute fit residuals. This method also implements the "reduce residuals near contact point" feature. You may define your own `residual` function in your model file, but this is discouraged. The same is true for the `model` function, which defaults to *nanite.model.residuals.model_direction_agnostic()*.
- You should always name the contact point parameter `contact_point`. Otherwise fitting will not work. If the *geometrical correction factor* k is used, the `contact_point` parameter is modified internally before and after the fit. If you don't use `contact_point`, then your fit results will be wrong when using $k \neq 1$.
- You should always name the parameter describing the Young's modulus `E`. This is important for higher-level functionalities in e.g. PyJibe and for plotting the Young's modulus over the indentation depth.

Now it is time for a quick sanity check:

```
from nanite import model
assert "unique_model_key" in model.models_available
```

6.3.2 Ancillary parameters

For more elaborate models, you might need additional parameters from the *nanite.indent.Indentation* instance. This is where ancillary parameters come into play.

You can define an arbitrary number of ancillary parameters in your `model_unique_name.py` file:

```
def compute_ancillaries(idnt):
    """Compute ancillaries for my model

    Parameters
    ----------
    idnt: nanite.indent.Indentation
        Indentation dataset from which to extract the ancillary
        parameters.

    Returns
    -------
    example: dict
        Dictionary with ancillary parameters. In this example:

        - "force_range": total force range covered by approach and retract
    """
    # You have access to the initial fit parameters (including a
    # good contact point estimate) with this line:
    parms = idnt.get_initial_fit_parameters(model_key=model_key,
                                            model_ancillaries=False)
```

(continues on next page)

(continued from previous page)

```
    # You can access individual columns...
    force = idnt.data["force"]
    segment = idnt.data["segment"]  # `False` for approach; `True` for retract
    tip_position = idnt.data["tip position"]

    # ...and segments
    force_approach = force[~segment]  # equivalent to force[segment == False]
    force_retract = force[segment]

    # Initialize ancillary dictionary.
    anc_dict = dict()

    # This is the exemplary force parameter
    anc_dict["force_range"] = np.ptp(force)

    return anc_dict

# And below the other `parameter_keys` etc.:
parameter_anc_keys = ["force_range"]
parameter_anc_names = ["Overall peak-to-peak force"]
parameter_anc_units = ["N"]
```

You should know:

- If an ancillary parameter key matches that of a fitting parameter (defined in `get_parameter_defaults` above), then the ancillary parameter can be used as an initial value for fitting (see *nanite.fit.guess_initial_parameters()*).
- If `compute_ancillaries` does not know how to compute a certain parameter, it shoud set it to `np.nan` instead of `None` (compatibility with PyJibe).
- If you would like to define an ancillary parameter that depends on a successful fit, you could first check against `idnt.fit_properties["success"]` and then compute your parameter (else set it to `np.nan`).

CHAPTER

SEVEN

CODE REFERENCE

7.1 Module level aliases

For user convenience, the following objects are available at the module level.

class nanite.**Indentation**
alias of *nanite.indent.Indentation*

class nanite.**IndentationGroup**
alias of *nanite.group.IndentationGroup*

class nanite.**IndentationRater**
alias of nanite.rate.IndentationRater

class nanite.**QMap**
alias of *nanite.qmap.QMap*

nanite.**load_group**()
alias of *nanite.group.load_group*

7.2 Force-indentation data

class nanite.indent.**Indentation**(*data*, *metadata*, *diskcache=None*)
Additional functionalities for afmformats.AFMForceDistance

apply_preprocessing(*preprocessing=None*, *options=None*, *ret_details=False*)
Perform curve preprocessing steps

Parameters

- **preprocessing** (*list*) – A list of preprocessing method identifiers that are stored in the *nanite.preproc.PREPROCESSORS* list. If set to *None*, *self.preprocessing* will be used.
- **options** (*dict of dict*) – Dictionary of keyword arguments for each preprocessing step (if applicable)
- **ret_details** – Return preprocessing details dictionary

compute_emodulus_mindelta(*callback=None*)
Elastic modulus in dependency of maximum indentation

The fitting interval is varied such that the maximum indentation depth ranges from the lowest tip position to the estimated contact point. For each interval, the current model is fitted and the elastic modulus is extracted.

Parameters **callback** (*callable*) – A method that is called with the *emoduli* and *indentations* as the computation proceeds every five steps.

Returns **emoduli, indentations** – The fitted elastic moduli at the corresponding maximal indentation depths.

Return type 1d ndarrays

Notes

The information about emodulus and mindelta is also stored in *self.fit_properties* with the keys "optimal_fit_E_array" and "optimal_fit_delta_array", if *self.fit_model* is called with the argument *search_optimal_fit* set to *True*.

estimate_contact_point_index(*method='deviation_from_baseline'*)
Estimate the contact point index

See the *poc* submodule for more information.

estimate_optimal_mindelta()
Estimate the optimal indentation depth

This is a convenience function that wraps around *compute_emodulus_mindelta* and *IndentationFitter.compute_opt_mindelta*.

fit_model(***kwargs*)
Fit the approach-retract data to a model function

Parameters

- **model_key** (*str*) – A key referring to a model in *nanite.model.models_available*
- **params_initial** (*instance of lmfit.Parameters or dict*) – Parameters for fitting. If not given, default parameters are used.
- **range_x** (*tuple of 2*) – The range for fitting, see *range_type* below.
- **range_type** (*str*) – One of:
  - **absolute:** Set the absolute fitting range in values given by the *x_axis*.
  - **relative cp:** In some cases it is desired to be able to fit a model only up until a certain indentation depth (tip position) measured from the contact point. Since the contact point is a fit parameter as well, this requires a two-pass fitting.
- **preprocessing** (*list of str*) – Preprocessing
- **preprocessing_options** (*list of str*) – Preprocessing
- **segment** (*str*) – Segment index (e.g. 0 for approach)
- **weight_cp** (*float*) – Weight the contact point region which shows artifacts that are difficult to model with e.g. Hertz.
- **gcf_k** (*float*) – Geometrical correction factor k for non-single-contact data. The measured indentation is multiplied by this factor to correct for experimental geometries during fitting, e.g. `gcf_k=0.5` for parallel-place compression.
- **optimal_fit_edelta** (*bool*) – Search for the optimal fit by varying the maximal indentation depth and determining a plateau in the resulting Young's modulus (fitting parameter "E").

get_ancillary_parameters(*model_key=None*)
Compute ancillary parameters for the current model

get_initial_fit_parameters(*model_key=None*, *common_ancillaries=True*, *model_ancillaries=True*)
Return the initial fit parameters

If there are not initial fit parameters set in *self.fit_properties*, then they are computed.

Parameters

- **model_key** (*str*) – Optionally set a model key. This will override the "model_key" key in *self.fit_properties*.
- **common_ancillaries** (*bool*) – Guess global ancillaries such as the contact point.
- **model_ancillaries** (*bool*) – Guess model-related ancillaries

Notes

global_ancillaries and *model_ancillaries* only have an effect if self.fit_properties["params_initial"] is set.

get_rating_parameters()
Return current rating parameters

rate_quality(*regressor='Extra Trees'*, *training_set='zef18'*, *names=None*, *lda=None*)
Compute the quality of the obtained curve

Uses heuristic approaches to rate a curve.

Parameters

- **regressor** (*str*) – The regressor name used for rating.
- **training_set** (*str*) – A label for a training set shipped with nanite or a path to a training set.
- **names** (*list of str*) – Only use these features for rating
- **lda** (*bool*) – Perform linear discriminant analysis

Returns **rating** – A value between 0 and 10 where 0 is the lowest rating. If no fit has been performed, a rating of -1 is returned.

Return type float

Notes

The rating is cached based on the fitting hash (see *IndentationFitter._hash*).

property data

property fit_properties
Fitting results, see *Indentation.fit_model()*)

preprocessing
Default preprocessing steps, see *Indentation.apply_preprocessing()*.

preprocessing_options
Preprocessing options

7.3 Groups

class nanite.group.**IndentationGroup**(*path=None*, *meta_override=None*, *callback=None*)
Group of Indentation

Parameters

- **path** (*str or pathlib.Path or None*) – The path to the data file. The data format is determined and the file is loaded using index.
- **meta_override** (*dict*) – if specified, contains key-value pairs of metadata that should be used when loading the files (see afmformats.meta.META_FIELDS)
- **callback** (*callable or None*) – A method that accepts a float between 0 and 1 to externally track the process of loading the data.

append(*afmdata*)
Append a new instance of AFMData

This subclassed method makes sure that "spring constant" is set if "tip position" has to be computed in the future.

Parameters **afmdata** (*afmformats.afm_data.AFMData*) – AFM data

nanite.group.**load_group**(*path*, *callback=None*, *meta_override=None*)
Load indentation data from disk

Parameters

- **path** (*path-like*) – Path to experimental data
- **callback** (*callable*) – function for tracking progress; must accept a float in [0, 1] as an argument.
- **meta_override** (*dict*) – if specified, contains key-value pairs of metadata that should be used when loading the files (see afmformats.meta.META_FIELDS)

Returns **group** – Indentation group with force-distance data

Return type nanite.IndetationGroup

7.4 Loading data

nanite.read.**get_data_paths**(*path*)
Return list of data paths with force-distance data

DEPRECATED

nanite.read.**get_data_paths_enum**(*path*, *skip_errors=False*)
Return a list with paths and their internal enumeration

Parameters

- **path** (*str or pathlib.Path or list of str or list of pathlib.Path*) – path to data files or directory containing data files; if directories are given, they are searched recursively
- **skip_errors** (*bool*) – skip paths that raise errors

Returns **path_enum** – each entry in the list is a list of [pathlib.Path, int], enumerating all curves in each file

Return type list of lists

nanite.read.**get_load_data_modality_kwargs**()
Return imaging modality kwargs for afmformats.load_data

Uses *DEFAULT_MODALITY*.

Returns **kwargs** – keyword arguments for `afmformats.load_data()`

Return type dict

nanite.read.**load_data**(*path*, *callback=None*, *meta_override=None*)
Load data and return list of `afmformats.AFMForceDistance`

This is essentially a wrapper around `afmformats.formats.find_data()` and `afmformats.formats.load_data()` that returns force-distance datasets.

Parameters

- **path** (*str or pathlib.Path or list of str or list of pathlib.Path*) – path to data files or directory containing data files; if directories are given, they are searched recursively
- **callback** (*callable*) – function for progress tracking; must accept a float in [0, 1] as an argument.
- **meta_override** (*dict*) – if specified, contains key-value pairs of metadata that are used when loading the files (see `afmformats.meta.META_FIELDS`)

nanite.read.**DEFAULT_MODALITY = 'force-distance'**
The default imaging modality when loading AFM data. Set this to *None* to also be able to load e.g. creep-compliance data. See issue https://github.com/AFM-analysis/nanite/issues/11 for more information. Note that especially the export of rating containers may not work with any imaging modality other than force-distance.

7.5 Preprocessing

exception nanite.preproc.**CannotSplitWarning**

class nanite.preproc.**IndentationPreprocessor**

apply(***kwargs*)

autosort(***kwargs*)

available(***kwargs*)

check_order(***kwargs*)

get_func(***kwargs*)

get_name(***kwargs*)

get_steps_required(***kwargs*)

nanite.preproc.**apply**(*apret*, *identifiers=None*, *options=None*, *ret_details=False*, *preproc_names=None*)
Perform force-distance preprocessing steps

Parameters

- **apret** (`nanite.Indentation`) – The afm data to preprocess
- **identifiers** (*list*) – A list of preprocessing identifiers that will be applied (in the order given).

- **options** (*dict of dict*) – Preprocessing options for each identifier
- **ret_details** – Return preprocessing details dictionary
- **preproc_names** (*list*) – Deprecated - use *identifiers* instead

nanite.preproc.**autosort**(*identifiers*)
Automatically sort preprocessing identifiers

This takes into account *steps_required* and *steps_optional*.

nanite.preproc.**available**()
Return list of available preprocessor identifiers

nanite.preproc.**check_order**(*identifiers*)
Check preprocessing steps for correct order

nanite.preproc.**get_func**(*identifier*)
Return preprocessor function for identifier

nanite.preproc.**get_name**(*identifier*)
Return preprocessor name for identifier

nanite.preproc.**get_steps_required**(*identifier*)
Return requirement identifiers for identifier

nanite.preproc.**preproc_compute_tip_position**(*apret*)
Perform tip-sample separation

Populate the "tip position" column by adding the force normalized by the spring constant to the cantilever height ("height (measured)").

This computation correctly reproduces the column "Vertical Tip Position" as it is exported by the JPK analysis software with the checked option "Use Unsmoothed Height".

nanite.preproc.**preproc_correct_force_offset**(*apret*)
Correct the force offset with an average baseline value

nanite.preproc.**preproc_correct_split_approach_retract**(*apret*)
Split the approach and retract curves (farthest point method)

Approach and retract curves are defined by the microscope. When the direction of piezo movement is flipped, the force at the sample tip is still increasing. This can be either due to a time lag in the AFM system or due to a residual force acting on the sample due to the bent cantilever.

To repair this time lag, we append parts of the retract curve to the approach curve, such that the curves are split at the minimum height.

nanite.preproc.**preproc_correct_tip_offset**(*apret*, *method='deviation_from_baseline'*, *ret_details=False*)
Estimate the point of contact

An estimate of the contact point is subtracted from the tip position.

nanite.preproc.**preproc_smooth_height**(*apret*)
Make height data monotonic

For the columns "height (measured)", "height (piezo), and "tip position", this method ensures that the approach and retract segments are monotonic.

nanite.preproc.**preprocessing_step**(*identifier*, *name*, *steps_required=None*, *steps_optional=None*, *options=None*)
Decorator for Indentation preprocessors

The name and identifier are stored as a property of the wrapped function.

Parameters

- **identifier** (*str*) – identifier of the preprocessor (e.g. "correct_tip_offset")
- **name** (*str*) – human-readble name of the preprocessor (e.g. "Estimate contact point")
- **steps_required** (*list of str*) – list of preprocessing steps that must be added before this step
- **steps_optional** (*list of str*) – unlike *steps_required*, these steps do not have to be set, but if they are set, they should come before this step
- **options** (*list of dict*) – if the preprocessor accepts optional keyword arguments, this list yields valid values or dtypes

nanite.preproc.**PREPROCESSORS = [<function preproc_compute_tip_position>, <function preproc_correct_force_offset>, <function preproc_correct_tip_offset>, <function preproc_correct_split_approach_retract>, <function preproc_smooth_height>]**
Available preprocessors

7.6 Contact point estimation

Methods for estimating the point of contact (POC)

nanite.poc.**compute_poc**(*force*, *method='deviation_from_baseline'*, *ret_details=False*)
Compute the contact point from force data

Parameters

- **force** (*1d ndarray*) – Force data
- **method** (*str*) – Name of the method for computing the POC (see *POC_METHODS*)
- **ret_details** (*bool*) – Whether or not to return a dictionary with details alongside the POC estimate.

Notes

If the POC method returns np.nan, then the center of the force data is returned (to allow fitting algorithms to proceed).

nanite.poc.**compute_preproc_clip_approach**(*force*)
Clip the approach part (discard the retract part)

This POC preprocessing method may be applied before applying the POC estimation method.

nanite.poc.**poc**(*identifier*, *name*, *preprocessing*)
Decorator for point of contact (POC) methods

The name and identifier are stored as a property of the wrapped function.

Parameters

- **identifier** (*str*) – identifier of the POC method (e.g. "baseline_deviation")
- **name** (*str*) – human-readble name of the POC method (e.g. "Deviation from baseline")
- **preprocessing** (*list of str*) – list of preprocessing methods that should be applied; may contain ["clip_approach"].

nanite.poc.**poc_deviation_from_baseline**(*force*, *ret_details=False*)
Deviation from baseline

1. Obtain the baseline (initial 10% of the gradient curve)
2. Compute average and maximum deviation of the baseline
3. The CP is the index of the curve where it exceeds twice of the maximum deviation

nanite.poc.**poc_fit_constant_line**(*force*, *ret_details=False*)
Piecewise fit with constant and line

Fit a piecewise function (constant+linear) to the baseline and indentation part:

$$F = \max(d, m\delta + d)$$

The point of contact is the intersection of a horizontal line at d (baseline) and a linear function with slope m for the indentation part.

The point of contact is defined as $\delta = 0$ (It's another fitting parameter).

nanite.poc.**poc_fit_constant_polynomial**(*force*, *ret_details=False*)
Piecewise fit with constant and polynomial

Fit a piecewise function (constant + polynomial) to the baseline and indentation part.

$$F = \frac{\delta^3}{a\delta^2 + b\delta + c} + d$$

This function is defined for all $\delta > 0$. For all $\delta < 0$ the model evaluates to d (baseline).

I'm not sure where this has been described initially, but it is used e.g. in [RZSK19].

For small indentations, this function exhibits a cubic behavior:

$$y \approx \delta^3/c + d$$

And for large indentations, this function is linear:

$$y \approx \delta/a$$

The point of contact is defined as $\delta = 0$ (It's another fitting parameter).

nanite.poc.**poc_fit_line_polynomial**(*force*, *ret_details=False*)
Piecewise fit with line and polynomial

Fit a piecewise function (line + polynomial) to the baseline and indentation part.

The linear basline ($\delta < 0$) is modeled with:

$$F = m\delta + d$$

The indentation part ($\delta > 0$) is modeled with:

$$F = \frac{\delta^3}{a\delta^2 + b\delta + c} + m\delta + d$$

For small indentations, this function exhibits a linear and only slightly cubic behavior:

$$y \approx \delta^3/c + m\delta + d$$

And for large indentations, this function is linear:

$$y \approx \left(\frac{1}{a} + m\right)\delta$$

The point of contact is defined as $\delta = 0$ (It's another fitting parameter).

See also:

poc_fit_constant_polynomial polynomial-only version

nanite.poc.**poc_frechet_direct_path**(*force*, *ret_details=False*)
Fréchet distance to direct path

The indentation part is transformed to normalized coordinates (force and corresponding x in range [0, 1]). The point with the largest distance to the line from (0, 0) to (1, 1) is the contact point.

This method is robust with regard to tilted baselines and is a good initial guess for fitting-based POC estimation approaches.

Note that the length of the baseline influences the returned contact point. For shorter baselines, the contact point will be closer to the point of maximum indentation.

nanite.poc.**poc_gradient_zero_crossing**(*force*, *ret_details=False*)
Gradient zero-crossing of indentation part

1. Apply a moving average filter to the curve
2. Compute the gradient
3. Cut off gradient at maximum with a 10 point reserve
4. Apply a moving average filter to the gradient
5. The POC is the index of the averaged gradient curve where the values are below 1% of the gradient maximum, measured from the indentation maximum (not from baseline).

nanite.poc.**POC_METHODS = [<function poc_deviation_from_baseline>, <function poc_fit_constant_line>, <function poc_fit_constant_polynomial>, <function poc_fit_line_polynomial>, <function poc_frechet_direct_path>, <function poc_gradient_zero_crossing>]**
List of all methods available for contact point estimation

7.7 Modeling

7.7.1 Methods and constants

nanite.model.**compute_anc_parms**(*idnt*, *model_key*)
Compute ancillary parameters for a force-distance dataset

Ancillary parameters include parameters that:

- are unrelated to fitting: They may just be important parameters to the user.
- require the entire dataset: They cannot be extracted during fitting, because they require more than just the approach xor retract curve to compute (e.g. hysteresis, jump of retract curve at maximum indentation). They may, additionally, depend on initial fit parameters set by the user.
- require a fit: They are dependent on fitting parameters but are not required during fitting.

Notes

If an ancillary parameter name matches that of a fitting parameter, then it is assumed that it can be used for fitting. Please see *nanite.indent.Indentation.get_initial_fit_parameters()* and *nanite.fit.guess_initial_parameters()*.

Ancillary parameters are set to *np.nan* if they cannot be computed.

Parameters

- **idnt** (nanite.indent.Indentation) – The force-distance data for which to compute the ancillary parameters
- **model_key** (*str*) – Name of the model

Returns **ancillaries** – key-value dictionary of ancillary parameters

Return type collections.OrderedDict

nanite.model.**get_anc_parm_keys**(*model_key*)
Return the key names of a model's ancillary parameters

nanite.model.**get_anc_parms**(*idnt*, *model_key*)

nanite.model.**get_init_parms**(*model_key*)
Get initial fit parameters for a model

nanite.model.**get_model_by_name**(*name*)
Convenience function to obtain a model by name instead of by key

nanite.model.**get_parm_name**(*model_key*, *parm_key*)
Return parameter label

Parameters

- **model_key** (*str*) – The model key (e.g. "hertz_cone")
- **parm_key** (*str*) – The parameter key (e.g. "E")

Returns **parm_name** – The parameter label (e.g. "Young's Modulus")

Return type str

nanite.model.**get_parm_unit**(*model_key*, *parm_key*)
Return parameter unit

Parameters

- **model_key** (*str*) – The model key (e.g. "hertz_cone")
- **parm_key** (*str*) – The parameter key (e.g. "E")

Returns **parm_unit** – The parameter unit (e.g. "Pa")

Return type str

7.7.2 Modeling core class

exception nanite.model.core.**ModelError**

exception nanite.model.core.**ModelImplementationError**

exception nanite.model.core.**ModelImplementationWarning**

exception nanite.model.core.**ModelImportError**

exception nanite.model.core.**ModelIncompleteError**

class nanite.model.core.**NaniteFitModel**(*model_module*)

Initialize the model with an imported Python module

compute_ancillaries(*fd*)

Compute ancillary parameters for a force-distance dataset

Ancillary parameters include parameters that:

- are unrelated to fitting: They may just be important parameters to the user.
- require the entire dataset: They cannot be extracted during fitting, because they require more than just the approach xor retract curve to compute (e.g. hysteresis, jump of retract curve at maximum indentation). They may, additionally, depend on initial fit parameters set by the user.
- require a fit: They are dependent on fitting parameters but are not required during fitting.

Notes

If an ancillary parameter name matches that of a fitting parameter, then it is assumed that it can be used for fitting. Please see *nanite.indent.Indentation.get_initial_fit_parameters()* and *nanite.fit.guess_initial_parameters()*.

Ancillary parameters are set to *np.nan* if they cannot be computed.

Parameters **fd** (nanite.indent.Indentation) – The force-distance data for which to compute the ancillary parameters

Returns **ancillaries** – key-value dictionary of ancillary parameters

Return type collections.OrderedDict

get_anc_parm_keys()

Return the key names of a model's ancillary parameters

get_parm_name(*key*)

Return parameter label

Parameters **key** (*str*) – The parameter key (e.g. "E")

Returns **parm_name** – The parameter label (e.g. "Young's Modulus")

Return type str

get_parm_unit(*key*)

Return parameter unit

Parameters **key** (*str*) – The parameter key (e.g. "E")

Returns **parm_unit** – The parameter unit (e.g. "Pa")

Return type str

nanite.model.core.**compute_anc_max_indent**(*fd*)
Compute ancillary parameter 'Maximum indentation'

nanite.model.core.**ANCILLARY_COMMON = {'max_indent': ('Maximum indentation', 'm', <function compute_anc_max_indent>)}**
Common ancillary parameters

7.7.3 Residuals and weighting

nanite.model.residuals.**compute_contact_point_weights**(*cp*, *delta*, *weight_dist=5e-07*)
Compute contact point weights

Parameters

- **cp** (*float*) – Fitted contact point value
- **delta** (*1d ndarray of length N*) – The indentation array along which weights will be computed.
- **weight_width** (*float*) – The distance from *cp* until which weights will be applied.

Returns **weights** – The weights.

Return type 1d ndarray of length N

Notes

All variables should be given in the same units. The weights increase linearly from increasing distances of *delta-cp* from 0 to 1 and are 1 outside of the weight width *abs(delta-cp)>weight_width*.

nanite.model.residuals.**get_default_modeling_wrapper**(*model_function*)
Return a wrapper around the default nanite modeling function

nanite.model.residuals.**get_default_residuals_wrapper**(*model_function*)
Return a wrapper around the default nanite residual function

nanite.model.residuals.**model_direction_agnostic**(*model_function*, *params*, *delta*)
Call *model_function* while making sure that data are in correct order

TODO: Re-evaluate usefulness of this method.

nanite.model.residuals.**residual**(*params*, *delta*, *force*, *model*, *weight_cp=5e-07*)
Compute residuals for fitting

Parameters

- **params** (*lmfit.Parameters*) – The fitting parameters for *model*
- **delta** (*1D ndarray of lenght M*) – The indentation distances
- **force** (*1D ndarray of length M*) – The corresponding force data
- **model** (*callable*) – A model function accepting the arguments `params` and `delta`
- **weight_cp** (*positive float or zero/False*) – The distance from the contact point until which linear weights will be applied. Set to zero to disable weighting.

nanite.model.weight.**weight_cp**(**args*, ***kwargs*)

7.7.4 Models

Each model is implemented as a submodule in `nanite.model`. For instance `nanite.model.model_hertz_parabolic`. Each of these modules implements the following functions (which are not listed for each model in the subsections below), here with the (non-existent) example module `model_submodule`:

`nanite.model.model_submodule.`**`get_parameter_defaults`**`()`
Return the default parameters of the model.

`nanite.model.model_submodule.`**`model`**`()`
Wrap the actual model for fitting.

`nanite.model.model_submodule.`**`residual`**`()`
Compute the residuals during fitting (optional).

In addition, each submodule contains the following attributes:

`nanite.model.model_submodule.`**`model_doc`**
The doc-string of the model function.

`nanite.model.model_submodule.`**`model_key`**
The model key used in the command line interface and during scripting.

`nanite.model.model_submodule.`**`model_name`**
The name of the model.

`nanite.model.model_submodule.`**`parameter_keys`**
Parameter keys of the model for higher-level applications.

`nanite.model.model_submodule.`**`parameter_names`**
Parameter names of the model for higher-level applications.

`nanite.model.model_submodule.`**`parameter_units`**
Parameter units for higher-level applications.

Ancillary parameters may also be defined like so:

`nanite.model.model_submodule.`**`compute_ancillaries`**`()`
Function that returns a dictionary with ancillary parameters computed from an *Indentation* instance.

`nanite.model.model_submodule.`**`parameter_anc_keys`**
Ancillary parameter keys

`nanite.model.model_submodule.`**`parameter_anc_names`**
Ancillary parameter names

`nanite.model.model_submodule.`**`parameter_anc_units`**
Ancillary parameter units

conical indenter (Hertz)

model key	hertz_cone
model name	conical indenter (Hertz)
model location	nanite.model.model_conical_indenter

`nanite.model.model_conical_indenter.`**`hertz_conical`**`(`*delta*, *E*, *alpha*, *nu*, *contact_point=0*, *baseline=0*`)`
Hertz model for a conical indenter

$$F = \frac{2\tan\alpha}{\pi}\frac{E}{1-\nu^2}\delta^2$$

Parameters

- **delta** (*1d ndarray*) – Indentation [m]
- **E** (*float*) – Young's modulus [N/m^2]
- **alpha** (*float*) – Half cone angle [degrees]
- **nu** (*float*) – Poisson's ratio
- **contact_point** (*float*) – Indentation offset [m]
- **baseline** (*float*) – Force offset [N]

Returns **F** – Force [N]

Return type float

Notes

These approximations are made by the Hertz model:

- The sample is isotropic.
- The sample is a linear elastic solid.
- The sample is extended infinitely in one half space.
- The indenter is not deformable.
- There are no additional interactions between sample and indenter.

Additional assumptions:

- infinitely sharp probe

References

Love (1939) [Lov39], Sneddon (1965) [Sne65] (equation 6.4)

parabolic indenter (Hertz)

model key	hertz_para
model name	parabolic indenter (Hertz)
model location	nanite.model.model_hertz_paraboloidal

nanite.model.model_hertz_paraboloidal.**hertz_paraboloidal**(*delta*, *E*, *R*, *nu*, *contact_point=0*, *baseline=0*)

Hertz model for a paraboloidal indenter

$$F = \frac{4}{3}\frac{E}{1-\nu^2}\sqrt{R}\delta^{3/2}$$

Parameters

- **delta** (*1d ndarray*) – Indentation [m]
- **E** (*float*) – Young's modulus [N/m^2]
- **R** (*float*) – Tip radius [m]

- **nu** (*float*) – Poisson's ratio
- **contact_point** (*float*) – Indentation offset [m]
- **baseline** (*float*) – Force offset [N]

Returns **F** – Force [N]

Return type float

Notes

The derivation in [Sne65] reads

$$F = \frac{4}{3}\frac{E}{1-\nu^2}\sqrt{2k}\delta^{3/2},$$

where k is defined by the paraboloid equation

$$\rho^2 = 4kz.$$

As follows from the derivations in [LL59], this model is valid for either of the two cases:

- Indentation of a plane (infinite radius) with Young's modulus E by a sphere with infinite Young's modulus and radius R, or
- Indentation of a sphere with radius R and Young's modulus E by a plane (infinite radius) with infinite Young's modulus.

These approximations are made by the Hertz model:

- The sample is isotropic.
- The sample is a linear elastic solid.
- The sample is extended infinitely in one half space.
- The indenter is not deformable.
- There are no additional interactions between sample and indenter.

Additional assumptions:

- no surface forces
- If the indenter is spherical, then its radius R is much larger than the indentation depth δ.

References

Sneddon (1965) [Sne65] (equation 6.9), Theory of Elasticity by Landau and Lifshitz (1959) [LL59] (§9 Solid bodies in contact, equation 9.14)

pyramidal indenter, three-sided (Hertz)

model key	hertz_pyr3s
model name	pyramidal indenter, three-sided (Hertz)
model location	nanite.model.model_hertz_three_sided_pyramid

nanite.model.model_hertz_three_sided_pyramid.**hertz_three_sided_pyramid**(*delta*, *E*, *alpha*, *nu*, *contact_point=0*, *baseline=0*)

Hertz model for three sided pyramidal indenter

$$F = 0.887 \tan\alpha \cdot \frac{E}{1-\nu^2}\delta^2$$

Parameters

- **delta** (*1d ndarray*) – Indentation [m]
- **E** (*float*) – Young's modulus [N/m^2]
- **alpha** (*float*) – Inclination angle of the pyramidal face [degrees]
- **nu** (*float*) – Poisson's ratio
- **contact_point** (*float*) – Indentation offset [m]
- **baseline** (*float*) – Force offset [N]

Returns **F** – Force [N]

Return type float

Notes

These approximations are made by the Hertz model:

- The sample is isotropic.
- The sample is a linear elastic solid.
- The sample is extended infinitely in one half space.
- The indenter is not deformable.
- There are no additional interactions between sample and indenter.
- The inclination angle of the pyramidal face (in radians) must be close to zero.

References

Bilodeau et al. 1992 [Bil92]

spherical indenter (Sneddon)

model key	sneddon_spher
model name	spherical indenter (Sneddon)
model location	nanite.model.model_sneddon_spherical

nanite.model.model_sneddon_spherical.**delta_of_a**(*a*, *R*)
Compute indentation from contact area radius (wrapper)

nanite.model.model_sneddon_spherical.**get_a**(*R*, *delta*, *accuracy=1e-12*)
Compute the contact area radius (wrapper)

nanite.model.model_sneddon_spherical.**hertz_spherical**(*delta*, *E*, *R*, *nu*, *contact_point=0.0*, *baseline=0.0*)
Hertz model for Spherical indenter - modified by Sneddon

$$F = \frac{E}{1-\nu^2}\left(\frac{R^2+a^2}{2}\ln\left(\frac{R+a}{R-a}\right) - aR\right)$$
$$\delta = \frac{a}{2}\ln\left(\frac{R+a}{R-a}\right)$$

(a is the radius of the circular contact area between bead and sample.)

Parameters

- **delta** (*1d ndarray*) – Indentation [m]
- **E** (*float*) – Young's modulus [N/m^2]
- **R** (*float*) – Tip radius [m]
- **nu** (*float*) – Poisson's ratio
- **contact_point** (*float*) – Indentation offset [m]
- **baseline** (*float*) – Force offset [N]

Returns **F** – Force [N]

Return type float

Notes

These approximations are made by the Hertz model:

- The sample is isotropic.
- The sample is a linear elastic solid.
- The sample is extended infinitely in one half space.
- The indenter is not deformable.
- There are no additional interactions between sample and indenter.

Additional assumptions:

- no surface forces

References

Sneddon (1965) [Sne65] (equations 6.13 and 6.15)

spherical indenter (Sneddon, truncated power series)

model key	sneddon_spher_approx
model name	spherical indenter (Sneddon, truncated power series)
model location	nanite.model.model_sneddon_spherical_approximation

nanite.model.model_sneddon_spherical_approximation.**hertz_sneddon_spherical_approx**(*delta*, *E*, *R*, *nu*, *contact_point=0*, *baseline=0*)

Hertz model for Spherical indenter - approximation

$$F = \frac{4}{3}\frac{E}{1-\nu^2}\sqrt{R}\delta^{3/2}\left(1 - \frac{1}{10}\frac{\delta}{R} - \frac{1}{840}\left(\frac{\delta}{R}\right)^2 + \frac{11}{15120}\left(\frac{\delta}{R}\right)^3 + \frac{1357}{6652800}\left(\frac{\delta}{R}\right)^4\right)$$

Parameters

- **delta** (*1d ndarray*) – Indentation [m]
- **E** (*float*) – Young's modulus [N/m^2]
- **R** (*float*) – Tip radius [m]
- **nu** (*float*) – Poisson's ratio
- **contact_point** (*float*) – Indentation offset [m]
- **baseline** (*float*) – Force offset [N]

Returns **F** – Force [N]

Return type float

Notes

These approximations are made by the Hertz model:

- The sample is isotropic.
- The sample is a linear elastic solid.
- The sample is extended infinitely in one half space.
- The indenter is not deformable.
- There are no additional interactions between sample and indenter.

Additional assumptions:

- no surface forces

Truncated power series approximation:

This model is a truncated power series approximation of *spherical indenter (Sneddon)*. The expected error is more than four magnitues lower than the signal (see e.g. *Approximating the Hertzian model with a spherical indenter*). The Bio-AFM analysis software by JPK/Bruker uses the same model.

References

Sneddon (1965) [Sne65] (equations 6.13 and 6.15), Dobler (personal communication, 2018) [Dob18]

7.8 Fitting

exception nanite.fit.**FitDataError**

exception nanite.fit.**FitKeyError**

exception nanite.fit.**FitWarning**

class nanite.fit.**FitProperties**

Fit property manager class

Provide convenient access to fit properties as a dictionary and dynamically manage resets due to new initial parameters.

Dynamic properties include:

- set "params_initial" to *None* if the "model_key" changes
- remove all keys except those in *FP_DEFAULT* if a key that is in *FP_DEFAULT* changes (All other keys are considered to be obsolete fitting results).

Additional attributes:

reset()

restore(*props*)

update the dictionary without removing any keys

class nanite.fit.**IndentationFitter**(*idnt*, ***kwargs*)

Fit force-distance curves

Parameters

- **idnt** (nanite.indent.Indentation) – The dataset to fit
- **model_key** (*str*) – A key referring to a model in *nanite.model.models_available*
- **params_initial** (*instance of lmfit.Parameters*) – Parameters for fitting. If not given, default parameters are used.
- **range_x** (*tuple of 2*) – The range for fitting, see *range_type* below.
- **range_type** (*str*) – One of:
  - **absolute:** Set the absolute fitting range in values given by the *x_axis*.
  - **relative cp:** In some cases it is desired to be able to fit a model only up until a certain indentation depth (tip position) measured from the contact point. Since the contact point is a fit parameter as well, this requires a two-pass fitting.
- **preprocessing** (*list of str*) – Preprocessing step identifiers

- **preprocessing_options** (*dict of dicts*) – Preprocessing keyword arguments of steps (if applicable)
- **segment** (*int*) – Segment index (e.g. 0 for approach)
- **weight_cp** (*float*) – Weight the contact point region which shows artifacts that are difficult to model with e.g. Hertz.
- **gcf_k** (*float*) – Geometrical correction factor k for non-single-contact data. The measured indentation is multiplied by this factor to correct for experimental geometries during fitting, e.g. `gcf_k=0.5` for parallel-place compression.
- **optimal_fit_edelta** (*bool*) – Search for the optimal fit by varying the maximal indentation depth and determining a plateau in the resulting Young's modulus (fitting parameter "E").
- **optimal_fit_num_samples** (*int*) – Number of samples to use for searching the optimal fit

compute_emodulus_vs_mindelta(*callback=None*)
Compute elastic modulus vs. minimal indentation curve

static compute_opt_mindelta(*emoduli*, *indentations*)
Determine the plateau of an emodulus-indentation curve

The following procedure is performed:

1. Smooth the emodulus data with a Butterworth filter
2. Label sequences that have similar values by binning into ten regions between the min and max.
3. Ignore sequences with emodulus that is smaller than the binning size.
4. Determine the longest sequence.

fit()
Fit the approach-retract data to a model function

get_initial_parameters(*idnt=None*, *model_key='hertz_para'*)
Get initial fit parameters for a specific model

nanite.fit.**guess_initial_parameters**(*idnt=None*, *model_key='hertz_para'*, *common_ancillaries=True*, *model_ancillaries=True*)
Guess initial fitting parameters

Parameters

- **idnt** (nanite.indent.Indentation) – The dataset to use for guessing initial fitting parameters using ancillary parameters
- **model_key** (*str*) – The model key
- **common_ancillaries** (*bool*) – Guess global ancillary parameters (such as contact point)
- **model_ancillaries** (*bool*) – Use model-related ancillary parameters

nanite.fit.**obj2bytes**(*obj*)
Bytes representation of an object for hashing

7.9 Rating

7.9.1 Features

class nanite.rate.features.**IndentationFeatures**(*dataset=None*)

static compute_features(*idnt*, *which_type='all'*, *names=None*, *ret_names=False*)
Compute the features for a data set

Parameters

- **idnt** (nanite.Indentation) – A dataset to rate
- **names** (*list of str*) – The names of the rating methods to use, e.g. ["rate_apr_bumps", "rate_apr_mon_incr"]. If None (default), all available rating methods are used.

Notes

names may include features that are excluded by *which_type*. E.g. if a "bool" feature is in *names* but *which_type* is "float", then the "bool" feature will be silently ignored.

feat_bin_apr_spikes_count()
spikes during IDT

Sudden spikes in indentation curve

feat_bin_cp_position()
CP outside of data range

Contact point position outside of range

feat_bin_size()
dataset too small

Number of points in indentation curve

feat_con_apr_flatness()
flatness of APR residuals

fraction of the positive-gradient residuals in the approach part

feat_con_apr_size()
relative APR size

length of the approach part relative to the indentation part

feat_con_apr_sum()
residuals of APR

absolute sum of the residuals in the approach part

feat_con_bln_slope()
slope of BLN

slope obtained from a linear least-squares fit to the baseline

feat_con_bln_variation()
variation in BLN

comparison of the forces at the beginning and at the end of the baseline

feat_con_cp_curvature()
curvature at CP

curvature of the force-distance data at the contact point

feat_con_cp_magnitude()
residuals at CP

mean value of the residuals around the contact point

feat_con_idt_maxima_75perc()
maxima in IDT residuals

sum of the indentation residuals' maxima in three intervals in-between 25% and 100% relative to the maximum indentation

feat_con_idt_monotony()
monotony of IDT

change of the gradient in the indentation part

feat_con_idt_spike_area()
area of IDT spikes

area of spikes appearing in the indentation part

feat_con_idt_sum()
overall IDT residuals

sum of the residuals in the indentation part

feat_con_idt_sum_75perc()
residuals in 75% IDT

sum of the residuals in the indentation part in-between 25% and 100% relative to the maximum indentation

classmethod get_feature_funcs(*which_type='all'*, *names=None*)
Return functions that compute features from a dataset

Parameters

- **names** (*list of str*) – The names of the rating methods to use, e.g. ["rate_apr_bumps", "rate_apr_mon_incr"]. If None (default), all available rating methods are returned.
- **which_type** (*str*) – Which features to return: ["all", "bool", "float"].

Returns **raters** – Each item in the list consists contains the name of the rating method and the corresponding rating method.

Return type list of tuples (name, callable)

classmethod get_feature_names(*which_type='all'*, *names=None*, *ret_indices=False*)
Get features names

Parameters

- **which_type** (*str or list of str*) – Return only features that are of a certain type. See *VALID_FEATURE_TYPES* for valid strings.
- **names** (*list of str*) – Return only features that are in this list.
- **ret_indices** (*bool*) – If True, also return the internal feature indices.

Returns **name_list** – List of feature names (callables of this class)

Return type list of str

property contact_point

property datafit_apr

property datares_apr

dataset
current dataset from which features are computed

property datax_apr

property datay_apr

property has_contact_point

property is_fitted

property is_valid

property meta

nanite.rate.features.**VALID_FEATURE_TYPES = ['all', 'binary', 'continuous']**
Valid keyword arguments for feature types

7.9.2 Rater

class nanite.rate.rater.**IndentationRater**(*regressor=None*, *scale=None*, *lda=None*, *training_set=None*, *names=None*, *weight=True*, *sample_weight=None*, **args*, ***kwargs*)

Rate quality

Parameters

- **regressor** (*sciki-learn RegressorMixin*) – The regressor used for rating
- **scale** (*bool*) – If True, apply a Standard Scaler. If a regressor based on decision trees is used, the Standard Scaler is not used by default, otherwise it is.
- **lda** (*bool*) – If True, apply a Linear Discriminant Analysis (LDA). If a regressor based on a decision tree is used, LDA is not used by default, otherwise it is.
- **training_set** (*tuple of (X, y)*) – The training set (samples, response)
- **names** (*list of str*) – Feature names to use
- **weight** (*bool*) – Weight the input samples by the number of occurrences or with *sample_weight*. For tree-based classifiers, set this to True to avoid bias.
- **sample_weight** (*list-like*) – The sample weights. If set to *None* sample weights are computed from the training set.
- ***args** (*list*) – Positional arguments for `IndentationFeatures`
- ****kwargs** – Keyword arguments for `IndentationFeatures`

See also:

sklearn.preprocessing.StandardScaler Standard scaler

sklearn.discriminant_analysis.LinearDiscriminantAnalysis Linear discriminant analysis

nanite.rate.regressors.reg_trees List of regressors that are identified as tree-based

static compute_sample_weight(*X*, *y*)
Weight samples according to occurrence in y

static get_training_set_path(*label='zef18'*)
Return the path to a training set shipped with nanite

Training sets are stored in the *nanite.rate* module path with `ts_` prepended to *label*.

classmethod load_training_set(*path=None*, *names=None*, *which_type=['continuous']*, *remove_nan=True*, *ret_names=False*)
Load a training set from a directory

By default, only the "continuous" features are imported. The "binary" features are not needed for training; they are used to sort out new force-distance data.

rate(*samples=None*, *datasets=None*)
Perform rating step

Parameters

- **samples** (*1d or 2d ndarray (cast to 2d ndarray) or None*) – Measured samples, if set to None, *dataset* must be given.
- **dataset** (*list of nanite.Indentation*) – Full, fitted measurement

Returns **ratings** – Resulting ratings

Return type list

names
feature names used by the regressor pipeline

pipeline
sklearn pipeline with transforms (and regressor if given)

nanite.rate.rater.**get_available_training_sets**()
List of internal training sets

nanite.rate.rater.**get_rater**(*regressor*, *training_set='zef18'*, *names=None*, *lda=None*, ***reg_kwargs*)
Convenience method to get a rater

Parameters

- **regressor** (*str or RegressorMixin*) – If a string, must be in *reg_names*.
- **training_set** (*str or pathlib.Path or tuple (X, y)*) – A string label representing a training set shipped with nanite, the path to a training set, or a tuple representing the training set (samples, response) for use with sklearn.
- **names** (*list of str*) – Only use these features for rating
- **lda** (*bool*) – Perform linear discriminant analysis

Returns **irater** – The rating instance.

Return type *nanite.IndentationRater*

7.9.3 Regressors

scikit-learn regressors and their keyword arguments

nanite.rate.regressors.**reg_names = ['AdaBoost', 'Decision Tree', 'Extra Trees', 'Gradient Tree Boosting', 'Random Forest', 'SVR (RBF kernel)', 'SVR (linear kernel)']**
List of available default regressor names

nanite.rate.regressors.**reg_trees = ['AdaBoostRegressor', 'DecisionTreeRegressor', 'ExtraTreesRegressor', 'GradientBoostingRegressor', 'RandomForestRegressor']**
List of tree-based regressor class names (used for keyword defaults in `IndentationRater`)

7.9.4 Manager

Save and load user-rated datasets

class nanite.rate.io.**RateManager**(*path*, *verbose=0*)
Manage user-defined rates

export_training_set(*path*)

get_cross_validation_score(*regressor*, *training_set=None*, *n_splits=20*, *random_state=42*)
Regressor cross-validation scoring

Cross-validation is used to identify regressors that over-fit the train set by splitting the train set into multiple learn/test sets and quantifying the regressor performance for each split.

Parameters

- **regressor** (*str or RegressorMixin*) – If a string, must be in *reg_names*.
- **training_set** (*X, y*) – If given, do not use self.samples

Notes

A `skimage.model_selection.KFold` cross validator is used in combination with the mean squared error score.

Cross-validation score is computed from samples that are filtered with the binary features and only from samples that do not contain any nan values.

get_rates(*which='user'*, *training_set='zef18'*)

which: str Which rating to return: "user" or a regressor name

get_training_set(*which_type='all'*, *prefilter_binary=False*, *remove_nans=False*, *transform=False*)
Return (X, y) training set

property datasets

path
Path to the manual ratings (directory or .h5 file)

property ratings

property samples
The individual sample ratings computed by afmlib

verbose
verbosity level

nanite.rate.io.**hash_file**(*path*, *blocksize=65536*)
Compute sha256 hex-hash of a file

Parameters

- **path** (*str or pathlib.Path*) – path to the file
- **blocksize** (*int*) – block size read from the file

Returns **hex** – The first six characters of the hash

Return type str

nanite.rate.io.**hdf5_rated**(*h5path*, *indent*)
Test whether an indentation has already been rated

Returns

Return type is_rated, rating, comment

nanite.rate.io.**load**(*path*, *meta_only=False*, *verbose=0*)

Notes

The .fit_properties attribute of each Indentation instance is overridden by a simple dictionary, so its functionalities are not available anymore.

nanite.rate.io.**load_hdf5**(*path*, *meta_only=False*)

nanite.rate.io.**save_hdf5**(*h5path*, *indent*, *user_rate*, *user_name*, *user_comment*, *h5mode='a'*)
Store all relevant data of a user rating into an hdf5 file

Parameters

- **h5path** (*str or pathlib.Path*) – Path to HDF5 rating container where data will be stored
- **indent** (nanite.Indentation) – The experimental data processed and fitted with nanite
- **user_rate** (*float*) – Rating given by the user
- **user_name** (*str*) – Name of the rating user

7.10 Quantitative maps

exception nanite.qmap.**DataMissingWarning**

class nanite.qmap.**QMap**(*path_or_group*, *meta_override=None*, *callback=None*)
Quantitative force spectroscopy map handling

Parameters

- **path_or_group** (*str or pathlib.Path or afmformats.afm_group.AFMGroup*) – The path to the data file or an instance of *AFMGroup*
- **meta_override** (*dict*) – Dictionary with metadata that is used when loading the data in *path*.
- **callback** (*callable or None*) – A method that accepts a float between 0 and 1 to externally track the process of loading the data.

static feat_fit_contact_point(*idnt*)
Contact point of the fit

static feat_fit_youngs_modulus(*idnt*)
Young's modulus

static feat_meta_rating(*idnt*)
Rating

CHAPTER

EIGHT

CHANGELOG

List of changes in-between nanite releases.

8.1 version 3.5.4

- setup: drop support for Python 3.7

8.2 version 3.5.3

- setup: bump numpy to 1.22.0

8.3 version 3.5.2

- build: add wheels for Python up to 3.11
- ref: fix DeprecationWarnings
- docs: make docs build on Windows
- docs: update GH actions badge
- tests: loosen np.allclose calls

8.4 version 3.5.1

- docs: minor update

8.5 version 3.5.0

- feat: allow to specify geometric correction factor k during fit (tip position is multiplied by k and contact point is modified directly before and after fit)

8.6 version 3.4.0

- feat: allow to load fitting models from external Python files
- enh: add method to deregister NaniteFitModels

8.7 version 3.3.1

- docs: added some clarifications in the model docs
- enh: check for leading/trailing spaces in models
- enh: add more checks during loading models

8.8 version 3.3.0

- feat: introduce new NaniteFitModel class
- ref: deprecated get_anc_parms in favor of compute_anc_parms
- docs: add more information on how to write own model functions

8.9 version 3.2.1

- enh: more sensible default parameters for POC estimation fits

8.10 version 3.2.0

- feat: allow specifying the minimizer method for fitting (#21)
- feat: new contact point estimation with linear fit for baseline and polynomial fit for indentation part
- feat: new contact point estimation with Fréchet distance between curve and direct path in normalized coordinates
- tests: rename experimental test data files according to afmformats scheme (#20)

8.11 version 3.1.4

- docs: preprocessing methods not correctly rendered

8.12 version 3.1.3

- fix: make sure that AFM metadata contain the spring constant and raise a MissingMetaDataError if this is not the case
- ref: deprecate *IndentationPreprocessor* class in favor of a more flat submodule (#17)

8.13 version 3.1.2

- docs: add POC table to docs

8.14 version 3.1.1

- fix: do not modify preprocessing options when applying them (create a copy of the dictionary)
- ref: remove *Indentation.reset* in favor of *AFMData.reset_data*

8.15 version 3.1.0

- feat: new contact point estimation method "fit_constant_polynomial" which applies a piece-wise constant and polynomial fit
- feat: contact point estimation methods now return detailed information about the procedure (currently plottable data to understand the process)
- fix: spatial smoothing not working in some cases (#15)
- enh: add *steps_optional* in preprocessing to allow fine-grained control about order of application
- ref: remove "…smoothed" column data (which was never used anyway); instead, apply smoothing directly to AFMData subclass
- ref: rename *require_steps* to *steps_required* in preprocessing decorator
- setup: bump afmformats from 0.16.0 to 0.16.4

8.16 version 3.0.0

- BREAKING CHANGE: The contact point estimation method "scheme_2020" has been removed, although it has been the default for some time. It turns out that it does not perform so well and there are other more stable methods (to be implemented). Furthermore, some of the contact point estimation methods were improved so that basically many tests had to be updated. This will not break your analysis, it just means your contact points will change.
- feat: implement options for preprocessing methods

- feat: the “correct_tip_offset” preprocessing method now accepts the “method” argument (see new poc submodule)
- fix: contact point estimation with gradient-based method “poc_gradient_zero_crossing” did not really work
- enh: improve contact point estimation with “fit_constant_line”
- enh: speed-up contact point estimation with “deviation_from_baseline”
- ref: CLI profiles now use JSON format by default (old format still supported)
- ref: move contact point estimation to new ‘poc’ submodule

8.17 version 2.0.1

- enh: implement ‘require_steps’ in preprocessing to make sure that steps are executed in the correct order
- enh: add several helper functions for preprocessing

8.18 version 2.0.0

- BREAKING CHANGE: segment in FitProperties is now an integer
- setup: bump afmformats from 0.15.0 to 0.16.0
- docs: update doc strings for the “sneddon_spher_approx” model
- docs: remove duplcate docs for model functions

8.19 version 1.7.8

- ref: introduce preprocessing_step decorator for managing preprocessing steps
- ref: explicitly request “force-distance” data from afmformats (can be lifted by `setting nanite.read.DEFAULT_MODALITY` to `None`)
- setup: bump afmformats from 0.14.3 to 0.15.0 (initial support for loading creep-compliance data)

8.20 version 1.7.7

- docs: fix build

8.21 version 1.7.6

- setup: bump afmformats from 0.14.1 to 0.14.3 (adjust tests, speed)

8.22 version 1.7.5

- ref: migrate *QMap* and *Group* code to afmformats 0.14.1
- ref: *Indentation* is now a subclass for *afmformats.AFMForceDistance*
- ref: *QMap* is now a subclass for *afmformats.AFMQMap*
- ref: *Group* is now a subclass for *afmformats.AFMGroup*

8.23 version 1.7.4

- enh: allow passing metadata to the IndentationGroup initializer
- setup: bump afmformats from 0.10.2 to 0.13.2
- ref: deprecate get_data_paths in favor of afmformats.find_data

8.24 version 1.7.3

- build: move windows pipeline to GH Actions
- ref: better warning traceback for deprecated weight_cp method
- ref: DeprecationWarning: np.int from numpy 1.20

8.25 version 1.7.2

- build: use oldest-supported-numpy in pyproject.toml

8.26 version 1.7.1

- build: migrate to GitHub Actions

8.27 version 1.7.0

- enh: simplified writing new model functions by introducing default modeling and residual wrappers
- ref: improve code readability

8.28 version 1.6.3

- tests: fix fails due to tifffile upgrade
- setup: lift historic pinning of lmfit==0.9.5

8.29 version 1.6.2

- tests: improve coverage
- enh: add sanity checks for models during registration (#5)

8.30 version 1.6.1

- enh: if the contact point estimate is not possible, use a fit with a partially constant and linear function

8.31 version 1.6.0

- enh: improve contact point estimation by computing the gradient first; resolves issues with tilted baselines (#6) (This may affect fitting results slightly, hence the new minor release)

8.32 version 1.5.5

- setup: make tkinter optional for frozen applications

8.33 version 1.5.4

- setup: bump scikit-learn from 0.18.0 to 0.23.0 (different model results due to bugfixes, enhancements, or random sampling procedures; the tests have been updated accordingly)
- setup: bump afmformats from 0.10.0 to 0.10.2

8.34 version 1.5.3

- setup: new builds for Python 3.8

8.35 version 1.5.2

- enh: be more verbose when tip position cannot be computed
- setup: bump afmformats from 0.7.0 to 0.10.0

8.36 version 1.5.1

- setup: bump afmformats from 0.6.0 to 0.7.0 (metadata fixes)

8.37 version 1.5.0

- feat: IndentationGroup.get_enum returns a curve from an enum value
- setup: bump afmformats from 0.5.0 to 0.6.0 (hdf5 export, improved tab export)

8.38 version 1.4.1

- enh: set parameter *baseline* to "vary" for all models
- fix: make sure that *model_key* is set before *params_initial* when fitting with kwargs (otherwise, *params_initial* might reset)

8.39 version 1.4.0

- feat: add function *Indentation.get_rating_parameters*
- feat: compute additional ancillary parameter "Maximum indentation"
- feat: new functions *model.get_parm_unit* and updated *model.get_parm_name* to work with ancillary parameters as well

8.40 version 1.3.0

- feat: allow to define ancillary parameters for models and use them during fitting by default
- feat: *Indentation.get_initial_fit_parameters* now automatically computes common and model-related ancillary parameters if no initial parameters are present
- enh: allow to set the *model_key* in more functions of *Indentation*
- ref: use *idnt* to represent Indentation instances
- fix: preprocessing steps not stored in *Indentation.preprocessing*
- setup: bump afmformats from 0.4.1 to 0.5.0

8.41 version 1.2.4

- enh: update boundaries and default values for model parameters

8.42 version 1.2.3

- fix: FitProperties did not detect changes in "params_initial"

8.43 version 1.2.2

- setup: bump afmformats version from 0.3.0 to 0.4.1

8.44 version 1.2.1

- enh: skip computation of tip position if it is already in the dataset and cannot be computed e.g. due to missing spring constant
- fix: typo in get_data_paths_enum
- setup: bump afmformats version from 0.2.0 to 0.3.0

8.45 version 1.2.0

- tests: np.asscalar is deprecated
- ref: migrate to afmformats (#1)
- docs: minor improvements

8.46 version 1.1.2

- fix: add `__version__` property
- tests: use time.perf_counter for timing tests
- docs: improved LaTeX rendering

8.47 version 1.1.1

- setup: migrate to PEP 517 (pyproject.toml)
- docs: minor update

8.48 version 1.1.0

- feat: add contact point to available features in qmap visualization
- fix: avoid two invalid operations when computing features

8.49 version 1.0.1

- fix: invalid operation when loading data with a callback function

8.50 version 1.0.0

- docs: minor update

8.51 version 0.9.3

- enh: store nanite and h5py library versions in rating container
- enh: update hyperparameters of rating regressors
- ref: deprecation in h5py: replace dataset.value by dataset[...]

8.52 version 0.9.2

- ref: renamed the mode *model_hertz_parabolic* to *model_hertz_paraboloidal* to be consistent
- docs: updat code reference and other minor improvements

8.53 version 0.9.1

- fix: *preprocessing* keyword not working in *Indentation.fit_model*
- docs: add another scripting example and minor improvements
- tests: increase coverage

8.54 version 0.9.0

- ref: remove legacy "discrete" feature type
- ref: renamed kwargs for *Indetation.rate_quality*
- ref: new method *nanite.load_group* for loading experimental data
- ref: new class read.data.IndentationData for managing data
- ref: replace dataset.IndentationDataSet with group.IndentationGroup to avoid ambiguities
- fix: add missing "zef18" training set

- fix: sample weight computation failed when a rating level was missing
- enh: add *nanite-generate-training-set* command line program
- tests: reduce warnings and increase coverage
- cleanup: old docs in nanite.rate.io
- docs: major update using helper extensions

8.55 version 0.8.0

- initial release

CHAPTER

NINE

BILBLIOGRAPHY

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Bil92] G. G. Bilodeau. Regular pyramid punch problem. *Journal of Applied Mechanics*, 59(3):519, 1992. doi:10.1115/1.2893754.

[Dob18] Wolfgang Dobler. Truncated power series approximation for the relationship between indentation and force of a spherical indenter in atomic force microscopy. personal communication with Wolfgang Dobler, JPK Instruments, Berlin, November 2018.

[GMP+14] Michael Glaubitz, Nikolay Medvedev, Daniel Pussak, Laura Hartmann, Stephan Schmidt, Christiane A. Helm, and Mihaela Delcea. A novel contact model for AFM indentation experiments on soft spherical cell-like particles. *Soft Matter*, 10(35):6732, jun 2014. doi:10.1039/c4sm00788c.

[LL59] L. D. Landau and E. M. Lifshitz. *Theory of Elasticity*, chapter Fundamental Equations, pages 1–43. Volume 7. Pergamon Press, 1959.

[Lov39] A. E. H. Love. Boussinesq's problem for a rigid cone. *The Quarterly Journal of Mathematics*, os-10(1):161–175, 1939. doi:10.1093/qmath/os-10.1.161.

[MKH+19] Stephanie Möllmert, Maria A. Kharlamova, Tobias Hoche, Anna V. Taubenberger, Shada Abuhattum, Veronika Kuscha, Michael Brand, and Jochen Guck. Zebrafish spinal cord repair is accompanied by transient tissue stiffening. *(manuscript in preparation)*, 2019.

[MAM+19] Paul Müller, Shada Abuhattum, Stephanie Möllmert, Elke Ulbricht, Anna V. Taubenberger, and Jochen Guck. Nanite: using machine learning to assess the quality of atomic force microscopy-enabled nanoindentation data. *BMC Bioinformatics*, 20(1):1–9, sep 2019. doi:10.1186/s12859-019-3010-3.

[MMG18] Paul Müller, Stephanie Möllmert, and Jochen Guck. Atomic force microscopy indentation data of zebrafish spinal cord sections. *Figshare*, 11 2018. doi:10.6084/m9.figshare.7297202.v1.

[RZSK19] M. Rusaczonek, B. Zapotoczny, M. Szymonski, and J. Konior. Application of a layered model for determination of the elasticity of biological systems. *Micron*, 124:102705, sep 2019. doi:10.1016/j.micron.2019.102705.

[Sne65] Ian N. Sneddon. The relation between load and penetration in the axisymmetric boussinesq problem for a punch of arbitrary profile. *International Journal of Engineering Science*, 3(1):47–57, may 1965. doi:10.1016/0020-7225(65)90019-4.

PYTHON MODULE INDEX

n

INDEX

G

O

P

Q

R

S

V

W