refellips

Hayden Robertson, Isaac Gresham, Andrew Nelson

Feb 17, 2023

CONTENTS:

refellips is a Python package designed for the analysis of variable angle spectroscopic ellipsometry (VASE) data. It is a flexible package built around the widely used refnx package used extensively for the analysis of neutron and X-ray reflectivity data.

The package makes use of TMM package written by Steven Byrnes to calculate ellipsometric parameters. The physics background behind those calculations can be found here.

Demonstrations of *refellips* are available here as well as on the GitHub repository.

The *refellips* package is free software distributed under the BSD 3-clause license. If you are interested in participating in this project, please use the GitHub repository; all contributions are welcomed.

CHAPTER

ONE

INSTALLATION

refellips has been tested on Python 3.8, 3.9. As *refellips* utilises the same *refnx* codebase, the *refnx* package and its dependencies must be also installed.

The *refellips* wheels are readily available on PyPI, it is a pure Python package.

```
pip install refellips
```

CHAPTER

TWO

GETTING STARTED

Here we will briefly demonstrate loading in ellipsometry datasets, creating a model and optimising that model to the dataset. For a more detailed description of model creation, please see the getting started tutorial on *refnx*.

2.1 Fitting an ellipsometry dataset

We begin by importing all of the relevant packages.

[1]:
```
import sys
import os
from os.path import join as pjoin
import numpy as np
import matplotlib.pyplot as plt
import scipy

import refnx
from refnx.analysis import CurveFitter
from refnx.reflect import Slab

import refellips
from refellips.dataSE import DataSE, open_EP4file
from refellips.reflect_modelSE import ReflectModelSE
from refellips.objectiveSE import ObjectiveSE
from refellips.structureSE import RI, Cauchy, load_material
```

For reproducibility, it is important to note the versions of software that you're using.

[2]:
```
print(
    f"refellips: {refellips.version.version}\n"
    f"refnx: {refnx.version.version}\n"
    f"scipy: {scipy.version.version}\n"
    f"numpy: {np.version.version}"
)
```

```
refellips: 0.0.3.dev0+92f1c50
refnx: 0.1.30
scipy: 1.8.0
numpy: 1.22.3
```

2.1.1 Loading a dataset

refellips has the capability of loading data directly from output files of both Accurion EP3 and EP4 ellipsometers, as well as Horiba ellipsometers using `open_EP4file` and `open_HORIBAfile` respectively.

Alternatively, other datasets can be imported using `DataSE`. The file must be formatted to contain four columns: wavelength, angle of incidence, psi and delta.

[3]:
```
pth = os.path.dirname(refellips.__file__)
dname = "testData1_11nm_PNIPAM_on_Si_EP4.dat"
file_path = pjoin(pth, "../", "demos", dname)
```

We will now use `DataSE` to import our dataset.

[4]:
```
data = DataSE(data=file_path)
```

2.1.2 Creating a model for our interface

As with *refnx*, `ComponentSE` objects are assembled into a `StructureSE` object which describes the interface. The simplest of these `ComponentSE` objects is a `SlabSE`, which is what we will use here.

We begin by loading in dispersion curves which describe the refractive index for each layer within our `StructureSE` (i.e., for each `ComponentSE`). *refellips* offers multiple ways to prescribe the refractive index of a layer. Here we will demonstrate the different ways to prescribe refractive indices. - `RI()`: Users can provide a refractive index (n) and extinction coefficient (k) for a given wavelength by `RI([n, k])` or alternatively, for spectroscopic analysis users load in their own dispersion curve of n,k as a function of wavelength by providing a path to the desired file as `RI('my_materials\material.csv')`. - `Cauchy()`: Creates a dispersion curve of a given material using the provided a, b and c values. - `load_material()`: Searches the *refellips* materials database for the provided material.

Note, dispersion curves provided in the *refellips* materials database are downloaded as a .csv file from refractiveindex.info. When providing a dispersion curve, files must contain at least two columns, assumed to be wavelength (in microns) and refractive index. If three columns are provided the third is loaded as the extinction coefficient. Further details on modelling material optical properties are provided on the FAQ page.

[5]:
```
si = load_material("silicon")
sio2 = RI([1.4563, 0])
PNIPAM = Cauchy(1.47, 0.00495)
air = RI(pjoin(pth, "materials/air.csv"))
```

Now we have defined the refractive indices of our layers, we can create a `Slab` object for each interfacial layer.

[6]:
```
# this is a 20 Angstrom layer
silica_layer = sio2(20)

polymer_layer = PNIPAM(200)
```

Each `Slab` has an associated thickness (as defined above), roughness, and volume fraction of solvent. As this is a dry film we will leave `vfsolv` as 0.

[7]:
```
silica_layer.name = "Silica"
silica_layer.thick.setp(vary=True, bounds=(1, 30))
silica_layer.vfsolv.setp(vary=False, value=0)

polymer_layer.name = "PNIPAM"
```

(continues on next page)

(continued from previous page)

```
polymer_layer.thick.setp(vary=True, bounds=(100, 500))
polymer_layer.vfsolv.setp(vary=False, value=0)
```

We now create the `Structure` by assembling the `Component` objects. Our structure is defined from fronting to backing, where the thickness of the fronting and backing are defined to be infinite (i.e., `np.inf`).

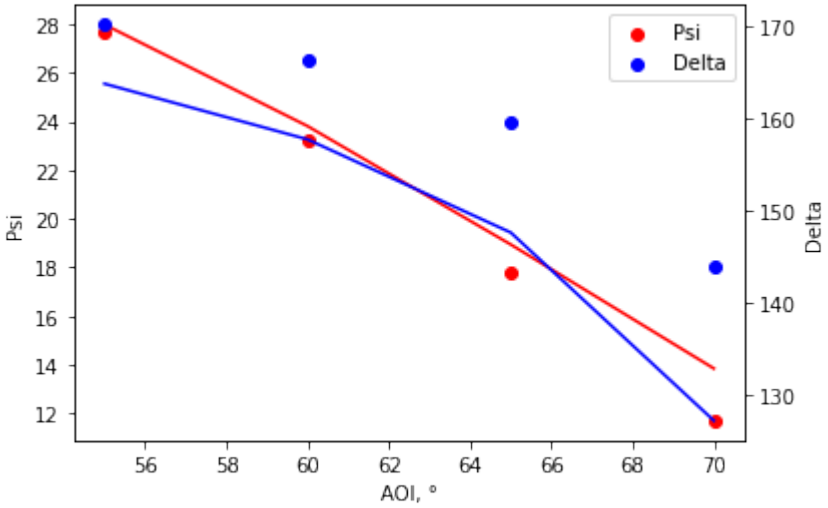
[8]:
```
structure = air() | polymer_layer | silica_layer | si()
```

Finally, we can create our model. A wavelength must be provided here, however, if your ellipsometry dataset contains a wavelength that will be automatically used. We have the option to define the `delta_offset` parameter here.

[9]:
```
model = ReflectModelSE(structure)

model.delta_offset.setp(value=0, vary=False, bounds=(-10, 10))
```

We can now have a quick preview of how our model compares to our dataset prior to fitting.

[10]:
```
fig, ax = plt.subplots()
axt = ax.twinx()

aois = np.linspace(50, 75, 100)

for dat in data.unique_wavelength_data():
    wavelength, aois, psi_d, delta_d = dat
    wavelength_aois = np.c_[np.ones_like(aois) * wavelength, aois]

    psi, delta = model(wavelength_aois)
    ax.plot(aois, psi, color="r")
    p = ax.scatter(data.aoi, data.psi, color="r")

    axt.plot(aois, delta, color="b")
    d = axt.scatter(data.aoi, data.delta, color="b")

ax.legend(handles=[p, d], labels=["Psi", "Delta"])
ax.set(ylabel="Psi", xlabel="AOI, °")
axt.set(ylabel="Delta")
plt.show()
```



2.1.3 Creating an objective

We will now create an objective. The `Objective` object is made by combining the model and the data, and is used to calculate statistics during the fitting process.

[11]:
```
objective = ObjectiveSE(model, data)
```

2.1.4 Fitting the data

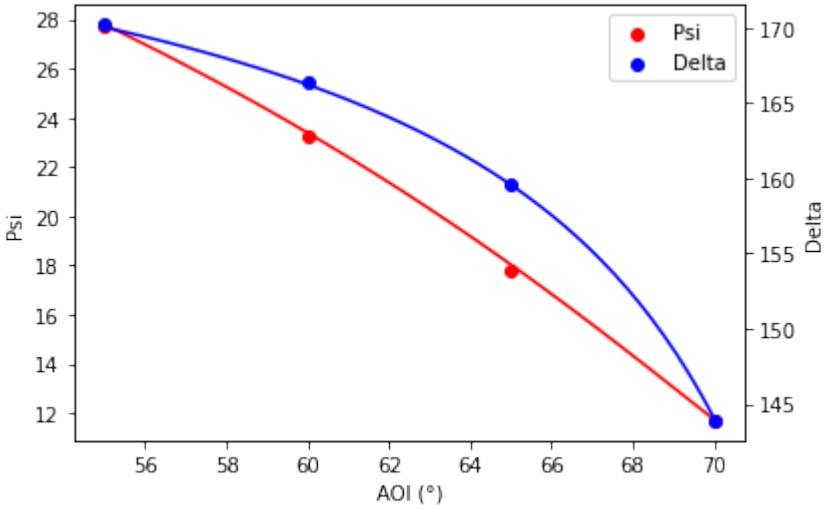
The optimisation of our `Objective` is performed by *refnx*'s `CurveFitter`. Data can be fit using a local optimisation such as `least_squares`, or a more global optimisation technique such as `differential_evolution`. For more information on the available fitting methods, see *refnx*.

[12]:
```
fitter = CurveFitter(objective)
fitter.fit(method="least_squares");
```

2.1.5 Optimised model and data post fit

We can now view our optimised objective, including our fit parameters. Users can plot this using the above method, or alternatively use the `objectiveSE.plot()` function.

[13]:
```
fig, ax = objective.plot()
```



[14]:
```
for i, x in enumerate(objective.model.parameters):
    print(x)
```

```
_____________________________________________________________________________________
Parameters: 'instrument parameters'
<Parameter:'delta offset' , value=0 (fixed)  , bounds=[-10.0, 10.0]>
_____________________________________________________________________________________
Parameters: 'Structure - '
_____________________________________________________________________________________
Parameters:       ''
<Parameter:  ' - thick'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:  ' - rough'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:' - volfrac solvent', value=0 (fixed)  , bounds=[0.0, 1.0]>
_____________________________________________________________________________________
Parameters:    'PNIPAM'
<Parameter:  ' - thick'   , value=132.339  +/- 982 , bounds=[100.0, 500.0]>
<Parameter: ' - cauchy A' , value=1.47 (fixed)  , bounds=[-inf, inf]>
<Parameter: ' - cauchy B' , value=0.00495 (fixed)  , bounds=[-inf, inf]>
<Parameter: ' - cauchy C' , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:  ' - rough'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:' - volfrac solvent', value=0 (fixed)  , bounds=[0.0, 1.0]>
_____________________________________________________________________________________
Parameters:    'Silica'
<Parameter:  ' - thick'   , value=1 +/- 1.01e+03, bounds=[1.0, 30.0]>
<Parameter:  ' - rough'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:' - volfrac solvent', value=0 (fixed)  , bounds=[0.0, 1.0]>
_____________________________________________________________________________________
Parameters:       ''
<Parameter:  ' - thick'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:  ' - rough'   , value=0 (fixed)  , bounds=[-inf, inf]>
<Parameter:' - volfrac solvent', value=0 (fixed)  , bounds=[0.0, 1.0]>
```

We can also view the resultant refractive index profile of the interface as well.

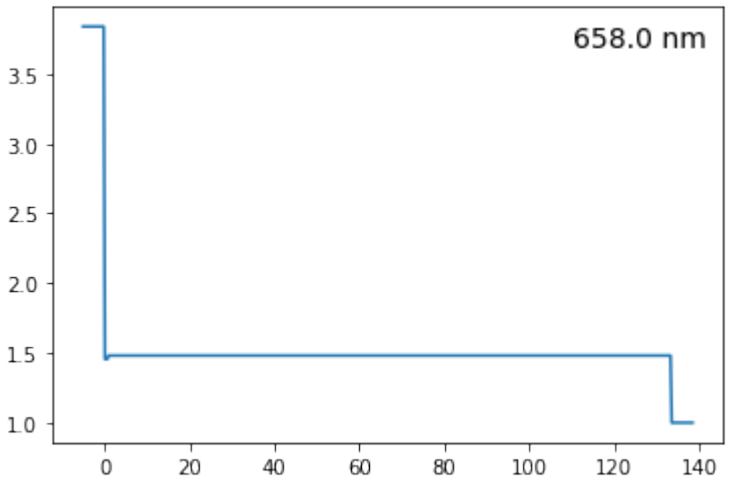
[15]:
```
structure.reverse_structure = True
plt.plot(*structure.ri_profile())
```

(continues on next page)

(continued from previous page)

```
plt.text(110, 3.7, f"{structure.wavelength} nm", fontsize=14)
```

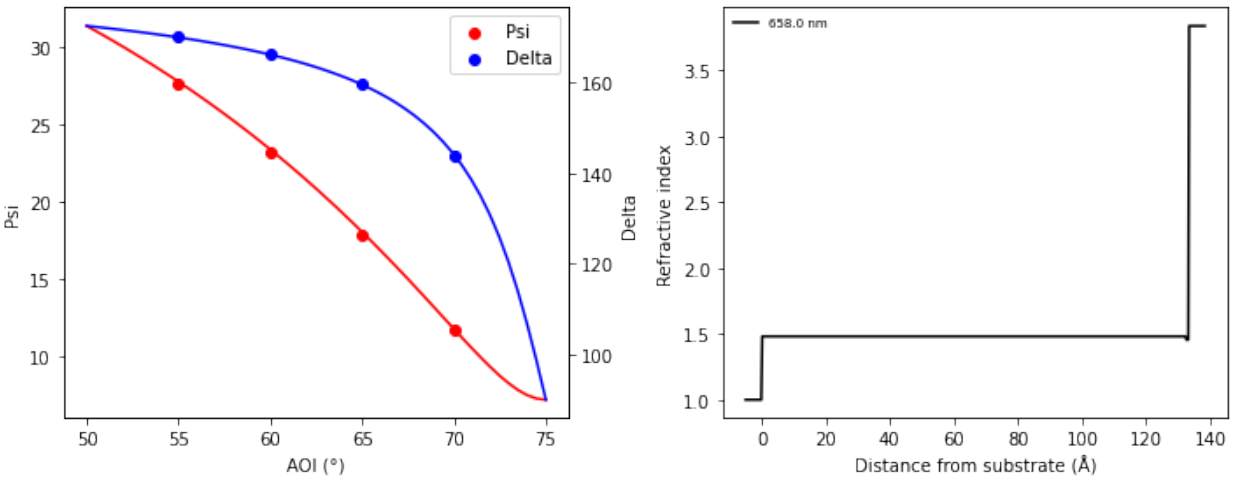
[15]:
```
Text(110, 3.7, '658.0 nm')
```



2.1.6 Using the plotting tools

[16]:
```
sys.path.insert(1, "../tools")
from plottools import plot_ellipsdata, plot_structure
```

[18]:
```
fig, ax = plt.subplots(1, 2, figsize=(10, 4))

plot_ellipsdata(ax[0], data=data, model=model, xaxis="aoi")
plot_structure(ax[1], objective=objective)

fig.tight_layout()
```



2.1.7 Saving the objective

If you would like to save the `Objective` or model to a file, this is best done through serialisation to a Python pickle.

[19]:
```
import pickle

pickle.dump(objective, open("my_objective.pkl", "wb"))
```

You can then simply reload your objective.

[20]:
```
objective = pickle.load(open("my_objective.pkl", "rb"))
```

If you would like to save your objective as a `.csv` file, this can be done as below.

[21]:
```
with open("my_objective.csv", "wb") as fh:
    data = objective.data
    wav = data.wavelength
    aoi = data.aoi
    psi_d = data.psi
    delta_d = data.delta
    for dat in data.unique_wavelength_data():
        wavelength, aois, psi_d, delta_d = dat
        wavelength_aois = np.c_[np.ones_like(aois) * wavelength, aois]
        psi_m, delta_m = objective.model(wavelength_aois)
    save_arr = np.array([wav, aoi, psi_d, delta_d, psi_m, delta_m])

    np.savetxt(
        fh,
        save_arr.T,
        delimiter=",",
        header="Wavelength, AOI, Measured Psi, Measured Delta, Modelled Psi, Modelled␣
↪Delta",
    )
```

CHAPTER

THREE

EXAMPLES

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS

A list of common questions.

4.1 What's the best way to ask for help or submit a bug report?

If you have any questions about using *refellips* or calculations performed by *refellips* please contact us or use the GitHub Issues tracker. If you find a bug in the code or documentation, please use GitHub Issues.

4.2 What are the 'fronting' and 'backing' media?

The 'fronting' and 'backing' media are infinite. The 'fronting' medium carries the incident beam of radiation, whilst the 'backing' medium will carry the transmitted beam away from the interface. In short, the fronting media is the medium that the radiation interacts with first, and the backing media is the medium which the radiation interacts with last.

For example, consider a system with an oxidised silicon wafer where the ambient material is air; the fronting media would be air and the backing media would be silicon.

4.3 What formats/types of ellipsometry data does *refellips* handle?

refellips has the capability of loading data directly from both Accurion EP3 and EP4 ellipsometers, as well Horiba ellipsometers using the *open_EP4file()* and *open_HORIBAfile()* functions, respectively.

Alternatively, users also have the option to load-in other datasets using *DataSE*. Files loaded using *DataSE* must contain four columns (with header): wavelength, angle of incidence, psi and delta.

4.4 Where do I find dispersion curves for a material?

refellips contains preloaded dispersion curves for select materials, which are accessible by the `load_material()` function. These materials are sourced from refractiveindex.info, and include air, a void, water, dimethyl sulfoxide, silicon, silica, gold, aluminium oxide, polystyrene, poly(N-isopropylacrylamide) (PNIPAM) and a material that represents a diffuse polymer.

If required, users can download their own dispersion curves from refractiveindex.info and load them into *refellips* using:

```
my_material = RI("my_dispersion.csv")
```

The loaded file must contain at least two columns, assumed to be wavelength (in microns) and refractive index. If three columns are provided, the third is loaded as the extinction coefficient. The *refellips* maintainers are happy to include additional dispersion curves with the package; please ask if you'd like this to happen.

Alternatively, users have the option to choose from any of the in-built oscillator functions to model the optical properties of their material: `Cauchy`, `Sellmeier`, `Lorentz`, `TaucLorentz` and `Gauss`. Both the *Cauchy* and *Sellmeier* oscillators monotonically decrease in refractive index with increasing wavelength and are therefore not Kramers-Kronig consistent. These optical models are frequently used to model the optical properties of transparent materials, however, the Sellmeier is more accurate at higher wavelengths, i.e., the infra-red region. Users can specify *Cauchy* and *Sellmeier* parameters for their material:

```
my_cauchy_material = Cauchy(A=a, B=b, C=c)
my_sellmeier_material = Sellmeier(Am, En, P, Einf)
```

The *Lorentz Tauc-Lorentz* and *Gaussian* functions are Kramers-Kronig consistent, and allow users to implement multiple oscillators. *Lorentz* oscillators are typically employed when working with materials above the fundamental band gap, describing well the optical properties of transparent and weakly absorbing materials. *Tauc-Lorentz* are often normally used for amorphous materials. *Gaussian* oscillators are typically used for absorbing materials, where the complex component models the Gaussian absorption and the real component is its Kramers-Kronig relation (a Hilbert transform). Users can implement a one *Lorentz* or *Tauc-Lorentz*, or a two *Gaussian* oscillator model for their material by:

```
my_lorentz_material = Lorentz([Am], [Br], [En], Einf)
my_TaucLorentz_material = TaucLorentz([Am], [C], [En], Eg, Einf)
my_gaussian_material = Gauss([Am_1, Am_2], [Br_1, Br_2], [En_1, En_2], Einf)
```

A demonstration on how to implement a user defined oscillator/dispersion curve is presented in the User defined oscillator notebook. Parameter values for Cauchy, Sellmeier, Lorentz and Tauc-Lorentz are provided by Horiba. Cauchy parameters can also be found on refractiveindex.info.

Alternatively, users can simply supply a refractive index (n) and extinction coefficient (k) for a single wavelength measurement:

```
my_material = RI([n, k])
```

4.5 How do I make my own dielectric function/dispersion curve?

A demonstration on how to implement a user defined oscillator/dispersion curve is presented in the User defined oscillator notebook.

4.6 What EMA methods does *refellips* provide?

refellips offers the three main methods of effective medium approximations (EMA): linear, Maxwell Garnett and Bruggeman. All EMA calculations performed in *refellips* are based on two-component mixing and done so using the complex dielectric function, not refractive indices and extinction coefficients.

For the examples below, ε_1 and f_1 relate to the complex dielectric function and volume fraction of the lower material (most commonly the host material) and ε_2 and f_2 relate to the complex dielectric function and volume fraction of the upper material (most commonly the inclusion material; e.g., solvent). It is important to note that $f_1 + f_2 = 1$.

For a linear EMA, the dielectric constant of the mixture is simply the sum of the products of the substituent dielectric

function and volume fraction (Equation (4.1)). *We hypothesise that the linear EMA will be sufficient for most use cases.*

$$\varepsilon_{\text{linear}} = f_1\varepsilon_1 + f_2\varepsilon_2 \tag{4.1}$$

For the Maxwell Garnett and Bruggeman EMA methods, a depolarisation factor (v) is included to account for potential electric field screening by anisotropic inclusions. When ($v = 1/3$), Equation (4.2) and (4.3) reduce down to the isotropic case, assuming all inclusions are spherical in nature. We anticipate that only expert users will use these EMA methods or alter the depolarisation factor.

The complex dielectric function for a mixed layer using the Maxwell-Garnett EMA is determined using Equation (4.2),

$$\varepsilon_{\text{MG}} = \varepsilon_1 \frac{\varepsilon_1 + (vf_1 + f_2)(\varepsilon_2 - \varepsilon_1)}{\varepsilon_1 + vf_1(\varepsilon_2 - \varepsilon_1)} \tag{4.2}$$

The Bruggeman EMA method is employed using Equation (4.3),

$$\varepsilon_{\text{BG}} = \frac{b + \sqrt{b^2 - 4(v-1)(e_1e_2v)}}{2(1-v)} \tag{4.3}$$

where $b = e_1(f_1 - v) + e_2(f_2 - v)$.

Further details surrounding these EMA methods and their derivations as well as the depolarisation factor and anisotropy are explored by both Markel and Humlicek.

4.7 Can I save models/objectives to a file?

Assuming that you have a *ReflectModelSE* or *ObjectiveSE* that you'd like to save to file, the easiest way to do this is via serialisation to a Python pickle:

```
import pickle
# save
with open('my_objective.pkl', 'wb+') as f:
    pickle.dump(objective, f)

# load
with open('my_objective.pkl', 'rb') as f:
    restored_objective = pickle.load(f)
```

The saved pickle files are in a binary format and are not human readable. It may also be useful to save the representation, `repr(objective)`.

Alternatively, modelled results can be exported into a *.csv* file. An example of this is provided in *Getting started*.

CHAPTER

FIVE

TESTIMONIALS

Please cite the *refellips* paper if you use it for data analysis in your own publications:

> "Robertson, H., Gresham, I.J., Prescott, S.W., Webber, G.B., Wanless, E.J., Nelson, A., 2022. *SoftwareX* 20, 101225. https://doi.org/10.1016/j.softx.2022.101225"

The following is a (possibly incomplete) list of publications that have used *refellips* let us know if your work should be included in this list or fork the repository and add it yourself.

1. Hayden Robertson, Isaac J Gresham, Stuart W Prescott, Grant B Webber, Erica J Wanless, and Andrew Nelson. refellips: A Python package for the analysis of variable angle spectroscopic ellipsometry data. *SoftwareX*, 20:101225, 2022. doi:10.1016/j.softx.2022.101225.

2. Hayden Robertson, Joshua D Willott, Kasimir P Gregory, Edwin C Johnson, Isaac J Gresham, Andrew R J Nelson, Vincent S J Craig, Stuart W Prescott, Robert Chapman, Grant B Webber, and Erica J Wanless. From Hofmeister to hydrotrope: Effect of anion hydrocarbon chain length on a polymer brush. *Journal of Colloid And Interface Science*, 634:983–994, 2023. doi:10.1016/j.jcis.2022.12.114.

3. Gregor Rudolph-Schöpping, Herje Schagerlöf, Ann-Sofi Jönsson, and Frank Lipnizki. Comparison of membrane fouling during ultrafiltration with adsorption studied by quartz crystal microbalance with dissipation monitoring (qcm-d). *Journal of Membrane Science*, 672:121313, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0376738822010584, doi:https://doi.org/10.1016/j.memsci.2022.121313.

CHAPTER

SIX

API REFERENCE

6.1 refellips

6.1.1 Modules

refellips.dataSE

" A basic representation of a 1D dataset

class refellips.dataSE.**DataSE**(*data=None*, *name=None*, *delimiter='\t'*, *reflect_delta=False*, ***kwds*)

Bases: object

A basic representation of a 1D dataset.

Parameters

- **data** (*{str, file-like, Path, tuple of np.ndarray}, optional*) – String pointing to a data file. Alternatively it is a tuple containing the data from which the dataset will be constructed. The tuple should have 4 members.
  - data[0] - Wavelength (nm)
  - data[1] - Angle of incidence (degree)
  - data[2] - Psi
  - data[3] - Delta

  data must be four long. All arrays must have the same shape.
- **mask** (*array-like*) – Specifies which data points are (un)masked. Must be broadcastable to the data. *Data1D.mask = None* clears the mask. If a mask value equates to *True*, then the point is included, if a mask value equates to *False* it is excluded.
- **reflect_delta** (*bool*) – Specifies whether delta values are reflected around 180 degrees (i.e., 360 - delta[delta > 180]), as is standard for some ellipsometry analysis packages (i.e., WVASE).

AOI

angle of incidence (degree)

Type
np.ndarray

mask

mask

Type
np.ndarray

filename

The file the data was read from

Type
str or None

weighted

Whether the y data has uncertainties

Type
bool

metadata

Information that should be retained with the dataset.

Type
dict

property aoi

Angle of incidence.

property data

4-tuple containing the (wavelength), AOI, psi, delta) data.

property delta

Ellipsometric parameter delta.

load(*f*)

Load a dataset from file. Must be a 4 column ASCII file with columns [wavelength, AOI, Psi, Delta].

Parameters
f (*file-handle or string*) – File to load the dataset from.

property psi

Ellipsometric parameter psi.

refresh()

Refreshes a previously loaded dataset.

save(*f*)

Save the data to file. Saves the data as a 4 column ASCII file.

Parameters
f (*file-handle or string*) – File to save the dataset to.

unique_wavelength_data()

Generator yielding wavelength, AOI, psi, delta tuples for the unique wavelengths in a dataset (i.e. all the data points for a given wavelength)

Return type
wavelength, AOI, psi, delta

property wavelength

refellips.dataSE.**custom_round**(*x*, *base=0.25*)

Perform rounding to a particular base. Default base is 0.25.

Parameters

- **x** (*DataFrame, array or list*) – Data to be rounded.
- **base** (*float*) – Base that the rounding will be with respect to.

Returns
Result of cutsom round

Return type
np.array

refellips.dataSE.**open_EP4file**(*fname*, *reflect_delta=False*)

Open and load in an Accurion EP4 formmated data file. Typically a .dat file.

Note: This file parser has been written for specific Accurion ellipsometers EP3 and EP4. No work has been done to ensure it is compatible with all Accurion ellipsometers. If you have trouble with this parser contact the maintainers through github.

Parameters

- **fname** (*file-handle or string*) – File to load the dataset from.
- **reflect_delta** (*bool*) – Option to reflect delta around 180 degrees (as WVASE would).

Returns
datasets – Structure containing wavelength, angle of incidence, psi and delta.

Return type
DataSE structure

refellips.dataSE.**open_HORIBAfile**(*fname*, *reflect_delta=False*, *lambda_cutoffs=[-inf, inf]*)

Opening and loading in a data file created by a Horiba ellipsometer. Data file loaded should be of the Horiba file format .spe.

Note: This file parser has been written for a specific ellipsometer, no work has been done to ensure it is compatable with all Horiba ellipsometers. If you have trouble with this parser contact the maintainers through github.

Parameters

- **fname** (*file-handle or string*) – File to load the dataset from.
- **reflect_delta** (*bool*) – Option to reflect delta around 180 degrees (as WVASE would).
- **lambda_cutoffs** (*list*) – Specifies the minimum and maximum wavelengths of data to be loaded. List has length 2.

Returns
DataSE – The data file structure from the loaded Horiba file.

Return type
DataSE structure

refellips.structureSE

class refellips.structureSE.**ComponentSE**(*name=''*)

Bases: Component

A base class for describing the structure of a subset of an interface.

Parameters
name (*str, optional*) – The name associated with the Component

Notes

By setting the *Component.interfaces* property one can control the type of interfacial roughness between all the layers of an interfacial profile.

class refellips.structureSE.**MixedSlabSE**(*thick*, *ri_A*, *ri_B*, *vf_B*, *rough*, *name=''*, *interface=None*)

Bases: *ComponentSE*

A slab component made of two materials.

Parameters

- **thick** (*refnx.analysis.Parameter or float*) – thickness of slab (Angstrom)
- **ri_A** (ScattererSE) – refractive index of first material
- **ri_B** (ScattererSE) – refractive index of second material
- **vf_B** (*float*) – volume fraction of B in the layer. Volume fraction of A is calculated as `1 - vf_B`.
- **rough** (*refnx.analysis.Parameter or float*) – roughness on top of this slab (Angstrom)
- **name** (*str*) – Name of this slab
- **interface** ({Interface, None}, optional) – The type of interfacial roughness associated with the Slab. If *None*, then the default interfacial roughness is an Error function (also known as Gaussian roughness).

property parameters

`refnx.analysis.Parameters` associated with this component

slabs(*structure=None*)

Slab representation of this component. See `Component.slabs`

class refellips.structureSE.**ScattererSE**(*name=''*, *wavelength=None*)

Bases: `Scatterer`

Abstract base class for something that will have a refractive index. Inherited from refnx.reflect.structure.Scatterer

complex(*wavelength*)

Calculate a complex RI

Parameters
wavelength (*float*) – wavelength of light in nm

Returns
RI – refractive index and extinction coefficient

Return type
complex

class refellips.structureSE.**SlabSE**(*thick*, *ri*, *rough*, *name=''*, *vfsolv=0*, *interface=None*)

Bases: *ComponentSE*

A slab component has uniform refractive index over its thickness

Parameters

- **thick** (*refnx.analysis.Parameter or float*) – thickness of slab (Angstrom)
- **ri** (`refellips.ScattererSE`) – (complex) RI of film

- **rough** (*refnx.analysis.Parameter or float*) – roughness on top of this slab (Angstrom)
- **name** (*str*) – Name of this slab
- **vfsolv** (*refnx.analysis.Parameter or float*) – Volume fraction of solvent [0, 1]
- **interface** ({Interface, None}, optional) – The type of interfacial roughness associated with the Slab. If *None*, then the default interfacial roughness is an Error function (also known as Gaussian roughness).

property parameters

refnx.analysis.Parameters associated with this component

slabs(*structure=None*)

Slab representation of this component. See Component.slabs

class refellips.structureSE.**StructureSE**(*components=()*, *name=''*, *solvent=None*, *reverse_structure=False*, *contract=0*, *wavelength=None*, *ema='linear'*, *depolarisation_factor=0.3333333333333333*)

Bases: Structure

Represents the interfacial Structure of an Ellipsometry sample. Successive Components are added to the Structure to construct the interface.

Parameters

- **components** (*sequence*) – A sequence of ComponentSE to initialise the Structure.
- **name** (*str*) – Name of this structure
- **solvent** (ScattererSE) – Specifies the refractive index of the solvent used for solvation. If no solvent is specified then the RI of the solvent is assumed to be the RI of *Structure[-1].slabs()[-1]* (after any possible slab order reversal).
- **reverse_structure** (*bool*) – If *StructureSE.reverse_structure* is *True* then the slab representation produced by *StructureSE.slabs* is reversed. The sld profile and calculated reflectivity will correspond to this reversed structure.
- **contract** (*float*) – If contract > 0 then an attempt to contract/shrink the slab representation is made. Use larger values for coarser profiles (and vice versa). A typical starting value to try might be 1.0.
- **wavelength** (*float, None*) – Wavelength the sample was measured at.
- **ema** (*{'linear', 'maxwell-garnett', 'bruggeman'}*) – Specifies the effective medium approximation for how the RI of a Component is mixed with the RI of the solvent. Further details regarding mixing are explained in the *slabs* method.
- **depolarisation_factor** (*float, int*) – The depolarisation factor is used only in the EMA calculations for the Maxwell-Garnett and Bruggeman methods. It describes the electric field screening: 0 prescribing no screening and 1 prescribing maximum screening.

append(*item*)

Append a Component to the Structure.

Parameters

item (*refnx.reflect.Component*) – The component to be added.

contract

float if contract > 0 then an attempt to contract/shrink the slab representation is made. Use larger values for coarser profiles (and vice versa). A typical starting value to try might be 1.0.

property depolarisation_factor

overall_ri(*slabs*, *solvent*)

Calculates the overall refractive index of the material and solvent RI in a layer.

Parameters

- **slabs** (*np.ndarray*) – Slab representation of the layers to be averaged.
- **solvent** (*complex or ScattererSE*) – RI of solvating material.

Returns
averaged_slabs – the averaged slabs.

Return type
np.ndarray

plot(*pvals=None*, *samples=0*, *fig=None*, *align=0*)

Plot the structure.

Requires matplotlib be installed.

Parameters

- **pvals** (*np.ndarray, optional*) – Numeric values for the Parameter's that are varying
- **samples** (*number*) – If this structures constituent parameters have been sampled, how many samples you wish to plot on the graph.
- **fig** (*Figure instance, optional*) – If *fig* is not supplied then a new figure is created. Otherwise the graph is created on the current axes on the supplied figure.
- **align** (*int, optional*) – Aligns the plotted structures around a specified interface in the slab representation of a Structure. This interface will appear at z = 0 in the sld plot. Note that Components can consist of more than a single slab, so some thought is required if the interface to be aligned around lies in the middle of a Component. Python indexing is allowed, e.g. supplying -1 will align at the backing medium.

Returns
fig, ax – *matplotlib* figure and axes objects.

Return type
matplotlib.Figure, matplotlib.Axes

reflectivity()

Calculate theoretical reflectivity of this structure

Parameters

- **q** (*array-like*) – Q values (Angstrom**-1) for evaluation
- **threads** (*int, optional*) – Specifies the number of threads for parallel calculation. This option is only applicable if you are using the _creflect module. The option is ignored if using the pure python calculator, _reflect. If *threads == 0* then all available processors are used.

Notes

Normally the reflectivity will be calculated using the Nevot-Croce approximation for Gaussian roughness between different layers. However, if individual components have non-Gaussian roughness (e.g. Tanh), then the overall reflectivity and SLD profile are calculated by micro-slicing. Micro-slicing involves calculating the specific SLD profile, dividing it up into small-slabs, and calculating the reflectivity from those. This normally takes much longer than the Nevot-Croce approximation. To speed the calculation up the *Structure.contract* property can be used.

ri_profile(*z=None*, *align=0*, *max_delta_z=None*)

Calculates an RI profile, as a function of distance through the interface.

Parameters

- **z** (*float*) – Interfacial distance (Angstrom) measured from interface between the fronting medium and the first layer.
- **align** (*int, optional*) – Places a specified interface in the slab representation of a Structure at z = 0. Python indexing is allowed, e.g. supplying -1 will place the backing medium at z = 0.
- **max_delta_z** (*{None, float}, optional*) – If specified this will control the maximum spacing between SLD points. Only used if *z is None*.

Returns
ri – refractive index

Return type
float

Notes

This can be called in vectorised fashion.

slabs(***kwds*)

The slab representation of this structure.

Returns

slabs – Slab representation of this structure. Has shape (N, 5).

- **slab[N, 0]**
  thickness of layer N
- **slab[N, 1]**
  overall RI.real of layer N (material AND solvent)
- **slab[N, 2]**
  overall RI.imag of layer N (material AND solvent)
- **slab[N, 3]**
  roughness between layer N and N-1
- **slab[N, 4]**
  volume fraction of solvent in layer N.

Return type
np.ndarray

Notes

If *Structure.reversed is True* then the slab representation order is reversed. The slab order is reversed before the solvation calculation is done. I.e. if *Structure.solvent == 'backing'* and *Structure.reversed is True* then the material that solvates the system is the component in *Structure[0]*, which corresponds to *Structure.slab[-1]*.

Users can simulate mixing between two adjacent layers by specifying a volume fraction of solvent (*vfsolv*). The *overall_ri* function then performs the EMA using the specified method: 'linear', 'maxwell-garnett' or 'bruggeman'. All EMA calculations are performed by using the complex dielectric function (i.e., square of refractive index and extinction coefficient). For a host layer (e_h) with volume fraction (vf) of impurities (e_i), the overall RI is calculated by

```
>>> StructureSE.ema = 'linear'
e_linear = e_h * (1 - vf) + e_i * vf
```

```
>>> StructureSE.ema = 'maxwell-garnett'
>>> StructureSE.depolarisation_factor = 1/3
top = e_h + (depolarisation_factor * (1 - vf) + vf) * (e_i - e_h)
bottom = e_h + depolarisation_factor * (1 - vf) * (e_i - e_h)
e_MG = e_h * top_r / bottom_r
```

```
>>> StructureSE.ema = 'bruggeman'
>>> StructureSE.depolarisation_factor = 1/3
b = e_h * ((1 - vf) - depolarisation_factor) + e_i * (vf - depolarisation_
↪factor)
e_BG = (b + np.sqrt(b**2 - 4 * (depolarisation_factor - 1) *
                          (vf * e_h * e_i * depolarisation_factor
                           ))) / (2 * (1 - depolarisation_factor))
```

sld_profile = None

property solvent

refellips.structureSE.**nm_eV_conversion**(*val*)

Convert wavelength from nm to eV -or- Convert wavelength from nm to eV

It does both. Visible light has a range of energies from 1.77 (red) to 3.26 (blue) eV

refellips.structureSE.**overall_ri**(*ri_A*, *ri_B*, *vf_B=0.0*, *ema='linear'*, *depolarisation_factor=0.3333333333333333*)

Calculates the overall refractive index of two materials.

Parameters

- **ri_A** (*complex, array-like*) – RI of material A
- **ri_B** (*complex*) – RI of material B
- **vf_B** (*float, optional*) – volume fraction of material B. The volume fraction of A is calculated as `1 - vf_B`.
- **ema** (*{'linear', 'maxwell-garnett', 'bruggeman'}*) – Specifies how refractive indices are mixed together.
- **depolarisation_factor** (*float, optional*) – Depolarisation factor. Default is 1/3.

Returns

ri_avg – the averaged material RI

Return type
complex

refellips.objectiveSE

class refellips.objectiveSE.**ObjectiveSE**(*model*, *data*, *lnsigma=None*, *use_weights=True*, *transform=None*, *logp_extra=None*, *name=None*)

Bases: `BaseObjective`

Objective function for using with curvefitters such as *refnx.analysis.curvefitter.CurveFitter*.

Parameters

- **model** (*refnx.analysis.Model*) – the residuals model function. One can also provide an object that inherits *refnx.analysis.Model*.
- **data** (*refnx.dataset.Data1D*) – data to be analysed.
- **lnsigma** (*float or refnx.analysis.Parameter, optional*) – Used if the experimental uncertainty (*data.y_err*) underestimated by a constant fractional amount. The experimental uncertainty is modified as:

  *s_n**2 = y_err**2 + exp(lnsigma * 2) * model**2*

  See *Objective.logl* for more details.
- **use_weights** (*bool*) – use experimental uncertainty in calculation of residuals and logl, if available. If this is set to False, then you should also set *self.lnsigma.vary = False*, it will have no effect on the fit.
- **transform** (*callable, optional*) – the model, data and data uncertainty are transformed by this function before calculating the likelihood/residuals. Has the signature *transform(data.x, y, y_err=None)*, returning the tuple (*transformed_y, transformed_y_err*).
- **logp_extra** (*callable, optional*) – user specifiable log-probability term. This contribution is in addition to the log-prior term of the *model* parameters, and *model.logp*, as well as the log-likelihood of the *data*. Has signature: *logp_extra(model, data)*. The *model* will already possess updated parameters. Beware of including the same log-probability terms more than once.
- **name** (*str*) – Name for the objective.

Notes

For parallelisation *logp_extra* needs to be picklable.

chisqr(*pvals=None*)

Calculates the chi-squared value for a given fitting system.

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

Returns
chisqr – Chi-squared value, *np.sum(residuals**2)*.

Return type
np.ndarray

covar()

Estimates the covariance matrix of the curvefitting system.

Returns
covar – Covariance matrix

Return type
np.ndarray

logl(*pvals=None*)

Calculate the log-likelihood of the system.

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

Returns
logl – log-likelihood probability

Return type
float

Notes

The log-likelihood is calculated as:

```
logl = -0.5 * np.sum(((y - model) / s_n)**2
                     + np.log(2 * pi * s_n**2))
logp += self.model.logp()
logp += self.logp_extra(self.model, self.data)
```

where

```
s_n**2 = y_err**2 + exp(2 * lnsigma) * model**2
```

At the moment s_n**2, the variance of the measurement uncertainties, is assumed to be unity. A future release may implement those uncertainties

logp(*pvals=None*)

Calculate the log-prior of the system.

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

Returns
logp – log-prior probability

Return type
float

Notes

The log-prior is calculated as:

```
logp = np.sum(param.logp() for param in
                  self.varying_parameters())
```

logpost(*pvals=None*)

Calculate the log-probability of the curvefitting system

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

Returns
logpost – log-probability

Return type
float

Notes

The overall log-probability is the sum of the log-prior and log-likelihood. The log-likelihood is not calculated if the log-prior is impossible (*logp == -np.inf*).

property npoints

int the number of points in the dataset.

property parameters

refnx.analysis.Parameters, all the Parameters contained in the fitting system.

pgen(*ngen=1000*, *nburn=0*, *nthin=1*)

Yield random parameter vectors from the MCMC samples. The objective state is not altered.

Parameters

- **ngen** (*int, optional*) – the number of samples to yield. The actual number of samples yielded is *min(ngen, chain.size)*
- **nburn** (*int, optional*) – discard this many steps from the start of the chain
- **nthin** (*int, optional*) – only accept every *nthin* samples from the chain

Yields
pvec (*np.ndarray*) – A randomly chosen parameter vector

plot(*xaxis=None*, *plot_labels=True*, *fig=None*)

Plot the data/model.

Requires matplotlib be installed.

Parameters

- **xaxis** (*String, optional*) – Either 'aoi' or 'wavelength'. If none specified, 'wavelength' will be chosen unless there is more than 1 unique aoi.
- **plot_labels** (*Bool, optional*) – Whether to plot axis labels. The default is True.
- **fig** (*Figure instance, optional*) – If *fig* is not supplied then a new figure is created. Otherwise the graph is created on the current axes on the supplied figure.

Returns
fig, ax – *matplotlib* figure and axes objects.

Return type
`matplotlib.Figure`, `matplotlib.Axes`

`prior_transform`(*u*)

Calculate the prior transform of the system.

Transforms uniform random variates in the unit hypercube, *u ~ uniform[0.0, 1.0)*, to the parameter space of interest, according to the priors on the varying parameters.

Parameters
u (*array-like*) – Size of the varying parameters

Returns
pvals – Scaled parameter values

Return type
array-like

Notes

If a parameter has bounds, *x ~ Unif[-10, 10)* then the scaling from *u* to *x* is done as follows:

```
x = 2. * u - 1.  # scale and shift to [-1., 1.)
x *= 10.  # scale to [-10., 10.)
```

`residuals`(*pvals=None*)

Calculates the residuals for a given fitting system.

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

Returns
residuals – Residuals, *(data.y - model) / y_err*.

Return type
np.ndarray

`setp`(*pvals*)

Set the parameters from pvals.

Parameters
pvals (*array-like or refnx.analysis.Parameters*) – values for the varying or entire set of parameters

`varying_parameters`()

Returns
varying_parameters – The varying Parameter objects allowed to vary during the fit.

Return type
refnx.analysis.Parameters

`property weighted`

bool Does the data have weights (*data.y_err*), and is the objective using them?

refellips.reflect_modelSE

refellips.reflect_modelSE.**Delta_Psi_TMM**(*AOI*, *layers*, *wavelength*, *delta_offset*, *reflect_delta=False*)

Get delta and psi using the transfer matrix method.

Parameters

- **AOI** (*array_like*) – the angle of incidence values required for the calculation. Units = degrees
- **Wavelength** (*float*) – Wavelength of light. Units = nm
- **layers** (*np.ndarray*) – coefficients required for the calculation, has shape (2 + N, 4), where N is the number of layers layers[0, 1] - refractive index of fronting layers[0, 2] - extinction coefficent of fronting layers[N, 0] - thickness of layer N layers[N, 1] - refractive index of layer N layers[N, 2] - extinction coefficent of layer N layers[N, 3] - roughness between layer N-1/N (IGNORED!) layers[-1, 1] - refractive index of backing layers[-1, 2] - extinction coefficent of backing layers[-1, 3] - roughness between backing and last layer (IGNORED!)

Returns

- **Psi** (*np.ndarray*) – Calculated Psi values for each aoi value.
- **Delta** (*np.ndarray*) – Calculated Delta values for each aoi value.

class refellips.reflect_modelSE.**ReflectModelSE**(*structure*, *delta_offset=0*, *name=None*)

Bases: object

Parameters

- **structure** (*refnx.reflect.Structure*) – The interfacial structure.
- **name** (*str, optional*) – Name of the Model

logp()

Additional log-probability terms for the reflectivity model. Do not include log-probability terms for model parameters, these are automatically included elsewhere.

Returns
logp – log-probability of structure.

Return type
float

model(*wavelength_aoi*, *p=None*)

Calculate the ellipsometric values (psi, delta) of this model

Parameters

- **wavelength_aoi** (*array-like*) – An array of shape (N, 2) corresponding to the wavelengths (nm) and angle of incidences (deg) the ellipsometric measurements were performed at.
- **p** (*refnx.analysis.Parameters, optional*) – parameters required to calculate the model

Returns
psi, delta – Calculated ellipsometric parameters

Return type
np.ndarray

property parameters

refnx.analysis.Parameters - parameters associated with this model.

property structure

refnx.reflect.Structure - object describing the interface of a reflectometry sample.

refellips.reflect_modelSE.**coh_tmm**(*n_list*, *d_list*, *th_0*, *lam_vac*)

Code adapted by that of Byrnes - see https://arxiv.org/abs/1603.02720

n_list is the list of refractive indices, in the order that the light would pass through them. The 0'th element of the list should be the semi-infinite medium from which the light enters, the last element should be the semi- infinite medium to which the light exits (if any exits).

th_0 is the angle of incidence: 0 for normal, pi/2 for glancing. Remember, for a dissipative incoming medium (n_list[0] is not real), th_0 should be complex so that n0 sin(th0) is real (intensity is constant as a function of lateral position).

d_list is the list of layer thicknesses (front to back). Should correspond one-to-one with elements of n_list. First and last elements should be "inf".

lam_vac is vacuum wavelength of the light.

refellips.reflect_modelSE.**interface_r_p**(*n_i*, *n_f*, *th_i*, *th_f*)

refellips.reflect_modelSE.**interface_r_s**(*n_i*, *n_f*, *th_i*, *th_f*)

refellips.reflect_modelSE.**interface_t_p**(*n_i*, *n_f*, *th_i*, *th_f*)

refellips.reflect_modelSE.**interface_t_s**(*n_i*, *n_f*, *th_i*, *th_f*)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

r

INDEX

R

S

U

V

W