likelihood

April 14, 2021

0.1 Likelihood sampling

Likelihood sampling of reflectometry models is a common tool in the analysis of specular reflectometry data. The Python programming language has a powerful infrastructure for this modelling, including packages such as PyMC3 and PyStan. Here, we will show how the emcee Python may be used to enable Bayesian sampling in BornAgain and the differential evoulation optimisation algorithm from the scipy.

First, we will import the necessary packages.

[1]:
```
from os import path
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import differential_evolution
import emcee
import corner
import bornagain as ba
from bornagain import ba_fitmonitor
```

The system that we are investigating in this tutorial is a Ni-Ti multilayer material at the interface between a Si substrate and a vacuum. For, this example, there are two parameters we are interested in, the thickness of the Ni and Ti layers. We know the scattering length densities for each, and that in total there are 10 repetitions of the Ni-Ti sandwich.

[2]:
```
def get_sample(ni_thickness, ti_thickness):
    """
    Creates a sample and returns it
    :float ni_thickness: a value of the Ni thickness in nanometres
    :float ti_thickness: a value of the Ti thickness in nanometres
    :return: the sample defined
    """

    # substrate (Si)
    si_sld_real = 2.0704e-06  # \AA^{-2}

    # layers' parameters
    n_repetitions = 10
    # Ni
    ni_sld_real = 9.4245e-06  # \AA^{-2}
```

```
    # Ti
    ti_sld_real = -1.9493e-06  # \AA^{-2}

    # defining materials
    m_vacuum = ba.MaterialBySLD()
    m_ni = ba.MaterialBySLD("Ni", ni_sld_real, 0)
    m_ti = ba.MaterialBySLD("Ti", ti_sld_real, 0)
    m_substrate = ba.MaterialBySLD("SiSubstrate", si_sld_real, 0)

    # vacuum layer and substrate form multi layer
    vacuum_layer = ba.Layer(m_vacuum)
    ni_layer = ba.Layer(m_ni, ni_thickness)
    ti_layer = ba.Layer(m_ti, ti_thickness)
    substrate_layer = ba.Layer(m_substrate)
    multi_layer = ba.MultiLayer()
    multi_layer.addLayer(vacuum_layer)
    for i in range(n_repetitions):
        multi_layer.addLayer(ti_layer)
        multi_layer.addLayer(ni_layer)
    multi_layer.addLayer(substrate_layer)
    return multi_layer
```

Having created a function to return the multilayer sample, we can now have another function that will obtain our data. For this to work locally, it will be necessary that you download the experimental data file and place it in the same directory as the python script. Then, you can add the following data collection function.

[3]:
```
def get_real_data():
    """
    Loading data from genx_interchanging_layers.dat
    Returns a Nx3 array (N - the number of experimental data entries)
    with first column being coordinates,
    second one being values, and the third the uncertainties.
    """
    if not hasattr(get_real_data, "data"):
        filepath = "../../../../../static/files/python/fitting/
↪ex03_ExtendedExamples/specular/genx_interchanging_layers.dat.gz"
        real_data = np.loadtxt(filepath, usecols=(0, 1, 2), skiprows=3)
        # translating axis values from double incident angle (degs)
        # to incident angle (radians)
        real_data[:, 0] *= np.pi/360
        # setting artificial uncertainties (uncertainty sigma equals a half
        # of experimental data value)
        real_data[:, 2] = real_data[:, 1] * 0.1
        get_real_data.data = real_data
    return get_real_data.data.copy()
```

It is then necessary that a simulation of the specular reflectometry is created, since the data in

the `genx_interchanging_layers.dat.gz` above is on an angular abscissa scale, we will use an `AngularSpecScan` below.

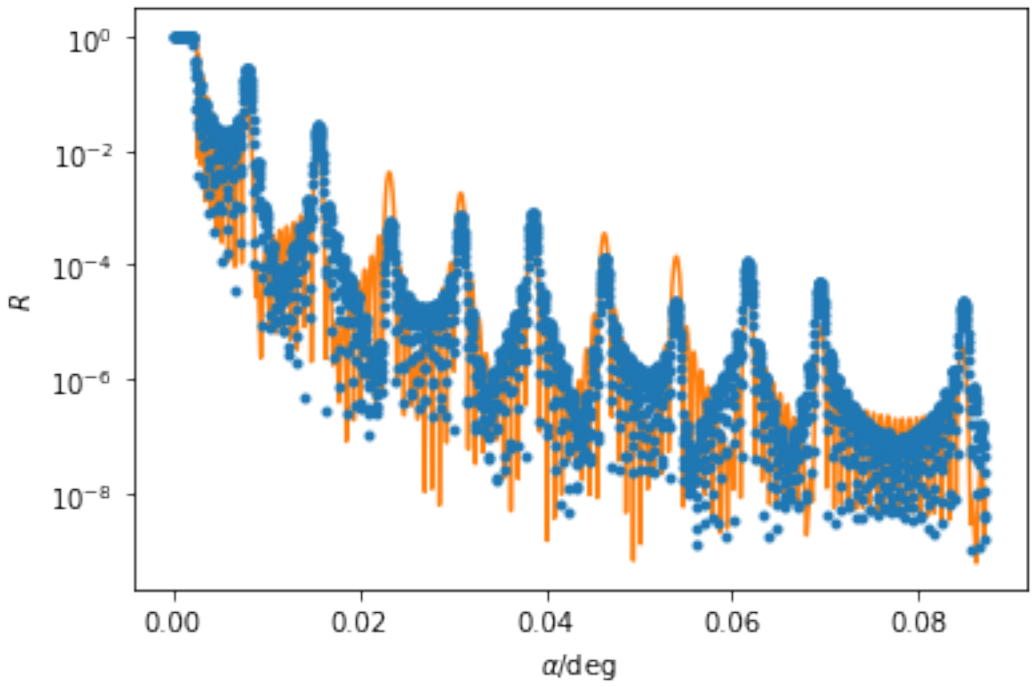
[4]:
```
def get_simulation(alpha):
    """
    Defines and returns a specular simulation.
    """
    wavelength = 1.54*ba.angstrom
    scan = ba.AngularSpecScan(wavelength,alpha)
    simulation = ba.SpecularSimulation()
    simulation.setScan(scan)
    return simulation
```

The final function that is necessary is the simulation of specular reflectometry. This will take the alpha-values to be simulated and thickness for the Ni and Ti layers and then return the result of the simulation as a `numpy.array`.

[5]:
```
def run_simulation(alpha, ni_thickness, ti_thickness):
    """
    Runs simulation and returns its result.
    :array q: q-values to be simulated
    :float ni_thickness: a value of the Ni thickness
    :float ti_thickness: a value of the Ti thickness
    :return: simulated reflected intensity
    """
    sample = get_sample(ni_thickness, ti_thickness)
    simulation = get_simulation(alpha)
    simulation.setSample(sample)
    simulation.runSimulation()
    return simulation.result().array()
```

It is then possible to plot some guess for the thicknesses of the layers along with our data.

[6]:
```
plt.errorbar(get_real_data()[:, 0], get_real_data()[:, 1], get_real_data()[:,␣
 ↪2], marker='.', ls='')
plt.plot(get_real_data()[:, 0], run_simulation(get_real_data()[:, 0], 9.0, 1.
 ↪0), '-')
plt.xlabel('$\\alpha$/deg')
plt.ylabel('$R$')
plt.yscale('log')
plt.show()
```



We will use the `emcee` package to sampling the likelihood of the data (we are follow their data fitting example). Therefore, it is necessary to define a likelihood objective.

[7]:
```
def log_likelihood(theta, x, y, yerr):
    """
    Calculate the log-likelihood for the normal uncertainties

    :tuple theta: the variable parameters
    :array x: the abscissa data (q-values)
    :array y: the ordinate data (R-values)
    :array x: the ordinate uncertainty (dR-values)
    :return: log-likelihood
    """
    model = run_simulation(x, *theta)
    sigma2 = yerr ** 2 + model ** 2
    return -0.5 * np.sum((y - model) ** 2 / sigma2 + np.log(sigma2))
```

First, we will find the maximum likleihood solution using a differential evolution algorithm from the `scipy.optimize` library.

[8]:
```
nll = lambda *args: -log_likelihood(*args)
initial = np.array([9.0, 1.0]) + 0.1 * np.random.randn(2)
soln = differential_evolution(nll, ((5.0, 9.0), (1.0, 10.0)),␣
 ↪args=(get_real_data()[:, 0], get_real_data()[:, 1], get_real_data()[:, 2]))
ni_thickness_ml, ti_thickness_ml = soln.x
```

```
print(ni_thickness_ml, ti_thickness_ml)
```

```
7.000335689791421 2.9998253518340725
```

Then we can use the `emcee.EnsembleSampler` to probe the uncertainties in the parameters and there correlation.

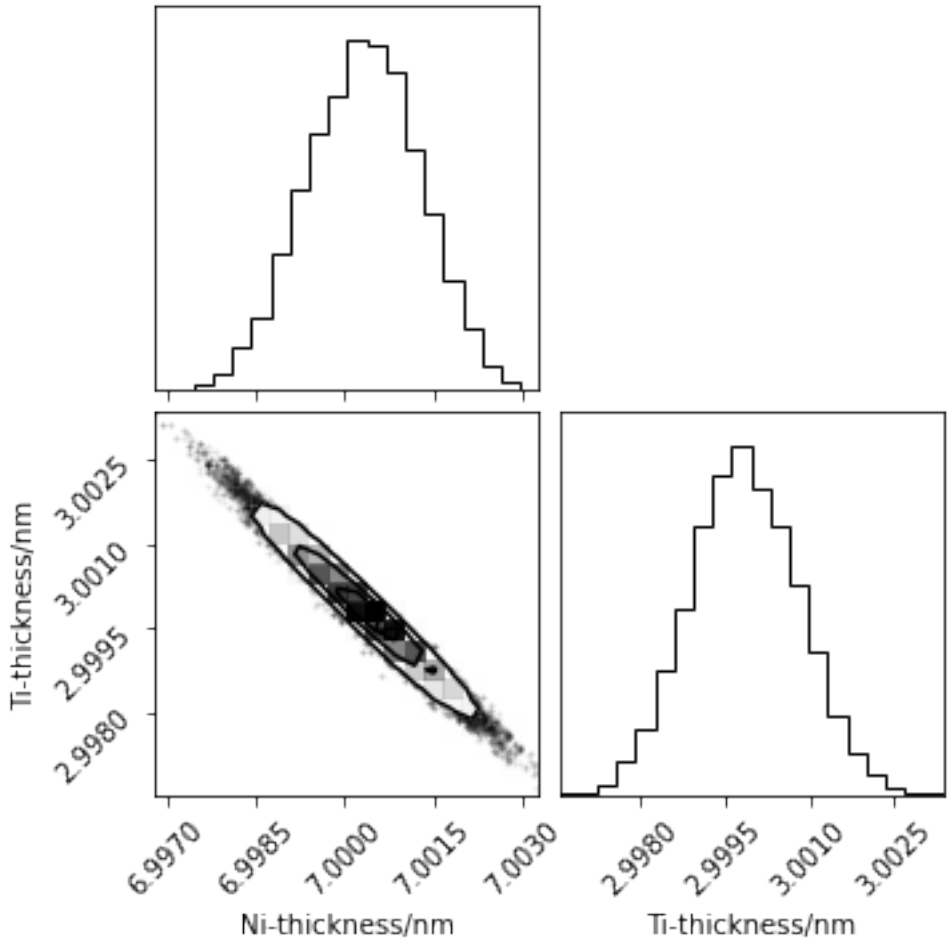
[9]:
```
pos = soln.x + 1e-4 * np.random.randn(32, 2)
nwalkers, ndim = pos.shape

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_likelihood,␣
↪args=(get_real_data()[:, 0], get_real_data()[:, 1], get_real_data()[:, 2]))
sampler.run_mcmc(pos, 1000, progress=True);
```

```
100%|      | 1000/1000 [02:28<00:00,  6.75it/s]
```

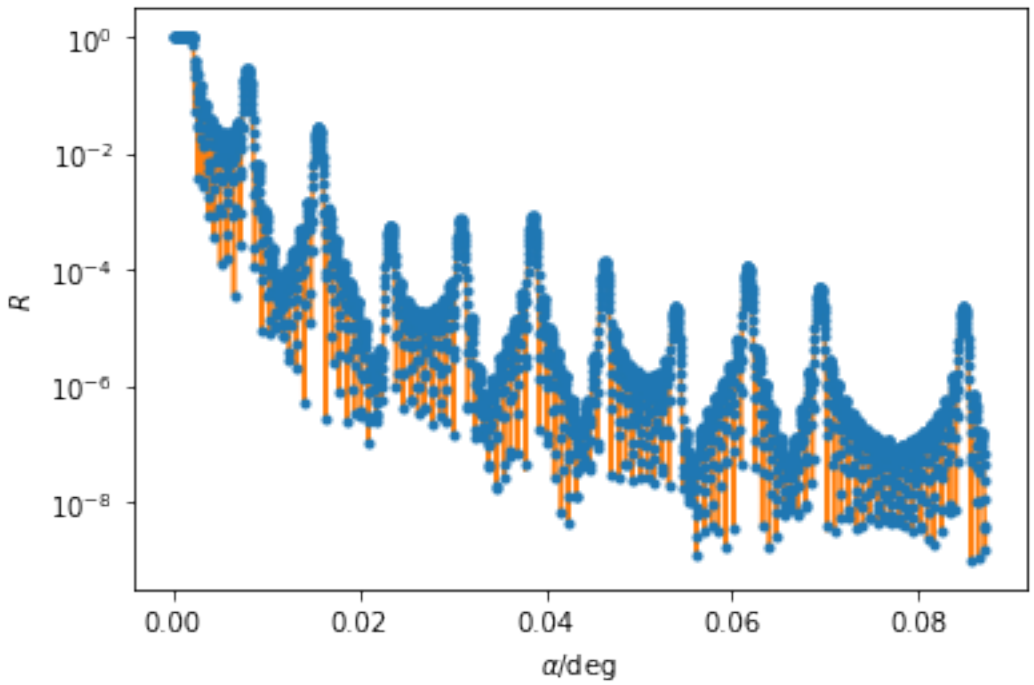
This will perform the sampling for some time (on my machine it took about 5 minutes to sample 1000 steps with 32 walkers). Having collected the samples, we can then unpack them and using the `corner` package visualise them.

[10]:
```
flat_samples = sampler.get_chain(flat=True)
corner.corner(flat_samples, labels=['Ni-thickness/nm', 'Ti-thickness/nm'])
plt.show()
```



Finally, we can plot the maximum likelihood estimate for the model along with the experimental data.

[11]:
```
plt.errorbar(get_real_data()[:, 0], get_real_data()[:, 1], get_real_data()[:,␣
 ↪2], marker='.', ls='')
plt.plot(get_real_data()[:, 0], run_simulation(get_real_data()[:, 0],␣
 ↪*flat_samples.mean(axis=0)), '-')
plt.xlabel('$\\alpha$/deg')
plt.ylabel('$R$')
plt.yscale('log')
plt.show()
```



Note that the `flat_samples` object describes the distributions shown in the corner plot. Therefore we can find values of interest, such as the standard deviation or confident intervals.