ForceMetric Documentation

Jacob Seifert

September 7, 2018

1 Download ForceMetric

ForceMetric can be downloaded on github at `https://github.com/jcbs/ForceMetric`. The setup via pip has not been implemented yet but might come in the future. To use the package clone the repository with git and add it to your python search path, so that you can import the package from any path you use for your data analysis via `import ForceMetric as fm`. This can be done on

Linux Add in you `$HOME/.bashrc` the line:

```
export PYTHONPATH="$PYTHONPATH:Path/to/ForceMetric/"
```

Windows and iOs in Spyder Click on Spyder tools menu → PYTHONPATH manager and add the folder where your ForceMetric.py file is located.

2 The structure of ForceMetric

ForceMetric is library based on python classes and because hierarchical structures become quickly very complex this section briefly describes the underlying ForceMetric classes and their hierarchical relations.

To read the IgorPro '.ibw' files it is possible to use the binarywave package from the python igor library. However, using this package the data is not brought into an appropriate format, where the metadata and the data can be accessed easily. Figure 1 shows a summary of the classes which build up an easily accessible data format suitable to describe gls:afm data. Additionally, there are classes which create the framework for static and dynamic mechanical analysis.

In a first step, I translated the data structure as read from the igor.binarywave package into the `Wave` structure, where the metadata can be easily accessed by a dictionary structure. This is implemented in the `Header` class. Furthermore, the data can be accessed in the `Wave` class through the `getData` function. The `Wave` class is then used to create the `AFMScan` class to represent the structure and useful functions for gls:afm scans.

Figure 1: Representation of the programmed AFM library showing hierarchy and dependencies of the python classes as well as the underlying functions of the individual classes.

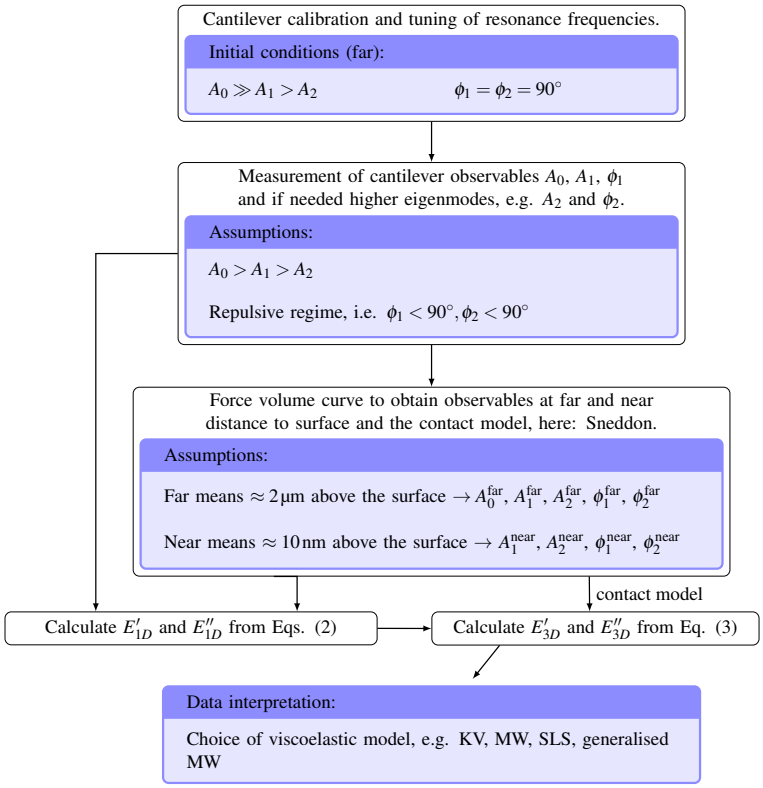

Figure 2: Processing flowchart including the assumptions for the experimental setup and calibration as well as the assumptions necessary for calculating E' and E''.

Similarly, I created a data structure for force-volume measurements that contains the class `FDIndices` and `FDData`. The first class only contains the indices at which sampling point the trace, dwell time, and retrace start and end. The second class returns the appropriate data in separate arrays for trace, retrace, and dwell time. In another step an object for static force measurements was created that includes calculating the contact point (class `ContactPoint`) and the Young's modulus (class `StaticYoung`). These two classes create in combination with the classes `Wave` and `FDData` the class ForceCurve, which provides the structure to analyse any kind of quantity vs distance data, including phase, amplitude, deflection, indentation, and force, with a focus on mechanical analysis. Finally, a structure for the analysis of dynamic mechanical analysis was created, represented by the class `DynamicViscoelastic`, and the translation of these one-dimensional values into three dimensional quantities, given by the class `DynamicYoung`. These two class create in combination with the class `AFMScan` the class `DynamicMechanicAFMScan`, which is suitable to translate amplitude and phase and appropriate calibration values into maps of dynamic mechanical properties.

To summarise, in order to obtain dynamic viscoelastic maps the following conditions and experimental order must be fulfilled:

- $A_0 \gg A_1 > A_2$ to ensure linearisation of the problem, i.e. contact area does not change significantly during oscillation (see also **??**)

- The experiment must take place in the repulsive regime, i.e. $\phi_1 < 90°$ and $\phi_2 < 90°$
- Force-volume curves must be obtained after every scan and record the parameters A_0, A_1, ϕ_1, and if measured in the experiment also higher eigenmodes. The calibration quantities near the surface should be measured at about 10 nm above the surface and the quantities far from the surface should be measured at least 2 µm above the surface. This curve is also important to determine the contact model, i.e. exponent γ in eq. (1).
- With the scan and the calibration quantities E'_{1D} and E''_{1D} can be calculated.
- With the contact model and the fitted Young's modulus E_0 from this model and the one dimensional quantities E'_{3D} and E''_{3D} can be determined

.

The choice of the viscoelastic model, e.g. gls:mw, gls:kv, gls:sls, is important for the data interpretation but not for the experiment.

The whole process, including mathematical and experimental conditions is summarized in a flowchart (fig. 2).

$$F = g\frac{E}{1-\nu^2}\delta^\gamma, \tag{1}$$

$$\delta_0 = \sqrt[\gamma]{\frac{F_0}{E_0}\frac{1-\nu^2}{g}} \tag{2a}$$

$$E'_{3D} = \frac{E'_{1D}}{\gamma}\sqrt[\gamma]{\frac{1-\nu^2}{g}}\sqrt[\gamma]{\frac{E_0}{F_0}}^{\gamma-1} \tag{2b}$$

$$E''_{3D} = \frac{E''_{1D}}{\gamma}\sqrt[\gamma]{\frac{1-\nu^2}{g}}\sqrt[\gamma]{\frac{E_0}{F_0}}^{\gamma-1}. \tag{2c}$$

$$E'_{3D} = \frac{E'_{1D}}{2}\sqrt{\frac{1-\nu^2}{\tan(\alpha)}\frac{\pi}{2}\frac{E_0}{F_0}} \tag{3a}$$

$$E''_{3D} = \frac{E''_{1D}}{2}\sqrt{\frac{1-\nu^2}{\tan(\alpha)}\frac{\pi}{2}\frac{E_0}{F_0}}. \tag{3b}$$

3 Module documentation and examples

3.1 Waves

Wave is the name Igor's binary data structure. Similarly to HDF5 (hierarchical data format 5) they are containers with data matrices and headers. The waves in ForceMetric convert the igor wave format into an easily accesible python structure. Waves are the foundation of all the other AFM data structures and therefore they will be treated first. A wave can be loaded by simply adding the filename to the class.

```
import ForceMetric as fm

wave = fm.Wave('Filename.ibw')
```

The Wave class is based on the Header class, which creates a header as a python dictionary. In this way information from the header can be read as `wave[key]` where the key is a string, e.g. for the cantilever spring constant `wave['SpringConstant']`. Alternatively, header information can be obtained by the getParam function as `wave.getParam('SpringConstant')`. This is also based on the dictionary and only for readability in the code. Because AFM headers usually contain a lot of information the Header class has a search function which returns all possible entries which contain a certain string, such as 'Spring' `wave.SearchHeader('Spring')`

gives as output:

```
['DisplaySpringConstant2',
'AdjustSpringConstant',
'Springconstant2',
'DisplaySpringConstant',
'SpringConstant']
```

It is important to note that the search function is case sensitive, so that `wave.SearchHeader('spring')` gives no output.

Finally, the data of the wave can be accessed with `wave.getData(key)` where the 'key' is a string describing the quantity, e.g. 'Defl', 'HeightRetrace' to obtain the deflection data in a force-volume curve or the height retrace from an AFM scan. The names of the keys can be found in `wave.labels`.

3.2 Force curves

Force-volume curves a based on the classes FDIndices and FDData, which both provide an appropriate structure for the force-curve. FDIndices simply returns the indices for trace (approach), retrace (retract), and if exists the dwell data of the experiment. Of course, also the whole data set can be returned. These different quantities can be accessed via the following commands:

```
FDData.Trace()
FDData.Retrace()
FDData.Dwell()
FDData.Data()
```

Force-volume curves are represented by the ForceCurve class, which is explicitly based on the Wave and StaticYoung classes and therefore has all the functions of them (cf. section 3.1 and below for StaticYoung). It also depends implicitly on the class FDData, which means that only certain variables are part of that class. These variables are the following:

- indentations
- force
- zsnr
- time
- deflection

- amp1
- amp2
- phase1
- phase2

While the first five quantities exit for every force curve, the lower four are only added if they were measured in the experiment. Further quantities can be added if measured but this is not done automatically. To add another quantity just put

```
fc = fm.ForceCurve(path)
fc.quantity = fm.FDData(fc.getData(key), fc.idxs))
```

So far all functions are from other classes but ForceCurve also has functions itself. It is possible to create an empty instance of ForceCurve by setting `path=None`. This instance can load a curve later by `fc.load(path)`. In this way it is also possible to reload a curve by `fc.load(fc.path)`.

The most important function to analyse force-volume data is to determine the contact point. Many procedures exist to do this as accurately as possible and the library has a few implemented from which some well known are the goodness of fit (gof) and ratio of variation (rov) method. A method I newly developed (unpublished) is based on the principle of virtual work, where I vary the force and the indentation, hence the name force indentation variation (fiv). These methods can be used to find the contact point

with the function correct (see example below). The function correct has the following parameters: `stds, method, fitrange, cix, fmin, fmax, surface_effect, surface_range`. The method requires a string, such as 'gof', 'rov', or 'fiv'. The parameters for stds and fitrange are for a very simplistic inaccurate method to determine the contact point, where the contact point is determined through a subtraction of a linear fit. The range of the linear fit is determined by fitrange, where `fitrange=0.6` does a linear fit of the first $SI60$. The contact point is then the value which is stds standard deviations above the subtracted baseline, i.e. `stds=4` means 4 standard deviations above the noise level. The parameter cix is in the case that the contact point is already known, e.g. determined manually. The parameters fmin and fmax are the minimal and maximal force to use for a goodness of fit approach. This assumes that the baseline starts at force zero. Finally, the parameters `surface_effect` and `surface_range` enable a correction of the force curve due to surface effects. This is particularly important for the force indentation variation method which always returns the first deflection as contact point and because hydrodynamic drag often leads to a deflection before a physical contact between the probe and the sample is established a correction of this effect is necessary. So far only a polynomial correction is implemented so that `surface_effect` gives the polynomial order of the subtracted surface effect and `surface_range` the percentage of the fit range. Note that these corrections are only to determine the contact point and do not change the data. Additionally, it should be mentioned that the fit range is only from the 'wrong' contact point caused by the surface effect.

Further functions are:

- `difference`: calculates the z-sensor difference between trace and retrace at the baseline (contact point position), i.e. the deformation of the material.
- `surface_idx`: determines the index where the contact is lost for the retrace, i.e. $F = 0$
- `get_idx_at`: determines the index closest to a given value of a quantity like force or indentation of the force-curve.
- `get_ind_at`: determines the indentation at a certain value of the deflection or force
- `get_force_at`: determines the force at a certain value of the deflection or indentation
- `ContactPhase`: determines the phase at the contact point
- `ContactAmp`: determines the amplitude at the contact point
- `plot`: plots the force curve with some default parameters

Finally, it is possible to determine Young's modulus from the force-curves. As indicated above this is done by the class `StaticYoung`, which is designed to determine Young's modulus for any force-indentation data and a few models based on contact mechanics are implemented, namely the Hertz (spherical indenter), Sneddon (conical indenter), flat

end of a cylindrical punch, and neohookean (for a spherical indenter). It is also possible to just fit the curve with empirical parameters but then the fitting parameter for the slope of the does not return Young's modulus.

```
"""Example for a force–volume curve to load it and find the contact point"""
import ForceMetric as fm
from matplotlib import pyplot as plt

if __name__ == "__main__":
    path = "LRCol0_MS_LG7d0001.ibw"

    # Default plot settings
    fc = fm.ForceCurve(path)
    fc.correct('fiv')
    fc.plot()

    # customized plots
    fig, ax = plt.subplots()
    ax.plot(fc.indentation.Trace() * 1e6, fc.force.Trace() * 1e9, c='r', lw=2)
    fig.suptitle('Custumized plot')
    ax.set_xlabel(r"$\delta$ (um)")
    ax.set_ylabel(r"$F$ (nN)")

    ax.grid()

    # determine and print Young's modulus
    E = fc.Young(model='sneddon', fmin=10e-9, fmax=500e-9)
    print("E = %.2e Pa" % E)

    # plot trace and fit
    fig1, ax1 = fc.PlotFit()
    fig1.suptitle("Plot Fit")

    plt.show()
```

3.3 Multiple force curves

To handle multiple force curves the class `Multicurve` can be used. Multicurve gets the path of one ForceCurve and then loads all other force curves in this directory which have the same basename (see example below).

Multicurve has a function to correct all curves and another one to fit all of them. Both functions can take the same arguments as the corresponding ones for single force curves (see above). Additionally, the class has a function (`mc.Scatter()`) which creates indentation and force arrays with all the data points to create a scatter plot of the data

set and it can plot a fit with the average fitting parameters and the rolling mean of the arrays with the function `mc.PlotAverageFit()`. The window of the rolling mean is 1 % of the total number of data points.

The Multicurve class can also be used for the analysis of force-volume maps by loading all curves, analyzing them, and reshaping the Young's modulus array, e.g.

```
mc = fm.Mulitcurve('path/to/force/curves.ibw')
mc.CorrectAll(method='fiv')
mc.FitAll(model='hertz')
E = mc.E.reshape(dimy, dimx)
```

The following code displays a full example file including analysis and plotting of several force curves aquired on alginate beads with a spherical indenter of radius $R = 10\,\mu m$. The data set is a set of repeated indentations on the same position.

```
"""Example to load multiple curves and fit the Young's modulus"""
import ForceMetric as fm
from matplotlib import pyplot as plt
import os
import numpy as np

if __name__ == "__main__":
    path = os.path.join("Data", "X60_3_x4_0100.ibw")
    mc = fm.Multicurve(path)  # loads all force curves with the name X60_3_x4_*.ibw
    Youngs = []

    mc.CorrectAll(method='fiv')
    mc.FitAll(model='Hertz', fmin=10e-9, fmax=100e-9)
    Youngs = mc.E

    mc.Scatter()  # creates indentation and force array for all curves to
                  # display scatter plot

    print("E = %.2e +- %.2e Pa" % (np.nanmean(Youngs), np.nanstd(Youngs)))

    fig, ax = plt.subplots()
    ax.plot(mc.indentation * 1e6, mc.force * 1e9, '.')
    ax.set_xlabel(r"$\delta$ (um)")
    ax.set_ylabel(r"$F$ (nN)")

    ax.grid()

    # plot fit
    mc.PlotAverageFit()
```

```
    plt.show()
```

3.4 AFM scans

AFM scans are important for any kind of AFM imaging and is represented by the class AFMScan. AFM scan is based on the Wave class and thus has all the functions of wave (section 3.1). Additionally, it has it's own functions, which are: `PlaneSubtraction, Display` and the experimental function `ProjectOnHeight`. The last function only works if line 27 to 32 are uncommented.