udkm1Dsim Documentation

Release 2.0.1

Daniel Schick, et.al.

Feb 01, 2024

CONTENTS

The *udkm1Dsim* toolbox is a collection of Python classes and routines to simulate the thermal, structural, and magnetic dynamics after laser excitation as well as the according X-ray scattering response in one-dimensional sample structures after ultrafast excitation.

The toolbox provides the capabilities to define arbitrary layered structures on the atomic level including a rich database of element-specific physical properties. The excitation of ultrafast dynamics is represented by an N-temperature-model which is commonly applied for ultrafast optical excitations. Structural dynamics due to thermal stresses are calculated by a linear-chain model of masses and springs. The implementation of magnetic dynamics can be easily accomplished by the user for the individual problem.

The resulting X-ray diffraction response is computed by kinematical or dynamical X-ray theory which can also include magnetic scattering.

The *udkm1Dsim* toolbox is highly modular and allows to introduce user-defined results at any step in the simulation procedure.

The *udkm1Dsim* toolbox was initially developed for MATLAB® in the *Ultrafast Dynamics in Condensed Matter* group of Prof. Matias Bargheer at the *University of Potsdam*, Germany. The MATLAB® source code is still available at github.com/dschick/udkm1DsimML.

The current toolbox, written in Python, is maintained by Daniel Schick at the *Max-Born-Institute*, Berlin, Germany.

CHAPTER

ONE

DOCUMENTATION

The documentation can be found at udkm1Dsim.readthedocs.io.

CHAPTER

TWO

CITATION

Please cite the current preprint if you use the toolbox in your own work:

D. Schick, *udkm1Dsim - A Python toolbox for simulating 1D ultrafast dynamics in condensed matter*, Comput. Phys. Commun. 266, 108031 (2021) (preprint).

You can also cite the original publication if appropriate:

D. Schick, A. Bojahr, M. Herzog, R. Shayduk, C. von Korff Schmising & M. Bargheer, *udkm1Dsim - A Simulation Toolkit for 1D Ultrafast Dynamics in Condensed Matter*, Comput. Phys. Commun. 185, 651 (2014) (preprint).

CHAPTER

THREE

INSTALLATION

You can either install directly from pypi.org using the command

```
pip install udkm1Dsim
```

or if you want to work on the latest develop release you can clone udkm1Dsim from the main git repository:

```
git clone https://github.com/dschick/udkm1Dsim.git udkm1Dsim
```

To work in editable mode (source is only linked but not copied to the python site-packages), just do:

```
pip install -e ./udkm1Dsim
```

Or to do a normal install with

```
pip install ./udkm1Dsim
```

Optionally, you can also let pip install directly from the repository:

```
pip install git+https://github.com/dschick/udkm1Dsim.git
```

You can have the following optional installation to enable parallel computations, unit tests, as well as building the documentation:

```
pip install udkm1Dsim[parallel]
pip install udkm1Dsim[testing]
pip install udkm1Dsim[documentation]
```

CHAPTER

FOUR

CONTRIBUTE & SUPPORT

If you are having issues please let us know via the issue tracker.

You can also ask questions, share ideas, or engage with community members via the discussions.

You can contribute to the project via pull-requests following the GitHub flow concept.

CHAPTER

FIVE

LICENSE

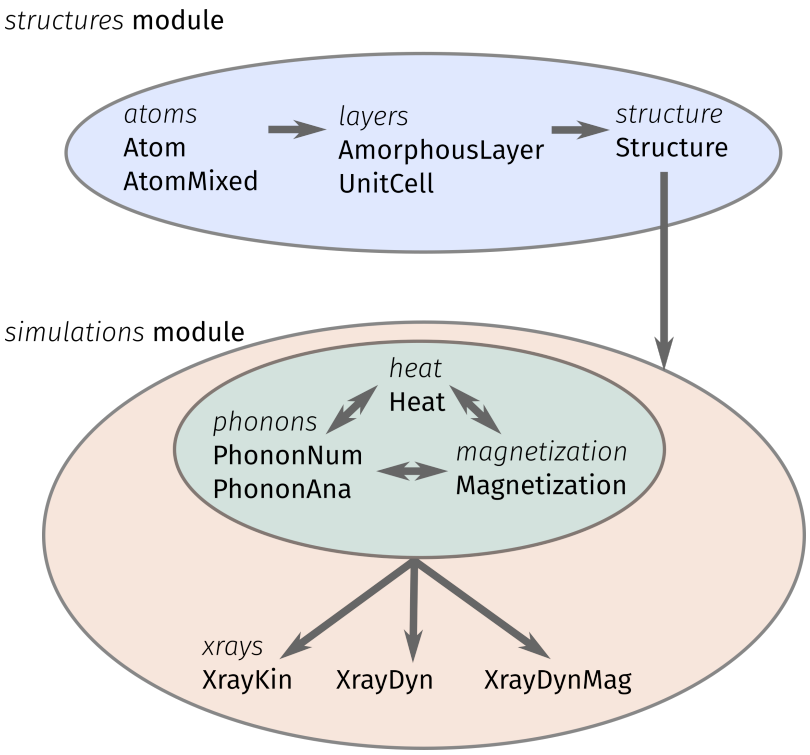
The project is licensed under the MIT license.

5.1 User Guide

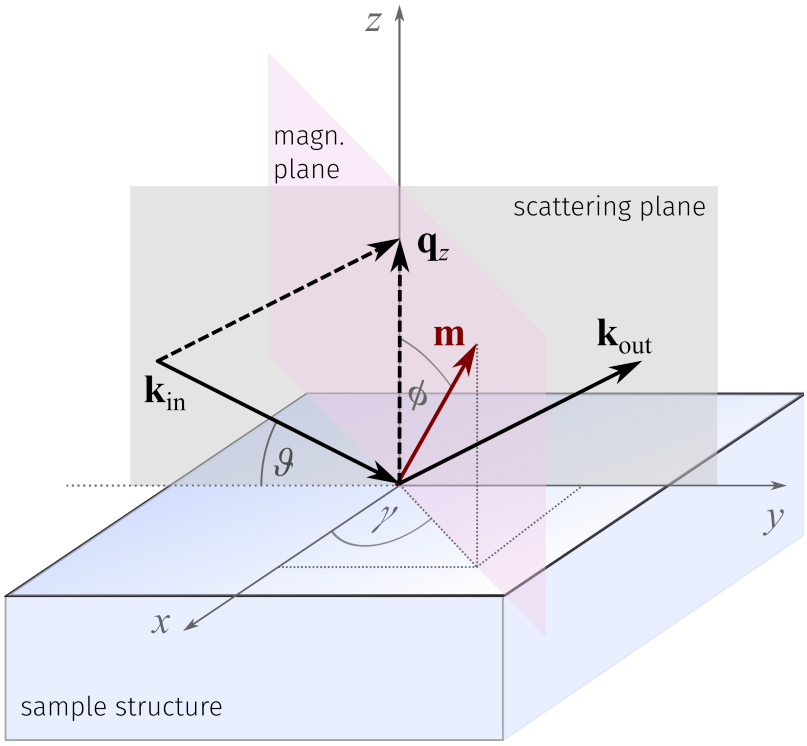
The *udkm1Dsim* toolbox comes as a Python package which can be easily installed with all of its dependencies.

In case you encounter any issues during the installation or usage of the package, please open an issue at GitHub.

The internal structure of the *udkm1Dsim* toolbox is sketched in the lower figure:



The common experimental scheme is sketched below and shows the general definition of coordinates and angles that are used throughout the *udkm1Dsim* toolbox. The incoming and outgoing wavevectors $\vec{k}_{\text{in}}$ and $\vec{k}_{\text{out}}$ define the scattering plane which must be co-planar to the sample surface, hence only accessing scattering vectors $\vec{q}_z$ along the surface normal. The pointing of the magnetization vector $\vec{m}$ is defined by the two angles ϕ and γ and allows any direction in the xyz coordinate system. The sample structure is defined from top to bottom. Accordingly the distances with in the sample structure start at its surface and proceed along $-z$ direction.



To get started, please work yourself through the Examples.

For more detailed information, also regarding the physics, you might refer to the API Documentation.

5.2 Publications

5.2.1 2023

1. Maximilian Mattern, Alexander von Reppert, Steffen Peer Zeuschner, Marc Herzog, Jan Etienne Pudell, and Matias Bargheer. Concepts and use cases for picosecond ultrasonics with x-rays. *Photoacoustics*, 31:100503, 2023. doi:10.1016/j.pacs.2023.100503.

2. Denys Naumenko, Max Burian, Benedetta Marmiroli, Richard Haider, Andrea Radeticchio, Lucas Wagner, Luca Piazza, Lisa Glatt, Stefan Brandstetter, Simone Dal Zilio, Giorgio Biasiol, and Heinz Amenitsch. Implication of the double-gating mode in a hybrid photon counting detector for measurements of transient heat conduction in GaAs/AlAs superlattice structures. *J. Appl. Crystallogr.*, 56(4):961–966, 2023. doi:10.1107/s1600576723004302.

3. Martin Borchert, Dieter Engel, Clemens Von Korff Schmising, Bastian Pfau, Stefan Eisebitt, and Daniel Schick. X-ray magnetic circular dichroism spectroscopy at the fe l edges with a picosecond laser-driven plasma source. *OPTICA*, 10(4):450–455, APR 20 2023. doi:10.1364/OPTICA.480221.

4. E. Rongione, O. Gueckstock, M. Mattern, O. Gomonay, H. Meer, C. Schmitt, R. Ramos, T. Kikkawa, M. Micica, E. Saitoh, J. Sinova, H. Jaffres, J. Mangeney, S. T. B. Goennenwein, S. Gepraegs, T. Kampfrath, M. Klaeui, M. Bargheer, T. S. Seifert, S. Dhillon, and R. Lebrun. Emission of coherent thz magnons in an antiferromagnetic insulator triggered by ultrafast spin-phonon interactions. *NATURE COMMUNICATIONS*, MAR 31 2023. doi:10.1038/s41467-023-37509-6.

5. Peter Gaal, Daniel Schmidt, Mallika Khosla, Carsten Richter, Peter Boesecke, Dmitri Novikov, Martin Schmidbauer, and Jutta Schwarzkopf. Self-stabilization of the equilibrium state in ferroelectric thin films. *APPLIED SURFACE SCIENCE*, MAR 15 2023. doi:10.1016/j.apsusc.2022.155891.

6. Kathinka Gerlinger, Bastian Pfau, Martin Hennecke, Lisa-Marie Kern, Ingo Will, Tino Noll, Markus Weigand, Joachim Graefe, Nick Traeger, Michael Schneider, Christian M. Guenther, Dieter Engel, Gisela Schuetz, and Stefan Eisebitt. Pump-probe x-ray microscopy of photo-induced magnetization dynamics at mhz repetition rates. *STRUCTURAL DYNAMICS-US*, MAR 2023. doi:10.1063/4.0000167.

5.2.2 2022

1. Roman Shayduk, Joerg Hallmann, Angel Rodriguez-Fernandez, Markus Scholz, Wei Lu, Ulrike Boesenberg, Johannes Moeller, Alexey Zozulya, Man Jiang, Ulrike Wegner, Radu-Costin Secareanu, Guido Palmer, Moritz Emons, Max Lederer, Sergey Volkov, Ionela Lindfors-Vrejoiu, Daniel Schick, Marc Herzog, Matias Bargheer, and Anders Madsen. Femtosecond x-ray diffraction study of multi-thz coherent phonons in srtio3. *APPLIED PHYSICS LETTERS*, MAY 16 2022. doi:10.1063/5.0083256.

2. Di Zhao, Pengxian You, Jing Yang, Junhong Yu, Hang Zhang, Min Liao, and Jianbo Hu. A highly stable-output kilohertz femtosecond hard x-ray pulse source for ultrafast x-ray diffraction. *APPLIED SCIENCES-BASEL*, MAY 2022. doi:10.3390/app12094723.

3. Marc Herzog, Alexander von Reppert, Jan-Etienne Pudell, Carsten Henkel, Matthias Kronseder, Christian H. Back, Alexei A. Maznev, and Matias Bargheer. Phonon-dominated energy transport in purely metallic heterostructures. *ADVANCED FUNCTIONAL MATERIALS*, OCT 2022. doi:10.1002/adfm.202206179.

4. Felix Steinbach, Nele Stetzuhn, Dieter Engel, Unai Atxitia, Clemens von Korff Schmising, and Stefan Eisebitt. Accelerating double pulse all-optical write/erase cycles in metallic ferrimagnets. *APPLIED PHYSICS LETTERS*, MAR 14 2022. doi:10.1063/5.0080351.

5. M. Mattern, A. von Reppert, S. P. Zeuschner, J. -E. Pudell, F. Kuhne, D. Diesing, M. Herzog, and M. Bargheer. Electronic energy transport in nanoscale au/fe hetero-structures in the perspective of ultrafast lattice dynamics. *APPLIED PHYSICS LETTERS*, FEB 28 2022. doi:10.1063/5.0080378.

6. Theodor Secanell Holstad, Trygve Magnus Raeder, Mads Carlsen, Erik Bergback Knudsen, Leora Dresselhaus-Marais, Kristoffer Haldrup, Hugh Simons, Martin Meedom Nielsen, and Henning Friis Poulsen. X-ray free-electron laser based dark-field x-ray microscopy: a simulation-based study. *JOURNAL OF APPLIED CRYSTALLOGRAPHY*, 55(1):112–121, FEB 2022. doi:10.1107/S1600576721012760.

5.2.3 2021

7. Daniel Schick. Udkm1dsim-a python toolbox for simulating 1d ultrafast dynamics in condensed matter. *COMPUTER PHYSICS COMMUNICATIONS*, SEP 2021. doi:10.1016/j.cpc.2021.108031.

8. N. Bach and S. Schaefer. Ultrafast strain propagation and acoustic resonances in nanoscale bilayer systems. *STRUCTURAL DYNAMICS-US*, MAY 2021. doi:10.1063/4.0000079.

9. Matthias Roessle, Wolfram Leitenberger, Matthias Reinhardt, Azize Koc, Jan Pudell, Christelle Kwamen, and Matias Bargheer. The time-resolved hard x-ray diffraction endstation kmc-3 xpp at bessy ii. *JOURNAL OF SYNCHROTRON RADIATION*, 28(3):948–960, MAY 2021. doi:10.1107/S1600577521002484.

10. Lukas Alber, Valentino Scalera, Vivek Unikandanunni, Daniel Schick, and Stefano Bonetti. Ntmpy: an open source package for solving coupled parabolic differential equations in the framework of the three-temperature model. *COMPUTER PHYSICS COMMUNICATIONS*, AUG 2021. doi:10.1016/j.cpc.2021.107990.

11. M. Mattern, J. -E. Pudell, G. Laskin, A. von Reppert, and M. Bargheer. Analysis of the temperature- and fluence-dependent magnetic stress in laser-excited srruo3. *STRUCTURAL DYNAMICS-US*, MAR 2021. doi:10.1063/4.0000072.

12. Marwan Deb, Elena Popova, Steffen Peer Zeuschner, Michel Hehn, Niels Keller, Stephane Mangin, Gregory Malinowski, and Matias Bargheer. Generation of spin waves via spin-phonon interaction in a buried dielectric thin film. *PHYSICAL REVIEW B*, JAN 11 2021. doi:10.1103/PhysRevB.103.024411.

13. Daniel Schick, Martin Borchert, Julia Braenzel, Holger Stiel, Johannes Tuemmler, Daniel E. Buergler, Alexander Firsov, Clemens von Korff Schmising, Bastian Pfau, and Stefan Eisebitt. Laser-driven resonant magnetic soft-x-ray scattering for probing ultrafast antiferromagnetic and structural dynamics. *OPTICA*, 8(9):1237–1242, SEP 20 2021. doi:10.1364/OPTICA.435522.

5.2.4 2020

14. Jan-Etienne Pudell, Maximilian Mattern, Michel Hehn, Gregory Malinowski, Marc Herzog, and Matias Bargheer. Heat transport without heating?-an ultrafast x-ray perspective into a metal heterostructure. *ADVANCED FUNCTIONAL MATERIALS*, NOV 11 2020. doi:10.1002/adfm.202004555.

15. V. Bragaglia, M. Ramsteiner, D. Schick, J. E. Boschker, R. Mitzner, R. Calarco, and K. Holldack. Phonon anharmonicities and ultrafast dynamics in epitaxial sb2te3. *SCIENTIFIC REPORTS*, JUL 31 2020. doi:10.1038/s41598-020-69663-y.

16. Daniel X. Du and David J. Flannigan. Imaging phonon dynamics with ultrafast electron microscopy: kinematical and dynamical simulations. *STRUCTURAL DYNAMICS-US*, MAR 2020. doi:10.1063/1.5144682.

17. Diana Bachiller-Perea, Yves-Marie Abiven, Jerome Bisou, Pierre Fertey, Pawel Grybos, Amelie Jarnac, Brahim Kanout, Anna Koziol, Florent Langlois, Claire Laulh, Fabien Legrand, Piotr Maj, Claude Menneglier, Arafat Noureddine, Fabienne Orsini, Gauthier Thibaux, and Arkadiusz Dawiec. First pump-probe-probe hard x-ray diffraction experiments with a 2d hybrid pixel detector developed at the soleil synchrotron. *JOURNAL OF SYNCHROTRON RADIATION*, 27(2):340–350, MAR 2020. doi:10.1107/S1600577520000612.

18. A. von Reppert, M. Mattern, J. -E. Pudell, S. P. Zeuschner, K. Dumesnil, and M. Bargheer. Unconventional picosecond strain pulses resulting from the saturation of magnetic stress within a photoexcited rare earth layer. *STRUCTURAL DYNAMICS-US*, MAR 2020. doi:10.1063/1.5145315.

19. Max Burian, Benedetta Marmiroli, Andrea Radeticchio, Christian Morello, Denys Naumenko, Giorgio Biasiol, and Heinz Amenitsch. Picosecond pump-probe x-ray scattering at the elettra saxs beamline. *JOURNAL OF SYNCHROTRON RADIATION*, 27(1):51–59, JAN 1 2020. doi:10.1107/S1600577519015728.

5.2.5 2019

20. J. -E. Pudell, M. Sander, R. Bauer, M. Bargheer, M. Herzog, and P. Gaal. Full spatiotemporal control of laser-excited periodic surface deformations. *PHYSICAL REVIEW APPLIED*, AUG 19 2019. doi:10.1103/PhysRevApplied.12.024036.

21. Mathias Sander, Roman Bauer, Victoria Kabanova, Matteo Levantino, Michael Wulff, Daniel Pfuetzenreuter, Jutta Schwarzkopf, and Peter Gaal. Demonstration of a picosecond bragg switch for hard x-rays in a synchrotron-based pump-probe experiment. *JOURNAL OF SYNCHROTRON RADIATION*, 26(4, SI):1253–1259, JUL 2019. doi:10.1107/S1600577519005356.

22. J. Pudell, A. von Reppert, D. Schick, F. Zamponi, M. Roessle, M. Herzog, H. Zabel, and M. Bargheer. Ultrafast negative thermal expansion driven by spin disorder. *PHYSICAL REVIEW B*, MAR 11 2019. doi:10.1103/PhysRevB.99.094304.

23. S. P. Zeuschner, T. Parpiiev, T. Pezeril, A. Hillion, K. Dumesnil, A. Anane, J. Pudell, L. Willig, M. Roessle, M. Herzog, A. von Reppert, and M. Bargheer. Tracking picosecond strain pulses in heterostructures that exhibit giant magnetostriction. *STRUCTURAL DYNAMICS-US*, MAR 2019. doi:10.1063/1.5084140.

5.2.6 2018

24. A. von Reppert, L. Willig, J. -E. Pudell, M. Roessle, W. Leitenberger, M. Herzog, F. Ganss, O. Hellwig, and M. Bargheer. Ultrafast laser generated strain in granular and continuous fept thin films. *APPLIED PHYSICS LETTERS*, SEP 17 2018. doi:10.1063/1.5050234.

25. J. Pudell, A. A. Maznev, M. Herzog, M. Kronseder, C. H. Back, G. Malinowski, A. von Reppert, and M. Bargheer. Layer specific observation of slow thermal equilibration in ultrathin metallic nanostructures by femtosecond x-ray diffraction. *NATURE COMMUNICATIONS*, AUG 20 2018. doi:10.1038/s41467-018-05693-5.

26. Runze Li, Kyle Sundqvist, Jie Chen, H. E. Elsayed-Ali, Jie Zhang, and Peter M. Rentzepis. Transient lattice deformations of crystals studied by means of ultrafast time-resolved x-ray and electron diffraction. *STRUCTURAL DYNAMICS-US*, JUL 2018. doi:10.1063/1.5029970.

5.2.7 2017

27. A. Jarnac, Xiaocui Wang, A. U. J. Bengtsson, J. C. Ekstrom, H. Enquist, A. Jurgilaitis, D. Kroon, A. I. H. Persson, V. -T. Pham, C. M. Tu, and J. Larsson. Communication: demonstration of a 20 ps x-ray switch based on a photoacoustic transducer. *STRUCTURAL DYNAMICS-US*, SEP 2017. doi:10.1063/1.4993730.

28. M. Sander, M. Herzog, J. E. Pudell, M. Bargheer, N. Weinkauf, M. Pedersen, G. Newby, J. Sellmann, J. Schwarzkopf, V. Besse, V. V. Temnov, and P. Gaal. Spatiotemporal coherent control of thermal excitations in solids. *PHYSICAL REVIEW LETTERS*, AUG 18 2017. doi:10.1103/PhysRevLett.119.075901.

29. A. Koc, M. Reinhardt, A. von Reppert, M. Roessle, W. Leitenberger, K. Dumesnil, P. Gaal, F. Zamponi, and M. Bargheer. Ultrafast x-ray diffraction thermometry measures the influence of spin excitations on the heat transport through nanolayers. *PHYSICAL REVIEW B*, JUL 25 2017. doi:10.1103/PhysRevB.96.014306.

30. Stefano Cecchi, Eugenio Zallo, Jamo Momand, Ruining Wang, Bart J. Kooi, Marcel A. Verheijen, and Raffaella Calarco. Improved structural and electrical properties in native sb2te3/gexsb2te3+x van der waals superlattices due to intermixing mitigation. *APL MATERIALS*, FEB 2017. doi:10.1063/1.4976828.

5.2.8 2016

31. M. Sander, A. Koc, C. T. Kwamen, H. Michaels, A. v. Reppert, J. Pudell, F. Zamponi, M. Bargheer, J. Sellmann, J. Schwarzkopf, and P. Gaal. Characterization of an ultrafast bragg-switch for shortening hard x-ray pulses. *JOURNAL OF APPLIED PHYSICS*, NOV 21 2016. doi:10.1063/1.4967835.

32. A. I. H. Persson, A. Jarnac, Xiaocui Wang, H. Enquist, A. Jurgilaitis, and J. Larsson. Studies of electron diffusion in photo-excited ni using time-resolved x-ray diffraction. *APPLIED PHYSICS LETTERS*, NOV 14 2016. doi:10.1063/1.4967470.

33. E. S. Pavlenko, M. Sander, Q. Cui, and M. Bargheer. Gold nanorods sense the ultrafast viscoelastic deformation of polymers upon molecular strain actuation. *JOURNAL OF PHYSICAL CHEMISTRY C*, 120(43):24957–24964, NOV 3 2016. doi:10.1021/acs.jpcc.6b06915.

34. Daniel Schick, Sebastian Eckert, Niko Pontius, Rolf Mitzner, Alexander Foehlisch, Karsten Holldack, and Florian Sorgenfrei. Versatile soft x-ray-optical cross-correlator for ultrafast applications. *STRUCTURAL DYNAMICS-US*, SEP 2016. doi:10.1063/1.4964296.

35. A. von Reppert, J. Pudell, A. Koc, M. Reinhardt, W. Leitenberger, K. Dumesnil, F. Zamponi, and M. Bargheer. Persistent nonequilibrium dynamics of the thermal energies in the spin and phonon systems of an antiferromagnet. *STRUCTURAL DYNAMICS-US*, SEP 2016. doi:10.1063/1.4961253.

36. Kirill V. Mitrofanov, Paul Fons, Kotaro Makino, Ryo Terashima, Toru Shimada, Alexander V. Kolobov, Junji Tominaga, Valeria Bragaglia, Alessandro Giussani, Raffaella Calarco, Henning Riechert, Takahiro Sato, Tetsuo Katayama, Kanade Ogawa, Tadashi Togashi, Makina Yabashi, Simon Wall, Dale Brewe, and Muneaki Hase.

Sub-nanometre resolution of atomic motion during electronic excitation in phase-change materials. *SCIENTIFIC REPORTS*, FEB 12 2016. doi:10.1038/srep20633.

37. E. S. Pavlenko, M. Sander, S. Mitzscherling, J. Pudell, F. Zamponi, M. Roessle, A. Bojahr, and M. Bargheer. Azobenzene - functionalized polyelectrolyte nanolayers as ultrafast optoacoustic transducers. *NANOSCALE*, 8(27):13297–13302, 2016. doi:10.1039/c6nr01448h.

5.2.9 2015

38. L. Maerten, A. Bojahr, and M. Bargheer. Observing backfolded and unfolded acoustic phonons by broadband optical light scattering. *ULTRASONICS*, 56:148–152, FEB 2015. doi:10.1016/j.ultras.2014.08.023.

5.2.10 2014

39. Daniel Schick, Marc Herzog, Andre Bojahr, Wolfram Leitenberger, Andreas Hertwig, Roman Shayduk, and Matias Bargheer. Ultrafast lattice response of photoexcited thin films studied by x-ray diffraction. *STRUCTURAL DYNAMICS-US*, NOV 2014. doi:10.1063/1.4901228.

40. Paul Fons, Peter Rodenbach, Kirill V. Mitrofanov, Alexander V. Kolobov, Junji Tominaga, Roman Shayduk, Alessandro Giussani, Raffaella Calarco, Michael Hanke, Henning Riechert, Robert E. Simpson, and Muneaki Hase. Picosecond strain dynamics in ge2sb2te5 monitored by time-resolved x-ray diffraction. *PHYSICAL REVIEW B*, SEP 25 2014. doi:10.1103/PhysRevB.90.094305.

41. B. B. Zhang, S. S. Sun, D. R. Sun, and Y. Tao. Note: a novel normalization scheme for laser-based plasma x-ray sources. *REVIEW OF SCIENTIFIC INSTRUMENTS*, SEP 2014. doi:10.1063/1.4896252.

42. Daniel Schick, Marc Herzog, Haidan Wen, Pice Chen, Carolina Adamo, Peter Gaal, Darrell G. Schlom, Paul G. Evans, Yuelin Li, and Matias Bargheer. Localized excited charge carriers generate ultrafast inhomogeneous strain in the multiferroic bifeo3. *PHYSICAL REVIEW LETTERS*, MAR 3 2014. doi:10.1103/PhysRevLett.112.097602.

43. Peter Gaal, Daniel Schick, Marc Herzog, Andre Bojahr, Roman Shayduk, Jevgeni Goldshteyn, Wolfram Leitenberger, Ionela Vrejoiu, Dmitry Khakhulin, Michael Wulff, and Matias Bargheer. Ultrafast switching of hard x-rays. *JOURNAL OF SYNCHROTRON RADIATION*, 21(2):380–385, MAR 2014. doi:10.1107/S1600577513031949.

44. H. A. Navirian, D. Schick, P. Gaal, W. Leitenberger, R. Shayduk, and M. Bargheer. Thermoelastic study of nanolayered structures using time-resolved x-ray diffraction at high repetition rate. *APPLIED PHYSICS LETTERS*, JAN 13 2014. doi:10.1063/1.4861873.

5.3 Examples

5.3.1 Structure

In order to carry out any simulations using the `udkm1Dsim` package, an according one-dimensional `Structure` needs to be created in advance.

This `Structure` object can consists of one or many sub-structures and/or `Layers` of type `UnitCell` or `AmorphousLayer`.

`UnitCell`s and `AmorphousLayer`s consist of the fundamental building blocks, namely `Atom`s and `AtomMixed`.

In this example the basic concepts of creating the above mentioned objects are introduced. Furthermore one should easily see how to set and access all the physical properties of these physical objects.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Atoms

The `atoms` module contains two classes: `Atom` and `AtomMixed`.

Atom

The `Atom` object represents a real physical atom as it can be found in the periodic table. Accordingly, it is initialized with the required `symbol` of the element. Then all necessary data are loaded from parameter files linked to this element.

An optional `ID` can be given as *keyword parameter* if atoms of the same element but with different properties are used. Another keyword argument is the `ionicity` of the atom, which has to be present in the parameter files.

The magnetization of every atom can be set by the three *keyword parameters*

- `mag_amplitude`
- `mag_phi`
- `mag_gamma`

If no individual paths to the atomic- and/or magnetic scattering factors are given, by `atomic_form_factor_path` and `magnetic_form_factor_path`, respectively, the defaults parameters are used from the *Chantler tables*:

C.T. Chantler, K. Olsen, R.A. Dragoset, J. Chang, A.R. Kishore, S.A. Kotochigova, & D.S. Zucker,
Detailed Tabulation of Atomic Form Factors, Photoelectric Absorption and Scattering Cross Section, and Mass Attenuation Coefficients for Z = 1-92 from E = 1-10 eV to E = 0.4-1.0 MeV
NIST Standard Reference Database 66.

as well as Project Dyna, respectively

```
O = ud.Atom('O')
Om1 = ud.Atom('O', id='Om1', ionicity=-1)
Om2 = ud.Atom('O', id='Om2', ionicity=-2)
Fe = ud.Atom('Fe', mag_amplitude=1, mag_phi=0*u.deg, mag_gamma=90*u.deg)
Cr = ud.Atom('Cr')
Ti = ud.Atom('Ti')
Sr = ud.Atom('Sr')
Ru = ud.Atom('Ru')
Pb = ud.Atom('Pb')
Zr = ud.Atom('Zr')
```

One can easily print all properties of a single atom:

```
print(Sr)
```

```
Atom with the following properties
================  ================================
              id  Sr
          symbol  Sr
            name  Strontium
  atomic number Z  38
    mass number A  87.62
            mass  1.455×10^25 kg
        ionicity  0
Cromer Mann coeff  [38.      0.     17.5663  9.8184]
              ..  [ 5.422   2.6694  1.5564 14.0988]
              ..  [  0.1664 132.376    2.5064]
  magn. amplitude  0
        magn. phi  0.0 deg
      magn. gamma  0.0 deg
================  ================================
```

Or just a single property:

```
print(Ti.mass)
```

```
7.948502350094219×10^26 kg
```

Mixed Atom

The `AtomMixed` class allows for solid solutions that can easily achieved by the following lines of code. The input for the initialization of the `AtomMixed` object are the `symbol`, `id`, and `name`, whereas only the first is required.

```
ZT = ud.AtomMixed('ZT', id='ZT', name='Zircon-Titan 0.2 0.8')
ZT.add_atom(Zr, 0.2)
ZT.add_atom(Ti, 0.8)
print(ZT)
```

```
AtomMixed with the following properties
===============  ====================
            id  ZT
        symbol  ZT
          name  Zircon-Titan 0.2 0.8
atomic number Z  25.6
  mass number A  56.538399999999996
          mass  9.388×10^26 kg
      ionicity  0.0
magn. amplitude  0
      magn. phi  0.0 deg
    magn. gamma  0.0 deg
===============  ====================
2 Constituents:
---------  ------
Zirconium  20.0 %
```

(continues on next page)

(continued from previous page)

```
 Titanium  80.0 %
---------  ------
```

Layers

The atoms created above can be used to build `Layer`s. There are two types of layers available: `AmorphousLayer` and crystalline `UnitCell`. Both share many common physical properties which are relevant for the later simulations. Please refer to a complete list or properties and methods in the **API documentation**.

Amorphous Layers

The `AmorphousLayer` must be initialized with an `id`, `name`, `thickness`, and `density`. All other properties are optional and must be set to carry out the according simulations.

```
amorph_Fe = ud.AmorphousLayer('amorph_Fe', 'amorph_Fe',
                              20*u.nm, 7.874*u.g/u.cm**3)
# print the layer properties
print(amorph_Fe)
```

```
Amorphous layer with the following properties

========================  ===============
               parameter  value
========================  ===============
                      id  amorph_Fe
                    name  amorph_Fe
               thickness  20.0 nm
                    area  0.01 nm^2
                  volume  0.2 nm^3
                    mass  1.5748×10^24 kg
     mass per unit area  1.5748×10^24 kg
                 density  7.874×10^3 kg/m^3
               roughness  0.0 nm
    Debye Waller Factor  0 m^2
         sound velocity  0.0 m/s
        spring constant  [0.0] kg/s^2
         phonon damping  0.0 kg/s
        opt. pen. depth  0.0 nm
   opt. refractive index  0
  opt. ref. index/strain  0
       thermal conduct.  0 W/(m K)
linear thermal expansion  0
          heat capacity  0 J/(kg K)
     subsystem coupling  0 W/m^3
             no atom set
========================  ===============
```

The physical properties can be also given during initialization using a `dict`:

```
params = {
    'opt_pen_depth': 10*u.nm,
    'sound_vel': 5*(u.nm/u.ps),
}

amorph_Cr = ud.AmorphousLayer('amorph_Cr', 'amorph_Cr',
                              40*u.nm, 7.14*u.g/u.cm**3, atom=Cr, **params)
# print the layer properties
print(amorph_Cr)
```

```
Amorphous layer with the following properties

========================  ============================
                parameter  value
========================  ============================
                       id  amorph_Cr
                     name  amorph_Cr
                thickness  40.0 nm
                     area  0.01 nm^2
                   volume  0.4 nm^3
                     mass  2.856×10^24 kg
      mass per unit area  2.856×10^24 kg
                  density  7.14×10^3 kg/m^3
                roughness  0.0 nm
     Debye Waller Factor  0 m^2
          sound velocity  5×10^3 m/s
         spring constant  [0.044625000000000001] kg/s^2
          phonon damping  0.0 kg/s
         opt. pen. depth  9.999999999999998 nm
   opt. refractive index  0
  opt. ref. index/strain  0
        thermal conduct.  0 W/(m K)
linear thermal expansion  0
           heat capacity  0 J/(kg K)
      subsystem coupling  0 W/m^3
                    atom  Chromium
            magnetization
                amplitude  0
                  phi [°]  0.0 deg
                gamma [°]  0.0 deg
========================  ============================
```

Unit Cells

The `UnitCell` requires an `id`, `name`, and `c_axis` upon initialization. Multiple atoms can be added to relative positions along the *c*-Axis in the 1D `UnitCell`. Note that all temperature-dependent properties can be given either as scalar (constant) value or as string that represents a temperature-dependent *lambda*-function:

```
# c-axis lattice constants of the two layers
c_STO_sub = 3.905*u.angstrom
c_SRO = 3.94897*u.angstrom
```

(continues on next page)

(continued from previous page)

```
# sound velocities [nm/ps] of the two layers
sv_SRO = 6.312*u.nm/u.ps
sv_STO = 7.800*u.nm/u.ps

# SRO layer
prop_SRO = {}
prop_SRO['a_axis'] = c_STO_sub  # a-Axis
prop_SRO['b_axis'] = c_STO_sub  # b-Axis
prop_SRO['deb_Wa l_Fac'] = 0  # Debye-Waller factor
prop_SRO['sound_ve l'] = sv_SRO  # sound velocity
prop_SRO['opt_ref_index'] = 2.44+4.32j
prop_SRO['therm_ond'] = 5.72*u.W/(u.m*u.K)  # heat conductivity
prop_SRO['lin_therm_ exp'] = 1.03e-5  # linear thermal expansion
prop_SRO['heat_capacity'] = '455.2 + 0.112*T  - 2.1935e6/T**2'  # [J/kg K]

SRO = ud.UnitCell('SRO', 'Strontium Ruthenate', c_SRO, **prop_SRO)
SRO.add_atom(O, 0)
SRO.add_atom(Sr, 0)
SRO.add_atom(O, 0.5)
SRO.add_atom(O, 0.5)
SRO.add_atom(Ru, 0.5)

print(SRO)
```

```
Unit Cell with the following properties

======================  ========================================
               parameter  value
======================  ========================================
                      id  SRO
                    name  Strontium Ruthenate
                  a-axis  0.3905 nm
                  b-axis  0.3905 nm
                  c-axis  0.3949 nm
                    area  0.1525 nm^2
                  volume  0.06022 nm^3
                    mass  3.93×10^25 kg
     mass per unit area  2.577×10^26 kg
                    area  0.1525 nm^2
                  volume  0.06022 nm^3
                    mass  3.930300027032341×10^25 kg
     mass per unit area  2.5774107046400294×10^26 kg
                 density  6.527×10^3 kg/m^3
               roughness  0.0 nm
    Debye Waller Factor  0 m^2
         sound velocity  0.0 m/s
        spring constant  [0.0] kg/s^2
         phonon damping  0.0 kg/s
        opt. pen. depth  0.0 nm
  opt. refractive index  (2.44+4.32j)
 opt. ref. index/strain  0
       thermal conduct.  0 W/(m K)
```

(continues on next page)

(continued from previous page)

```
linear thermal expansion  0
           heat capacity  455.2 + 0.112*T  - 2.1935e6/T**2 J/(kg K)
      subsystem coupling  0 W/m^3
========================  ========================================

5 Constituents:
=========  ==========  ==================  =======  ==========  =========  ==========
atom         position    position function  magn.      amplitude    phi [°]    gamma [°]
=========  ==========  ==================  =======  ==========  =========  ==========
Oxygen            0                   0                    0          0           0
Strontium         0                   0                    0          0           0
Oxygen            0.5                 0.5                  0          0           0
Oxygen            0.5                 0.5                  0          0           0
Ruthenium         0.5                 0.5                  0          0           0
=========  ==========  ==================  =======  ==========  =========  ==========
```

Non-Linear Strain Dependence

In general the position of each atom in a unit cell depends linearly from an external strain. In some cases this linear behavior has to be altered. This can be easily achieved by providing a string representation of a strain-dependent *lambda*-function for the atom position when the atom is added to the unit cell.

```
# STO substrate
prop_STO_sub = {}
prop_STO_sub['a_axis'] = c_STO_sub  # a-Axis
prop_STO_sub['b_axis'] = c_STO_sub  # b-Axis
prop_STO_sub['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_STO_sub['sound_vel'] = sv_STO  # sound velocity
prop_STO_sub['opt_ref_index'] = 2.1+0j
prop_STO_sub['therm_cond'] = 12*u.W/(u.m *u.K)  # heat conductivity
prop_STO_sub['lin_therm_exp'] = 1e-5  # linear thermal expansion
prop_STO_sub['heat_capacity'] = '733.73 + 0.0248*T - 6.531e6/T**2'  # [J/kg K]

STO_sub = ud.UnitCell('STOsub', 'Strontium Titanate Substrate',
                      c_STO_sub, **prop_STO_sub)
STO_sub.add_atom(O, '0.1*(s**2+1)')
STO_sub.add_atom(Sr, 0)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(Ti, 0.5)

print(STO_sub)
```

```
Unit Cell with the following properties

========================  ========================================
               parameter  value
========================  ========================================
                      id  STOsub
                    name  Strontium Titanate Substrate
```

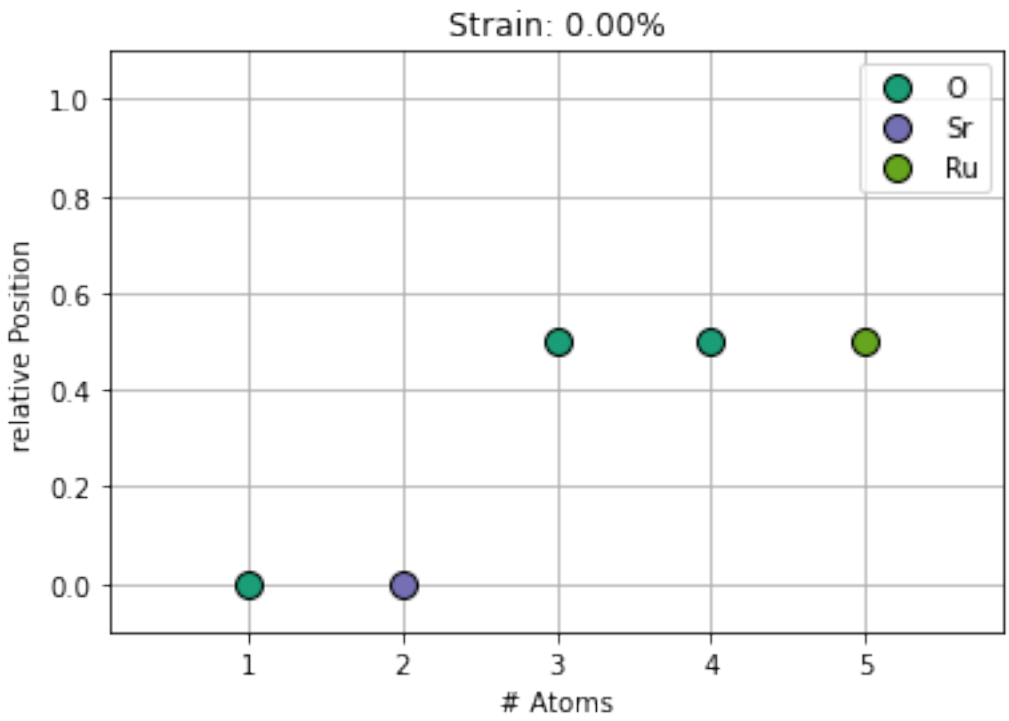
(continues on next page)

(continued from previous page)

```
                  a-axis  0.3905 nm
                  b-axis  0.3905 nm
                  c-axis  0.3905 nm
                    area  0.1525 nm^2
                  volume  0.05955 nm^3
                    mass  3.047×10^25 kg
     mass per unit area  1.998×10^26 kg
                    area  0.1525 nm^2
                  volume  0.05955 nm^3
                    mass  3.046843427429143×10^25 kg
     mass per unit area  1.9980578610298977×10^26 kg
                 density  5.117×10^3 kg/m^3
               roughness  0.0 nm
     Debye Waller Factor  0 m^2
          sound velocity  7.8×10^3 m/s
         spring constant  [7.971777885147345] kg/s^2
          phonon damping  0.0 kg/s
         opt. pen. depth  0.0 nm
   opt. refractive index  (2.1+0j)
  opt. ref. index/strain  0
        thermal conduct.  12.0 W/(m K)
linear thermal expansion  1e-05
           heat capacity  733.73 + 0.0248*T - 6.531e6/T**2 J/(kg K)
      subsystem coupling  0 W/m^3
========================  ========================================

5 Constituents:
=========  ==========  ==================  =======  ==========  =========  ==========
atom         position  position function    magn.      amplitude    phi [°]    gamma [°]
=========  ==========  ==================  =======  ==========  =========  ==========
Strontium         0    0                                     0          0           0
Oxygen            0.1  0.1*(s**2+1)                          0          0           0
Oxygen            0.5  0.5                                   0          0           0
Oxygen            0.5  0.5                                   0          0           0
Titanium          0.5  0.5                                   0          0           0
=========  ==========  ==================  =======  ==========  =========  ==========
```

A simple visualization is also available:

```
SRO.visualize()
```

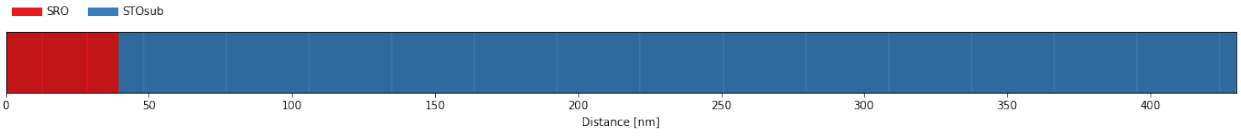


Structure

The `AmorphousLayer` and `UnitCell` can now be added to an actual structure:

```
S = ud.Structure('Single Layer')
S.add_sub_structure(SRO, 100)  # add 100 layers of SRO to sample
S.add_sub_structure(STO_sub, 1000)  # add 1000 layers of STO substrate

print(S)

S.visualize()
```

```
Structure properties:

Name   : Single Layer
Thickness : 429.99 nanometer
Roughness : 0.00 nanometer
----
100 times Strontium Ruthenate: 39.49 nanometer
1000 times Strontium Titanate Substrate: 390.50 nanometer
----
no substrate
```



There are various methods available to determine specific properties of a `Structure`, please also refer to the **API**

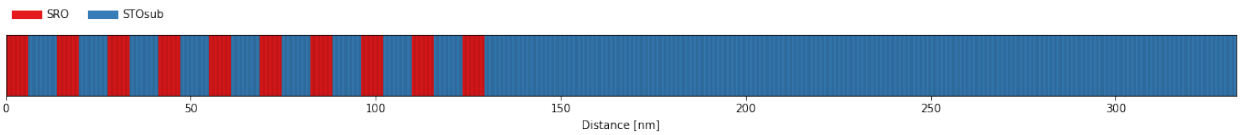
documentation for more details:

```
[d_start, d_end, d_mid] = S.get_distances_of_layers()
K = S.get_number_of_sub_structures()
L = S.get_number_of_unique_layers()
M = S.get_number_of_layers()
P = S.get_all_positions_per_unique_layer()
I = S.get_distances_of_interfaces()
c_axis = S.get_layer_property_vector('c_axis')
```

The `Structure` class also allows to nest multiple substructures in order to build more complex samples easily:

```
S2 = ud.Structure('Super Lattice')
# define a single double layer
DL = ud.Structure('Double Layer')
DL.add_sub_structure(SRO, 15)  # add 15 layers of SRO
DL.add_sub_structure(STO_sub, 20)  # add 20 layers of STO substrate

# add the double layer to the super lattice
S2.add_sub_structure(DL, 10)  # add 10 double layers to super lattice
S2.add_sub_structure(STO_sub, 500)  # add 500 layers of STO substrate

print(S2)

S2.visualize()
```

```
Structure properties:

Name   : Super Lattice
Thickness : 332.58 nanometer
Roughness : 0.00 nanometer
----
sub-structure 10 times:
        Structure properties:

        Name   : Double Layer
        Thickness : 13.73 nanometer
        Roughness : 0.00 nanometer
        ----
        15 times Strontium Ruthenate: 5.92 nanometer
        20 times Strontium Titanate Substrate: 7.81 nanometer
        ----
        no substrate
500 times Strontium Titanate Substrate: 195.25 nanometer
----
no substrate
```



There a few more interfaces than before (includes also the top and bottom interface):

```
I2 = S2.get_distances_of_interfaces();
print(len(I2))
```

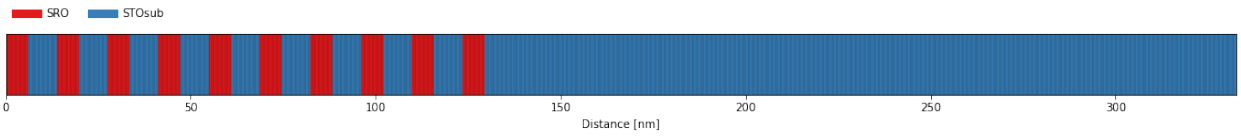
```
21
```

Static Substrate

Mainly for X-ray scattering simulations it might be helpful to add a `static` substrate to the structure, which is not included in the `dynamic` simulations of `heat`, `phonons`, and `magnetization`. Hence the simulation time can be kept short while the scattering result include thick substrate contributions.

```
substrate = ud.Structure('Static Substrate')
substrate.add_sub_structure(STO_sub, 1000000)

S2.add_substrate(substrate)

print(S2)

S2.visualize()
```

```
Structure properties:

Name   : Super Lattice
Thickness : 332.58 nanometer
Roughness : 0.00 nanometer
----
sub-structure 10 times:
        Structure properties:

        Name   : Double Layer
        Thickness : 13.73 nanometer
        Roughness : 0.00 nanometer
        ----
        15 times Strontium Ruthenate: 5.92 nanometer
        20 times Strontium Titanate Substrate: 7.81 nanometer
        ----
        no substrate
500 times Strontium Titanate Substrate: 195.25 nanometer
----
Substrate:
----
1000000 times Strontium Titanate Substrate: 390500.00 nanometer
```



5.3.2 Heat

In this example the laser-excitation of a sample `Structure` is shown. It includes the actual absorption of the laser light as well as the transient temperature profile calculation.

Setup

Do all necessary imports and settings.

```
%load_ext autoreload
%autoreload 2
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

Refer to the *structure-example* for more details.

```
O = ud.Atom('O')
Ti = ud.Atom('Ti')
Sr = ud.Atom('Sr')
Ru = ud.Atom('Ru')
Pb = ud.Atom('Pb')
Zr = ud.Atom('Zr')
```

```
# c-axis lattice constants of the two layers
c_STO_sub = 3.905*u.angstrom
c_SRO = 3.94897*u.angstrom
# sound velocities [nm/ps] of the two layers
sv_SRO = 6.312*u.nm/u.ps
sv_STO = 7.800*u.nm/u.ps

# SRO layer
prop_SRO = {}
prop_SRO['a_axis'] = c_STO_sub  # aAxis
prop_SRO['b_axis'] = c_STO_sub  # bAxis
prop_SRO['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_SRO['sound_vel'] = sv_SRO  # sound velocity
prop_SRO['opt_ref_index'] = 2.44+4.32j
prop_SRO['therm_cond'] = 5.72*u.W/(u.m*u.K)  # heat conductivity
prop_SRO['lin_therm_exp'] = 1.03e-5  # linear thermal expansion
prop_SRO['heat_capacity'] = '455.2 + 0.112*T - 2.1935e6/T**2'  # [J/kg K]

SRO = ud.UnitCell('SRO', 'Strontium Ruthenate', c_SRO, **prop_SRO)
SRO.add_atom(O, 0)
SRO.add_atom(Sr, 0)
```

(continues on next page)

(continued from previous page)

```
SRO.add_atom(O, 0.5)
SRO.add_atom(O, 0.5)
SRO.add_atom(Ru, 0.5)

# STO substrate
prop_STO_sub = {}
prop_STO_sub['a_axis'] = c_STO_sub  # aAxis
prop_STO_sub['b_axis'] = c_STO_sub  # bAxis
prop_STO_sub['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_STO_sub['sound_vel'] = sv_STO  # sound velocity
prop_STO_sub['opt_ref_index'] = 2.1+0j
prop_STO_sub['therm_cond'] = 12*u.W/(u.m*u.K)  # heat conductivity
prop_STO_sub['lin_therm_exp'] = 1e-5  # linear thermal expansion
prop_STO_sub['heat_capacity'] = '733.73 + 0.0248*T - 6.531e6/T**2'  #  [J/kg K]

STO_sub = ud.UnitCell('STOsub', 'Strontium Titanate Substrate',
                      c_STO_sub, **prop_STO_sub)
STO_sub.add_atom(O, 0)
STO_sub.add_atom(Sr, 0)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(Ti, 0.5)
```

```
S = ud.Structure('Single Layer')
S.add_sub_structure(SRO, 100)  # add 100 layers of SRO to sample
S.add_sub_structure(STO_sub, 200)  # add 200 layers of STO substrate
```

Initialize Heat

The `Heat` class requires a `Structure` object and a boolean `force_recalc` in order overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can using the boolean switch `progress_bar`.

```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True

print(h)
```

```
Heat simulation properties:

This is the current structure for the simulations:

Structure properties:

Name   : Single Layer
Thickness : 117.59 nanometer
Roughness : 0.00 nanometer
```

(continues on next page)

(continued from previous page)

```
----
100 times Strontium Ruthenate: 39.49 nanometer
200 times Strontium Titanate Substrate: 78.10 nanometer
----
no substrate


Display properties:

================================  ============================================================
                        parameter  value
================================  ============================================================
                     force recalc  True
                   cache directory  ./
                  display messages  True
                         save data  False
                      progress bar  True
                excitation fluence  [] mJ/cm²
                  excitation delay  [0.0] ps
           excitation pulse length  [0.0] ps
             excitation wavelength  799.9999999999999 nm
                  excitation theta  90.0 deg
excitation multilayer absorption  True
               excitation backside  False
                    heat diffusion  False
         interpolate at interfaces  11
                           backend  scipy
                         distances  no distance mesh is set for heat diffusion calculations
                top boundary type  isolator
             bottom boundary type  isolator
================================  ============================================================
```

Simple Excitation

In order to calculate the temperature of the sample after quasi-instantaneous (delta) photoexcitation the `excitation` must be set with the following parameters:

- `fluence`
- `delay_pump`
- `pulse_width`
- `multilayer_absorption`
- `wavelength`
- `theta`
- `backside`

The angle of incidence `theta` does change the footprint of the excitation on the sample for any type excitation. The `wavelength` and `theta` angle of the excitation are also relevant if `multilayer_absorption = True`. Otherwise the *Lambert_Beer*-law is used and its absorption profile is independent of `wavelength` and `theta`. With `backside = True` the excitation is calculated to happen from the backside of the sample structure.
Note: the `fluence`, `delay_pump`, and `pulse_width` must be given as `array` or `list`.

The simulation requires also a `delay` array as temporal grid as well as an initial temperature `init_temp`. The later can be either a scalar which is then the constant temperature of the whole sample structure, or the initial temperature can be an array of temperatures for each single layer in the structure.

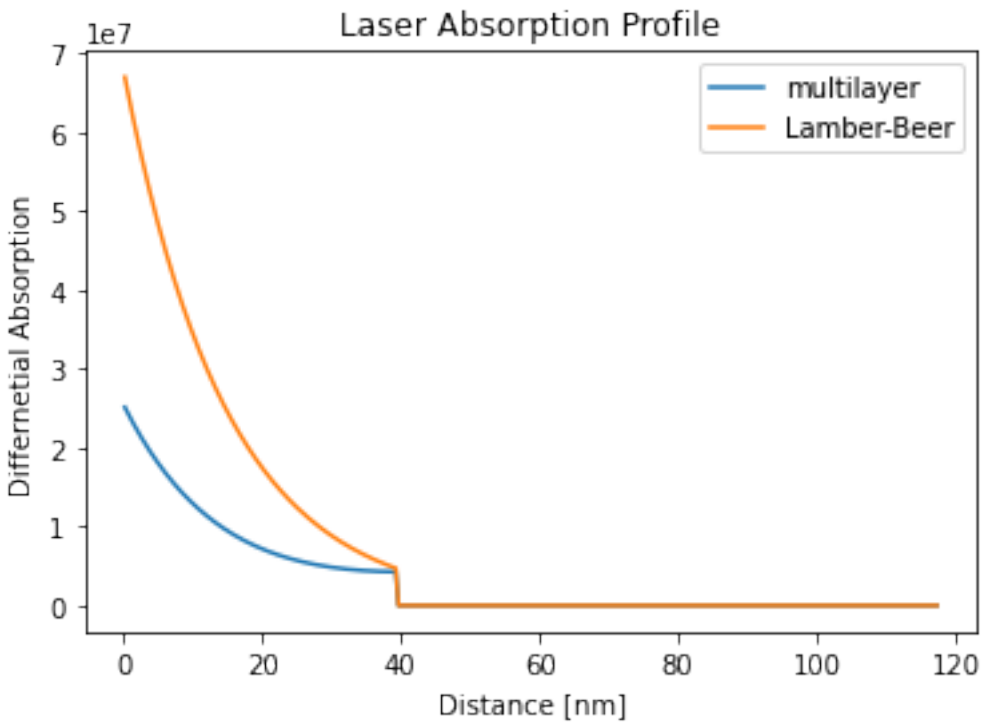
```
h.excitation = {'fluence': [5]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg,
                'backside': False}

# when calculating the laser absorption profile using Lamber-Beer-law
# the opt_pen_depth must be set manually or calculated from the refractive index
SRO.set_opt_pen_depth_from_ref_index(800*u.nm)
STO_sub.set_opt_pen_depth_from_ref_index(800*u.nm)

# temporal and spatial grid
delays = np.r_[-10:200:0.1]*u.ps
_, _, distances = S.get_distances_of_layers()
```

Laser Absorption Profile

Here the difference in the spatial laser absorption profile is shown between the multilayer absorption algorithm and the Lambert-Beer law.
Note that Lambert-Beer does not include reflection of the incident light from the surface of the sample structure:

```
plt.figure()
dAdz, _, _, _ = h.get_multilayers_absorption_profile()
plt.plot(distances.to('nm'), dAdz, label='multilayer')
dAdz = h.get_Lambert_Beer_absorption_profile()
plt.plot(distances.to('nm'), dAdz, label='Lamber-Beer')
plt.legend()
plt.xlabel('Distance [nm]')
plt.ylabel('Differnetial Absorption')
plt.title('Laser Absorption Profile')
plt.show()
```

```
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
Absorption profile is calculated by Lambert-Beer's law.
```

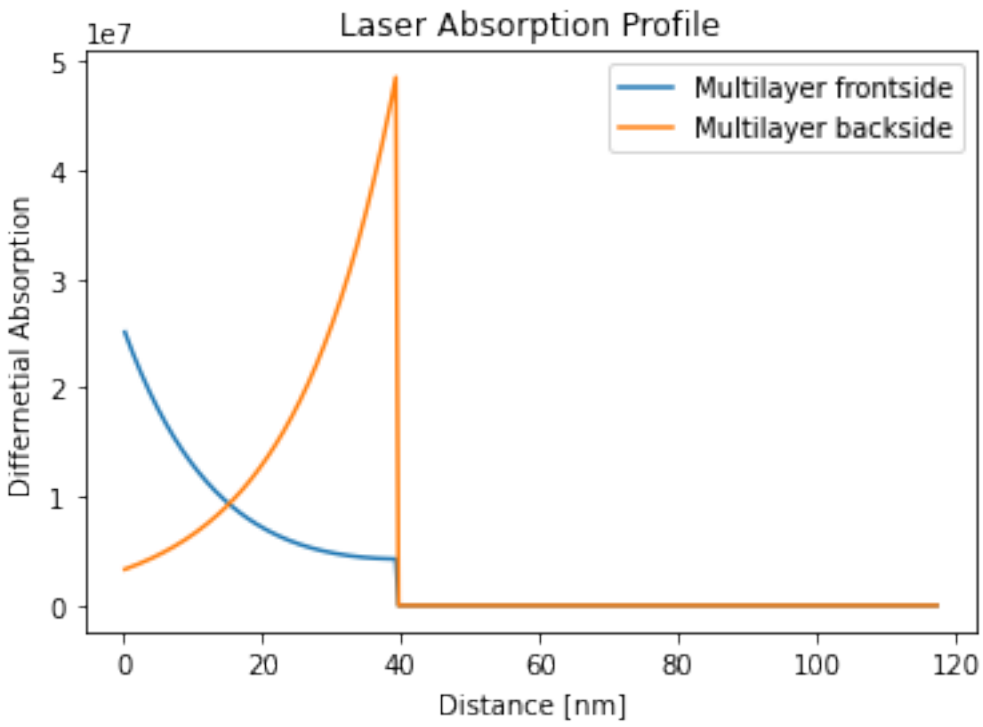


Backside Excitation

The same calculation is repeated with the pump laser entering from the backside of the sample of the structure within the multilayer formalismus.
The clear difference in absorbed energy stems from the significantly different reflectivity of the pump light for front- or backside excitation due to better or worse refractive-index matching.

```
plt.figure()
dAdz, _, _, _ = h.get_multilayers_absorption_profile(backside=False)
plt.plot(distances.to('nm'), dAdz, label='Multilayer frontside')
dAdz, _, _, _ = h.get_multilayers_absorption_profile(backside=True)
plt.plot(distances.to('nm'), dAdz, label='Multilayer backside')
plt.legend()
plt.xlabel('Distance [nm]')
plt.ylabel('Differnetial Absorption')
plt.title('Laser Absorption Profile')
plt.show()
```

```
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
Absorption profile is calculated by multilayer formalism.
Backside excitation is enabled.
Total reflectivity of 28.2 % and transmission of 71.8 %.
```



Temperature Map

Calculating a tranient temperature map is only a one-line command:

```
temp_map, delta_temp = h.get_temp_map(delays, 300*u.K)
```

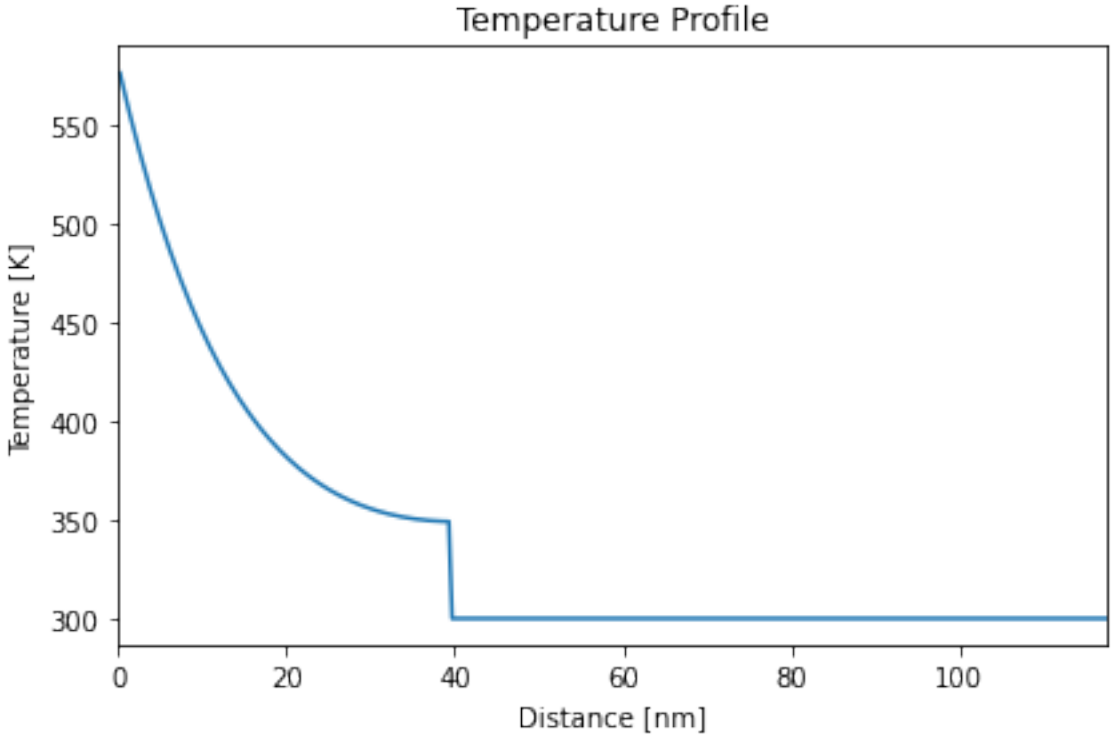
```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
Elapsed time for _temperature_after_delta_excitation_: 0.006274 s
Elapsed time for _temp_map_: 0.049654 s
```

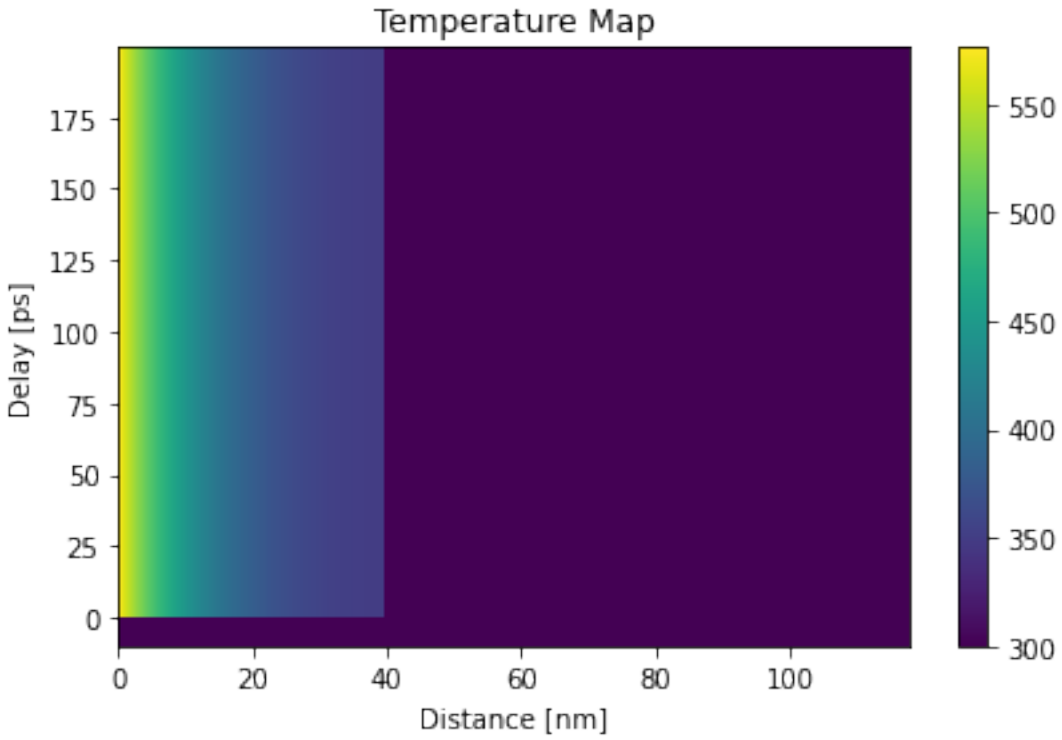
```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :])
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.title('Temperature Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map')
```

(continues on next page)

(continued from previous page)

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





Heat Diffusion

In order to enable heat diffusion the boolean switch `heat_diffusion` must be `True`.
It is also reasonable to define a finte pump `pulse_width`.

```
# enable heat diffusion
h.heat_diffusion = True
# set the boundary conditions
h.boundary_conditions = {'top_type': 'isolator', 'bottom_type': 'isolator'}
# change only the pulse duration
h.excitation = {'pulse_width':  [0.1]*u.ps}
# The resulting temperature profile is calculated in one line:
temp_map, delta_temp = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

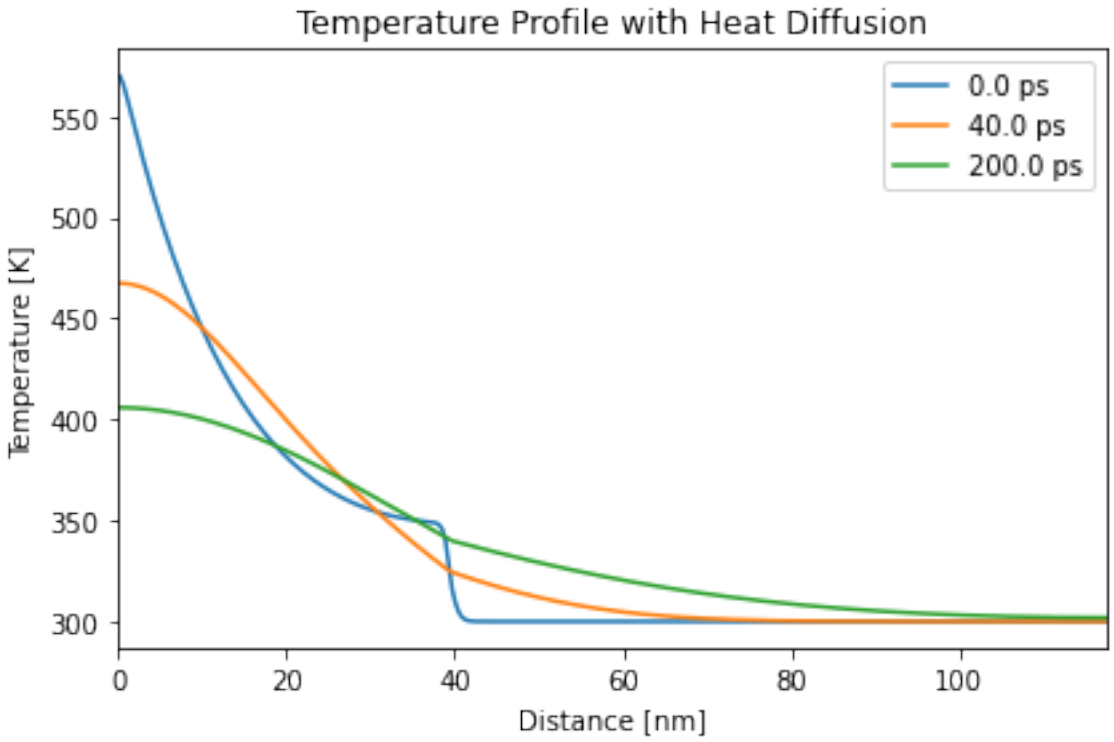
```
0it [00:00, ?it/s]
```

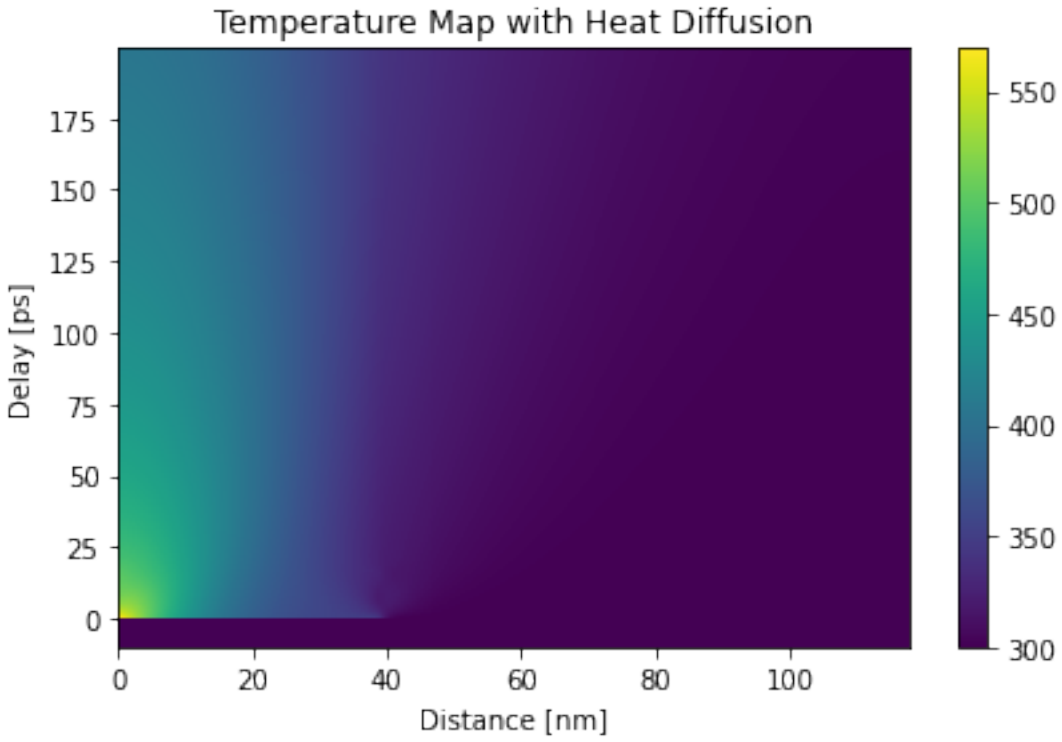
```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 6.334593 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 8.037995 s
Elapsed time for _temp_map_: 14.586600 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :], label=np.round(delays[101]))
plt.plot(distances.to('nm').magnitude, temp_map[501, :], label=np.round(delays[501]))
plt.plot(distances.to('nm').magnitude, temp_map[-1, :], label=np.round(delays[-1]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.legend()
plt.title('Temperature Profile with Heat Diffusion')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map with Heat Diffusion')

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


Temperature Profile with Heat Diffusion
0.0 ps
40.0 ps
200.0 ps
Temperature [K]
550
500
450
400
350
300
0
20
40
60
80
100
Distance [nm]


Temperature Map with Heat Diffusion
Delay [ps]
175
150
125
100
75
50
25
0
0
20
40
60
80
100
Distance [nm]
550
500
450
400
350
300

Heat Diffusion Parameters

For heat diffusion simulations various parameters for the underlying pdepe solver can be altered.

By default, the `backend` is set to `scipy` but can be switched to `matlab`. Currently, the is no obvious reason to choose *MATLAB* above *SciPy*.

Depending on the `backend` either the `ode_options` or `ode_options_matlab` can be configured and are directly handed to the actual solver. Please refer to the documentation of the actual backend and solver and the **API documentation** for more details.

The speed but also the result of the heat diffusion simulation strongly depends on the spatial grid handed to the solver. By default, one spatial grid point is used for every `Layer` (`AmorphousLayer` or `UnitCell`) in the `Structure`. The resulting `temp_map` will also always be interpolated in this spatial grid which is equivalent to the distance vector returned by `S.get_distances_of_layers()`.

As the solver for the heat diffusion usually suffers from large gradients, e.g. of thermal properties or initial temperatures, additional spatial grid points are added by default only for internal calculations. The number of additional points (should be an odd number, default is 11) is set by:

```
h.intp_at_interface = 11
```

The internally used spatial grid can be returned by:

```
dist_interp, original_indicies = S.interp_distance_at_interfaces(h.intp_at_interface)
```

The internal spatial grid can also be given by hand, e.g. to realize logarithmic steps for rather large `Structure`:

```
h.distances = np.linspace(0, distances.magnitude[-1], 100)*u.m
```

As already shown above, the heat diffusion simulation supports also an top and bottom boundary condition. The can have the types:

- `isolator`
- `temperature`
- `flux`

For the later types also a value must be provides:

```
h.boundary_conditions = {'top_type': 'temperature', 'top_value': 500*u.K,
                         'bottom_type': 'flux', 'bottom_value': 5e11*u.W/u.m**2}

print(h)
```

```
Heat simulation properties:

This is the current structure for the simulations:

Structure properties:

Name   : Single Layer
Thickness : 117.59 nanometer
Roughness : 0.00 nanometer
----
100 times Strontium Ruthenate: 39.49 nanometer
```

(continues on next page)

(continued from previous page)

```
200 times Strontium Titanate Substrate: 78.10 nanometer
----
no substrate


Display properties:

================================  ============================================================
                        parameter  value
================================  ============================================================
                     force recalc  True
                  cache directory  ./
                 display messages  True
                        save data  False
                     progress bar  True
               excitation fluence  [5.0] mJ/cm²
                 excitation delay  [0.0] ps
          excitation pulse length  [0.1] ps
            excitation wavelength  800.0 nm
                 excitation theta  45.0 deg
excitation multilayer absorption  True
              excitation backside  False
                   heat diffusion  True
       interpolate at interfaces  11
                         backend  scipy
                       distances  a distance mesh is set for heat diffusion calculations.
                top boundary type  temperature
         top boundary temperature  500 K
             bottom boundary type  flux
             bottom boundary flux  5000000000000.0 W/m²
================================  ============================================================
```

```
# The resulting temperature profile is calculated in one line:
temp_map, delta_temp = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 1.938106 s
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

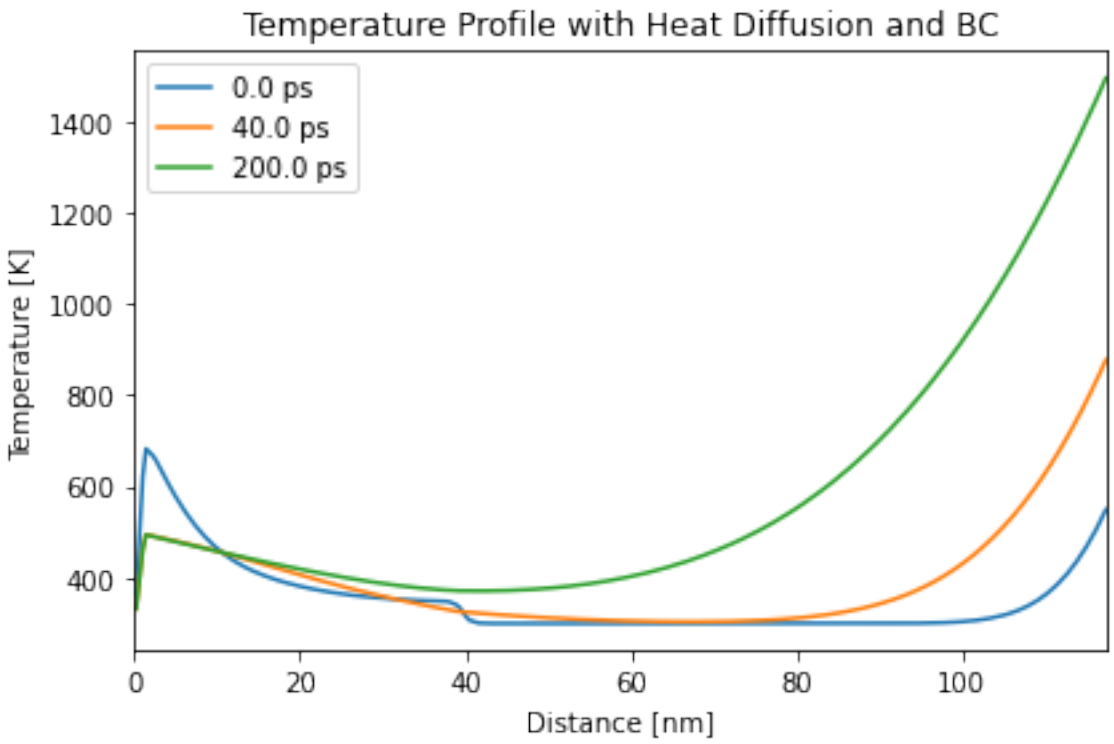
```
0it [00:00, ?it/s]
```

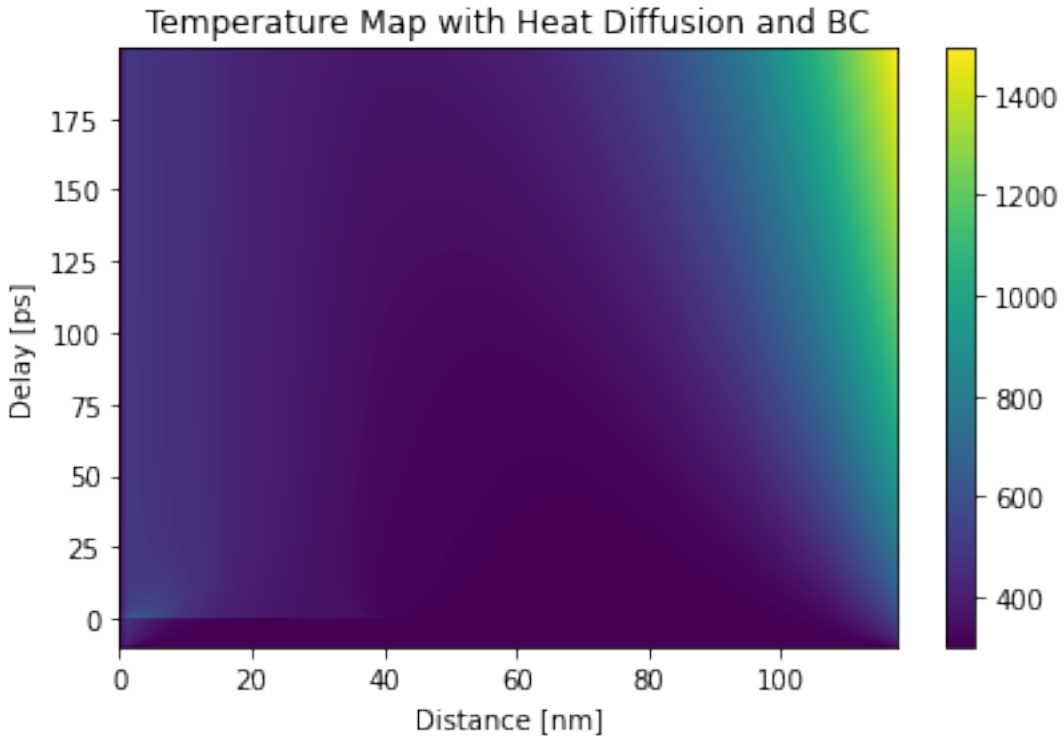
```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 1.289404 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 2.122720 s
Elapsed time for _temp_map_: 5.430157 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :], label=np.round(delays[101]))
plt.plot(distances.to('nm').magnitude, temp_map[501, :], label=np.round(delays[501]))
plt.plot(distances.to('nm').magnitude, temp_map[-1, :], label=np.round(delays[-1]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.legend()
plt.title('Temperature Profile with Heat Diffusion and BC')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map with Heat Diffusion and BC')

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


Temperature Profile with Heat Diffusion and BC
0.0 ps
40.0 ps
200.0 ps
Temperature [K]
1400
1200
1000
800
600
400
0
20
40
60
80
100
Distance [nm]


Temperature Map with Heat Diffusion and BC
Delay [ps]
175
150
125
100
75
50
25
0
0
20
40
60
80
100
Distance [nm]
1400
1200
1000
800
600
400

Multipulse Excitation

As already stated above, also multiple pulses of variable fluence, pulse width and, delay are possible.

The heat diffusion simulation automatically splits the calculation in parts with and without excitation and adjusts the initial temporal step width according to the pulse width. Hence the solver does not miss any excitation pulses when adjusting its temporal step size.

The temporal laser pulse profile is always assumed to be Gaussian and the pulse width must be given as FWHM:

```
h.excitation = {'fluence': [5, 5, 5, 5]*u.mJ/u.cm**2,
                'delay_pump':  [0, 10, 20, 20.5]*u.ps,
                'pulse_width':  [0.1, 0.1, 0.1, 0.5]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg,
                'backside': False
               }

h.boundary_conditions = {'top_type': 'isolator', 'bottom_type': 'isolator'}
```

```
# The resulting temperature profile is calculated in one line:
temp_map, delta_temp = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ for excitation 1:4 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 1.265777 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 1.407974 s
Calculating _heat_diffusion_ for excitation 2:4 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 0.938399 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 0.972477 s
Calculating _heat_diffusion_ for excitation 3-4:4...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

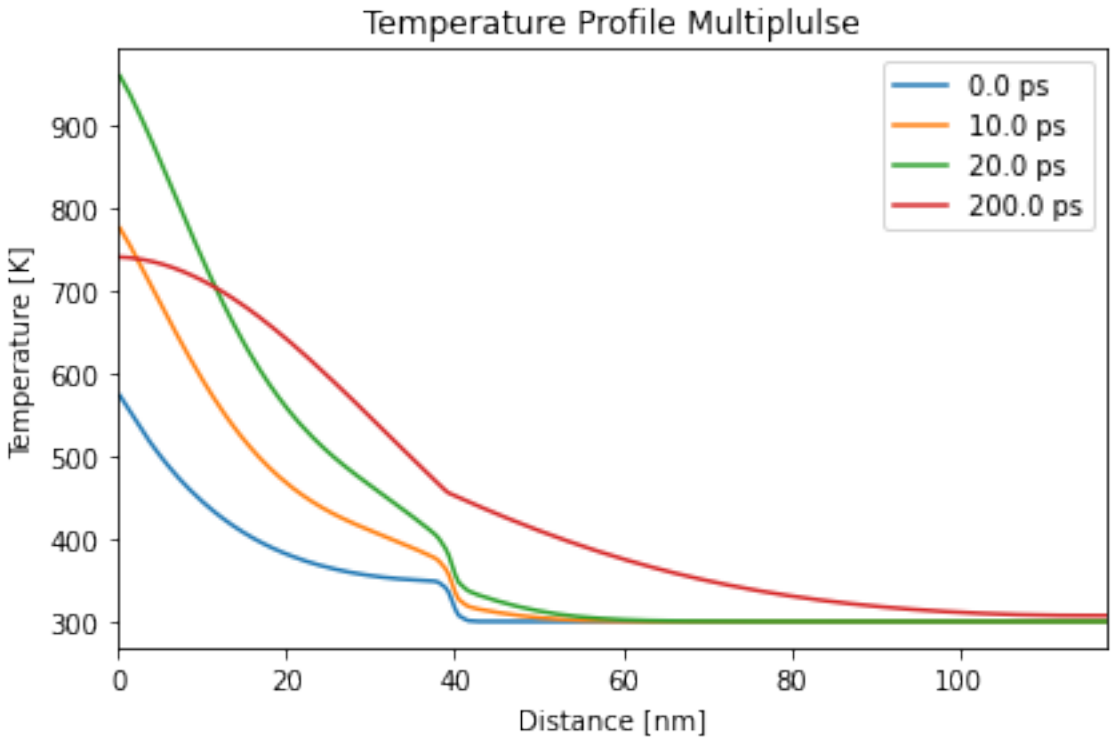
```
0it [00:00, ?it/s]
```

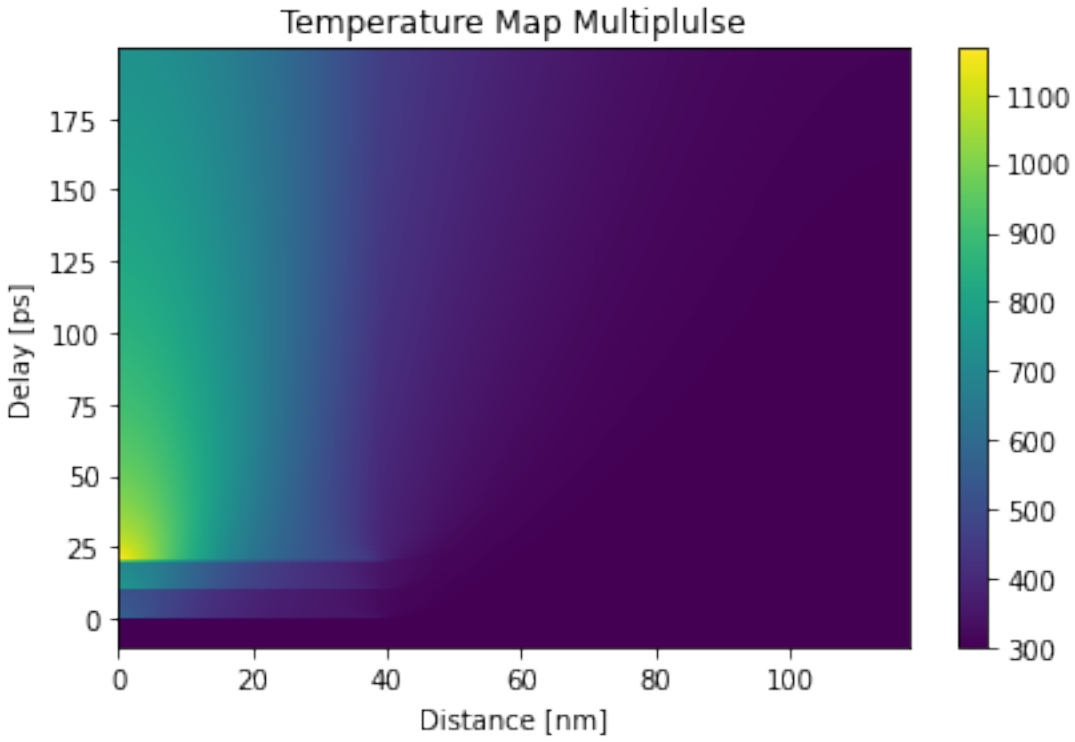
```
Elapsed time for _heat_diffusion_ with 2 excitation(s): 1.425995 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 1.414223 s
Elapsed time for _temp_map_: 7.698703 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :], label=np.round(delays[101]))
plt.plot(distances.to('nm').magnitude, temp_map[201, :], label=np.round(delays[201]))
plt.plot(distances.to('nm').magnitude, temp_map[301, :], label=np.round(delays[301]))
plt.plot(distances.to('nm').magnitude, temp_map[-1, :], label=np.round(delays[-1]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.legend()
plt.title('Temperature Profile Multiplulse')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Multiplulse')

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


Temperature Profile Multiplulse
0.0 ps
10.0 ps
20.0 ps
200.0 ps
Temperature [K]
900
800
700
600
500
400
300
0
20
40
60
80
100
Distance [nm]


Temperature Map Multiplulse
Delay [ps]
175
150
125
100
75
50
25
0
0
20
40
60
80
100
Distance [nm]
1100
1000
900
800
700
600
500
400
300

N-Temperature Model

The heat diffusion is also capable of simulating an N-temperature model which is often applied to empirically simulate the energy flow between *electrons*, *phonons*, and *spins*. In order to run the *NTM* all thermo-elastic properties must be given as a list of N elements corresponding to different sub-systems. The actual external laser-excitation is always set to happen within the **first** sub-system, which is usually the electron-system.

In addition the `sub_system_coupling` must be provided in order to allow for energy-flow between the sub-systems. `sub_system_coupling` is often set to a constant prefactor multiplied with the difference between the electronic and phononic temperatures, as in the example below. For sufficiently high temperatures, this prefactor also depdends on temperature. See here for an overview.

In case the thermo-elastic parameters are provided as functions of the temperature T, the `sub_system_coupling` requires the temperature `T` to be a vector of all sub-system-temperatures which can be accessed in the function string via the underscore-notation. The `heat_capacity` and `lin_therm_exp` instead require the temperature `T` to be a scalar of only the current sub-system-temperature. For the `therm_cond` both options are available.

```
# update the relevant thermo-elastic properties of the layers in the sample
# structure
SRO.therm_cond = [0,
                  5.72*u.W/(u.m*u.K)]
SRO.lin_therm_exp = [1.03e-5,
                     1.03e-5]
SRO.heat_capacity = ['0.112*T',
                     '455.2 - 2.1935e6/T**2']
SRO.sub_system_coupling = ['5e17*(T_1-T_0)',
                           '5e17*(T_0-T_1)']

STO_sub.therm_cond = [0,
                      12*u.W/(u.m*u.K)]
STO_sub.lin_therm_exp = [1e-5,
                         1e-5]
STO_sub.heat_capacity = ['0.0248*T',
                         '733.73 - 6.531e6/T**2']
STO_sub.sub_system_coupling = ['5e17*(T_1-T_0)',
                               '5e17*(T_0-T_1)']
```

```
Number of subsystems changed from 1 to 2.
Number of subsystems changed from 1 to 2.
```

As no new `Structure` is build, the `num_sub_systems` must be updated by hand. Otherwise this happens automatically.

```
S.num_sub_systems = 2
```

Set the excitation conditions:

```
h.excitation = {'fluence': [5]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0.25]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg,
                'backside': False
```

(continues on next page)

(continued from previous page)

```
                }

h.boundary_conditions = {'top_type': 'isolator', 'bottom_type': 'isolator'}

delays = np.r_[-5:15:0.01]*u.ps
```

```
# The resulting temperature profile is calculated in one line:
temp_map, delta_temp = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
```

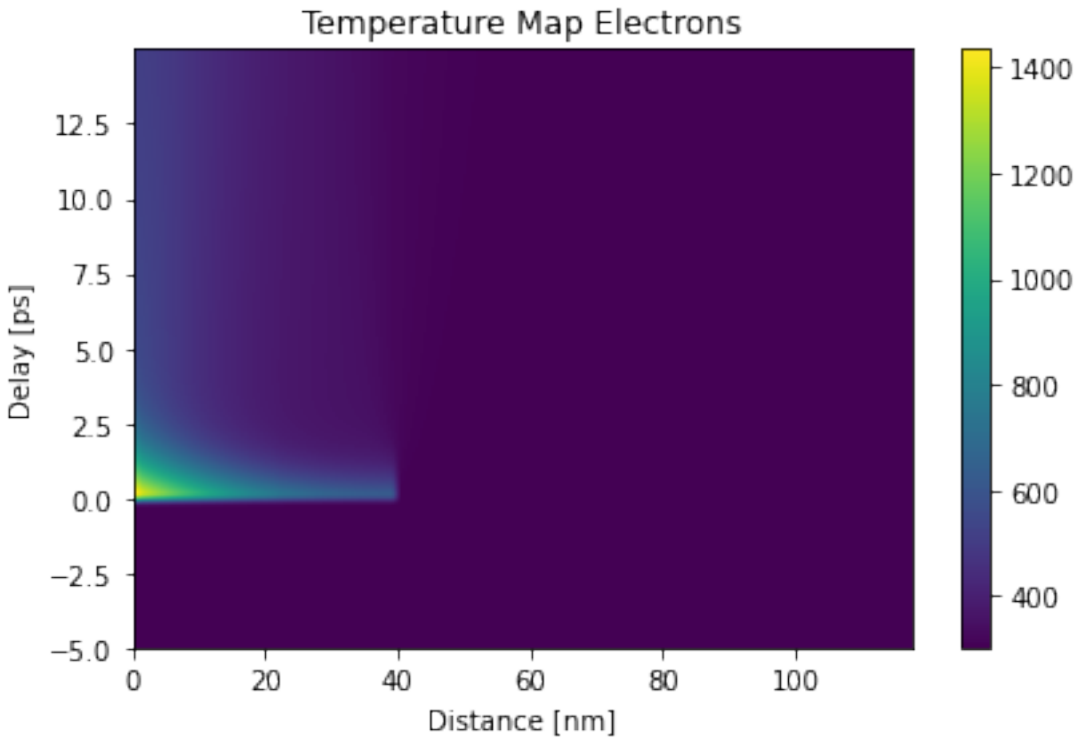
```
0it [00:00, ?it/s]
```

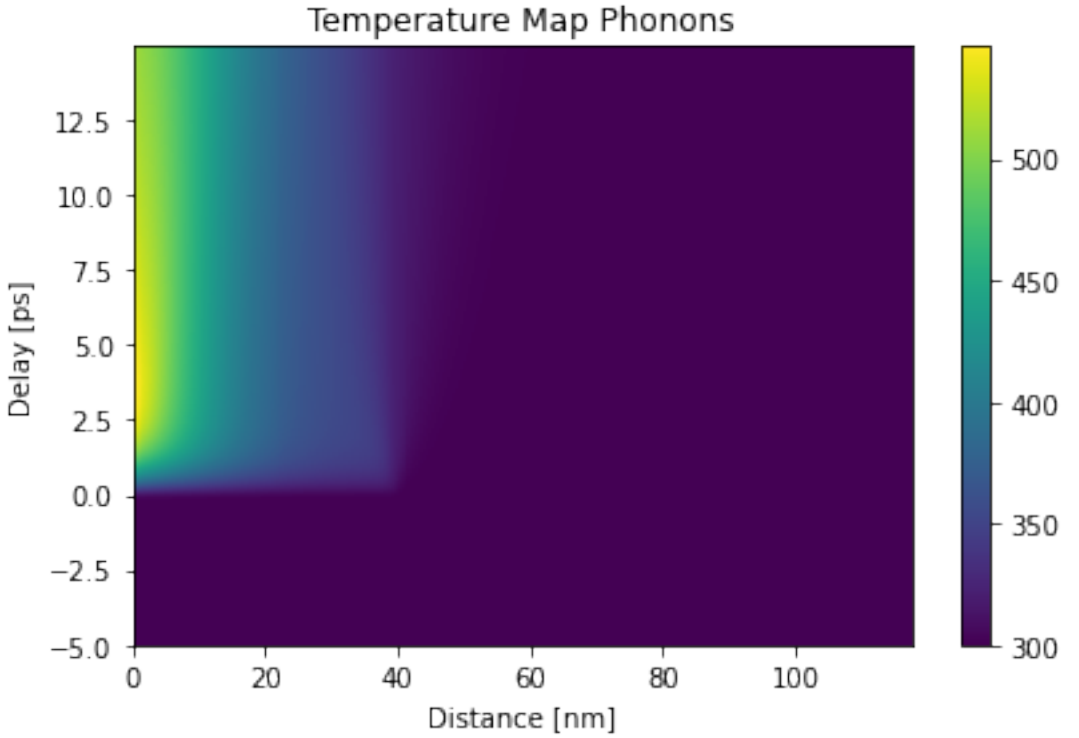
```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 3.675223 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 4.096187 s
Elapsed time for _temp_map_: 7.930553 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪0],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Electrons')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪1],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Phonons')

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



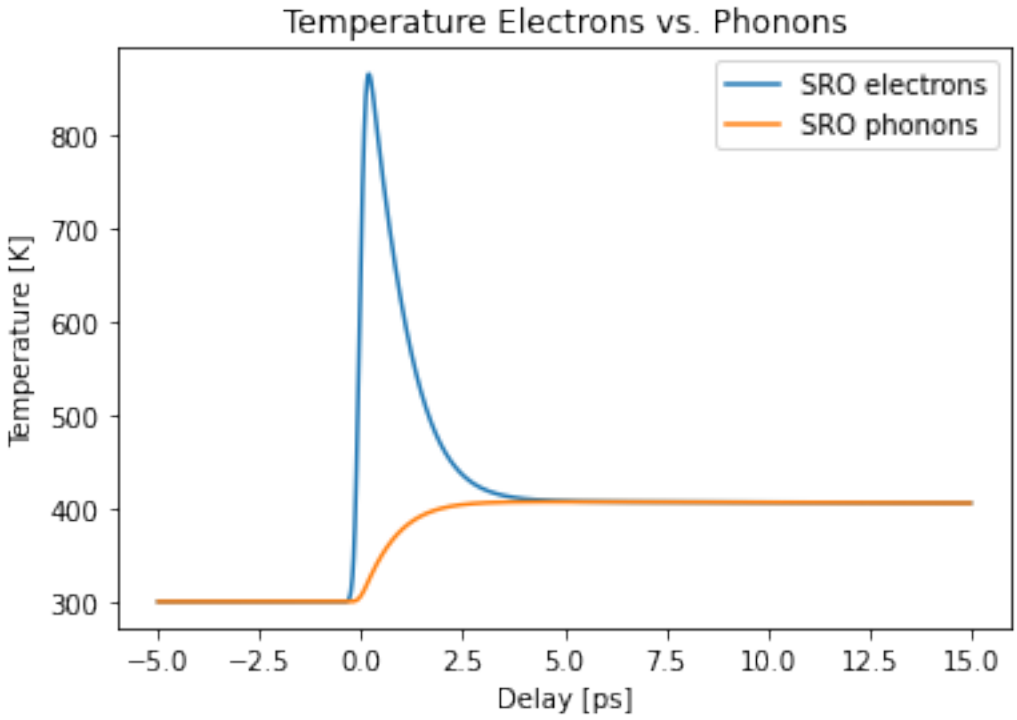



```
plt.figure()
select = S.get_all_positions_per_unique_layer()['SRO']
plt.plot(delays.to('ps'), np.mean(temp_map[:, select, 0], 1), label='SRO electrons')
plt.plot(delays.to('ps'), np.mean(temp_map[:, select, 1], 1), label='SRO phonons')
plt.ylabel('Temperature [K]')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Delay [ps]')
plt.legend()
plt.title('Temperature Electrons vs. Phonons')
plt.show()
```



5.3.3 Microscopic 3-Temperature-Model

Here we adapt the NTM from the last example to allow for calculations of the magnetization within the microscopic 3-temperature-model as proposed by:

Koopmans, B., Malinowski, G., Dalla Longa, F. et al.
Explaining the paradoxical diversity of ultrafast laser-induced demagnetization.
Nature Mater 9, 259–265 (2010).

We need to solve the following coupled differential equations:

$$
\begin{aligned}
c_e(T_e)\rho\frac{\partial T_e}{\partial t} &= \frac{\partial}{\partial z}\left(k_e(T_e)\frac{\partial T_e}{\partial z}\right) - G_{ep}(T_e - T_p) + S(z,t) \\
c_p(T_p)\rho\frac{\partial T_p}{\partial t} &= \frac{\partial}{\partial z}\left(k_p(T_p)\frac{\partial T_p}{\partial z}\right) + G_{ep}(T_e - T_p) \\
\frac{\partial m}{\partial t} &= Rm\frac{T_p}{T_C}\left(1 - m\coth\left(\frac{mT_C}{T_e}\right)\right)
\end{aligned}
$$

We treat the temperature of the 3rd subsystem as magnetization m. For that we have to set its `heat_capacity` to $1/\rho$ and `thermal_conductivity` to zero. We put the complete right term of the last equation in the `sub_system_coupling` term for the 3rd subsystem. Here, we need to rewrite the kotangens hyperbolicus in Python as

$$\coth(x) = 1 + \frac{2}{e^{2x} - 1}$$

The values of the used parameters are not experimentally verified.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

to the *structure-example* for more details.

```
Co = ud.Atom('Co')
Ni = ud.Atom('Ni')
CoNi = ud.AtomMixed('CoNi')
CoNi.add_atom(Co, 0.5)
CoNi.add_atom(Ni, 0.5)
Si = ud.Atom('Si')
```

```
prop_CoNi = {}
prop_CoNi['heat_capacity'] = ['0.1*T',
                              532*u.J/u.kg/u.K,
                              1/7000
                             ]
prop_CoNi['therm_cond'] = [20*u.W/(u.m*u.K),
                           80*u.W/(u.m*u.K),
                           0]

R = 25.3/1e-12
Tc = 1388
g = 4.0e18

prop_CoNi['sub_system_coupling'] = \
    ['-{:f}*(T_0-T_1)'.format(g),
     '{:f}*(T_0-T_1)'.format(g),
     '{0:f}*T_2*T_1/{1:f}*(1-T_2* (1 + 2/(exp(2*T_2*{1:f}/T_0) - 1) ))'.format(R, Tc)
    ]
prop_CoNi['lin_therm_exp'] = [0, 11.8e-6, 0]
prop_CoNi['sound_vel'] = 4.910*u.nm/u.ps
prop_CoNi['opt_ref_index'] = 2.9174+3.3545j

layer_CoNi = ud.AmorphousLayer('CoNi', 'CoNi amorphous', thickness=1*u.nm,
                               density=7000*u.kg/u.m**3, atom=CoNi, **prop_CoNi)
```

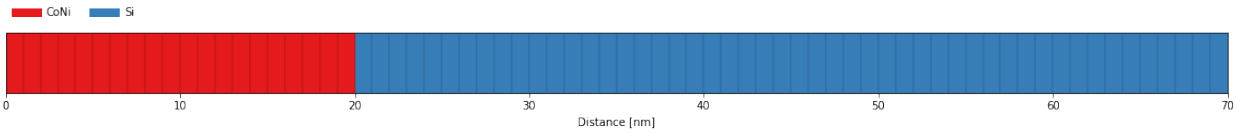
```
Number of subsystems changed from 1 to 3.
```

```
prop_Si = {}
prop_Si['heat_capacity'] = [100*u.J/u.kg/u.K, 603*u.J/u.kg/u.K, 1]
prop_Si['therm_cond'] = [0, 100*u.W/(u.m*u.K), 0]

prop_Si['sub_system_coupling'] = [0, 0, 0]

prop_Si['lin_therm_exp'] = [0, 2.6e-6, 0]
prop_Si['sound_vel'] = 8.433*u.nm/u.ps
prop_Si['opt_ref_index'] = 3.6941+0.0065435j

layer_Si = ud.AmorphousLayer('Si', "Si amorphous", thickness=1*u.nm, density=2336*u.kg/u.
↪m**3,
                             atom=Si, **prop_Si)
```

```
Number of subsystems changed from 1 to 3.
```

```
S = ud.Structure('CoNi')

S.add_sub_structure(layer_CoNi, 20)
S.add_sub_structure(layer_Si, 50)
```

```
S.visualize()
```



Initialize Heat and the Excitation

```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True
```

```
h.excitation = {'fluence': [30]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0.05]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg}
# temporal and spatial grid
delays = np.r_[-1:5:0.005]*u.ps
_, _, distances = S.get_distances_of_layers()
```

Calculate Heat Diffusion

The `init_temp` sets the magnetization in the 3rd subsystem to 1 for CoNi and `0` for Si.

```
# enable heat diffusion
h.heat_diffusion = True
# set the boundary conditions
h.boundary_conditions = {'top_type': 'isolator', 'bottom_type': 'isolator'}
# The resulting temperature profile is calculated in one line:

init_temp = np.ones([S.get_number_of_layers(), 3])
init_temp[:, 0] = 300
init_temp[:, 1] = 300
init_temp[20:, 2] = 0

temp_map, delta_temp = h.get_temp_map(delays, init_temp)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 1.647954 s
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 42.4 % and transmission of 28.4 %.
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 5.000816 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

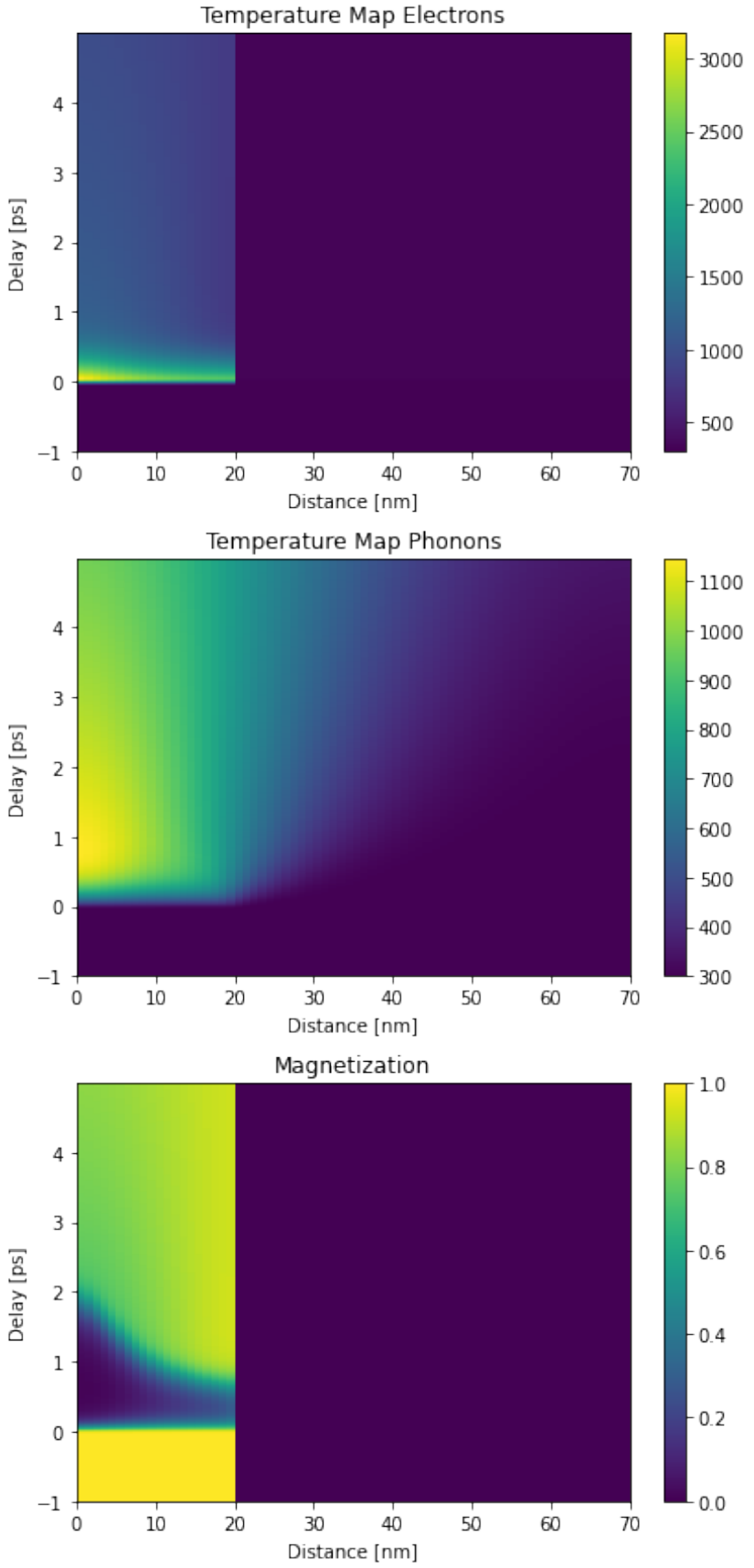
```
Elapsed time for _heat_diffusion_: 14.045075 s
Elapsed time for _temp_map_: 20.788493 s
```

```
plt.figure(figsize=[6, 12])
plt.subplot(3, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪0],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Electrons')

plt.subplot(3, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪1],
               shading='auto')
```

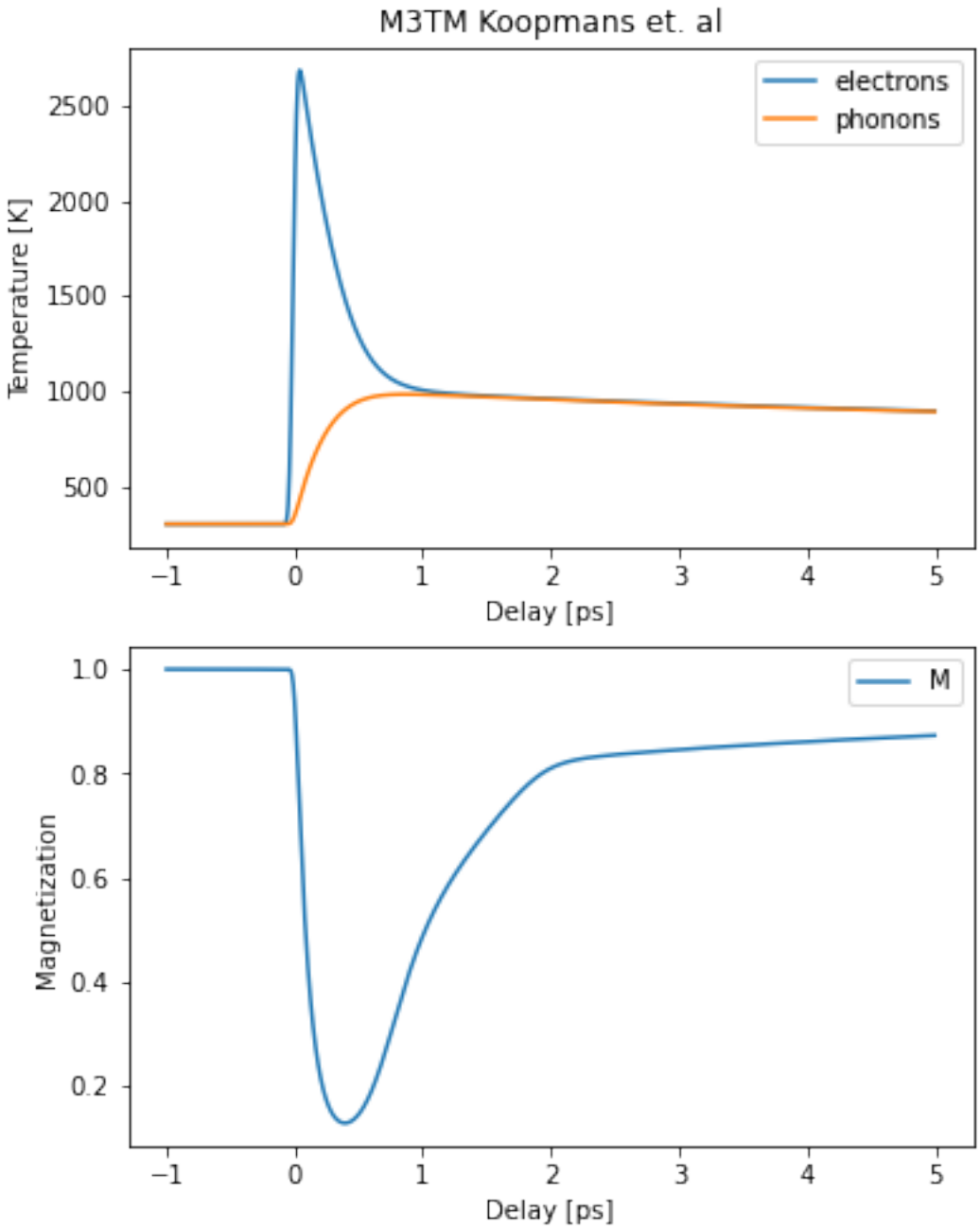
(continues on next page)

(continued from previous page)

```
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Phonons')

plt.subplot(3, 1, 3)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               temp_map[:, :, 2], shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Magnetization')

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


Temperature Map Electrons
Delay [ps]
Distance [nm]
Temperature Map Phonons
Delay [ps]
Distance [nm]
Magnetization
Delay [ps]
Distance [nm]

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
select = S.get_all_positions_per_unique_layer()['CoNi']
plt.plot(delays.to('ps'), np.mean(temp_map[:, select, 0], 1), label='electrons')
plt.plot(delays.to('ps'), np.mean(temp_map[:, select, 1], 1), label='phonons')
plt.ylabel('Temperature [K]')
plt.xlabel('Delay [ps]')
plt.legend()
plt.title('M3TM Koopmans et. al')

plt.subplot(2, 1, 2)

plt.plot(delays.to('ps'), np.mean(temp_map[:, select, 2], 1), label='M')
plt.ylabel('Magnetization')
plt.xlabel('Delay [ps]')
plt.legend()
plt.show()
```



5.3.4 Landau-Lifshitz-Bloch simulations

Here we calculate the vectorial magnetization dynamic in a magnetic heterostructure employing the mean-field quantum Landau-Lifshitz-Bloch (LLB) apporach. Please read the following review to get an overview of the LLB equation in ultrafast magnetism

U. Atxitia, D. Hinzke, and U. Nowak,
Fundamentals and Applications of the Landau-Lifshitz-Bloch Equation, J. Phys. D. Appl. Phys. 50, (2017).

Here we need to solve the following differential equation:

$$\frac{d\mathbf{m}}{dt} = \gamma_e \left(\mathbf{m} \times \mathbf{H}_{\text{eff}} + \frac{\alpha_\perp}{m^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{H}_{\text{eff}}) - \frac{\alpha_\parallel}{m^2} (\mathbf{m} \cdot \mathbf{H}_{\text{eff}}) \cdot \mathbf{m} \right)$$

The three terms describe

1. **precession** at Larmor frequency,
2. **transversal damping** (conserving the macrospin length), and
3. **longitudinal damping** (changing macrospin length due to incoherent atomistic spin excitations within the layer the macrospin is defined on).

$\alpha_\parallel$ and $\alpha_\perp$ are the longitudinal damping and transverse damping parameters, respectively. $\gamma_e = -1.761 \times 10^{11}\,\mathrm{rad\,s^{-1}\,T^{-1}}$ is the gyromagnetic ratio of an electron.

The effective magnetic field is the sum of all relevant magnetic interactions:

$$\mathbf{H}_{\text{eff}} = \mathbf{H}_{\text{ext}} + \mathbf{H}_{\text{A}} + \mathbf{H}_{\text{ex}} + \mathbf{H}_{\text{th}}$$

where

- $\mathbf{H}_{\text{ext}}$ is the external magnetic field
- $\mathbf{H}_{\text{A}}$ is the uniaxial anisotropy field
- $\mathbf{H}_{\text{ex}}$ is the exchange field
- $\mathbf{H}_{\text{th}}$ is the thermal field

The definitions of these subterms are described in their respective *API documentation*.

The material parameters of the current example arbitrarily chosen.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import time
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

Refer to the *structure-example* for more details.
We are providing a magnetization of the atoms which are used as initial magnetization in the LLB simulations, in case no specific initial condition is provided by the user.

```
Co = ud.Atom('Co', mag_amplitude=1, mag_gamma=0*u.deg, mag_phi=0*u.deg)
Ni = ud.Atom('Ni', mag_amplitude=1, mag_gamma=90*u.deg, mag_phi=0*u.deg)
Fe = ud.Atom('Fe', mag_amplitude=1, mag_gamma=0*u.deg, mag_phi=90*u.deg)
Si = ud.Atom('Si')
```

Solving the LLB requires several additional parameters of the according `Layer` objects:

- `eff_spin` - effective spin
- `curie_temp` - Curie temperature
- `lamda` - parameter used in the longitudinal & transversal damping terms $\alpha_{\parallel}$ & $\alpha_{\perp}$ (misspelled because of python `lambda`-functions)
- `mag_moment` - atomic magnetic moment
- `aniso_exponent` - exponent of the uniaxial anisotropy
- `anisotropy` - vector of the anisotropy $[K_x, K_y, K_z]$
- `exch_stiffness` - vector of the exchange stiffness $[A_{i\rightarrow(i-1)}, A_{i\rightarrow i}, A_{i\rightarrow(i+1)}]$
- `mag_saturation` - zero temperature saturation magnetization

The correct physical units and more details are documented in the *Layer API*.

```
prop_Ni = {}
# two-temperture model
prop_Ni['heat_capacity'] = ['0.1*T',
                            532*u.J/u.kg/u.K,
                           ]
prop_Ni['therm_cond'] = [20*u.W/(u.m*u.K),
                         80*u.W/(u.m*u.K),]
g = 4.0e18 # electron-phonon coupling
prop_Ni['sub_system_coupling'] = \
    ['-{:f}*(T_0-T_1)'.format(g),
     '{:f}*(T_0-T_1)'.format(g)
    ]
prop_Ni['lin_therm_exp'] = [0, 11.8e-6]
prop_Ni['opt_ref_index'] = 2.9174+3.3545j

# LLB parameters
prop_Ni['eff_spin'] = 0.5
prop_Ni['curie_temp'] = 630*u.K
prop_Ni['lamda'] = 0.005
prop_Ni['mag_moment'] = 0.393*u.bohr_magneton
prop_Ni['aniso_exponent'] = 3
prop_Ni['anisotropy'] = [0.45e6, 0.45e6, 0.45e6]*u.J/u.m**3
prop_Ni['exch_stiffness'] = [5e-14, 5e-14, 5e-14]*u.J/u.m
prop_Ni['mag_saturation'] = 500e3*u.J/u.T/u.m**3

# build the layer
layer_Ni = ud.AmorphousLayer('Ni', 'Ni amorphous', thickness=1*u.nm,
                             density=7000*u.kg/u.m**3, atom=Ni, **prop_Ni)
```

```
Number of subsystems changed from 1 to 2.
```

```
# similar to Ni layer
prop_Co = {}
prop_Co['heat_capacity'] = ['0.1*T',
                            332*u.J/u.kg/u.K,
                           ]
```

(continues on next page)

(continued from previous page)

```
prop_Co['therm_cond'] = [20*u.W/(u.m*u.K),
                         80*u.W/(u.m*u.K),]
g = 5.0e18
prop_Co['sub_system_coupling'] = \
    ['-{:f}*(T_0-T_1)'.format(g),
     '{:f}*(T_0-T_1)'.format(g)
    ]
prop_Co['lin_therm_exp'] = [0, 11.8e-6]
prop_Co['opt_ref_index'] = 2.9174+3.3545j

prop_Co['eff_spin'] = 3
prop_Co['curie_temp'] = 1480*u.K
prop_Co['lamda'] = 0.005
prop_Co['mag_moment'] = 0.393*u.bohr_magneton
prop_Co['aniso_exponent'] = 3
prop_Co['anisotropy'] = [0.45e6, 0.45e6, 0.45e6]*u.J/u.m**3
prop_Co['exch_stiffness'] = [5e-14, 5e-14, 5e-14]*u.J/u.m
prop_Co['mag_saturation'] = 1400e3*u.J/u.T/u.m**3

layer_Co = ud.AmorphousLayer('Co', 'Co amorphous', thickness=1*u.nm,
                             density=7000*u.kg/u.m**3, atom=Co, **prop_Co)
```

```
Number of subsystems changed from 1 to 2.
```

```
# similar to Ni layer
prop_Fe = {}
prop_Fe['heat_capacity'] = ['0.1*T',
                            732*u.J/u.kg/u.K,
                           ]
prop_Fe['therm_cond'] = [20*u.W/(u.m*u.K),
                         80*u.W/(u.m*u.K),]
g = 6.0e18
prop_Fe['sub_system_coupling'] = \
    ['-{:f}*(T_0-T_1)'.format(g),
     '{:f}*(T_0-T_1)'.format(g)
    ]
prop_Fe['lin_therm_exp'] = [0, 11.8e-6]
prop_Fe['opt_ref_index'] = 2.9174+3.3545j

prop_Fe['eff_spin'] = 2
prop_Fe['curie_temp'] = 1024*u.K
prop_Fe['lamda'] = 0.005
prop_Fe['mag_moment'] = 2.2*u.bohr_magneton
prop_Fe['aniso_exponent'] = 3
prop_Fe['anisotropy'] = [0.45e6, 0.45e6, 0.45e6]*u.J/u.m**3
prop_Fe['exch_stiffness'] = [5e-14, 5e-14, 5e-14]*u.J/u.m
prop_Fe['mag_saturation'] = 200e3*u.J/u.T/u.m**3

layer_Fe = ud.AmorphousLayer('Fe', 'Fe amorphous', thickness=1*u.nm,
                             density=7000*u.kg/u.m**3, atom=Fe, **prop_Fe)
```

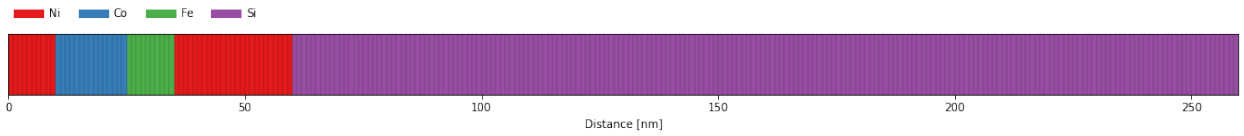
```
Number of subsystems changed from 1 to 2.
```

```
# this is the non-magnetic substrate
prop_Si = {}
prop_Si['heat_capacity'] = [100*u.J/u.kg/u.K, 603*u.J/u.kg/u.K]
prop_Si['therm_cond'] = [0, 100*u.W/(u.m*u.K)]

prop_Si['sub_system_coupling'] = [0, 0]

prop_Si['lin_therm_exp'] = [0, 2.6e-6]
prop_Si['sound_vel'] = 8.433*u.nm/u.ps
prop_Si['opt_ref_index'] = 3.6941+0.0065435j

layer_Si = ud.AmorphousLayer('Si', "Si amorphous", thickness=1*u.nm, density=2336*u.kg/u.
↪m**3,
                             atom=Si, **prop_Si)
```

```
Number of subsystems changed from 1 to 2.
```

```
S = ud.Structure('NiCoFeNi')

S.add_sub_structure(layer_Ni, 10)
S.add_sub_structure(layer_Co, 15)
S.add_sub_structure(layer_Fe, 10)
S.add_sub_structure(layer_Ni, 25)
S.add_sub_structure(layer_Si, 200)
```

```
S.visualize()
```



Initialize Heat and the Excitation

```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True
```

```
h.excitation = {'fluence': [25]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0.15]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg}
# temporal and spatial grid
delays = np.r_[-10:10:0.05, 10:200:0.5]*u.ps
_, _, distances = S.get_distances_of_layers()
```

Calculate Heat Diffusion for 2-Temperature Model

```
# enable heat diffusion
h.heat_diffusion = True
# set the boundary conditions
h.boundary_conditions = {'top_type': 'isolator', 'bottom_type': 'isolator'}
# The resulting temperature profile is calculated in one line:

temp_map, delta_temp = h.get_temp_map(delays, 300)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 46.5 % and transmission of 4.0 %.
```

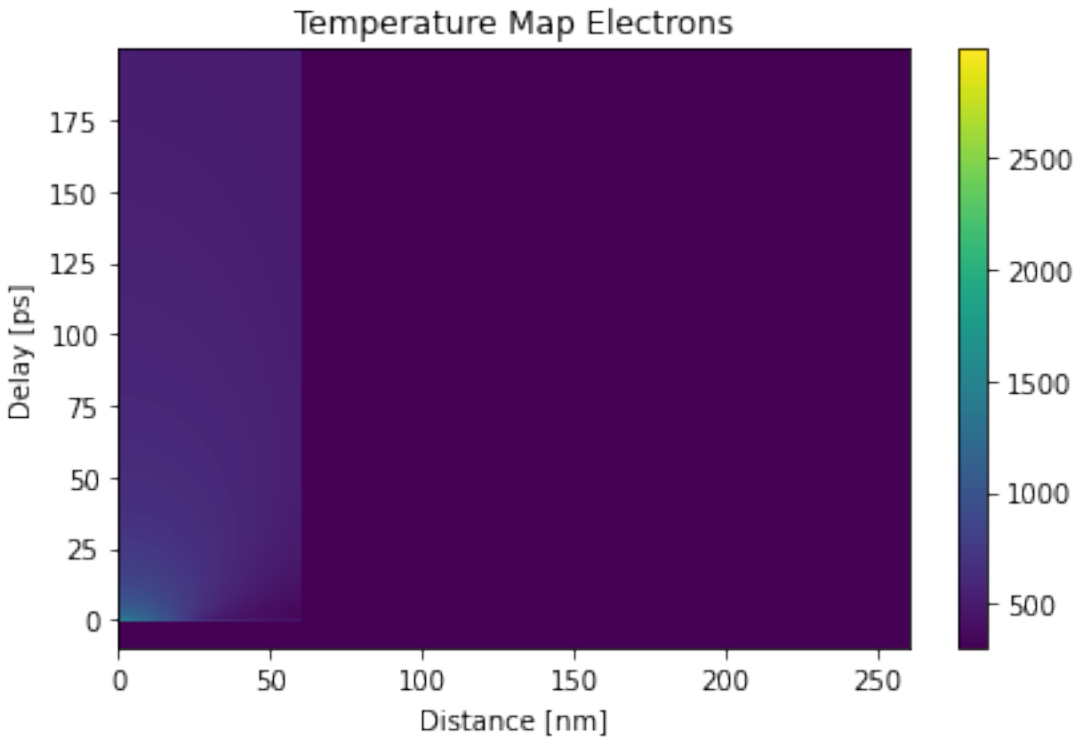
```
0it [00:00, ?it/s]
```

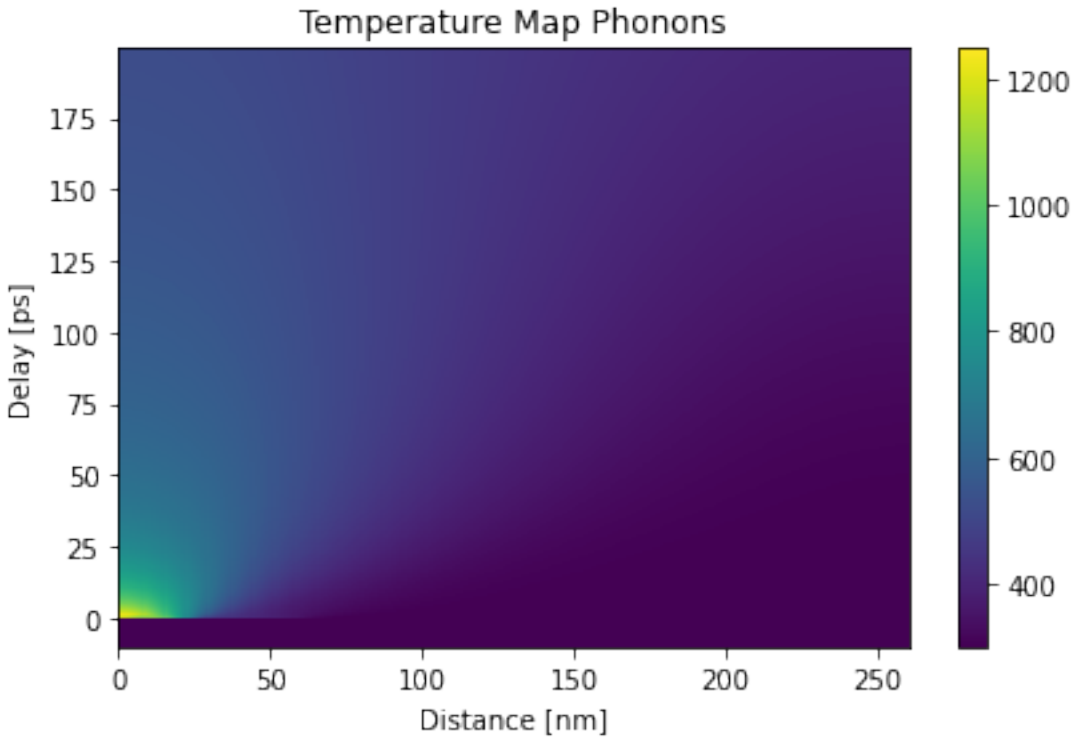
```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 20.389669 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 35.628406 s
Elapsed time for _temp_map_: 56.187758 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪0],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Electrons')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪1],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Phonons')

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


Temperature Map Electrons
Delay [ps]
Distance [nm]
0
25
50
75
100
125
150
175
0
50
100
150
200
250
500
1000
1500
2000
2500


Temperature Map Phonons
Delay [ps]
Distance [nm]
0
25
50
75
100
125
150
175
0
50
100
150
200
250
400
600
800
1000
1200

Landau-Lifshitz-Bloch Simulations

The `LLB` class requires a `Structure` object and a boolean `force_recalc` in order to overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can be enabled using the boolean switch `progress_bar`.

```
llb = ud.LLB(S, True)

llb.save_data = False
llb.disp_messages = True

print(llb)
```

```
Landau-Lifshitz-Bloch Magnetization Dynamics simulation properties:

Magnetization simulation properties:

This is the current structure for the simulations:

Structure properties:

Name   : NiCoFeNi
Thickness : 260.00 nanometer
Roughness : 0.00 nanometer
----
10 times Ni amorphous: 10.00 nanometer
15 times Co amorphous: 15.00 nanometer
10 times Fe amorphous: 10.00 nanometer
25 times Ni amorphous: 25.00 nanometer
200 times Si amorphous: 200.00 nanometer
----
no substrate


Display properties:

================  =======
       parameter  value
================  =======
    force recalc  True
 cache directory  ./
display messages  True
       save data  False
    progress bar  True
================  =======
```

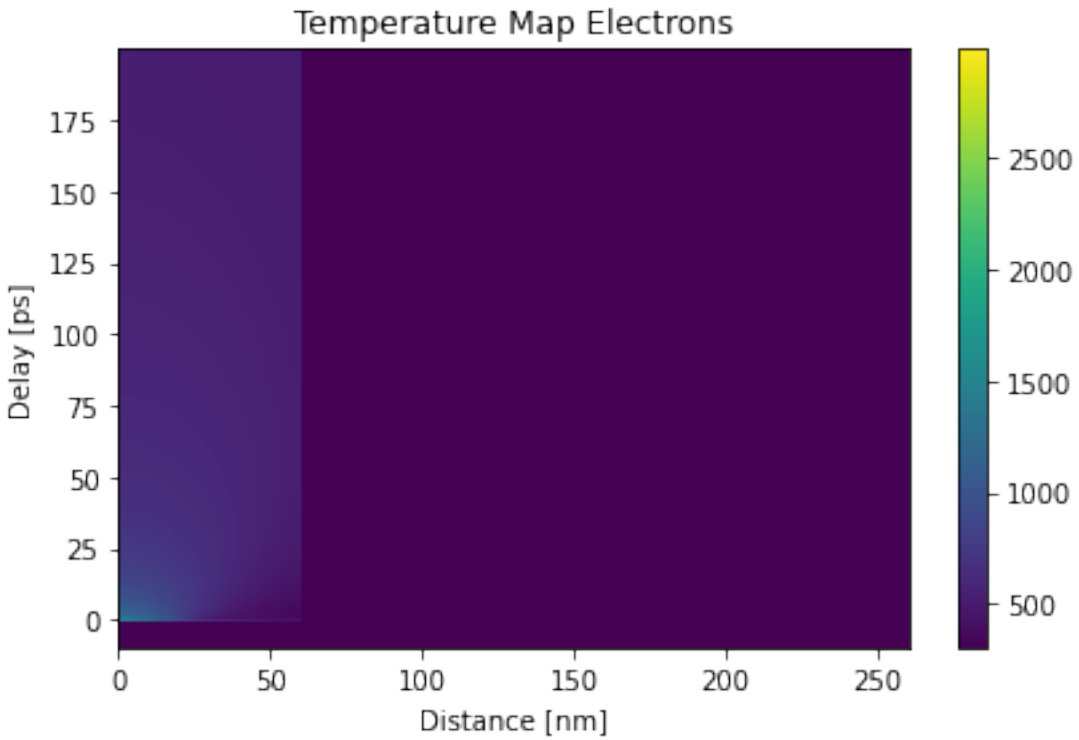
Brillouin Function

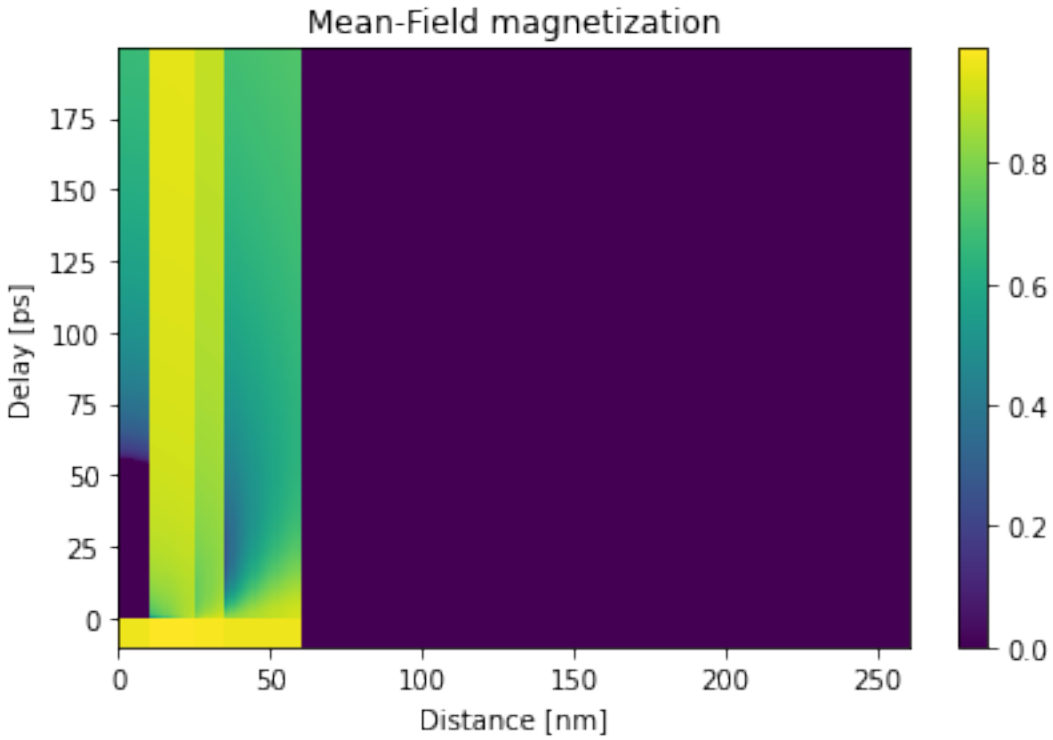
Internally, the LLB calculates a mean-field magnetization map for the according electron temperatures T_e at for every layer and for every time step. This is done by solving the *Brillouin* function of each layer and then mapping the result onto the according spatio-temporal grid, as given by the `temp_map`.

```
mean_field_mag_map = llb.get_mean_field_mag_map(temp_map[:, :, 0])
```

```
Calculating _mean_field_magnetization_map_ ...
Elapsed time for _mean_field_magnetization_map_: 1.736737 s
```

```
plt.figure(figsize=[6, 8])

plt.subplot(2, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map[:, :,␣
↪0],
               shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map Electrons')

plt.subplot(2, 1, 2)

plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, mean_field_mag_
↪map, shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Mean-Field magnetization')

plt.tight_layout()

plt.show()
```





In order to run the actual LLB simulation, an optional initial magnetization `init_mag` as well as an external magnetic field `H_ext` can be provided. While `H_ext` is provided in cartesian coordinates $[H_x, H_y, H_z]$ in Tesla, the initial magnetization is given in polar coordinates $[A, \phi, \gamma]$ in units of [none, rad, rad], following the definitions shown in the *user guide*.

Running the simulation is done by calling the `get_magnetization_map` method which requires a `delay` vector as

well as the temp_map. The resulting magnetization_map is returned in **polar coordinates**.

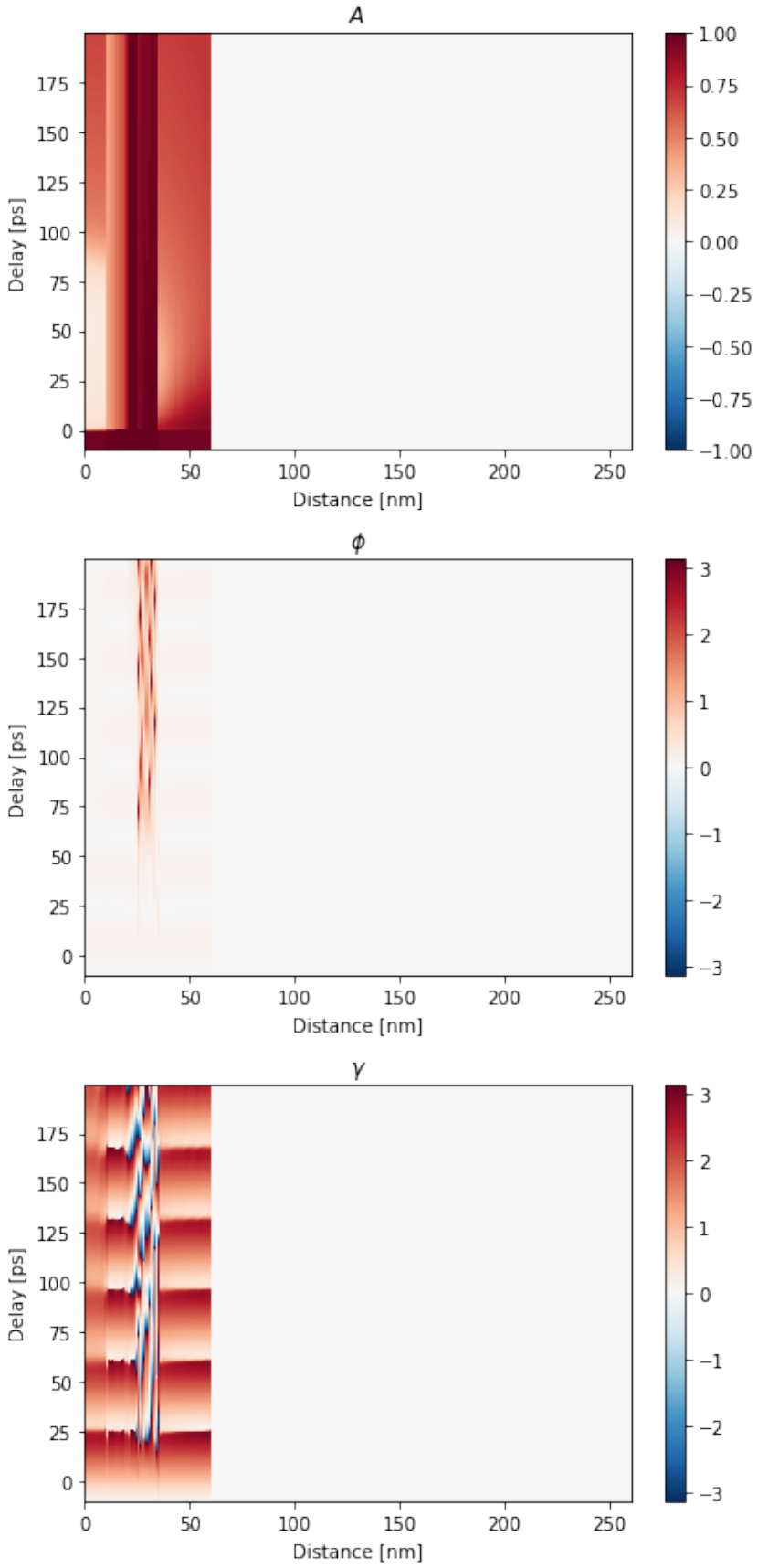
```
init_mag = np.array([1.0, (0.*u.deg).to('rad').magnitude, (0*u.deg).to('rad').magnitude])

magnetization_map = llb.get_magnetization_map(delays, temp_map=temp_map, init_mag=init_
↪mag,
                                              H_ext=np.array([0, 0.05, 1]))
```

```
Calculating _magnetization_map_ ...
Calculating _mean_field_magnetization_map_ ...
Elapsed time for _mean_field_magnetization_map_: 1.756780 s
```

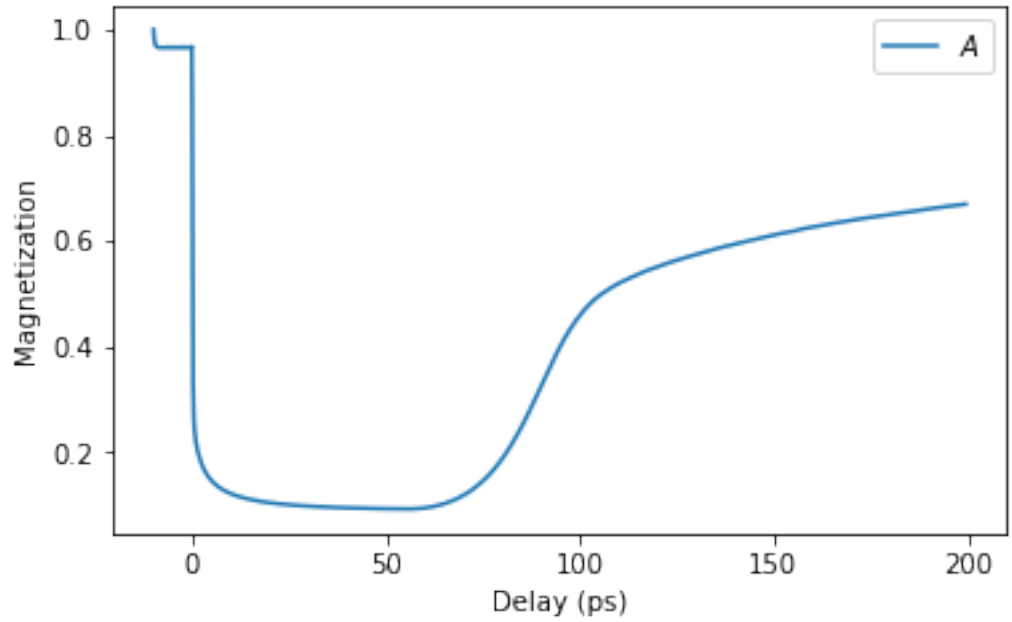
```
0it [00:00, ?it/s]
```

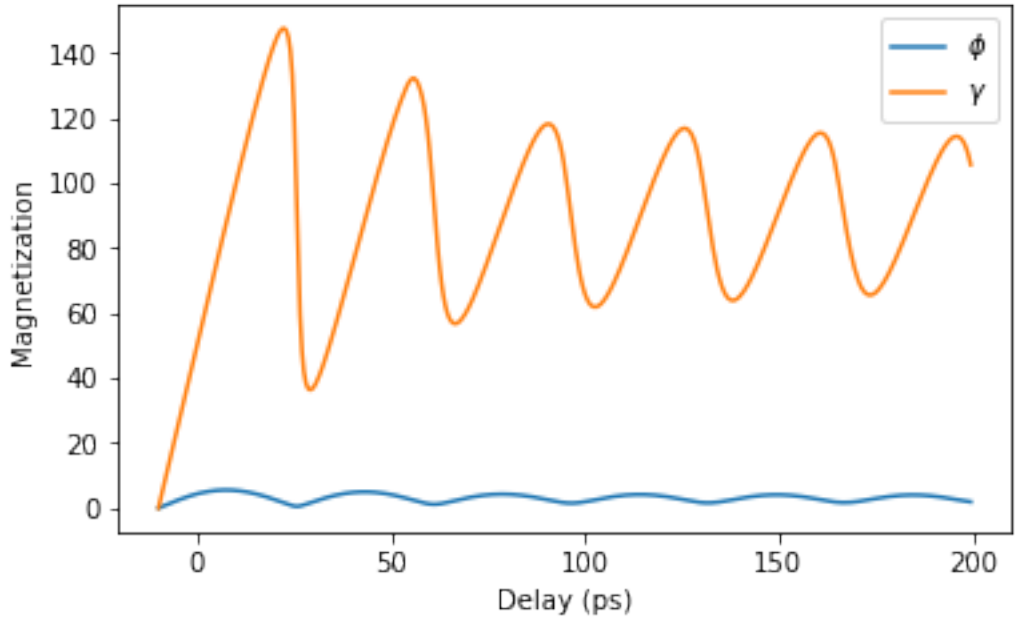
```
Elapsed time for _LLB_: 2.209331 s
Elapsed time for _magnetization_map_: 2.210315 s
```

```
plt.figure(figsize=[6, 12])
plt.subplot(3, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map[:, :, 0],
               shading='auto', cmap='RdBu_r', vmin=-1, vmax=1)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('$A$')

plt.subplot(3, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map[:, :, 1],
               shading='auto', cmap='RdBu_r', vmin=-3.14, vmax=3.14)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('$\phi$')

plt.subplot(3, 1, 3)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map[:, :, 2],
               shading='auto', cmap='RdBu_r', vmin=-3.14, vmax=3.14)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('$\gamma$')

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


A
Delay [ps]
Distance [nm]
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
ϕ
Delay [ps]
Distance [nm]
3
2
1
0
−1
−2
−3
γ
Delay [ps]
Distance [nm]
3
2
1
0
−1
−2
−3

```
plt.figure(figsize=[6,8])
plt.subplot(2,1,1)
plt.plot(delays, np.mean(magnetization_map[:, 0:10, 0], axis=1), label=r'$A$')
plt.legend()
plt.xlabel('Delay (ps)')
plt.ylabel('Magnetization')
plt.subplot(2,1,2)
plt.plot(delays, (np.mean(magnetization_map[:, 0:10, 1], axis=1)*u.rad).to('deg'),␣
↪label=r'$\phi$')
plt.plot(delays, (np.mean(magnetization_map[:, 0:10, 2], axis=1)*u.rad).to('deg'),␣
↪label=r'$\gamma$')
plt.legend()
plt.xlabel('Delay (ps)')
plt.ylabel('Magnetization')
plt.show()
```





The `helper` modules provides functions to convert the result also into cartesian coordinates:

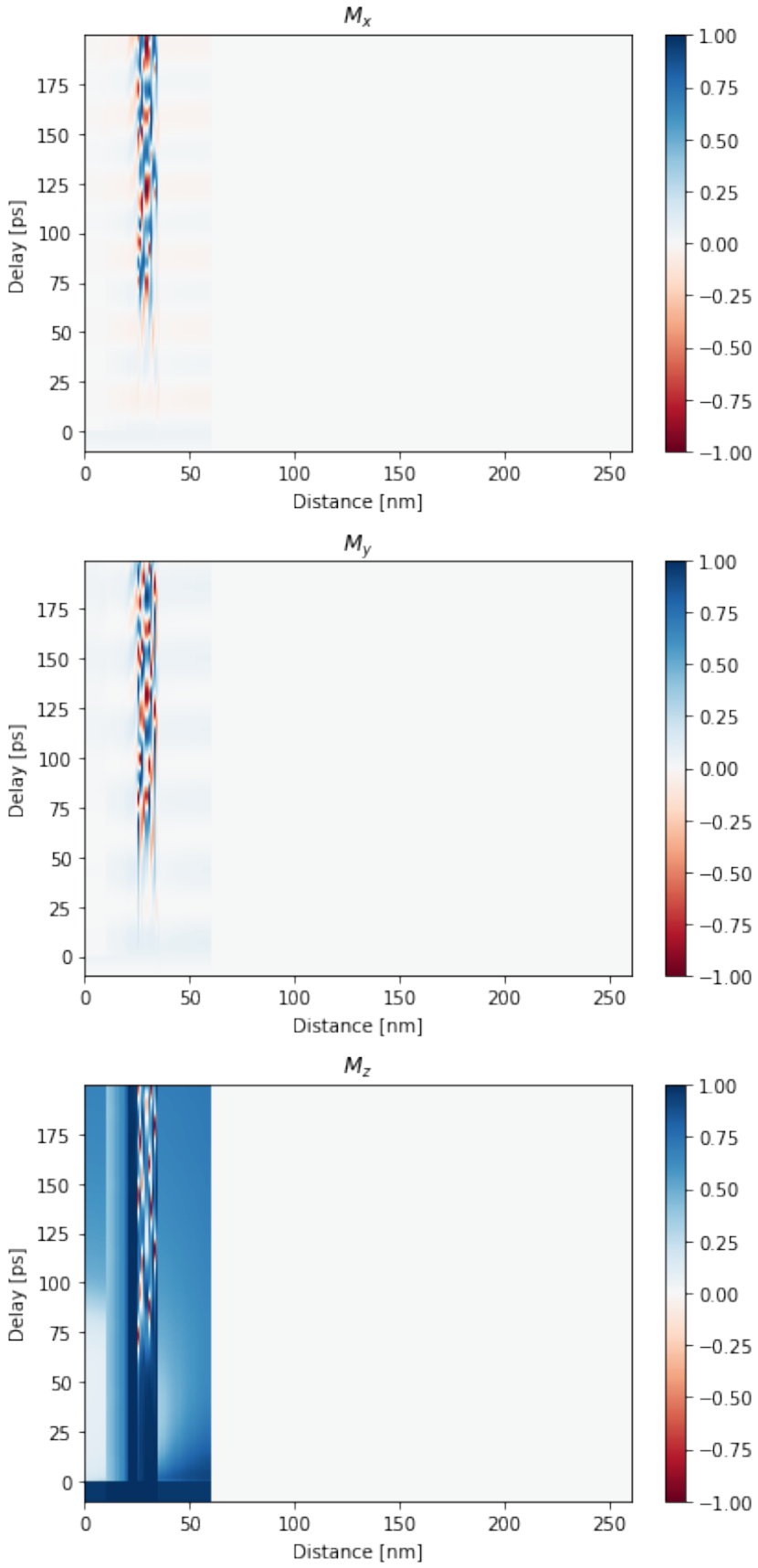
```
magnetization_map_xyz = ud.helpers.convert_polar_to_cartesian(magnetization_map)
```

```
plt.figure(figsize=[6, 12])
plt.subplot(3, 1, 1)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map_xyz[:, :, 0],
               shading='auto', cmap='RdBu', vmin=-1, vmax=1)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
```

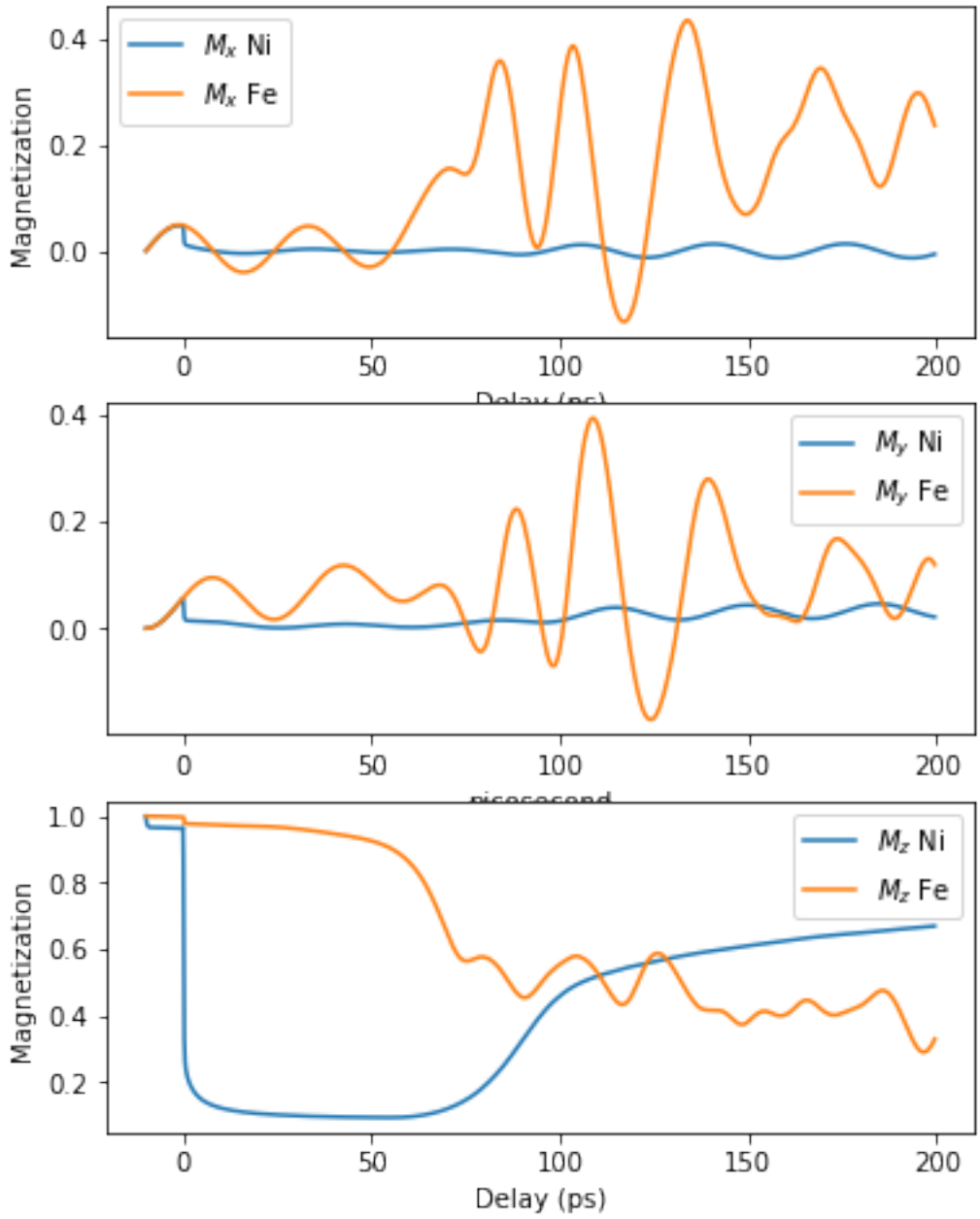
(continues on next page)

(continued from previous page)

```
plt.title('$M_x$')

plt.subplot(3, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map_xyz[:, :, 1],
               shading='auto', cmap='RdBu', vmin=-1, vmax=1)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('$M_y$')

plt.subplot(3, 1, 3)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, magnetization_
↪map_xyz[:, :, 2],
               shading='auto', cmap='RdBu', vmin=-1, vmax=1)
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('$M_z$')

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


M_x
M_y
M_z
Delay [ps]
Distance [nm]
0
25
50
75
100
125
150
175
0
50
100
150
200
250
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

```
plt.figure(figsize=[6,8])
plt.subplot(3,1,1)
plt.plot(delays, np.mean(magnetization_map_xyz[:, 0:10, 0], axis=1), label=r'$M_x$ Ni')
plt.plot(delays, np.mean(magnetization_map_xyz[:, 25:35, 0], axis=1), label=r'$M_x$ Fe')
plt.legend()
plt.xlabel('Delay (ps)')
plt.ylabel('Magnetization')
plt.subplot(3,1,2)
plt.plot(delays, np.mean(magnetization_map_xyz[:, 0:10, 1], axis=1), label=r'$M_y$ Ni')
plt.plot(delays, np.mean(magnetization_map_xyz[:, 25:35, 1], axis=1), label=r'$M_y$ Fe')
plt.legend()
plt.subplot(3,1,3)
plt.plot(delays, np.mean(magnetization_map_xyz[:, 0:10, 2], axis=1), label=r'$M_z$ Ni')
plt.plot(delays, np.mean(magnetization_map_xyz[:, 25:35, 2], axis=1), label=r'$M_z$ Fe')
plt.legend()
plt.xlabel('Delay (ps)')
plt.ylabel('Magnetization')
plt.show()
```



In both subplots it is obvious that the magnetization already starts to change before the actual laser excitation raises the electronic temperatures. This is because of the misalignment of the initial magnetization and the external magnetic field. To avoid this, the steady-state solution of the layer magnetization must be found first and used as `init_mag` for the actual transient simulations.

5.3.5 Phonons

In this example coherent acoustic phonon dynamics are calculated according to the results of the `heat` simulations.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

Refer to the *structure-example* for more details.

```
O = ud.Atom('O')
Ti = ud.Atom('Ti')
Sr = ud.Atom('Sr')
Ru = ud.Atom('Ru')
Pb = ud.Atom('Pb')
Zr = ud.Atom('Zr')
```

```
# c-axis lattice constants of the two layers
c_STO_sub = 3.905*u.angstrom
c_SRO = 3.94897*u.angstrom
# sound velocities [nm/ps] of the two layers
sv_SRO = 6.312*u.nm/u.ps
sv_STO = 7.800*u.nm/u.ps

# SRO layer
prop_SRO = {}
prop_SRO['a_axis'] = c_STO_sub  # aAxis
prop_SRO['b_axis'] = c_STO_sub  # bAxis
prop_SRO['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_SRO['sound_vel'] = sv_SRO  # sound velocity
prop_SRO['opt_ref_index'] = 2.44+4.32j
prop_SRO['therm_cond'] = 5.72*u.W/(u.m *u.K)  # heat conductivity
prop_SRO['lin_therm_exp'] = 1.03e-5  # linear thermal expansion
prop_SRO['heat_capacity'] = '455.2 + 0.112*T - 2.1935e6/T**2'  # [J/kg K]

SRO = ud.UnitCell('SRO', 'Strontium Ruthenate', c_SRO, **prop_SRO)
SRO.add_atom(O, 0)
SRO.add_atom(Sr, 0)
SRO.add_atom(O, 0.5)
SRO.add_atom(O, 0.5)
SRO.add_atom(Ru, 0.5)
```

(continues on next page)

(continued from previous page)

```
# STO substrate
prop_STO_sub = {}
prop_STO_sub['a_axis'] = c_STO_sub  # aAxis
prop_STO_sub['b_axis'] = c_STO_sub  # bAxis
prop_STO_sub['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_STO_sub['sound_vel'] = sv_STO  # sound velocity
prop_STO_sub['opt_ref_index'] = 2.1+0j
prop_STO_sub['therm_cond'] = 12*u.W/(u.m *u.K)  # heat conductivity
prop_STO_sub['lin_therm_exp'] = 1e-5  # linear thermal expansion
prop_STO_sub['heat_capacity'] = '733.73 + 0.0248*T - 6.531e6/T**2'  # [J/kg K]

STO_sub = ud.UnitCell('STOsub', 'Strontium Titanate Substrate',
                      c_STO_sub, **prop_STO_sub)
STO_sub.add_atom(O, 0)
STO_sub.add_atom(Sr, 0)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(Ti, 0.5)
```

```
S = ud.Structure('Single Layer')
S.add_sub_structure(SRO, 100)  # add 100 layers of SRO to sample
S.add_sub_structure(STO_sub, 2000)  # add 1000 layers of STO substrate
```

Heat

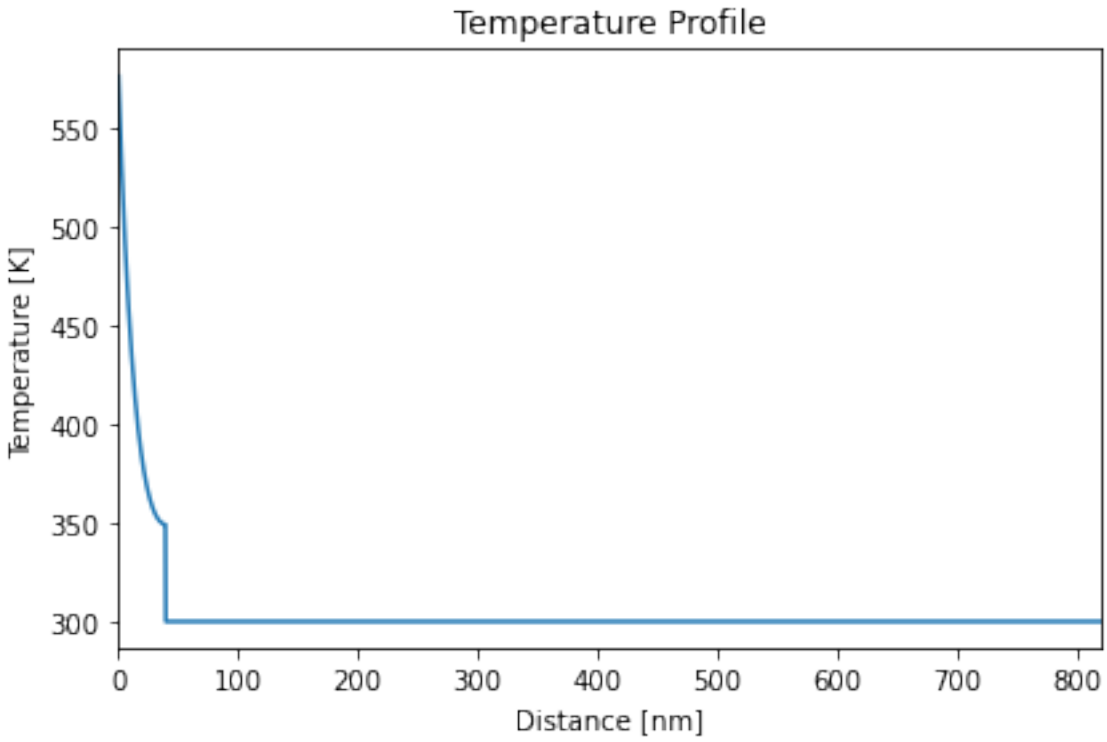
Refer to the *heat-example* for more details.

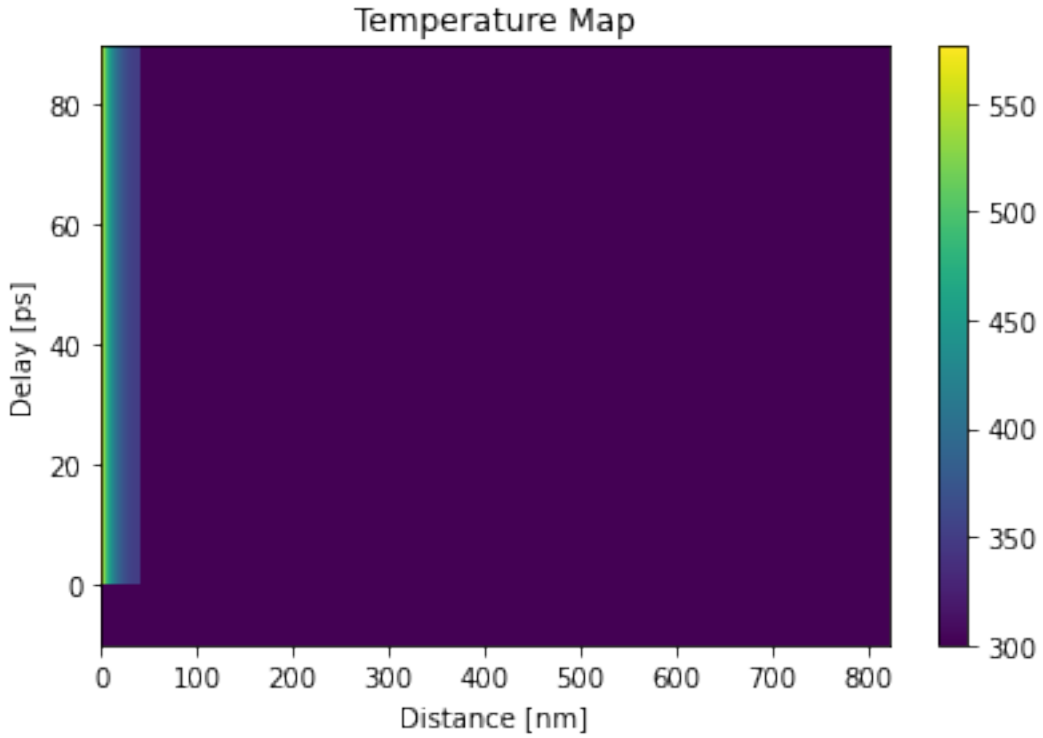
```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True

h.excitation = {'fluence': [5]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg}

# temporal and spatial grid
delays = np.r_[-10:90:0.1]*u.ps
_, _, distances = S.get_distances_of_layers()
```

```
temp_map, delta_temp_map = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 56.1 % and transmission of 5.7 %.
Elapsed time for _temperature_after_delta_excitation_: 0.035174 s
Elapsed time for _temp_map_: 0.289220 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :])
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.title('Temperature Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map')

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


Temperature Profile
Temperature [K]
550
500
450
400
350
300
0
100
200
300
400
500
600
700
800
Distance [nm]


Temperature Map
Delay [ps]
80
60
40
20
0
0
100
200
300
400
500
600
700
800
Distance [nm]
550
500
450
400
350
300

Analytical Phonons

The `PhononAna` class requires a `Structure` object and a boolean `force_recalc` in order overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can using the boolean switch `progress_bar`.

```
pana = ud.PhononAna(S, True)
pana.save_data = False
pana.disp_messages = True
```

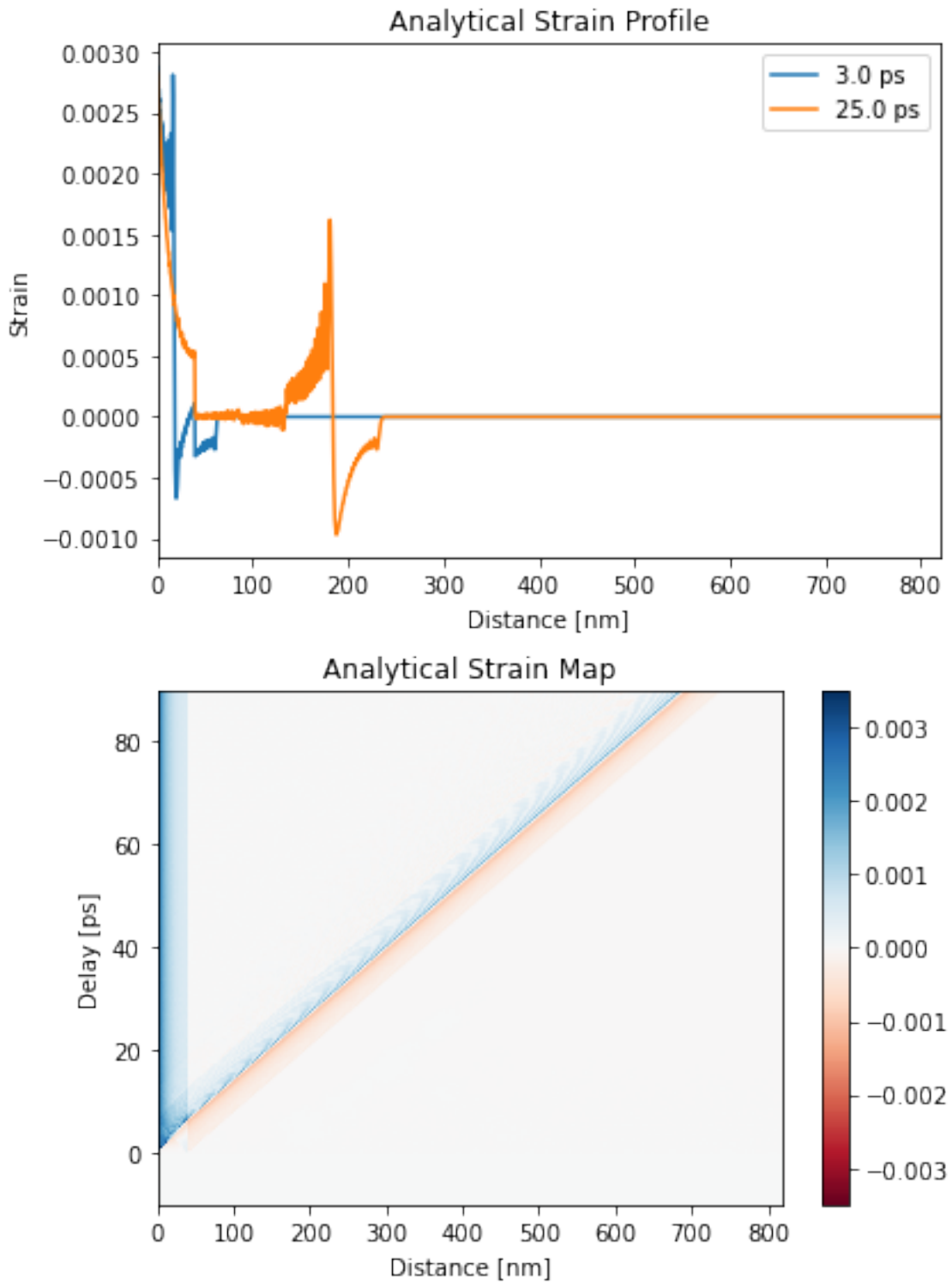
```
strain_map, A, B = pana.get_strain_map(delays, temp_map, delta_temp_map)
```

```
Calculating linear thermal expansion ...
Calculating _eigen_values_ ...
Elapsed time for _eigen_values_: 6.889534 s
Calculating _strain_map_ ...
```

```
Progress:   0%|          | 0/1000 [00:00<?, ?it/s]
```

```
Elapsed time for _strain_map_: 48.542959 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, strain_map[130, :], label=np.round(delays[130]))
plt.plot(distances.to('nm').magnitude, strain_map[350, :], label=np.round(delays[350]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Strain')
plt.legend()
plt.title('Analytical Strain Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               strain_map, cmap='RdBu', vmin=-np.max(strain_map),
               vmax=np.max(strain_map), shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Analytical Strain Map')

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


Analytical Strain Profile
3.0 ps
25.0 ps
Strain
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
−0.0005
−0.0010
0
100
200
300
400
500
600
700
800
Distance [nm]
Analytical Strain Map
Delay [ps]
80
60
40
20
0
0.003
0.002
0.001
0.000
−0.001
−0.002
−0.003
Distance [nm]

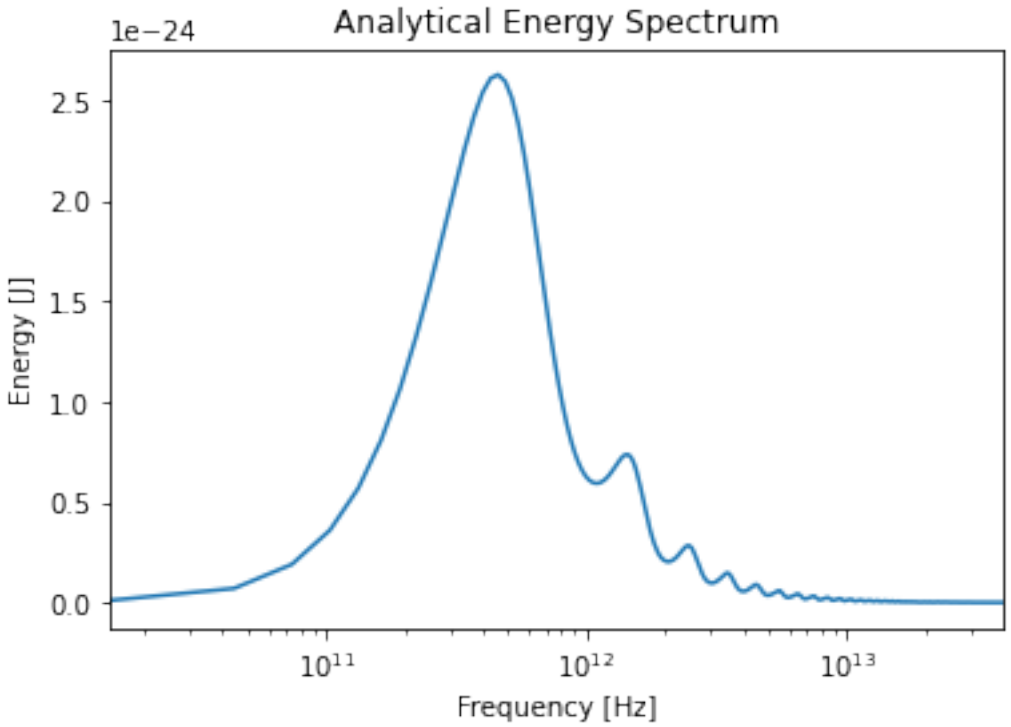
Energy Spectrum

The analytical phonon model easily allows for calculating the energy per eigenmode of the coherent acoustic phonon spectrum for every delay of the simulation.

```
omega, E = pana.get_energy_per_eigenmode(A, B)
```

```
Calculating _eigen_values_ ...
Elapsed time for _eigen_values_: 6.463842 s
```

```
plt.figure()
plt.plot(omega, E[-1, :])
plt.xlim(omega[0], omega[-1])
plt.xscale('log')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Energy [J]')
plt.title('Analytical Energy Spectrum')
plt.show()
```



Numerical Phonons

The `PhononNum` class requires a `Structure` object and a boolean `force_recalc` in order overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can using the boolean switch `progress_bar`.

```
pnum = ud.PhononNum(S, True)
pnum.save_data = False
pnum.disp_messages = True
```

The actual calculation is done in one line:

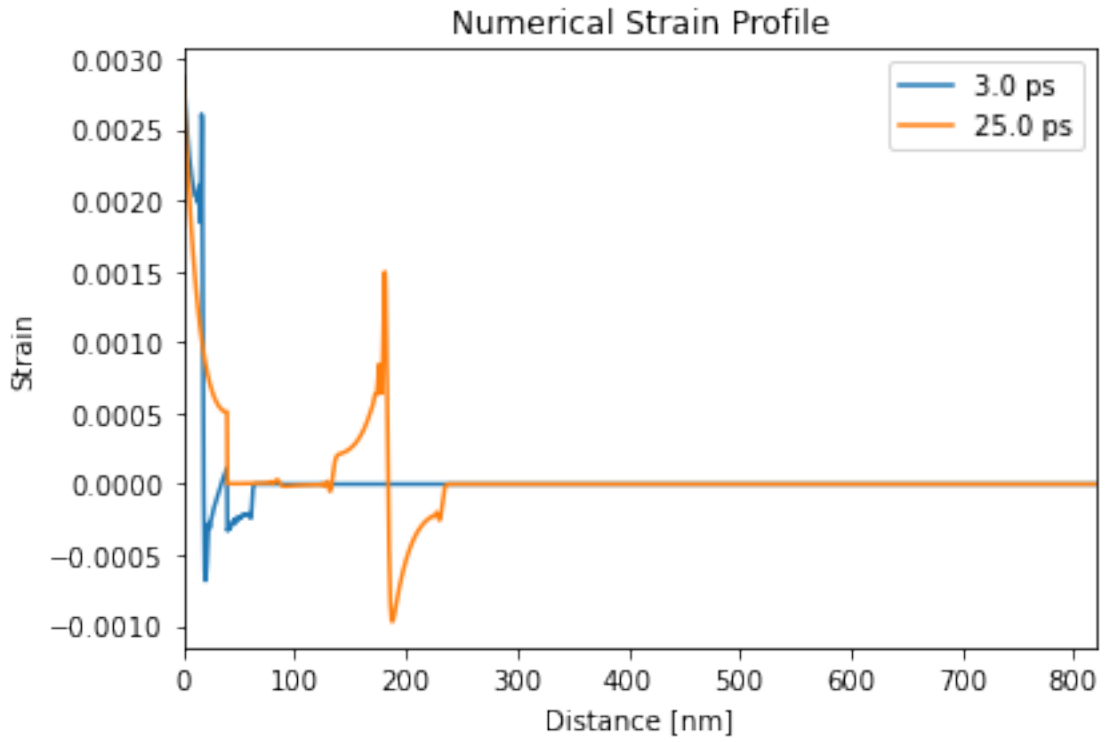
```
strain_map = pnum.get_strain_map(delays, temp_map, delta_temp_map)
```

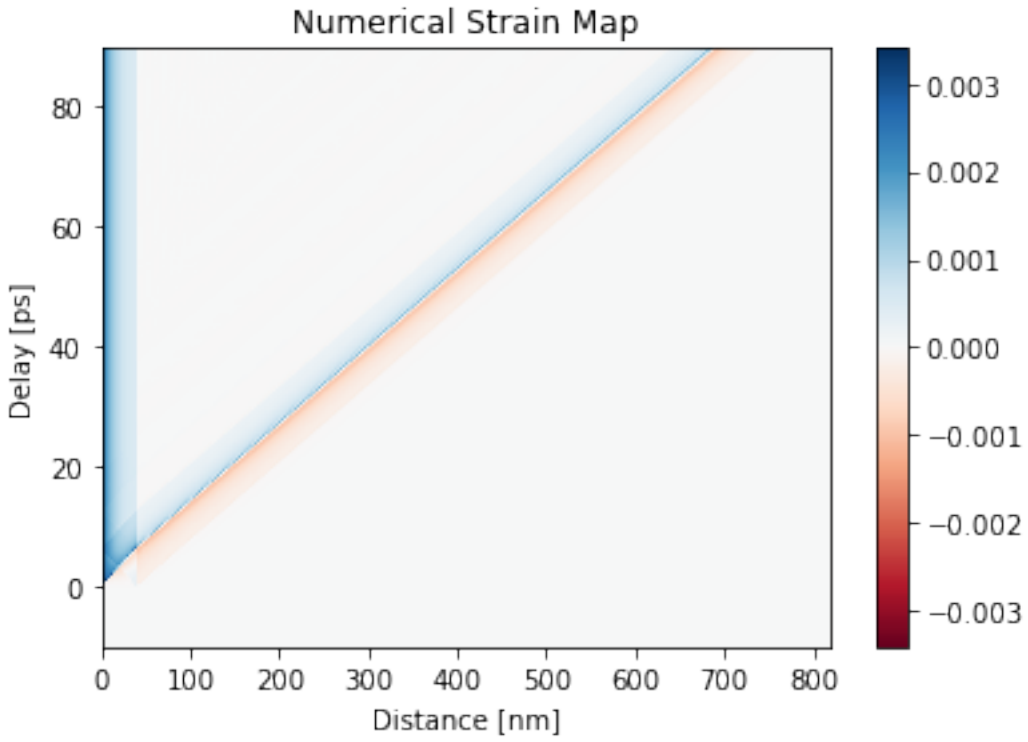
```
Calculating linear thermal expansion ...
Calculating coherent dynamics with ODE solver ...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _strain_map_: 2.325907 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, strain_map[130, :], label=np.round(delays[130]))
plt.plot(distances.to('nm').magnitude, strain_map[350, :], label=np.round(delays[350]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Strain')
plt.legend()
plt.title('Numerical Strain Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               strain_map, cmap='RdBu', vmin=-np.max(strain_map),
               vmax=np.max(strain_map), shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Numerical Strain Map')

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


Numerical Strain Profile
3.0 ps
25.0 ps
Strain
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
−0.0005
−0.0010
0
100
200
300
400
500
600
700
800
Distance [nm]


Numerical Strain Map
Delay [ps]
0
20
40
60
80
0
100
200
300
400
500
600
700
800
Distance [nm]
0.003
0.002
0.001
0.000
−0.001
−0.002
−0.003

Anharmonic Phonon Propagation

The numerical phonon dynamic calculations also allow for phonon damping and non-linear phonon propagation. This can be achieved by setting the `phonon_damping` property and using the `set_ho_spring_constants()` method of the according layers.

```
STO_sub.phonon_damping = -1e10*u.kg/u.s
STO_sub.set_ho_spring_constants([-7e11])
```

Recalculate the coherent phonon dynamics:

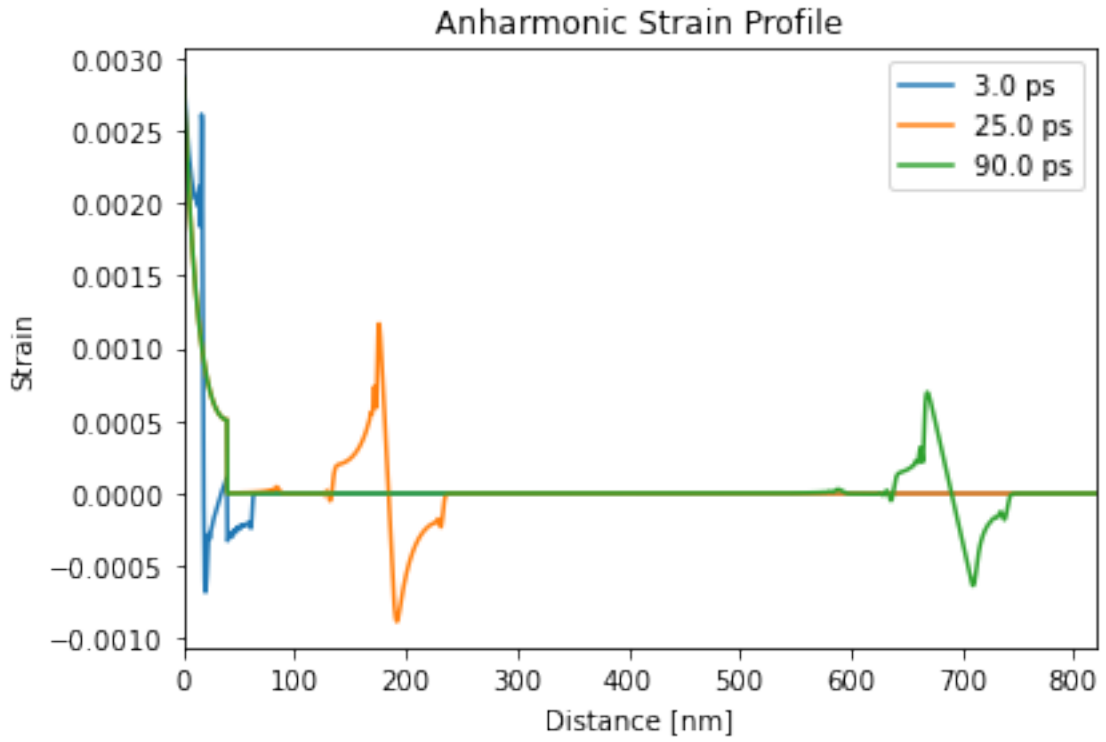
```
strain_map = pnum.get_strain_map(delays, temp_map, delta_temp_map)
```

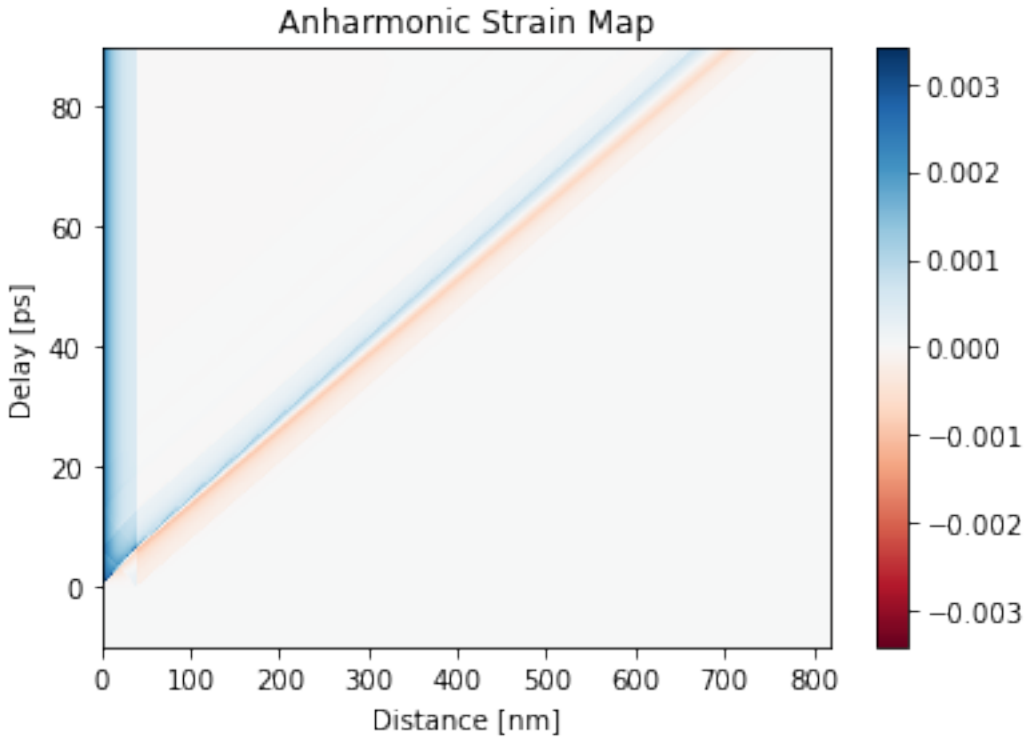
```
Calculating linear thermal expansion ...
Calculating coherent dynamics with ODE solver ...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _strain_map_: 3.084836 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, strain_map[130, :], label=np.round(delays[130]))
plt.plot(distances.to('nm').magnitude, strain_map[350, :], label=np.round(delays[350]))
plt.plot(distances.to('nm').magnitude, strain_map[-1, :], label=np.round(delays[-1]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Strain')
plt.legend()
plt.title('Anharmonic Strain Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               strain_map, cmap='RdBu', vmin=-np.max(strain_map),
               vmax=np.max(strain_map), shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Anharmonic Strain Map')

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


Anharmonic Strain Profile
3.0 ps
25.0 ps
90.0 ps
Strain
Distance [nm]


Anharmonic Strain Map
Delay [ps]
Distance [nm]

5.3.6 Dynamical X-ray Scattering

In this example static and transient X-ray simulations are carried out employing the dynamical X-ray scattering formalism.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import scipy.constants as constants
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

Refer to the *structure-example* for more details.

```
O = ud.Atom('O')
Ti = ud.Atom('Ti')
Sr = ud.Atom('Sr')
Ru = ud.Atom('Ru')
Pb = ud.Atom('Pb')
Zr = ud.Atom('Zr')
```

```
# c-axis lattice constants of the two layers
c_STO_sub = 3.905*u.angstrom
c_SRO = 3.94897*u.angstrom
# sound velocities [nm/ps] of the two layers
sv_SRO = 6.312*u.nm/u.ps
sv_STO = 7.800*u.nm/u.ps

# SRO layer
prop_SRO = {}
prop_SRO['a_axis'] = c_STO_sub  # aAxis
prop_SRO['b_axis'] = c_STO_sub  # bAxis
prop_SRO['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_SRO['sound_vel'] = sv_SRO  # sound velocity
prop_SRO['opt_ref_index'] = 2.44+4.32j
prop_SRO['therm_cond'] = 5.72*u.W/(u.m*u.K)  # heat conductivity
prop_SRO['lin_therm_exp'] = 1.03e-5  # linear thermal expansion
prop_SRO['heat_capacity'] = '455.2 + 0.112*T - 2.1935e6/T**2'  # [J/kg K]

SRO = ud.UnitCell('SRO', 'Strontium Ruthenate', c_SRO, **prop_SRO)
SRO.add_atom(O, 0)
SRO.add_atom(Sr, 0)
SRO.add_atom(O, 0.5)
SRO.add_atom(O, 0.5)
```

(continues on next page)

(continued from previous page)

```
SRO.add_atom(Ru, 0.5)

# STO substrate
prop_STO_sub = {}
prop_STO_sub['a_axis'] = c_STO_sub  # aAxis
prop_STO_sub['b_axis'] = c_STO_sub  # bAxis
prop_STO_sub['deb_Wal_Fac'] = 0  # Debye-Waller factor
prop_STO_sub['sound_vel'] = sv_STO  # sound velocity
prop_STO_sub['opt_ref_index'] = 2.1+0j
prop_STO_sub['therm_cond'] = 12*u.W/(u.m*u.K)  # heat conductivity
prop_STO_sub['lin_therm_exp'] = 1e-5  # linear thermal expansion
prop_STO_sub['heat_capacity'] = '733.73 + 0.0248*T - 6.531e6/T**2'  # [J/kg K]

STO_sub = ud.UnitCell('STOsub', 'Strontium Titanate Substrate',
                      c_STO_sub, **prop_STO_sub)
STO_sub.add_atom(O, 0)
STO_sub.add_atom(Sr, 0)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(O, 0.5)
STO_sub.add_atom(Ti, 0.5)
```

```
S = ud.Structure('Single Layer')
S.add_sub_structure(SRO, 200)  # add 100 layers of SRO to sample
S.add_sub_structure(STO_sub, 1000)  # add 1000 layers of dynamic STO substrate

substrate = ud.Structure('Static Substrate')
# add 1000000 layers of static STO substrate
substrate.add_sub_structure(STO_sub, 1000000)
S.add_substrate(substrate)
```

Heat

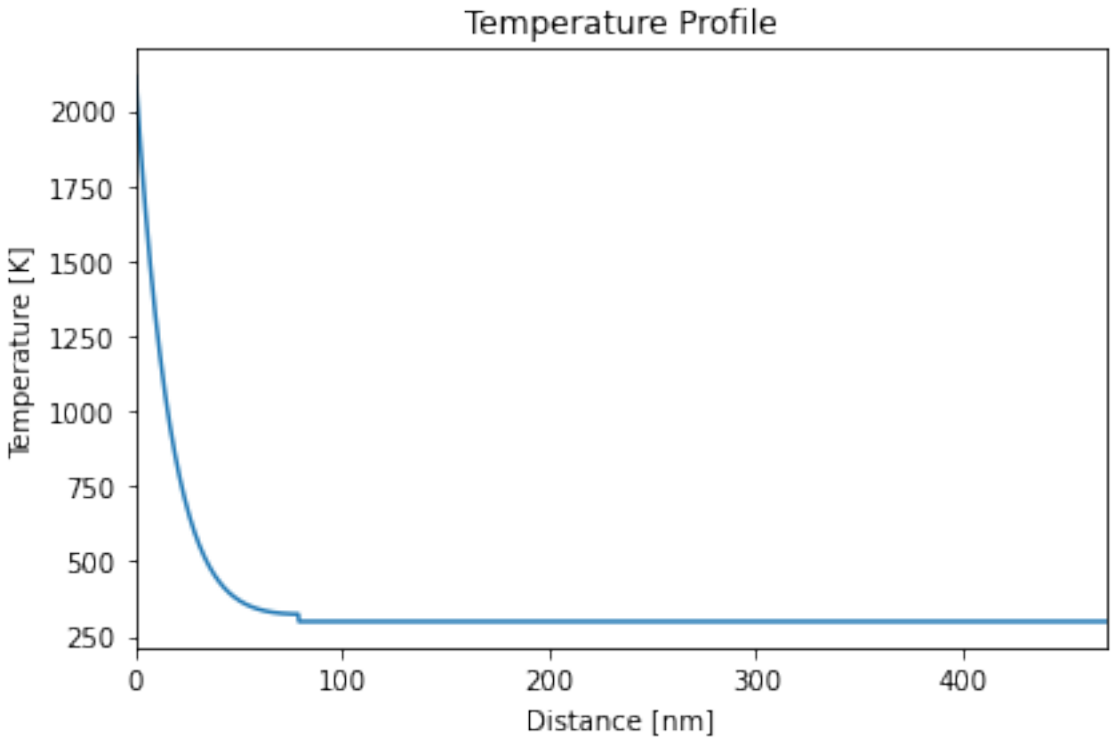
Refer to the *heat-example* for more details.

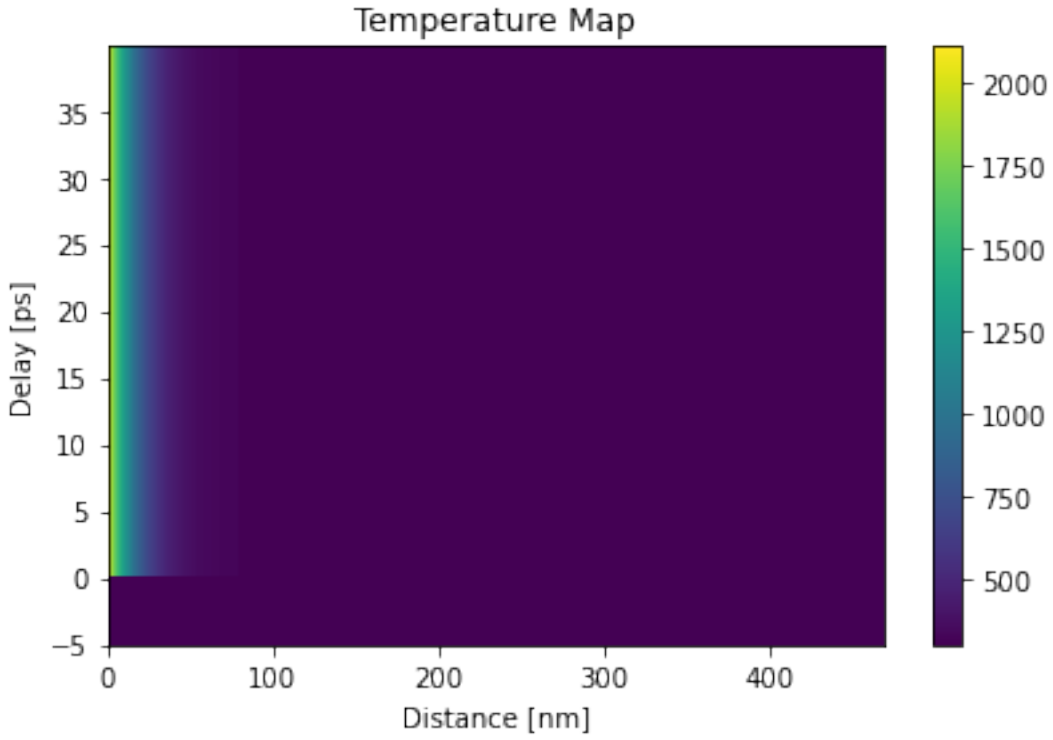
```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True

h.excitation = {'fluence': [35]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [0]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg}

# temporal and spatial grid
delays = np.r_[-5:40:0.1]*u.ps
_, _, distances = S.get_distances_of_layers()
```

```
temp_map, delta_temp_map = h.get_temp_map(delays, 300*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 58.5 % and transmission of 0.4 %.
Elapsed time for _temperature_after_delta_excitation_: 0.291286 s
Elapsed time for _temp_map_: 0.324429 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[101, :])
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.title('Temperature Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map')

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


Temperature Profile
Temperature [K]
2000
1750
1500
1250
1000
750
500
250
0
100
200
300
400
Distance [nm]


Temperature Map
Delay [ps]
35
30
25
20
15
10
5
0
−5
0
100
200
300
400
Distance [nm]
2000
1750
1500
1250
1000
750
500

Numerical Phonons

Refer to the *phonons-example* for more details.

```
p = ud.PhononNum(S, True)
p.save_data = False
p.disp_messages = True
```

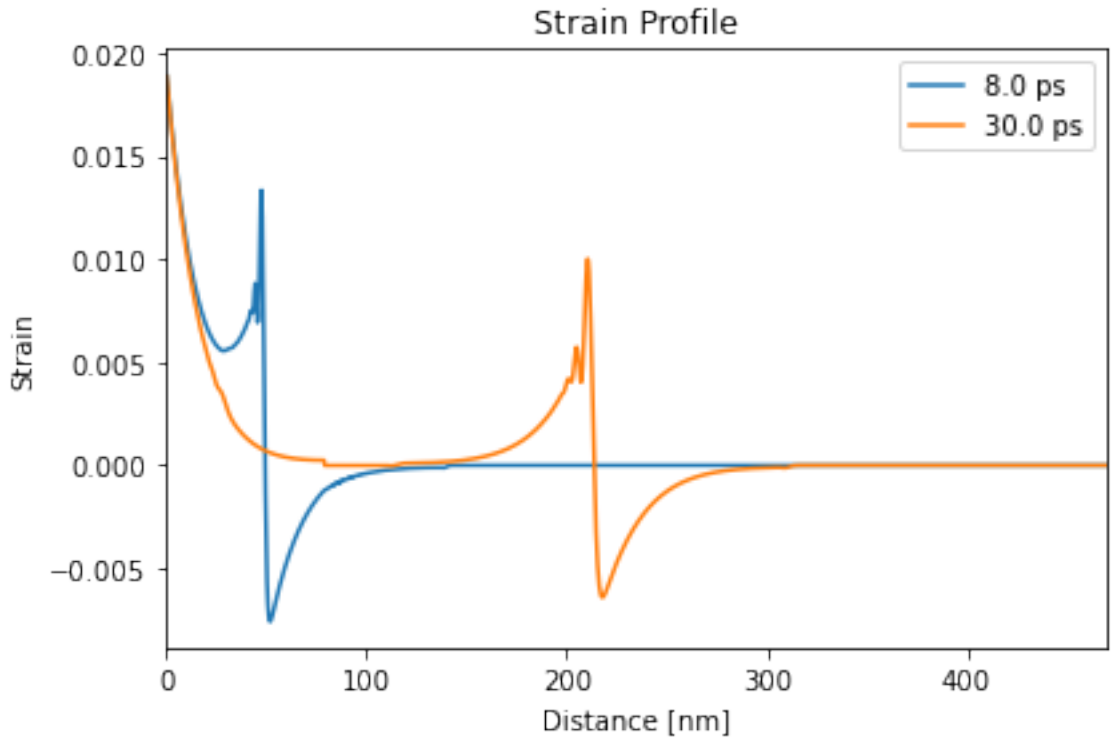
```
strain_map = p.get_strain_map(delays, temp_map, delta_temp_map)
```

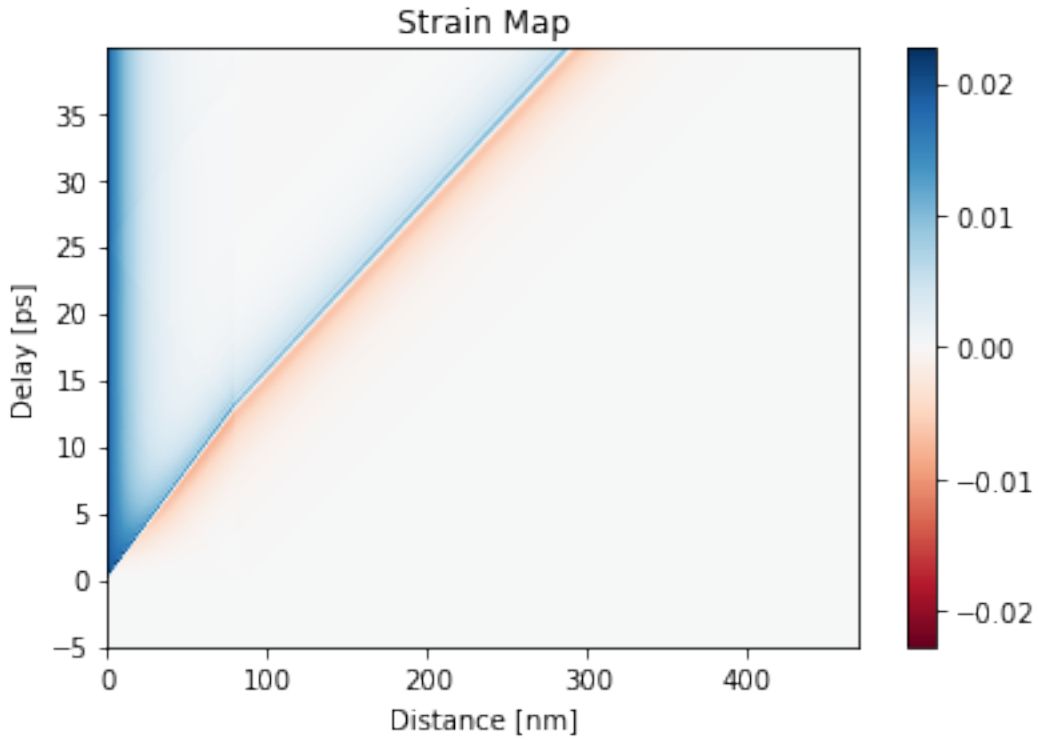
```
Calculating linear thermal expansion ...
Calculating coherent dynamics with ODE solver ...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _strain_map_: 0.723023 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, strain_map[130, :], label=np.round(delays[130]))
plt.plot(distances.to('nm').magnitude, strain_map[350, :], label=np.round(delays[350]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Strain')
plt.legend()
plt.title('Strain Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               strain_map, cmap='RdBu', vmin=-np.max(strain_map),
               vmax=np.max(strain_map), shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Strain Map')

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


Strain Profile
8.0 ps
30.0 ps
Strain
0.020
0.015
0.010
0.005
0.000
−0.005
0
100
200
300
400
Distance [nm]


Strain Map
Delay [ps]
35
30
25
20
15
10
5
0
−5
0
100
200
300
400
Distance [nm]
0.02
0.01
0.00
−0.01
−0.02

Initialize dynamical X-ray simulation

The `XrayDyn` class requires a `Structure` object and a boolean `force_recalc` in order overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can using the boolean switch `progress_bar`.

```
dyn = ud.XrayDyn(S, True)
dyn.disp_messages = True
dyn.save_data = False
```

```
incoming polarizations set to: sigma
analyzer polarizations set to: unpolarized
```

Homogeneous X-ray scattering

For the case of homogeneously strained samples, the dynamical X-ray scattering simulations can be greatly simplified, which saves a lot of computational time.

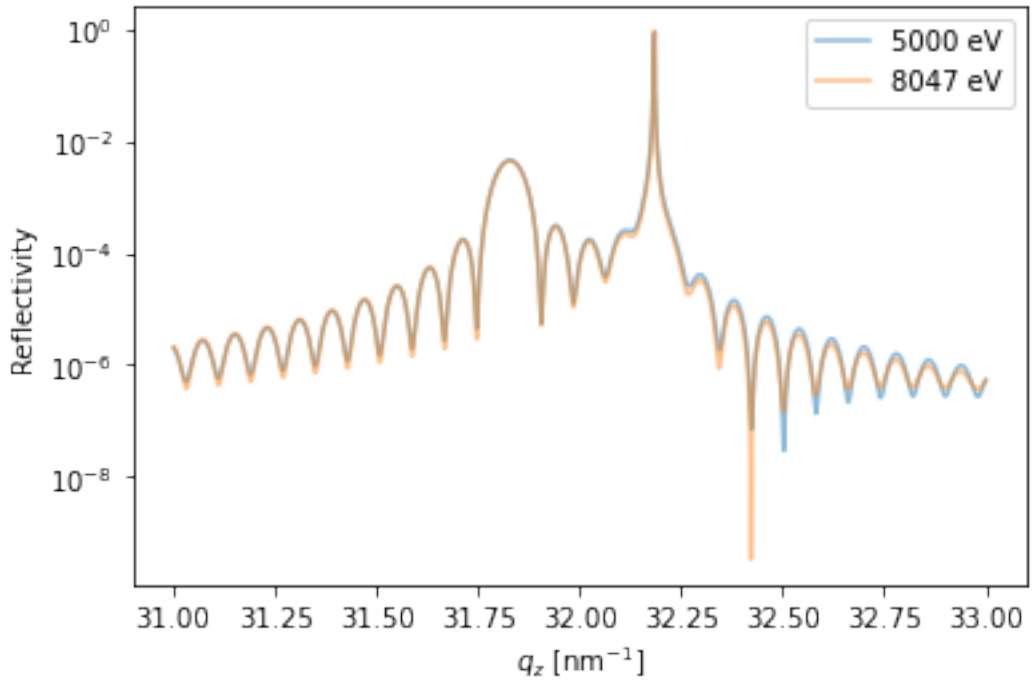
q_z-scan

The `XrayDyn` object requires an `energy` and scattering vector `qz` to run the simulations.
Both parameters can be arrays and the resulting reflectivity has a first dimension for the photon energy and the a second for the scattering vector.

```
dyn.energy = np.r_[5000, 8047]*u.eV  # set two photon energies
dyn.qz = np.r_[3.1:3.3:0.00001]/u.angstrom  # qz range

R_hom, A = dyn.homogeneous_reflectivity()  # this is the actual calculation
```

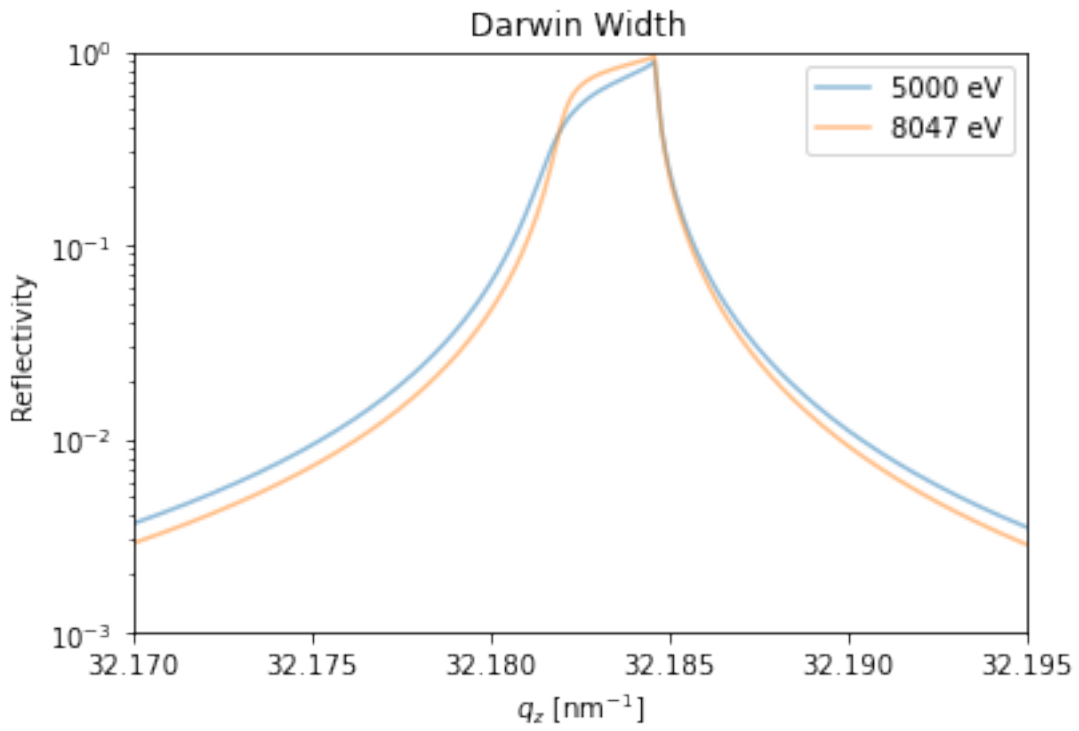
```
Calculating _homogenous_reflectivity_ ...
Elapsed time for _homogenous_reflectivity_: 3.990309 s
```

```
plt.figure()
plt.semilogy(dyn.qz[0, :], R_hom[0, :], label='{}'.format(dyn.energy[0]), alpha=0.5)
plt.semilogy(dyn.qz[1, :], R_hom[1, :], label='{}'.format(dyn.energy[1]), alpha=0.5)
plt.ylabel('Reflectivity')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.legend()
plt.show()
```



Due to the very thick static `substrate` in the `structure` and the very small step width in `qz` also the Darwin width of the substrate Bragg peak is nicely resolvable.

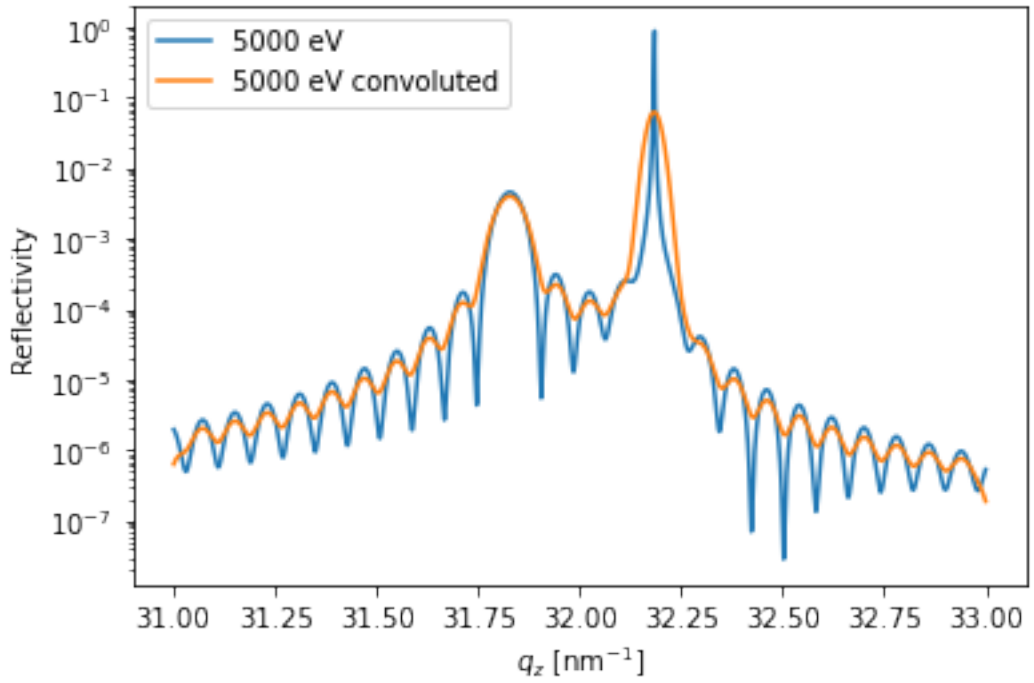
```
plt.figure()
plt.semilogy(dyn.qz[0, :], R_hom[0, :], label='{}'.format(dyn.energy[0]), alpha=0.5)
plt.semilogy(dyn.qz[1, :], R_hom[1, :], label='{}'.format(dyn.energy[1]), alpha=0.5)
plt.ylabel('Reflectivity')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.xlim(32.17, 32.195)
plt.ylim(1e-3, 1)
plt.legend()
plt.title('Darwin Width')
plt.show()
```



Post-Processing

All result can be convoluted with an arbitrary function handle, which e.g. mimics the instrumental resolution.

```
FWHM = 0.004/1e-10  # Angstrom
sigma = FWHM/2.3548

handle = lambda x: np.exp(-((x)/sigma)**2/2)
y_conv = dyn.conv_with_function(R_hom[0, :], dyn._qz[0, :], handle)

plt.figure()
plt.semilogy(dyn.qz[0, :], R_hom[0, :], label='{}'.format(dyn.energy[0]))
plt.semilogy(dyn.qz[0, :], y_conv, label='{} convoluted'.format(dyn.energy[0]))
plt.ylabel('Reflectivity')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.legend()
plt.show()
```



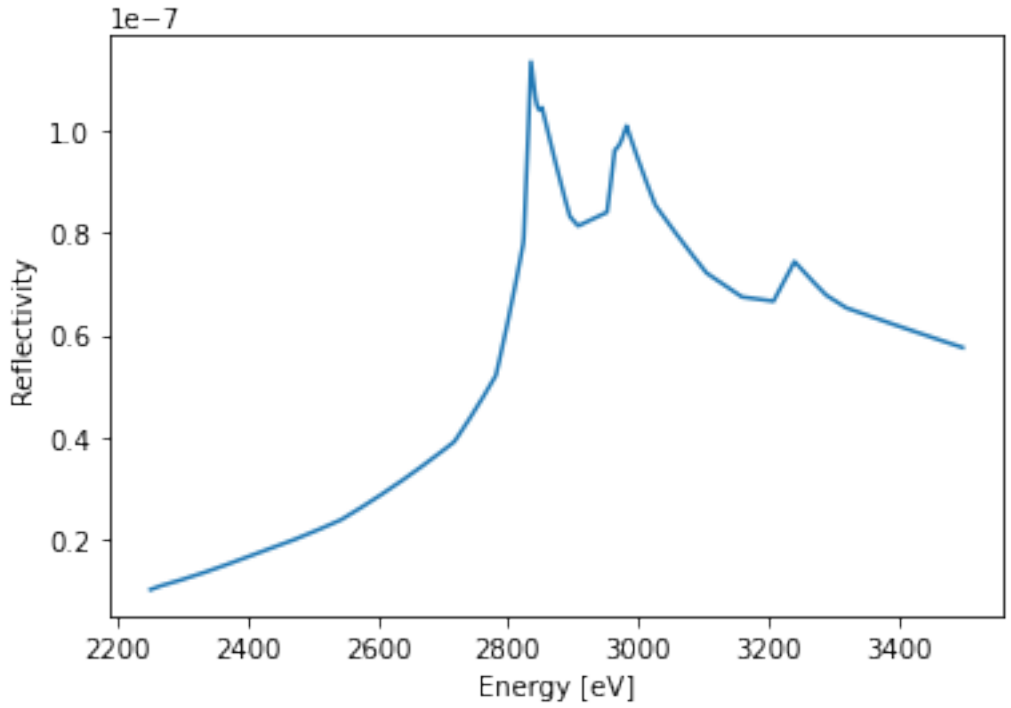
Energy-scan

Energy scans rely on experimental atomic scattering factors that are include also energy ranges around relevant resonances.
The warning message can be safely ignored as it results from the former q_z range which cannot be accessed with the new energy range.

```
dyn.energy = np.r_[2250:3500]*u.eV  # set the energy range
dyn.qz = np.r_[2]/u.angstrom  # qz range

R_hom, A = dyn.homogeneous_reflectivity()  # this is the actual calculation

plt.figure()
plt.plot(dyn.energy, R_hom[:, 0])
plt.ylabel('Reflectivity')
plt.xlabel('Energy [eV]')
plt.show()
```

```
c:\users\loc_schick\general\git\udkm1dsim\udkm1Dsim\simulations\xrays.py:239:␣
↪RuntimeWarning: invalid value encountered in arcsin
  self._theta = np.arcsin(np.outer(self._wl, self._qz[0, :])/np.pi/4)
```

```
Calculating _homogenous_reflectivity_ ...
Elapsed time for _homogenous_reflectivity_: 0.890771 s
```



Inhomogeneous X-ray scattering

The `inhomogeneous_reflectivity()` method allows to calculate the transient X-ray reflectivity according to a `strain_map`.

The actual strains per layer will be discretized and limited in order to save computational time using the `strain_vectors`.

```
dyn.energy = np.r_[8047]*u.eV  # set two photon energies
dyn.qz = np.r_[3.1:3.3:0.001]/u.angstrom  # qz range

strain_vectors = p.get_reduced_strains_per_unique_layer(strain_map)
R_seq = dyn.inhomogeneous_reflectivity(strain_map, strain_vectors, calc_type='sequential
↪')
```

```
Calculating _inhomogeneousReflectivity_ ...
Calculate all _ref_trans_matrices_ ...
Elapsed time for _ref_trans_matrices_: 1.557266 s
```

```
Progress:   0%|          | 0/450 [00:00<?, ?it/s]
```

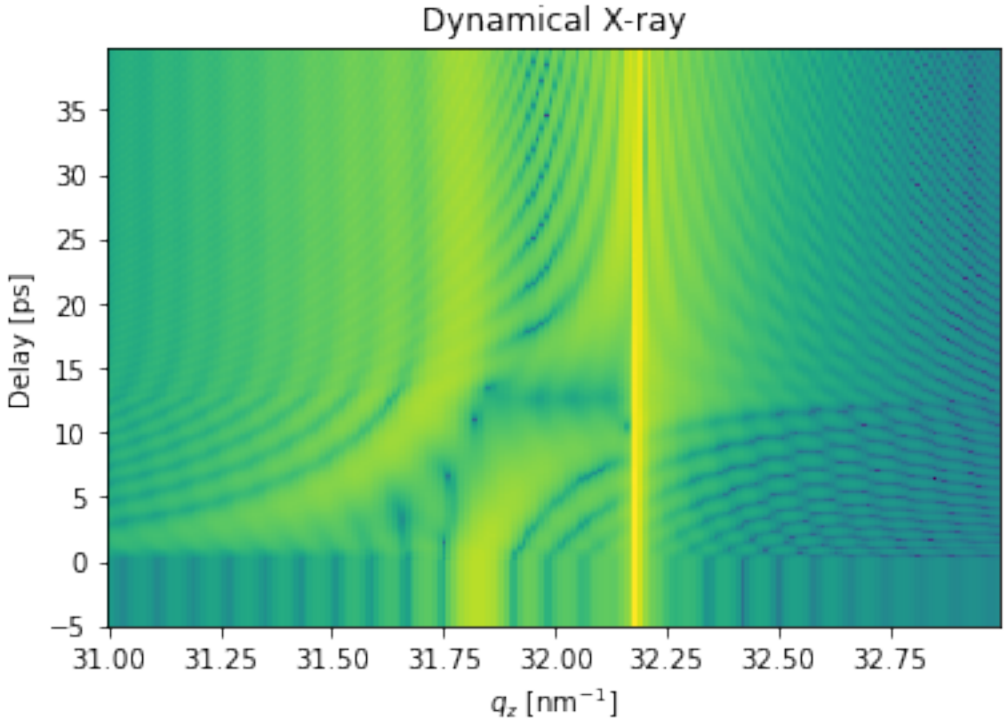
```
Elapsed time for _inhomogeneous_reflectivity_: 45.179404 s
```

```
plt.figure()
plt.pcolormesh(dyn.qz[0, :].to('1/nm').magnitude, delays.to('ps').magnitude, np.log10(R_
↪seq[:, 0, :]),
               shading='auto')
plt.title('Dynamical X-ray')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Delay [ps]')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.show()
```



The results can be convoluted again to mimic real experimental resolution:

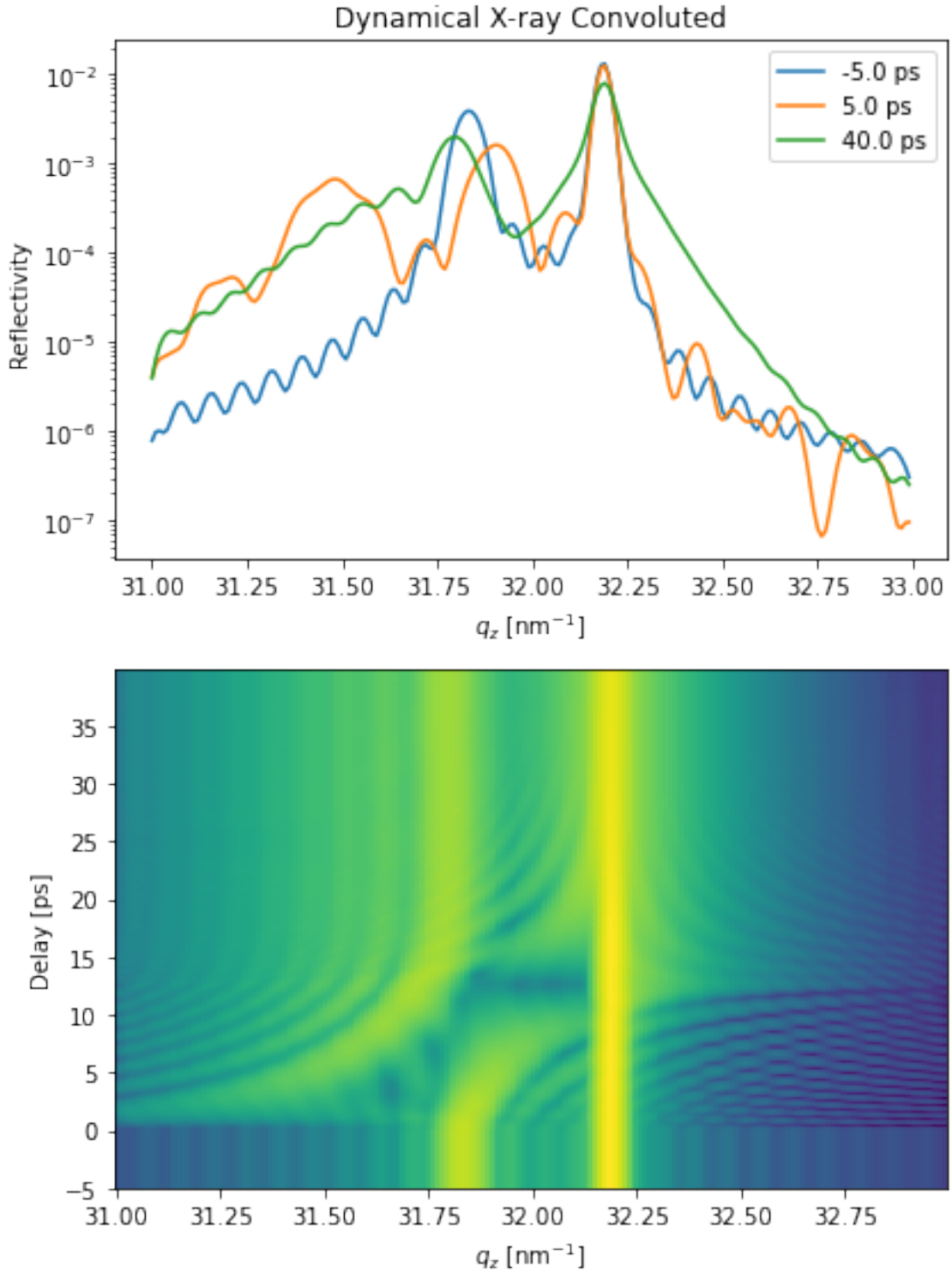
```
R_seq_conv = np.zeros_like(R_seq)
for i, delay in enumerate(delays):
    R_seq_conv[i, 0, :] = dyn.conv_with_function(R_seq[i, 0, :], dyn._qz[0, :], handle)
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.semilogy(dyn.qz[0, :].to('1/nm'), R_seq_conv[0, 0, :], label=np.round(delays[0]))
plt.semilogy(dyn.qz[0, :].to('1/nm'), R_seq_conv[100, 0, :], label=np.round(delays[100]))
plt.semilogy(dyn.qz[0, :].to('1/nm'), R_seq_conv[-1, 0, :], label=np.round(delays[-1]))

plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.ylabel('Reflectivity')
plt.legend()
plt.title('Dynamical X-ray Convoluted')

plt.subplot(2, 1, 2)
plt.pcolormesh(dyn.qz[0, :].to('1/nm').magnitude, delays.to('ps').magnitude,
               np.log10(R_seq_conv[:, 0, :]), shading='auto')
plt.ylabel('Delay [ps]')
plt.xlabel('$q_z$ [nm$^{-1}$]')

plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Parallel inhomogeneous X-ray scattering

You need to install the `udkm1Dsim` with the `parallel` option which essentially add the Dask package to the requirements:

```
> pip install udkm1Dsim[parallel]
```

You can also install/add Dask manually, e.g. via pip:

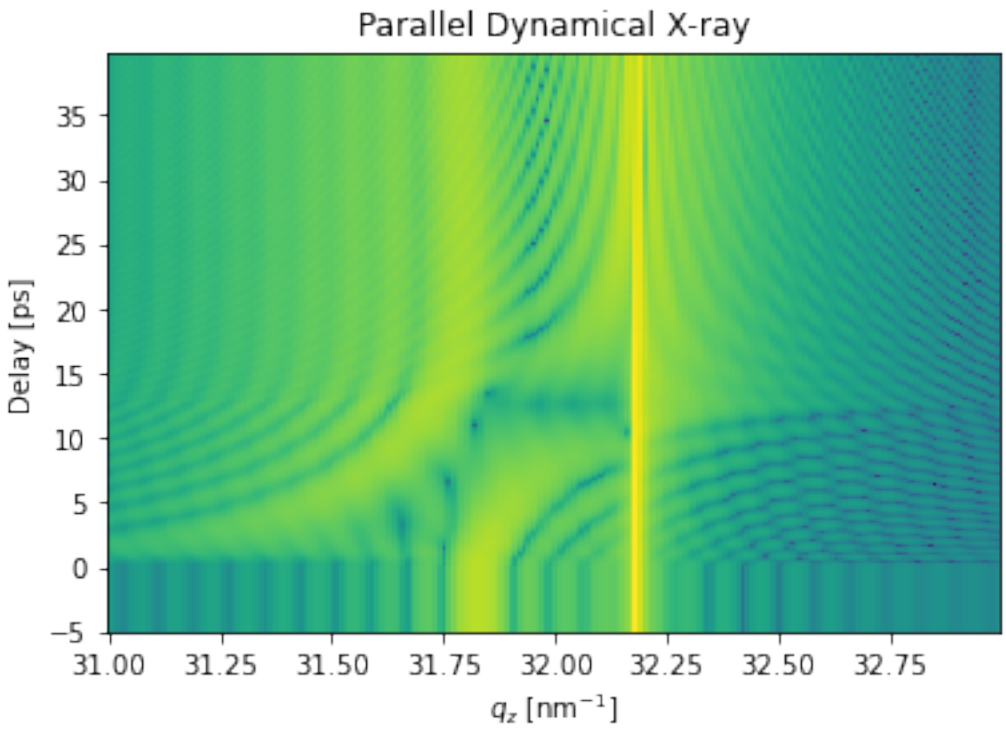
```
> pip install dask
```

Please refer to the Dask documentation for more details on parallel computing in Python.

```
try:
    from dask.distributed import Client
    client = Client()
    R_par = dyn.inhomogeneous_reflectivity(strain_map, strain_vectors, calc_type=
↪'parallel',
                                           dask_client=client)
    client.close()
except:
    pass
```

```
Calculating _inhomogeneousReflectivity_ ...
Calculate all _ref_trans_matrices_ ...
Elapsed time for _ref_trans_matrices_: 2.199932 s
Elapsed time for _inhomogeneous_reflectivity_: 18.328078 s
```

```
plt.figure()
plt.pcolormesh(dyn.qz[0, :].to('1/nm').magnitude, delays.to('ps').magnitude,
               np.log10(R_par[:, 0, :]), shading='auto')
plt.title('Parallel Dynamical X-ray')
plt.ylabel('Delay [ps]')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.show()
```



5.3.7 Dynamical Magnetic X-ray Scattering

In this example static and transient magnetic X-ray simulations are carried out employing a dynamical magnetic X-ray scattering formalism which was adapted from Project Dyna.

Setup

Do all necessary imports and settings.

```
import udkm1Dsim as ud
u = ud.u  # import the pint unit registry from udkm1Dsim
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
u.setup_matplotlib()  # use matplotlib with pint units
```

Structure

Refer to the *structure-example* for more details.

In this example the sample `Structure` consists of `AmorphousLayer`s.

In order to build an antiferromagnetic sample two different types of *Fe* `Atom`s and `AmorphousLayer`s are created with opposite in-plane magnetization.

Here a local file for the atomic scattering factor is read only for Fe atoms.

```
Fe_right = ud.Atom('Fe', mag_amplitude=1, mag_phi=90*u.deg, mag_gamma=90*u.deg,
                   atomic_form_factor_path='./Fe.cf')
Fe_left = ud.Atom('Fe', mag_amplitude=1, mag_phi=90*u.deg, mag_gamma=270*u.deg,
                  atomic_form_factor_path='./Fe.cf')
Cr = ud.Atom('Cr')
Si = ud.Atom('Si')
```

```
density_Fe = 7874*u.kg/u.m**3

prop_Fe = {}
prop_Fe['heat_capacity'] = 449*u.J/u.kg/u.K
prop_Fe['therm_cond'] = 80*u.W/(u.m *u.K)
prop_Fe['lin_therm_exp'] = 11.8e-6
prop_Fe['sound_vel'] = 4.910*u.nm/u.ps
prop_Fe['opt_ref_index'] = 2.9174+3.3545j

layer_Fe_left = ud.AmorphousLayer('Fe_left', 'Fe left amorphous', 1*u.nm,
                                  density_Fe, atom=Fe_left, **prop_Fe)
layer_Fe_right = ud.AmorphousLayer('Fe_right', 'Fe right amorphous', 1*u.nm,
                                   density_Fe, atom=Fe_right, **prop_Fe)
```

```
density_Cr = 7140*u.kg/u.m**3

prop_Cr = {}
prop_Cr['heat_capacity'] = 449*u.J/u.kg/u.K
prop_Cr['therm_cond'] = 94*u.W/(u.m*u.K)
prop_Cr['lin_therm_exp'] = 6.2e-6
prop_Cr['sound_vel'] = 5.940*u.nm/u.ps
prop_Cr['opt_ref_index'] = 3.1612+3.4606j

layer_Cr = ud.AmorphousLayer('Cr', "Cr amorphous", 1*u.nm, density_Cr, atom=Cr, **prop_
↪Cr)
```

```
density_Si = 2336*u.kg/u.m**3

prop_Si = {}
prop_Si['heat_capacity'] = 703*u.J/u.kg/u.K
prop_Si['therm_cond'] = 150*u.W/(u.m*u.K)
prop_Si['lin_therm_exp'] = 2.6e-6
prop_Si['sound_vel'] = 8.433*u.nm/u.ps
prop_Si['opt_ref_index'] = 3.6941+0.0065435j

layer_Si = ud.AmorphousLayer('Si', "Si amorphous", 1*u.nm, density_Si, atom=Si, **prop_
↪Si)
```

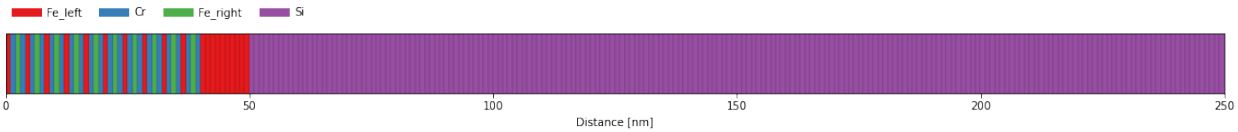
```
S = ud.Structure('Fe/Cr AFM Super Lattice')

# create a sub-structure
DL = ud.Structure('Two Fe/Cr Double Layers')
DL.add_sub_structure(layer_Fe_left, 1)
DL.add_sub_structure(layer_Cr, 1)
DL.add_sub_structure(layer_Fe_right, 1)
```

(continues on next page)

(continued from previous page)

```
DL.add_sub_structure(layer_Cr, 1)

S.add_sub_structure(DL, 10)
S.add_sub_structure(layer_Fe_left, 10)
S.add_sub_structure(layer_Si, 200)
```

```
S.visualize()
```



Heat

Refer to the *heat-example* for more details.

```
h = ud.Heat(S, True)

h.save_data = False
h.disp_messages = True

h.excitation = {'fluence': [40]*u.mJ/u.cm**2,
                'delay_pump':  [0]*u.ps,
                'pulse_width':  [1]*u.ps,
                'multilayer_absorption': True,
                'wavelength': 800*u.nm,
                'theta': 45*u.deg}

# enable heat diffusion
h.heat_diffusion = True

# temporal and spatial grid
delays = np.r_[-2:40:0.1]*u.ps
_, _, distances = S.get_distances_of_layers()
```

```
temp_map, delta_temp_map = h.get_temp_map(delays, 0*u.K)
```

```
Surface incidence fluence scaled by factor 0.7071 due to incidence angle theta=45.00 deg
Calculating _heat_diffusion_ for excitation 1:1 ...
Absorption profile is calculated by multilayer formalism.
Total reflectivity of 47.7 % and transmission of 6.4 %.
```

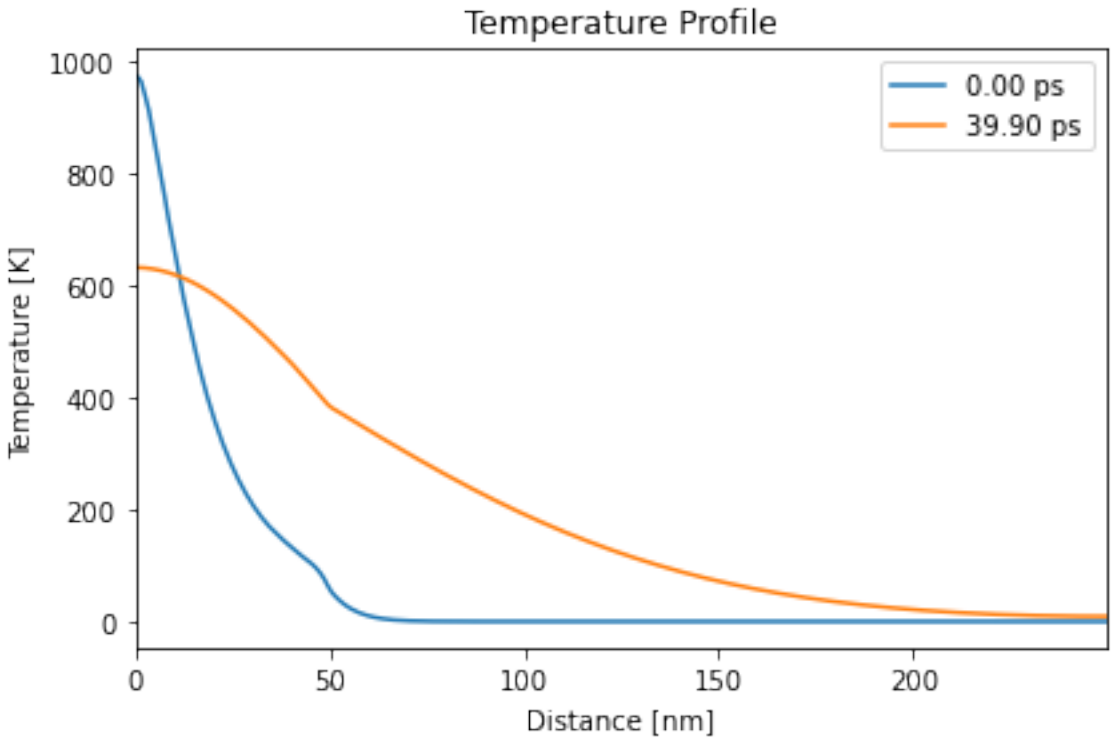
```
0it [00:00, ?it/s]
```

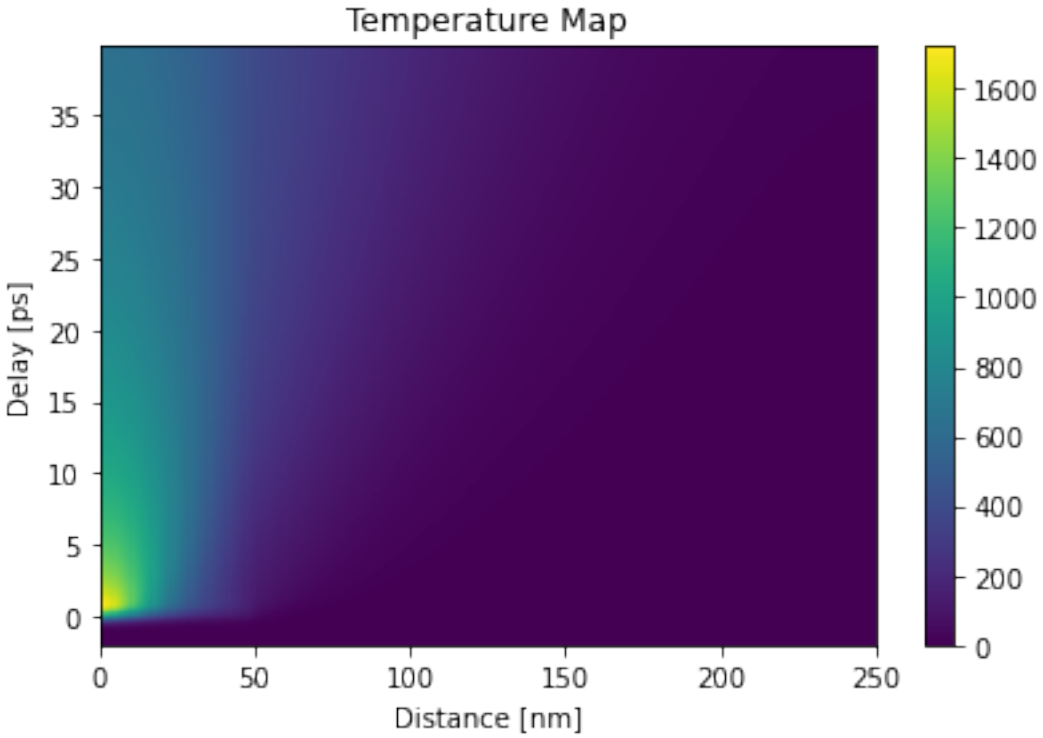
```
Elapsed time for _heat_diffusion_ with 1 excitation(s): 21.034174 s
Calculating _heat_diffusion_ without excitation...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _heat_diffusion_: 11.240211 s
Elapsed time for _temp_map_: 32.503744 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, temp_map[20, :],
         label='{:0.2f} ps'.format(delays[20].to('ps').magnitude))
plt.plot(distances.to('nm').magnitude, temp_map[-1, :],
         label='{:0.2f} ps'.format(delays[-1].to('ps').magnitude))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Temperature [K]')
plt.legend()
plt.title('Temperature Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude, temp_map,␣
↪shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Temperature Map')

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


Temperature Profile
0.00 ps
39.90 ps
Temperature [K]
Distance [nm]
0
200
400
600
800
1000
0
50
100
150
200


Temperature Map
Delay [ps]
Distance [nm]
0
5
10
15
20
25
30
35
0
50
100
150
200
250
0
200
400
600
800
1000
1200
1400
1600

Numerical Phonons

Refer to the *phonons-example* for more details.

```
p = ud.PhononNum(S, True)
p.save_data = False
p.disp_messages = True
```

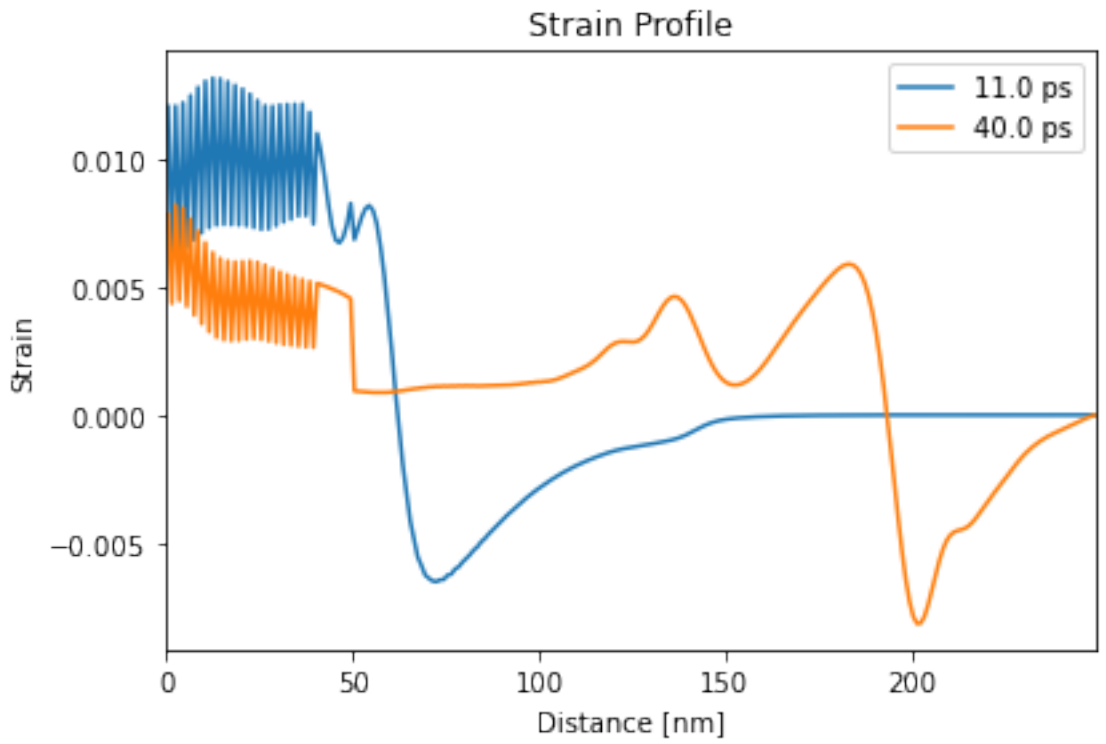
```
strain_map = p.get_strain_map(delays, temp_map, delta_temp_map)
```

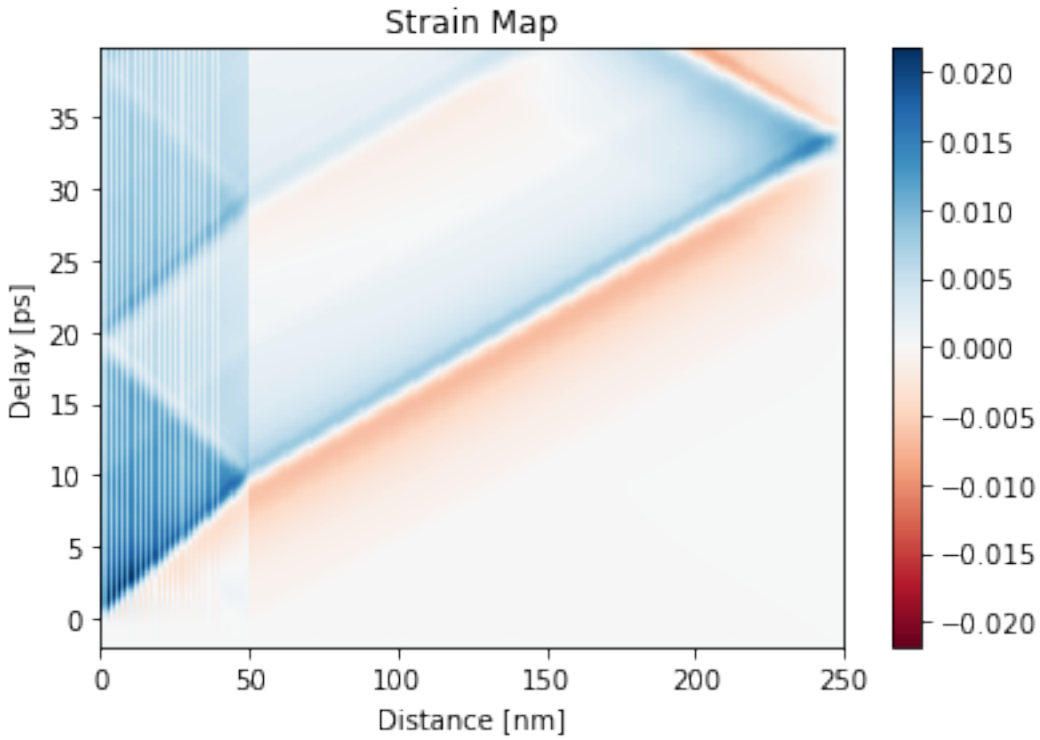
```
Calculating linear thermal expansion ...
Calculating coherent dynamics with ODE solver ...
```

```
0it [00:00, ?it/s]
```

```
Elapsed time for _strain_map_: 0.717197 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.plot(distances.to('nm').magnitude, strain_map[130, :], label=np.round(delays[130]))
plt.plot(distances.to('nm').magnitude, strain_map[-1, :], label=np.round(delays[-1]))
plt.xlim([0, distances.to('nm').magnitude[-1]])
plt.xlabel('Distance [nm]')
plt.ylabel('Strain')
plt.legend()
plt.title('Strain Profile')

plt.subplot(2, 1, 2)
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               strain_map, cmap='RdBu', vmin=-np.max(strain_map),
               vmax=np.max(strain_map), shading='auto')
plt.colorbar()
plt.xlabel('Distance [nm]')
plt.ylabel('Delay [ps]')
plt.title('Strain Map')

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


Strain Profile
11.0 ps
40.0 ps
0.010
0.005
0.000
−0.005
Strain
0
50
100
150
200
Distance [nm]

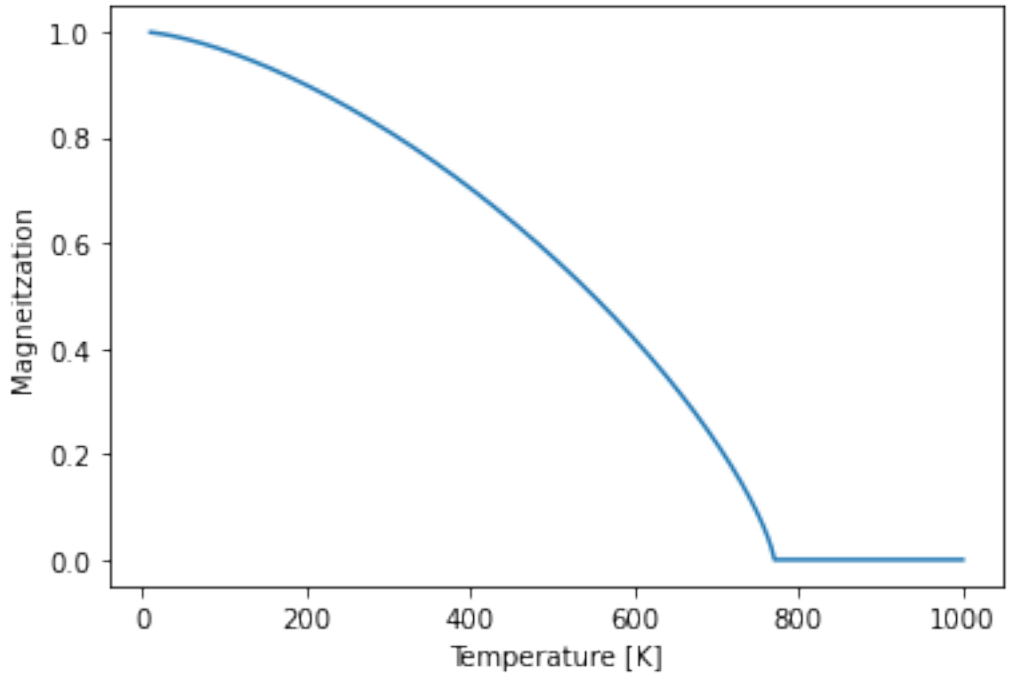

Strain Map
35
30
25
20
15
10
5
0
Delay [ps]
0
50
100
150
200
250
Distance [nm]
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020

Magnetization

The `Magnetization` class is currently only an interface to allow the user for defining specific magnetization dynamics depending on the `strain_map` and `temp_map`.

Here the magnetization as function of temperature is used as a simplified model to alter the transient magnetization amplitude.

```
def magnetization_bloch(x, Tc, M0, beta):
    m = M0*((1-(x/Tc)**(3/2))*np.heaviside(Tc-x, 0.5))**beta
    return m


plt.figure()
temperatures = np.r_[10:1000:0.1]
plt.plot(temperatures, magnetization_bloch(temperatures, 770, 1, 0.75))
plt.xlabel('Temperature [K]')
plt.ylabel('Magneitzation')
plt.show()
```



The `calc_magnetization_map` method must be overwritten to allow for calculating magnetization dynamics.

```
class SimpleMagnetization(ud.Magnetization):

    def calc_magnetization_map(self, delays, **kwargs):
        temp_map = kwargs['temp_map']
        T_c = kwargs['T_c']
        magnetization_map = np.zeros([len(delays),
                                      self.S.get_number_of_layers(), 3])
        handles = self.S.get_layer_vectors()[2]

        for i, handle in enumerate(handles):
```

(continues on next page)

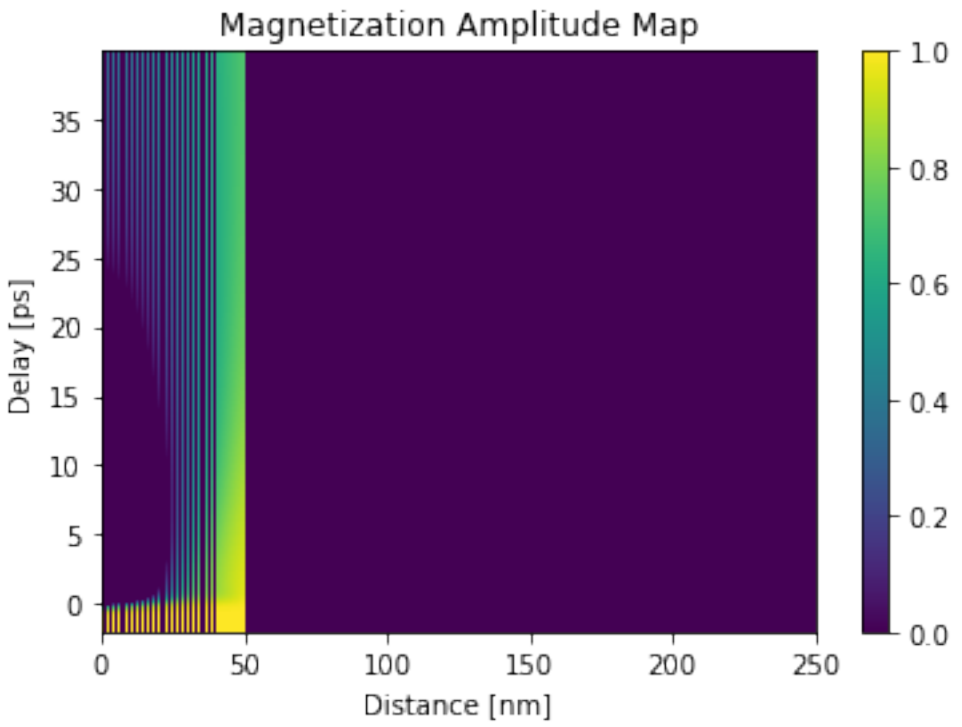
(continued from previous page)

```
            if handle.id in ['Fe_left', 'Fe_right']:
                magnetization_map[:, i, 0] = magnetization_bloch(temp_map[:, i], T_c, 1,␣
↪.75)
            handle.magnetization['phi']
            magnetization_map[:, i, 1] = handle.magnetization['phi'].to_base_units().
↪magnitude
            magnetization_map[:, i, 2] = handle.magnetization['gamma'].to_base_units().
↪magnitude

        return magnetization_map
```

```
mag = SimpleMagnetization(S, True)
mag.save_data = False

magnetization_map = mag.get_magnetization_map(delays,
                                              strain_map=strain_map,
                                              temp_map=temp_map,
                                              T_c=770)
```

```
Calculating _magnetization_map_ ...
Elapsed time for _magnetization_map_: 0.021199 s
```

```
plt.figure()
plt.pcolormesh(distances.to('nm').magnitude, delays.to('ps').magnitude,
               magnetization_map[:, :, 0], shading='auto')
plt.title('Magnetization Amplitude Map')
plt.ylabel('Delay [ps]')
plt.xlabel('Distance [nm]')
plt.colorbar()
plt.show()
```



Initialize dynamical magnetic X-ray simulations

The `XrayDynMag` class requires a `Structure` object and a boolean `force_recalc` in order overwrite previous simulation results.

These results are saved in the `cache_dir` when `save_data` is enabled. Printing simulation messages can be en-/disabled using `disp_messages` and progress bars can using the boolean switch `progress_bar`.

```
dyn_mag = ud.XrayDynMag(S, True)
dyn_mag.disp_messages = True
dyn_mag.save_data = False
```

```
incoming polarizations set to: sigma
analyzer polarizations set to: unpolarized
```

Homogeneous magnetic X-ray scattering

For the case of homogeneously strained/magnetized samples, the dynamical magnetic X-ray scattering simulations can be greatly simplified, which saves a lot of computational time.

q_z-scan

The `XrayDynMag` object requires an `energy` and scattering vector `qz` to run the simulations.
Both parameters can be arrays and the resulting reflectivity has a first dimension for the photon energy and the a second for the scattering vector.

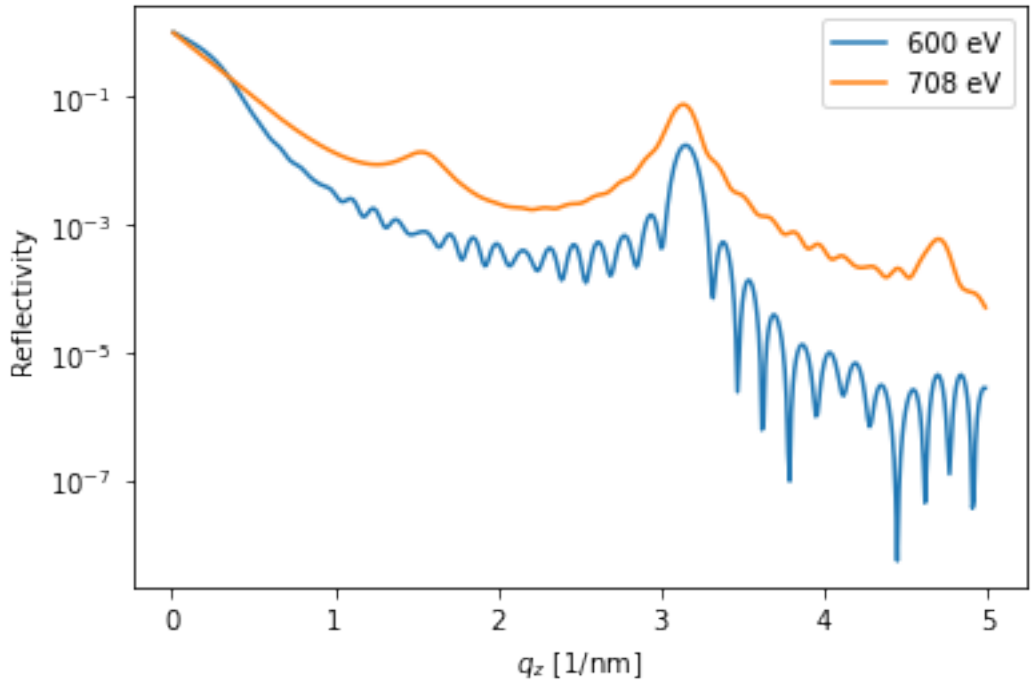
The resulting reflectivity is always calculated for the actual magnetization (`R_hom`) of the sample, as well as for opposite magnetization (`R_hom_phi`).

```
dyn_mag.energy = np.r_[600, 708]*u.eV  # set two photon energies
dyn_mag.qz = np.r_[0.01:5:0.01]/u.nm  # qz range
# this is the actual calculation
R_hom, R_hom_phi, _, _ = dyn_mag.homogeneous_reflectivity()
```

```
Calculating _homogeneous_reflectivity_ ...
Elapsed time for _homogeneous_reflectivity_: 0.715025 s
```

In this example half-order antiferromagnetic Bragg peaks appear only at the resonance.

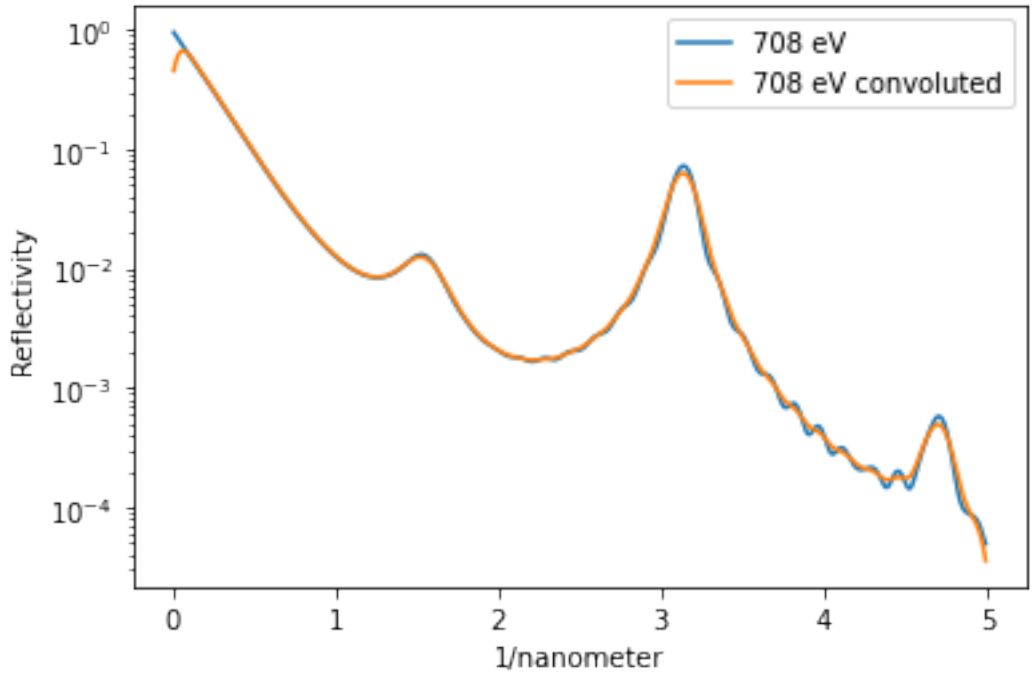
```
plt.figure()
plt.semilogy(dyn_mag.qz[0, :], R_hom[0, :], label='{}'.format(dyn_mag.energy[0]))
plt.semilogy(dyn_mag.qz[1, :], R_hom[1, :], label='{}'.format(dyn_mag.energy[1]))
plt.ylabel('Reflectivity')
plt.xlabel(r'$q_z$ [1/nm]')
plt.legend()
plt.show()
```



Post-Processing

Simple convolution of the results with an arbitrary function handle.

```
FWHM = 0.01/1e-10  # Angstrom
sigma = FWHM/2.3548

handle = lambda x: np.exp(-((x)/sigma)**2/2)
y_conv = dyn_mag.conv_with_function(R_hom[1, :], dyn_mag._qz[1, :], handle)

plt.figure()
plt.semilogy(dyn_mag.qz[0, :], R_hom[1, :], label='{}'.format(dyn_mag.energy[1]))
plt.semilogy(dyn_mag.qz[0, :], y_conv, label='{} convoluted'.format(dyn_mag.energy[1]))
plt.ylabel('Reflectivity')
plt.legend()
plt.show()
```



Energy- and q_z-scan

```
dyn_mag.energy = np.r_[690:730:0.1]*u.eV  # set the energy range
dyn_mag.qz = np.r_[0.01:5:0.01]/u.nm  # qz range
# this is the actual calculation
R_hom, R_hom_phi, _, _ = dyn_mag.homogeneous_reflectivity()
```

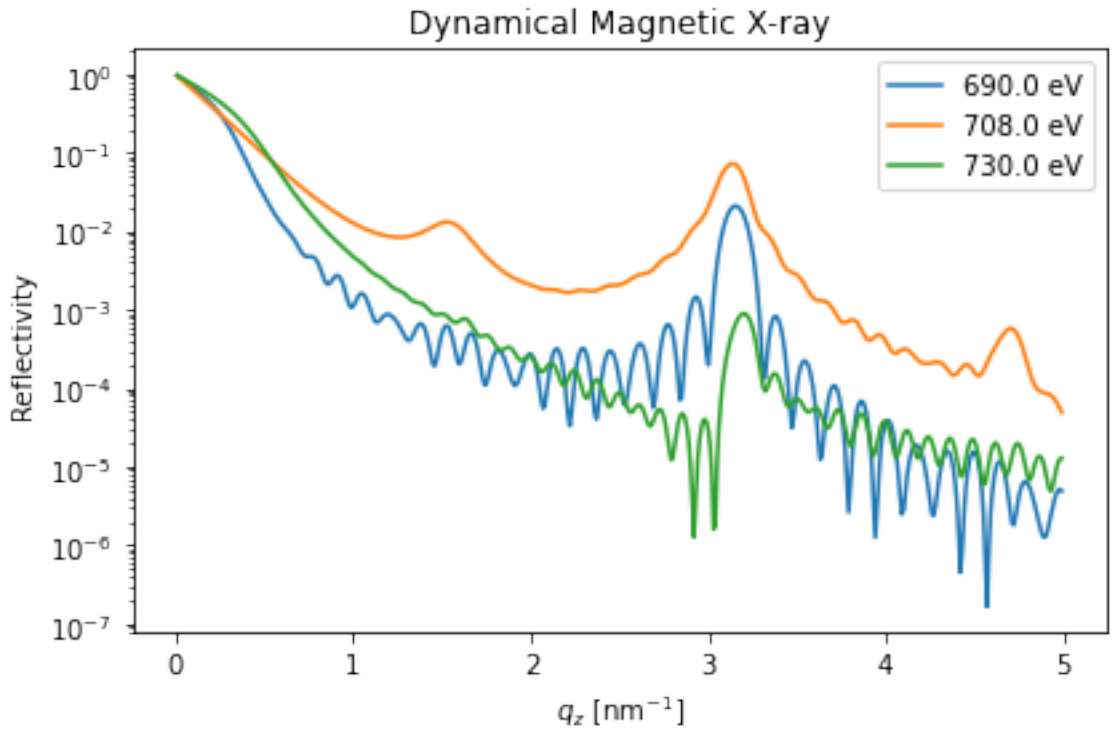
```
Calculating _homogeneous_reflectivity_ ...
Elapsed time for _homogeneous_reflectivity_: 141.994139 s
```

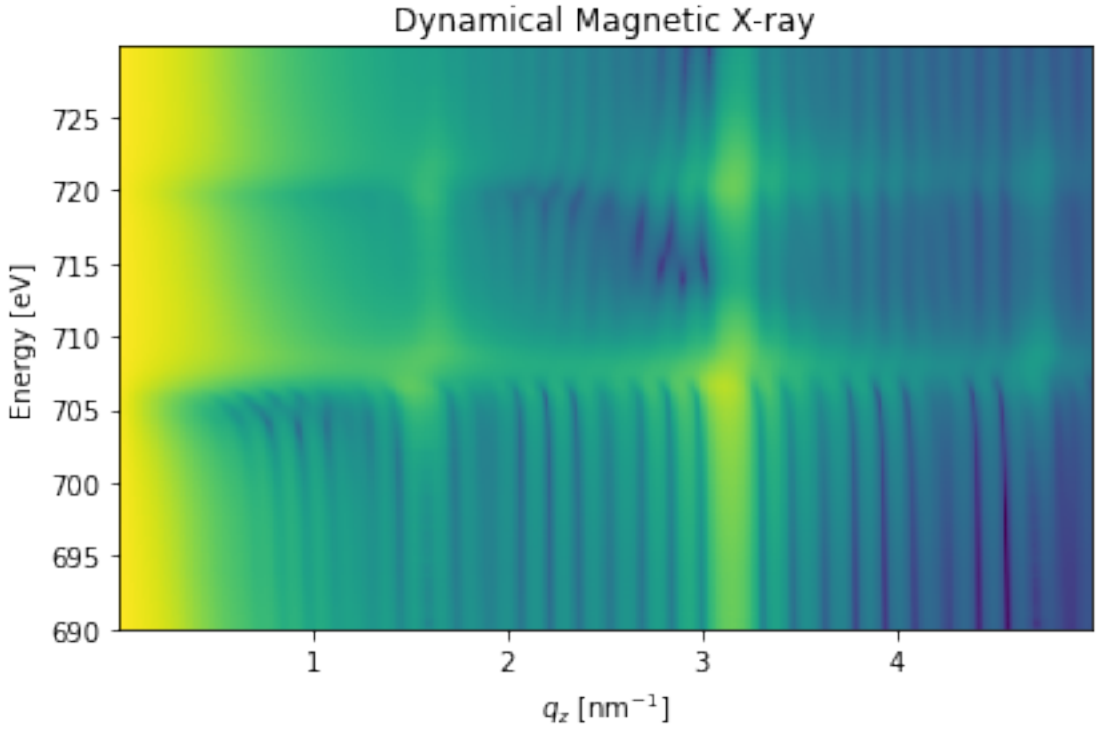
```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_hom[0, :],
             label=np.round(dyn_mag.energy[0]))
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_hom[180, :],
             label=np.round(dyn_mag.energy[180]))
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_hom[-1, :],
             label=np.round(dyn_mag.energy[-1]))

plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.ylabel('Reflectivity')
plt.legend()
plt.title('Dynamical Magnetic X-ray')

plt.subplot(2, 1, 2)
plt.pcolormesh(dyn_mag.qz[0, :].to('1/nm').magnitude, dyn_mag.energy.magnitude,
               np.log10(R_hom[:, :]), shading='auto')
plt.title('Dynamical Magnetic X-ray')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Energy [eV]')
plt.xlabel('$q_z$ [nm$^{-1}$]')

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


Dynamical Magnetic X-ray
690.0 eV
708.0 eV
730.0 eV
Reflectivity
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
0
1
2
3
4
5
q_z [nm^{-1}]

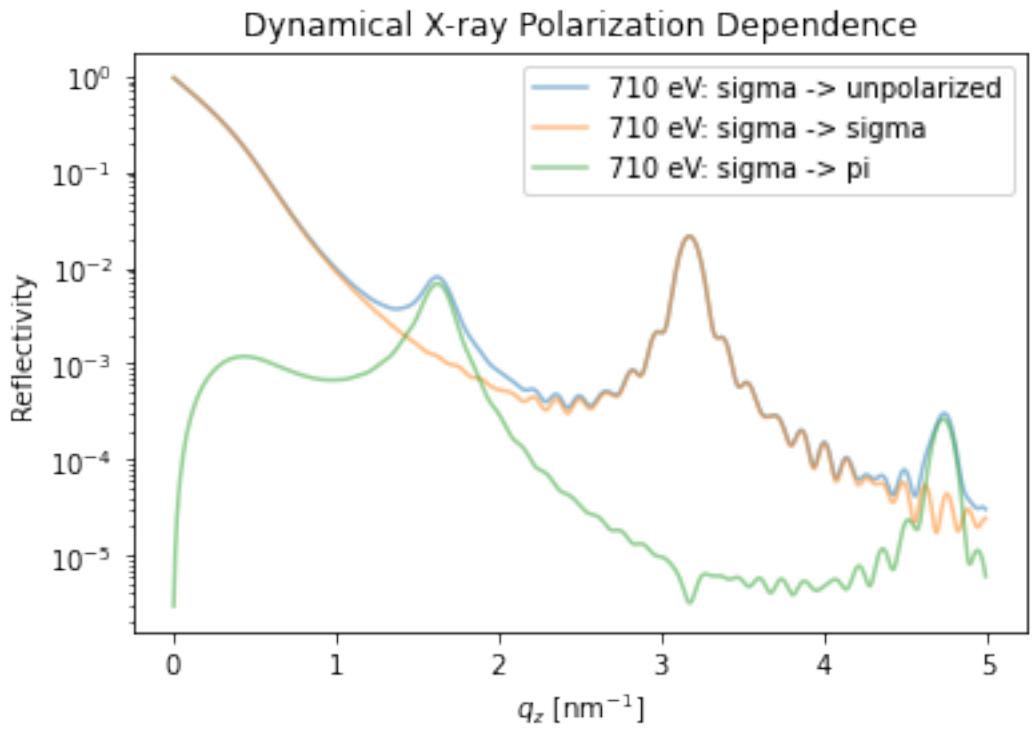

Dynamical Magnetic X-ray
Energy [eV]
725
720
715
710
705
700
695
690
1
2
3
4
q_z [nm^{-1}]

Polarization dependence

The `XrayDynMag` allows to set the incoming and outgoing polarization of the X-rays.

```
dyn_mag.energy = np.r_[710]*u.eV  # set two photon energies
dyn_mag.qz = np.r_[0.01:5:0.01]/u.nm  # qz range

plt.figure()
dyn_mag.set_polarization(3, 0)
R_hom, R_hom_phi, _, _ = dyn_mag.homogeneous_reflectivity()
plt.semilogy(dyn_mag.qz[0, :], R_hom[0, :],
             label='{}: sigma -> unpolarized'.format(dyn_mag.energy[0]),
             alpha=0.5)

dyn_mag.set_polarization(3, 3)
R_hom, R_hom_phi, _, _ = dyn_mag.homogeneous_reflectivity()
plt.semilogy(dyn_mag.qz[0, :], R_hom[0, :],
             label='{}: sigma -> sigma'.format(dyn_mag.energy[0]), alpha=0.5)

dyn_mag.set_polarization(3, 4)
R_hom, R_hom_phi, _, _ = dyn_mag.homogeneous_reflectivity()
plt.semilogy(dyn_mag.qz[0, :], R_hom[0, :],
             label='{}: sigma -> pi'.format(dyn_mag.energy[0]), alpha=0.5)

plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.ylabel('Reflectivity')
plt.title('Dynamical X-ray Polarization Dependence')
plt.legend()
plt.show()
```

```
incoming polarizations set to: sigma
analyzer polarizations set to: unpolarized
Calculating _homogeneous_reflectivity_ ...
Elapsed time for _homogeneous_reflectivity_: 0.480793 s
incoming polarizations set to: sigma
analyzer polarizations set to: sigma
Calculating _homogeneous_reflectivity_ ...
Elapsed time for _homogeneous_reflectivity_: 0.379073 s
incoming polarizations set to: sigma
analyzer polarizations set to: pi
Calculating _homogeneous_reflectivity_ ...
Elapsed time for _homogeneous_reflectivity_: 0.361267 s
```



Inhomogeneous dynamical magnetic X-ray scattering

The `inhomogeneous_reflectivity()` method allows to calculate the transient magnetic X-ray reflectivity according to a `strain_map` and/or `magnetization_map`.

```
dyn_mag.energy = np.r_[708]*u.eV  # set the energy range
dyn_mag.qz = np.r_[1:3.5:0.01]/u.nm  # qz range
dyn_mag.set_polarization(3, 0)
```

```
incoming polarizations set to: sigma
analyzer polarizations set to: unpolarized
```

```
R_seq, R_seq_phi, _, _ = dyn_mag.inhomogeneous_reflectivity(
    strain_map=strain_map,
    magnetization_map=magnetization_map
)
```

```
Calculating _inhomogeneous_reflectivity_ ...
```

```
Progress:   0%|          | 0/420 [00:00<?, ?it/s]
```

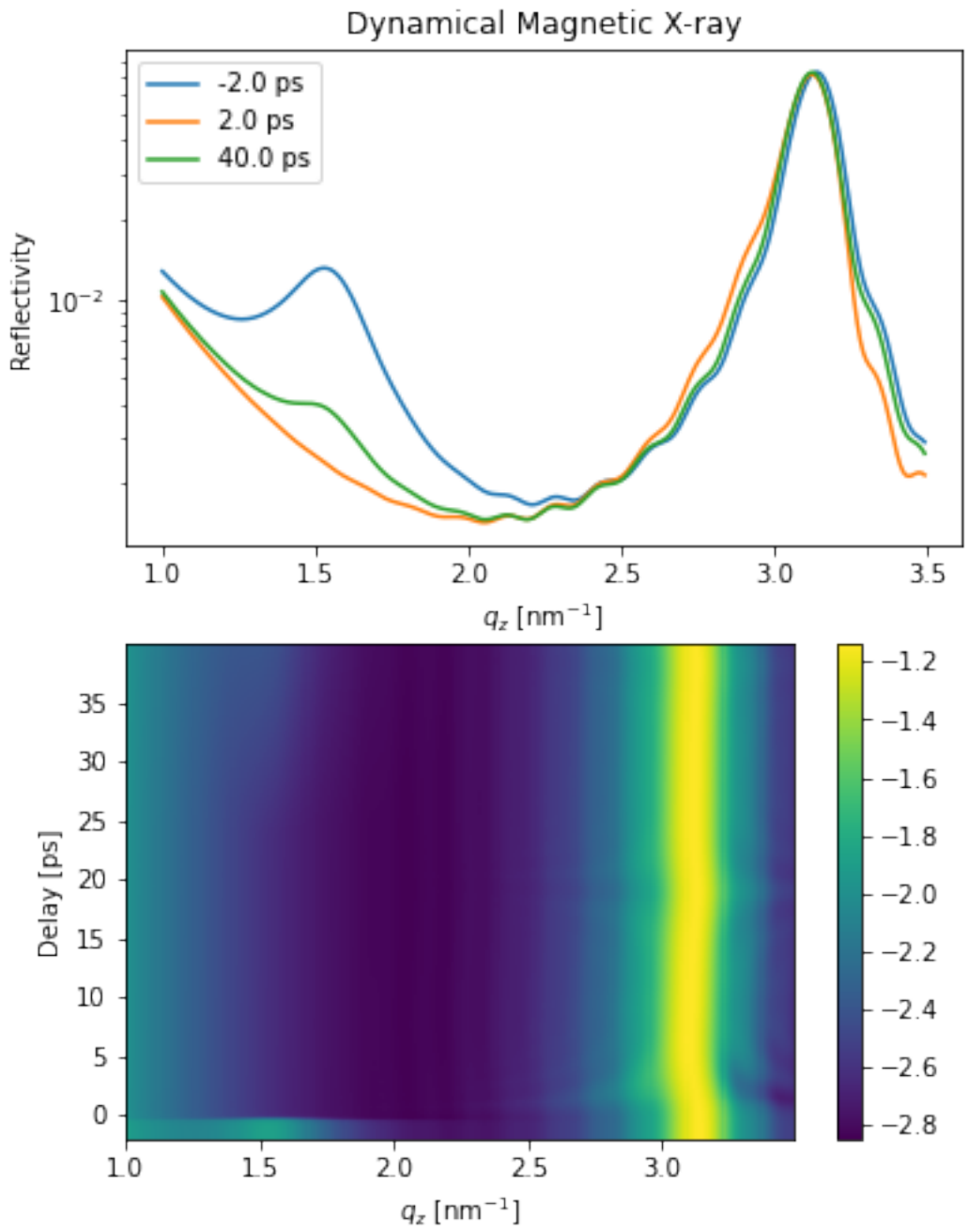
```
Elapsed time for _inhomogeneous_reflectivity_: 1834.662767 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq[0, 0, :], label=np.round(delays[0]))
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq[40, 0, :], label=np.round(delays[40]))
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq[-1, 0, :], label=np.round(delays[-1]))

plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.ylabel('Reflectivity')
plt.legend()
plt.title('Dynamical Magnetic X-ray')

plt.subplot(2, 1, 2)
plt.pcolormesh(dyn_mag.qz[0, :].to('1/nm').magnitude, delays.to('ps').magnitude,
               np.log10(R_seq[:, 0, :]), shading='auto')

plt.ylabel('Delay [ps]')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.colorbar()
plt.show()
```



Parallel dynamical X-ray scattering

Parallelization is fundamentally implemented but needs further improvements. It works similar as with `XrayDyn`.

You need to install the `udkm1Dsim` with the `parallel` option which essentially add the `Dask` package to the requirements:

```
> pip install udkm1Dsim[parallel]
```

You can also install/add `Dask` manually, e.g. via pip:

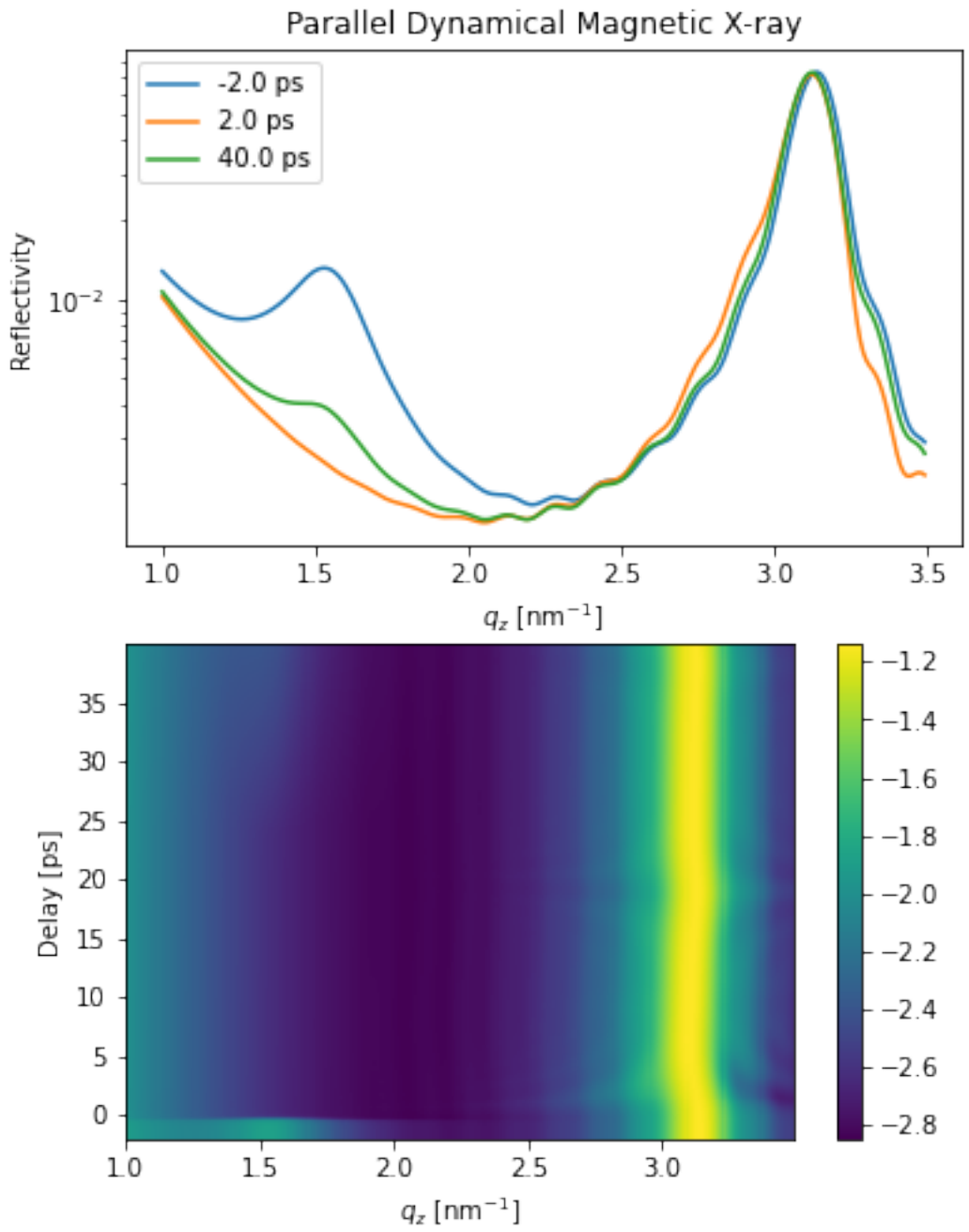
```
> pip install dask
```

Please refer to the Dask documentation for more details on parallel computing in Python.

```
try:
    from dask.distributed import Client
    client = Client()
    R_seq_par, R_seq_phi_par, _, _ = dyn_mag.inhomogeneous_reflectivity(
        strain_map=strain_map,
        magnetization_map=magnetization_map,
        calc_type='parallel', dask_client=client
    )
    client.close()
except:
    pass
```

```
Calculating _inhomogeneous_reflectivity_ ...
Elapsed time for _inhomogeneous_reflectivity_: 818.527532 s
```

```
plt.figure(figsize=[6, 8])
plt.subplot(2, 1, 1)
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq_par[0, 0, :], label=np.round(delays[0]))
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq_par[40, 0, :], label=np.
↪round(delays[40]))
plt.semilogy(dyn_mag.qz[0, :].to('1/nm'), R_seq_par[-1, 0, :], label=np.round(delays[-
↪1]))

plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.ylabel('Reflectivity')
plt.legend()
plt.title('Parallel Dynamical Magnetic X-ray')

plt.subplot(2, 1, 2)
plt.pcolormesh(dyn_mag.qz[0, :].to('1/nm').magnitude, delays.to('ps').magnitude,
               np.log10(R_seq_par[:, 0, :]), shading='auto')

plt.ylabel('Delay [ps]')
plt.xlabel('$q_z$ [nm$^{-1}$]')
plt.colorbar()
plt.show()
```



5.4 API Documentation

5.4.1 structures.atoms

Classes:

Atom(symbol, **kwargs)	Smallest structural unit of which larger structures can be build.
AtomMixed(symbol, **kwargs)	Representation of mixed atoms in alloys and stochiometric mixtures.

class udkm1Dsim.structures.atoms.**Atom**(*symbol*, ***kwargs*)

Bases: `object`

Smallest structural unit of which larger structures can be build.

It holds real physical properties of on the atomic level.

Parameters
symbol (*str*) – symbol of the atom.

Keyword Arguments

- **id** (*str*) – id of the atom, may differ from symbol and/or name.
- **ionicity** (*int*) – ionicity of the atom.
- **atomic_form_factor_path** (*str*) – path to atomic form factor coeffs.
- **atomic_form_factor_source** (*str*) – either _henke_ or default _chantler_
- **magnetic_form_factor_path** (*str*) – path to magnetic form factor coeffs.

Attributes

- **symbol** (*str*) – symbol of the element.
- **id** (*str*) – id of the atom, may differ from symbol and/or name.
- **name** (*str*) – name of the element (generic).
- **atomic_number_z** (*int*) – Z atomic number.
- **mass_number_a** (*float*) – A atomic mass number.
- **ionicity** (*int*) – ionicity of the atom.
- **mass** (*float*) – mass of the atom [kg].
- **atomic_form_factor_coeff** (*ndarray[float]*) – atomic form factor. coefficients for energy-dependent atomic form factor.
- **cromer_mann_coeff** (*ndarray[float]*) – cromer-mann coefficients for angular-dependent atomic form factor.
- **magnetic_form_factor_coeff** (*ndarray[float]*) – magnetic form factor coefficients for energy-dependent magnetic form factor.
- **mag_amplitude** (*float*) – magnetization amplitude -1 .. 1.
- **mag_phi** (*float*) – phi angle of magnetization [rad].
- **mag_gamma** (*float*) – gamma angle of magnetization [rad].

References

Methods:

read_atomic_form_factor_coeff([source, filename])	The coefficients for the atomic form factor f in dependence of the photon energy E is read from a parameter file given by[1] or by[2] as default.
get_atomic_form_factor(energy)	The complex atomic form factor for the photon energy E [eV] is calculated by:
read_cromer_mann_coeff()	The Cromer-Mann coefficients (Ref.
get_cm_atomic_form_factor(energy, qz)	The atomic form factor f is calculated in dependence of the photon energy E [eV] and the z-component of the scattering vector q_z [Å $^{-1}$] (Ref.
read_magnetic_form_factor_coeff([filename])	The coefficients for the magnetic form factor m in dependence of the photon energy E is read from a parameter file.
get_magnetic_form_factor(energy)	The complex magnetic form factor is claculated by:

read_atomic_form_factor_coeff(*source='chantler'*, *filename=''*)

The coefficients for the atomic form factor f in dependence of the photon energy E is read from a parameter file given by[1] or by[2] as default.

Parameters

- **source** (*str, optional*) – source of atmoic form factors can be either _henke_ or _chantler_. Defaults to _chantler_.
- **filename** (*str, optional*) – full path and filename to the atomic form factor coefficients.

Returns

f (ndarray[float]) – atomic form factor coefficients.

get_atomic_form_factor(*energy*)

The complex atomic form factor for the photon energy E [eV] is calculated by:

$$f(E) = f_1 - if_2$$

Convention of Ref.[3] (p. 11, footnote) is a negative f_2.

Parameters

energy (*ndarray[float]*) – photon energy [eV].

Returns

f (ndarray[complex]) – energy-dependent atomic form factors.

read_cromer_mann_coeff()

The Cromer-Mann coefficients (Ref.[4]) are read from a parameter file and are returned in the following order:

$$a_1\ a_2\ a_3\ a_4\ b_1\ b_2\ b_3\ b_4\ c$$

Returns

cm (ndarray[float]) – Cromer-Mann coefficients.

[1] B. L. Henke, E. M. Gullikson & J. C. Davis, *X-Ray Interactions: Photoabsorption, Scattering, Transmission, and Reflection at E = 50-30,000 eV, Z = 1-92*, Atomic Data and Nuclear Data Tables, 54(2), 181–342, (1993).

[2] C.T. Chantler, K. Olsen, R.A. Dragoset, J. Chang, A.R. Kishore, S.A. Kotochigova, & D.S. Zucker, *Detailed Tabulation of Atomic Form Factors, Photoelectric Absorption and Scattering Cross Section, and Mass Attenuation Coefficients for Z = 1-92 from E = 1-10 eV to E = 0.4-1.0 MeV*, NIST Standard Reference Database 66.

[3] J. Als-Nielson, & D. McMorrow, Elements of Modern X-Ray Physics. New York: John Wiley & Sons, Ltd. (2001)

[4] D. T. Cromer & J. B. Mann, *X-ray scattering factors computed from numerical Hartree–Fock wave functions*, Acta Crystallographica Section A, 24(2), 321–324 (1968).

get_cm_atomic_form_factor(*energy*, *qz*)

The atomic form factor f is calculated in dependence of the photon energy E [eV] and the z-component of the scattering vector q_z [Å $^{-1}$] (Ref.[4]). Note that the Cromer-Mann coefficients are fitted for q_z in [Å $^{-1}$]!

See Ref.[Page 118, 3] (p. 235).

$$f(q_z, E) = f_{CM}(q_z) + \delta f_1(E) - i f_2(E)$$

$f_{CM}(q_z)$ is given in Ref.[Page 118, 4]:

$$f_{CM}(q_z) = \sum (a_i \, \exp(-b_i \, (q_z/4\pi)^2)) + c$$

$\delta f_1(E)$ is the dispersion correction:

$$\delta f_1(E) = f_1(E) - \left(\sum_i^4 (a_i) + c \right)$$

Thus:

$$f(q_z, E) = \sum (a_i \, \exp(-b_i \, q_z/2\pi)) + c + f_1(E) - i f_2(E) - \left(\sum (a_i) + c \right)$$

$$f(q_z, E) = \sum (a_i \, \exp(-b_i \, q_z/2\pi)) + f_1(E) - i f_2(E) - \sum (a_i)$$

Parameters

- **energy** (*ndarray[float]*) – photon energy [eV].
- **qz** (*ndarray[float]*) – scattering vector [1/m].

Returns

f (ndarray[complex]) – energy- and qz-dependent Cromer-Mann atomic form factors.

read_magnetic_form_factor_coeff(*filename=''*)

The coefficients for the magnetic form factor m in dependence of the photon energy E is read from a parameter file.

Parameters

filename (*str*) – optional full path and filename to the magnetic form factor coefficients.

Returns

m (ndarray[float]) – magnetic form factor coefficients.

get_magnetic_form_factor(*energy*)

The complex magnetic form factor is claculated by:

$$m(E) = m_1 - i m_2$$

for the photon energy E [eV].

Convention of Ref.[Page 118, 3] (p. 11, footnote) is a negative m_2

Parameters

energy (*ndarray[float]*) – photon energy [eV].

Returns

m (ndarray[complex]) – energy-dependent magnetic form factors.

class udkm1Dsim.structures.atoms.**AtomMixed**(*symbol*, ***kwargs*)

Bases: *Atom*

Representation of mixed atoms in alloys and stochiometric mixtures.

All properties of the included sub-atoms of class Atom are averaged and weighted with their stochiometric ratio.

Parameters

symbol (*str*) – symbol of the atom.

Keyword Arguments

- **id** (*str*) – id of the atom, may differ from symbol and/or name.
- **name** (*str*) – name of the mixed atom, default is symbol.
- **atomic_form_factor_path** (*str*) – path to atomic form factor coeffs.
- **magnetic_form_factor_path** (*str*) – path to magnetic form factor coeffs.

Attributes

- **symbol** (*str*) – symbol of the element.
- **id** (*str*) – id of the atom, may differ from symbol and/or name.
- **name** (*str*) – name of the mixed atom, default is symbol.
- **atomic_number_z** (*int*) – Z atomic number.
- **mass_number_a** (*float*) – A atomic mass number.
- **ionicity** (*int*) – ionicity of the atom.
- **mass** (*float*) – mass of the atom [kg].
- **atomic_form_factor_coeff** (*ndarray[float]*) – atomic form factor. coefficients for energy-dependent atomic form factor.
- **magnetic_form_factor_coeff** (*ndarray[float]*) – magnetic form factor coefficients for energy-dependent magnetic form factor.
- **mag_amplitude** (*float*) – magnetization amplitude -1 .. 1.
- **mag_phi** (*float*) – phi angle of magnetization [rad].
- **mag_gamma** (*float*) – gamma angle of magnetization [rad].
- **atoms** (*list[Atoms]*) – list of Atoms.
- **num_atoms** (*int*) – number of atoms.

Methods:

add_atom(atom, fraction)	Add an Atom instance with its stochiometric fraction and recalculate averaged properties.
read_atomic_form_factor_coeff([filename])	The coefficients for the atomic form factor f in dependence of the photon energy E must be read from an external file given by `filename`.
get_atomic_form_factor(energy)	Averaged energy dependent atomic form factor.
get_cm_atomic_form_factor(energy, qz)	Averaged energy and qz-dependent atomic form factors.
read_magnetic_form_factor_coeff([filename])	The coefficients for the magnetic form factor m in dependence of the photon energy E must be read from an external file given by `filename`.
get_magnetic_form_factor(energy)	Mixed energy dependent magnetic form factors.

add_atom(*atom*, *fraction*)

Add an Atom instance with its stochiometric fraction and recalculate averaged properties.

Parameters

- **atom** (*Atom*) – atom to add.
- **fraction** (*float*) – fraction of the atom - sum of all fractions must be 1.

read_atomic_form_factor_coeff(*filename=''*)

The coefficients for the atomic form factor f in dependence of the photon energy E must be read from an external file given by `filename`.

Parameters
filename (*str, optional*) – full path and filename to the atomic form factor coefficients.

Returns
f (ndarray[float]) – atomic form factor coefficients.

get_atomic_form_factor(*energy*)

Averaged energy dependent atomic form factor. If `atomic_form_factor_path` was given on initialization this file will be used instead.

Parameters
energy (*ndarray[float]*) – photon energy [eV].

Returns
f (ndarray[complex]) – energy-dependent atomic form factors.

get_cm_atomic_form_factor(*energy*, *qz*)

Averaged energy and qz-dependent atomic form factors.

Parameters

- **energy** (*ndarray[float]*) – photon energy [eV].
- **qz** (*ndarray[float]*) – scattering vector [1/m].

Returns
f (ndarray[complex]) – energy- and qz-dependent Cromer-Mann atomic form factors.

read_magnetic_form_factor_coeff(*filename=''*)

The coefficients for the magnetic form factor m in dependence of the photon energy E must be read from an external file given by `filename`.

Parameters
filename (*str*) – optional full path and filename to the magnetic form factor coefficients.

Returns
m (ndarray[float]) – magnetic form factor coefficients.

get_magnetic_form_factor(*energy*)

Mixed energy dependent magnetic form factors.

Parameters
energy (*ndarray[float]*) – photon energy [eV].

Returns
f (ndarray[complex]) – energy-dependent magnetic form factors.

5.4.2 structures.layers

Classes:

Layer(id, name, **kwargs)	Base class of real physical layers, such as amorphous layers and unit cells.
AmorphousLayer(id, name, thickness, density, ...)	Representation of amorphous layers containing an Atom or AtomMixed.
UnitCell(id, name, c_axis, **kwargs)	Representation of unit cells made of one or multiple Atom or AtomMixed instances at defined positions.

class udkm1Dsim.structures.layers.**Layer**(*id*, *name*, ***kwargs*)

Bases: object

Base class of real physical layers, such as amorphous layers and unit cells.

It holds different structural, thermal, and elastic properties that are relevant for simulations.

Parameters

- **id** (*str*) – id of the layer.
- **name** (*str*) – name of the layer.

Keyword Arguments

- **roughness** (*float*) – gaussian width of the top roughness of a layer.
- **deb_wal_fac** (*float*) – Debye Waller factor.
- **sound_vel** (*float*) – sound velocity.
- **phonon_damping** (*float*) – phonon damping.
- **opt_pen_depth** (*float*) – optical penetration depth.
- **opt_ref_index** (*float*) – refractive index.
- **opt_ref_index_per_strain** (*float*) – change of refractive index per strain.
- **heat_capacity** (*float*) – heat capacity.
- **therm_cond** (*float*) – thermal conductivity.
- **lin_therm_exp** (*float*) – linear thermal expansion.
- **sub_system_coupling** (*float*) – sub-system coupling.

Attributes

- **id** (*str*) – id of the layer.
- **name** (*str*) – name of the layer.
- **thickness** (*float*) – thickness of the layer [m].
- **mass** (*float*) – mass of the layer [kg].
- **mass_unit_area** (*float*) – mass of layer normalized to unit area of 1 Å^2 [kg].
- **density** (*float*) – density of the layer [kg/m^3].
- **area** (*float*) – area of layer [m^2].
- **volume** (*float*) – volume of layer [m^3].
- **roughness** (*float*) – gaussian width of the top roughness of a layer [m].
- **deb_wal_fac** (*float*) – Debye-Waller factor [m^2].
- **sound_vel** (*float*) – longitudinal sound velocity in the layer [m/s].
- **spring_const** (*ndarray[float]*) – spring constant of the layer [kg/s^2] and higher orders.
- **phonon_damping** (*float*) – damping constant of phonon propagation [kg/s].
- **opt_pen_depth** (*float*) – optical penetration depth of the layer [m].
- **opt_ref_index** (*ndarray[float]*) – optical refractive index - real and imagenary part $n + i\kappa$.
- **opt_ref_index_per_strain** (*ndarray[float]*) – optical refractive index change per strain - real and imagenary part $\frac{dn}{d\eta} + i\frac{d\kappa}{d\eta}$.
- **therm_cond** (*list[@lambda]*) – list of T-dependent thermal conductivity [W/(m K)].
- **lin_therm_exp** (*list[@lambda]*) – list of T-dependent linear thermal expansion coefficient (relative).
- **int_lin_therm_exp** (*list[@lambda]*) – list of T-dependent integrated linear thermal expansion coefficient.
- **heat_capacity** (*list[@lambda]*) – list of T-dependent heat capacity function [J/(kg K)].
- **int_heat_capacity** (*list[@lambda]*) – list of T-dependent integrated heat capacity function.
- **sub_system_coupling** (*list[@lambda]*) – list of coupling functions of different subsystems [W/m^3].
- **num_sub_systems** (*int*) – number of subsystems for heat and phonons (electrons, lattice, spins, ...).
- **eff_spin** (*float*) – effective spin.
- **curie_temp** (*float*) – Curie temperature [K].
- **mf_exch_coupling** (*float*) – mean field exchange coupling constant [m^{2}kg/s^2].
- **lamda** (*float*) – intrinsic coupling to bath parameter.
- **mag_moment** (*float*) – atomic magnetic moment [mu_Bohr].
- **aniso_exponent** (*ndarray[float]*) – exponent of T-dependence uniaxial anisotropy.
- **anisotropy** (*float*) – anisotropy at T=0 K [J/m^3] as x,y,z component vector.
- **exch_stiffness** (*float*) – exchange stiffness at T=0 K [J/m].
- **mag_saturation** (*float*) – saturation magnetization at 0 K [J/T/m^3].

Methods:

check_input(inputs)	Checks the input and create a list of function handle strings with T as argument.
get_property_dict(**kwargs)	Returns a dictionary with all parameters.
get_acoustic_impedance()	Calculates the acoustic impedance.
set_ho_spring_constants(HO)	Set the higher orders of the spring constant for anharmonic phonon simulations.
set_opt_pen_depth_from_ref_index(wavelength	Set the optical penetration depth from the optical referactive index for a given wavelength.
calc_spring_const()	Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness
calc_mf_exchange_coupling()	Calculate the mean-field exchange coupling constant

check_input(*inputs*)

Checks the input and create a list of function handle strings with T as argument. Inputs can be strings, floats, ints, or pint quantaties.

Parameters
inputs (*list[str, int, float, Quantity]*) – list of strings, int, floats, or Pint quantities.

Returns

(tuple) –

- *output (list[@lambda])* - list of lambda functions.
- *output_strs (list[str])* - list of string-representations.

get_property_dict(***kwargs*)

Returns a dictionary with all parameters. objects or dicts and objects are converted to strings. if a type is given, only these properties are returned.

Parameters
****kwargs** (*list[str]*) – types of requested properties.

Returns
R (dict) – dictionary with requested properties.

get_acoustic_impedance()

Calculates the acoustic impedance.

Returns
Z (float) – acoustic impedance.

set_ho_spring_constants(*HO*)

Set the higher orders of the spring constant for anharmonic phonon simulations.

Parameters
HO (*ndarray[float]*) – higher order spring constants.

set_opt_pen_depth_from_ref_index(*wavelength*)

Set the optical penetration depth from the optical referactive index for a given wavelength.

Parameters
wavelength (*Quantity*) – wavelength as Pint Quantitiy.

`calc_spring_const()`

Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness

$$k = m \left(\frac{v}{c}\right)^2$$

`calc_mf_exchange_coupling()`

Calculate the mean-field exchange coupling constant

$$J = \frac{3}{S_{eff}+1} k_B T_C$$

`class` udkm1Dsim.structures.layers.**AmorphousLayer**(*id*, *name*, *thickness*, *density*, ***kwargs*)

Bases: *Layer*

Representation of amorphous layers containing an Atom or AtomMixed.

Parameters

- **id** (*str*) – id of the layer.
- **name** (*str*) – name of layer.
- **thickness** (*float*) – thickness of the layer.
- **density** (*float*) – density of the layer.

Keyword Arguments

- **atom** (*object*) – Atom or AtomMixed in the layer.
- **roughness** (*float*) – gaussian width of the top roughness of a layer.
- **deb_wal_fac** (*float*) – Debye Waller factor.
- **sound_vel** (*float*) – sound velocity.
- **phonon_damping** (*float*) – phonon damping.
- **roughness** – gaussian width of the top roughness of a layer.
- **opt_pen_depth** (*float*) – optical penetration depth.
- **opt_ref_index** (*float*) – refractive index.
- **opt_ref_index_per_strain** (*float*) – change of refractive index per strain.
- **heat_capacity** (*float*) – heat capacity.
- **therm_cond** (*float*) – thermal conductivity.
- **lin_therm_exp** (*float*) – linear thermal expansion.
- **sub_system_coupling** (*float*) – sub-system coupling.

Attributes

- **id** (*str*) – id of the layer.
- **name** (*str*) – name of the layer.
- **thickness** (*float*) – thickness of the layer [m].
- **mass** (*float*) – mass of the layer [kg].
- **mass_unit_area** (*float*) – mass of layer normalized to unit area of 1 Å^2 [kg].

- **density** (*float*) – density of the layer [kg/m^3].
- **area** (*float*) – area of layer [m^2].
- **volume** (*float*) – volume of layer [m^3].
- **roughness** (*float*) – gaussian width of the top roughness of a layer [m].
- **deb_wal_fac** (*float*) – Debye-Waller factor [m^2].
- **sound_vel** (*float*) – longitudinal sound velocity in the layer [m/s].
- **spring_const** (*ndarray[float]*) – spring constant of the layer [kg/s^2] and higher orders.
- **phonon_damping** (*float*) – damping constant of phonon propagation [kg/s].
- **opt_pen_depth** (*float*) – optical penetration depth of the layer [m].
- **opt_ref_index** (*ndarray[float]*) – optical refractive index - real and imagenary part $n + i\kappa$.
- **opt_ref_index_per_strain** (*ndarray[float]*) – optical refractive index change per strain - real and imagenary part $\frac{dn}{d\eta} + i\frac{d\kappa}{d\eta}$.
- **therm_cond** (*list[@lambda]*) – list of HANDLES T-dependent thermal conductivity [W/(m K)].
- **lin_therm_exp** (*list[@lambda]*) – list of T-dependent linear thermal expansion coefficient (relative).
- **int_lin_therm_exp** (*list[@lambda]*) – list of T-dependent integrated linear thermal expansion coefficient.
- **heat_capacity** (*list[@lambda]*) – list of T-dependent heat capacity function [J/(kg K)].
- **int_heat_capacity** (*list[@lambda]*) – list of T-dependent integrated heat capacity function.
- **sub_system_coupling** (*list[@lambda]*) – list of of coupling functions of different subsystems [W/m^3].
- **num_sub_systems** (*int*) – number of subsystems for heat and phonons (electrons, lattice, spins, ...).
- **eff_spin** (*float*) – effective spin.
- **curie_temp** (*float*) – Curie temperature [K].
- **mf_exch_coupling** (*float*) – mean field exchange coupling constant [m^{2}kg/s^2].
- **lamda** (*float*) – intrinsic coupling to bath parameter.
- **mag_moment** (*float*) – atomic magnetic moment [mu_Bohr].
- **aniso_exponent** (*ndarray[float]*) – exponent of T-dependence uniaxial anisotropy.
- **anisotropy** (*float*) – anisotropy at T=0 K [J/m^3] as x,y,z component vector.
- **exch_stiffness** (*float*) – exchange stiffness at T=0 K [J/m].
- **mag_saturation** (*float*) – saturation magnetization at 0 K [J/T/m^3].
- **magnetization** (*dict[float]*) – magnetization amplitude, phi and gamma angle inherited from the atom.
- **atom** (*object*) – Atom or AtomMixed in the layer.

Methods:

calc_mf_exchange_coupling()	Calculate the mean-field exchange coupling constant
calc_spring_const()	Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness
check_input(inputs)	Checks the input and create a list of function handle strings with T as argument.
get_acoustic_impedance()	Calculates the acoustic impedance.
get_property_dict(**kwargs)	Returns a dictionary with all parameters.
set_ho_spring_constants(HO)	Set the higher orders of the spring constant for anharmonic phonon simulations.
set_opt_pen_depth_from_ref_index(wavelength	Set the optical penetration depth from the optical referactive index for a given wavelength.

calc_mf_exchange_coupling()

Calculate the mean-field exchange coupling constant

$$J = \frac{3}{S_{eff}+1} k_B T_C$$

calc_spring_const()

Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness

$$k = m \left(\frac{v}{c}\right)^2$$

check_input(*inputs*)

Checks the input and create a list of function handle strings with T as argument. Inputs can be strings, floats, ints, or pint quantaties.

Parameters

inputs (*list[str, int, float, Quantity]*) – list of strings, int, floats, or Pint quantities.

Returns

(tuple) –

- *output (list[@lambda])* - list of lambda functions.
- *output_strs (list[str])* - list of string-representations.

get_acoustic_impedance()

Calculates the acoustic impedance.

Returns

Z (float) – acoustic impedance.

get_property_dict(***kwargs*)

Returns a dictionary with all parameters. objects or dicts and objects are converted to strings. if a type is given, only these properties are returned.

Parameters

****kwargs** (*list[str]*) – types of requested properties.

Returns

R (dict) – dictionary with requested properties.

set_ho_spring_constants(*HO*)

Set the higher orders of the spring constant for anharmonic phonon simulations.

Parameters
HO (*ndarray[float]*) – higher order spring constants.

set_opt_pen_depth_from_ref_index(*wavelength*)

Set the optical penetration depth from the optical referactive index for a given wavelength.

Parameters
wavelength (*Quantity*) – wavelength as Pint Quantitiy.

class udkm1Dsim.structures.layers.**UnitCell**(*id*, *name*, *c_axis*, ***kwargs*)

Bases: *Layer*

Representation of unit cells made of one or multiple Atom or AtomMixed instances at defined positions.

Parameters

- **id** (*str*) – id of the UnitCell.
- **name** (*str*) – name of the UnitCell.
- **c_axis** (*float*) – c-axis of the UnitCell.

Keyword Arguments

- **a_axis** (*float*) – a-axis of the UnitCell.
- **b_axis** (*float*) – b-axis of the UnitCell.
- **deb_wal_fac** (*float*) – Debye Waller factor.
- **sound_vel** (*float*) – sound velocity.
- **phonon_damping** (*float*) – phonon damping.
- **roughness** (*float*) – gaussian width of the top roughness of a layer.
- **opt_pen_depth** (*float*) – optical penetration depth.
- **opt_ref_index** (*float*) – refractive index.
- **opt_ref_index_per_strain** (*float*) – change of refractive index per strain.
- **heat_capacity** (*float*) – heat capacity.
- **therm_cond** (*float*) – thermal conductivity.
- **lin_therm_exp** (*float*) – linear thermal expansion.
- **sub_system_coupling** (*float*) – sub-system coupling.

Attributes

- **id** (*str*) – id of the layer.
- **name** (*str*) – name of the layer.
- **c_axis** (*float*) – out-of-plane c-axis [m].
- **a_axis** (*float*) – in-plane a-axis [m].
- **b_axis** (*float*) – in-plane b-axis [m].
- **thickness** (*float*) – thickness of the layer [m].
- **mass** (*float*) – mass of the layer [kg].

- **mass_unit_area** (*float*) – mass of layer normalized to unit area of 1 Å^2 [kg].
- **density** (*float*) – density of the layer [kg/m^3].
- **area** (*float*) – area of layer [m^2].
- **volume** (*float*) – volume of layer [m^3].
- **roughness** (*float*) – gaussian width of the top roughness of a layer [m].
- **deb_wal_fac** (*float*) – Debye-Waller factor [m^2].
- **sound_vel** (*float*) – longitudinal sound velocity in the layer [m/s].
- **spring_const** (*ndarray[float]*) – spring constant of the layer [kg/s^2] and higher orders.
- **phonon_damping** (*float*) – damping constant of phonon propagation [kg/s].
- **opt_pen_depth** (*float*) – optical penetration depth of the layer [m].
- **opt_ref_index** (*ndarray[float]*) – optical refractive index - real and imagenary part $n + i\kappa$.
- **opt_ref_index_per_strain** (*ndarray[float]*) – optical refractive index change per strain - real and imagenary part $\frac{dn}{d\eta} + i\frac{d\kappa}{d\eta}$.
- **therm_cond** (*list[@lambda]*) – list of HANDLES T-dependent thermal conductivity [W/(m K)].
- **lin_therm_exp** (*list[@lambda]*) – list of T-dependent linear thermal expansion coefficient (relative).
- **int_lin_therm_exp** (*list[@lambda]*) – list of T-dependent integrated linear thermal expansion coefficient.
- **heat_capacity** (*list[@lambda]*) – list of T-dependent heat capacity function [J/(kg K)].
- **int_heat_capacity** (*list[@lambda]*) – list of T-dependent integrated heat capacity function.
- **sub_system_coupling** (*list[@lambda]*) – list of of coupling functions of different subsystems [W/m^3].
- **num_sub_systems** (*int*) – number of subsystems for heat and phonons (electrons, lattice, spins, ...).
- **atoms** (*list[atom, @lambda]*) – list of atoms and function handle for strain dependent displacement.
- **num_atoms** (*int*) – number of atoms in unit cell.
- **eff_spin** (*float*) – effective spin.
- **curie_temp** (*float*) – Curie temperature [K].
- **mf_exch_coupling** (*float*) – mean field exchange coupling constant [m^{2}kg/s^2].
- **lamda** (*float*) – intrinsic coupling to bath parameter.
- **mag_moment** (*float*) – atomic magnetic moment [mu_Bohr].
- **aniso_exponent** (*ndarray[float]*) – exponent of T-dependence uniaxial anisotropy.
- **anisotropy** (*float*) – anisotropy at T=0 K [J/m^3] as x,y,z component vector.
- **exch_stiffness** (*float*) – exchange stiffness at T=0 K [J/m].
- **mag_saturation** (*float*) – saturation magnetization at 0 K [J/T/m^3].
- **magnetization** (*list[float]*) – magnetization amplitudes, phi, and gamma angle of each atom in the unit cell.

Methods:

calc_mf_exchange_coupling()	Calculate the mean-field exchange coupling constant
calc_spring_const()	Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness
check_input(inputs)	Checks the input and create a list of function handle strings with T as argument.
get_acoustic_impedance()	Calculates the acoustic impedance.
get_property_dict(**kwargs)	Returns a dictionary with all parameters.
set_ho_spring_constants(HO)	Set the higher orders of the spring constant for anharmonic phonon simulations.
set_opt_pen_depth_from_ref_index(wavelength	Set the optical penetration depth from the optical referactive index for a given wavelength.
visualize(**kwargs)	Allows for 3D presentation of unit cell by allow for a & b coordinate of atoms.
add_atom(atom, position)	Adds an AtomBase/AtomMixed at a relative position of the unit cell.
add_multiple_atoms(atom, position, Nb)	Adds multiple AtomBase/AtomMixed at a relative position of the unit cell.
get_atom_ids()	Provides a list of atom ids within the unit cell.
get_atom_positions(*args)	Calculates the relative positions of the atoms in the unit cell

calc_mf_exchange_coupling()

Calculate the mean-field exchange coupling constant

$$J = \frac{3}{S_{eff}+1} k_B T_C$$

calc_spring_const()

Calculates the spring constant of the layer from the mass per unit area, sound velocity and thickness

$$k = m \left(\frac{v}{c}\right)^2$$

check_input(*inputs*)

Checks the input and create a list of function handle strings with T as argument. Inputs can be strings, floats, ints, or pint quantaties.

Parameters

inputs (*list[str, int, float, Quantity]*) – list of strings, int, floats, or Pint quantities.

Returns

(tuple) –

- *output (list[@lambda])* - list of lambda functions.
- *output_strs (list[str])* - list of string-representations.

get_acoustic_impedance()

Calculates the acoustic impedance.

Returns

Z (float) – acoustic impedance.

get_property_dict(***kwargs*)

Returns a dictionary with all parameters. objects or dicts and objects are converted to strings. if a type is given, only these properties are returned.

Parameters
****kwargs** (*list[str]*) – types of requested properties.

Returns
R (dict) – dictionary with requested properties.

set_ho_spring_constants(*HO*)

Set the higher orders of the spring constant for anharmonic phonon simulations.

Parameters
HO (*ndarray[float]*) – higher order spring constants.

set_opt_pen_depth_from_ref_index(*wavelength*)

Set the optical penetration depth from the optical referactive index for a given wavelength.

Parameters
wavelength (*Quantity*) – wavelength as Pint Quantitiy.

visualize(***kwargs*)

Allows for 3D presentation of unit cell by allow for a & b coordinate of atoms. Also add magnetization per atom.

Todo: use the avogadro project as plugin

Todo: create unit cell from CIF file e.g. by xrayutilities plugin.

Parameters
****kwargs** (*str*) – strain or magnetization for manipulating unit cell visualization.

add_atom(*atom*, *position*)

Adds an AtomBase/AtomMixed at a relative position of the unit cell.

Sort the list of atoms by the position at zero strain.

Update the mass, density and spring constant of the unit cell automatically:

$$\kappa = m \cdot (v_s/c)^2$$

Parameters

- **atom** (Atom, AtomMixed) – Atom or AtomMixed added to unit cell.
- **position** (*float*) – relative position within unit cel [0 .. 1].

add_multiple_atoms(*atom*, *position*, *Nb*)

Adds multiple AtomBase/AtomMixed at a relative position of the unit cell.

Parameters

- **atom** (Atom, AtomMixed) – Atom or AtomMixed added to unit cell.
- **position** (*float*) – relative position within unit cel [0 .. 1].
- **Nb** (*int*) – repetition of atoms.

`get_atom_ids()`

Provides a list of atom ids within the unit cell.

Returns
ids (list[str]) – list of atom ids within unit cell

`get_atom_positions(*args)`

Calculates the relative positions of the atoms in the unit cell

Returns
res (ndarray[float]) – relative postion of the atoms within the unit cell.

5.4.3 structures.structure

Classes:

Structure(name)	Structure representation which holds various sub_structures.

`class udkm1Dsim.structures.structure.Structure(name)`

Bases: `object`

Structure representation which holds various sub_structures.

Each sub_structure can be either a layer of N UnitCell or AmorphousLayer instances or a structure by itself. It is possible to recursively build up 1D structures.

Parameters
name (*str*) – name of the sample.

Attributes

- **name** (*str*) – name of sample.
- **sub_structures** (*list[AmorphousLayer, UnitCell, Structure]*) – list of structures in sample.
- **substrate** (*Structure*) – structure of the substrate.
- **num_sub_systems** (*int*) – number of subsystems for heat and phonons (electronic, lattice, spins, ...).

Methods:

visualize([unit, fig_size, cmap, linewidth, ...])	Simple visualization of the structure.
get_hash(**kwargs)	Create an unique hash from all layer IDs in the correct order in the structure as well as the corresponding material properties which are given by the *kwargs*.
add_sub_structure(sub_structure[, N])	Add a sub_structure of N layers or sub-structures to the structure.
add_substrate(sub_structure)	Add a structure as static substrate to the structure.
get_number_of_sub_structures()	This methods does not return the number of all layers in the structure, see *get_number_of_layers()*.
get_number_of_layers()	Determines the number of all layers in the structure.
get_number_of_unique_layers()	Determines the number of unique layers in the structure.
get_thickness([units])	Determines the thickness of the structure.
get_unique_layers()	The uniqueness is determined by the handle of each layer instance.
get_layer_vectors(*args)	Returns three lists with the numeric index of all layers in a structure given by the get_unique_layers() method and additionally vectors with the ids and Handles of the corresponding layer instances.
get_all_positions_per_unique_layer()	Determines the position indices for each unique layer in the structure.
get_distances_of_layers([units])	Returns a vector of the distance from the surface for each layer starting at 0 (dStart) and starting at the end of the first layer (dEnd) and from the center of each layer (dMid).
get_distances_of_interfaces([units])	Calculates the distances of the interafaces of the structure.
interp_distance_at_interfaces(N[, units])	Interpolates the distances at the layer interfaces by an odd number N.
get_layer_property_vector(property_name)	Returns a vector for a property of all layers in the structure.
get_layer_handle(i)	Returns the handle to a layer at a given position index.
reverse()	Returns a reversed structure also reversing all nested sub_structure.
reverse_sub_structures(structure)	Reverse a *Structure* and recursively call itself if a sub_structure is a *Structure* itself.

visualize(*unit='nm'*, *fig_size=[20, 1]*, *cmap='Set1'*, *linewidth=0.1*, *show=True*)

Simple visualization of the structure.

Parameters

- **unit** (*str*) – SI unit of the distance of the Structure. Defaults to 'nm'.
- **fig_size** (*list[float]*) – figure size of the visualization plot. Defaults to [20, 1].
- **cmap** (*str*) – Matplotlib colormap for colors of layers.
- **linewidth** (*float*) – line width of the patches.
- **show** (*boolean*) – show visualization plot at the end.

get_hash(**kwargs*)

Create an unique hash from all layer IDs in the correct order in the structure as well as the corresponding material properties which are given by the *kwargs*.

Parameters
****kwargs** (*list[str]*) – types of requested properties..

Returns
hash (str) – unique hash.

add_sub_structure(*sub_structure*, *N=1*)

Add a sub_structure of N layers or sub-structures to the structure.

Parameters

- **sub_structure** (AmorphousLayer, UnitCell, Structure) – amorphous layer, unit cell, or structure to add as sub-structure.
- **N** (*int*) – number or repetitions.

add_substrate(*sub_structure*)

Add a structure as static substrate to the structure.

Parameters
sub_structure (Structure) – substrate structure.

get_number_of_sub_structures()

This methods does not return the number of all layers in the structure, see *get_number_of_layers()*.

Returns
N (int) – number of all sub structures.

get_number_of_layers()

Determines the number of all layers in the structure.

Returns
L (int) – number of all layers in the structure.

get_number_of_unique_layers()

Determines the number of unique layers in the structure.

Returns
N (int) – number of unique layers in the structure.

get_thickness(*units=True*)

Determines the thickness of the structure.

Parameters
units (*boolean, optional*) – whether units should be returned or not. Defaults to True.

Returns
thickness (float, Quantity) – the thickness from surface to bottom of the structure.

get_unique_layers()

The uniqueness is determined by the handle of each layer instance.

Returns

(tuple) –

- *layer_ids (list[str])* - ids of all unique layers instances in the structure.
- *layer_handles (list[AmorphousLayer, UnitCell, Structure])* - handles of all unique layers instances in the structure.

`get_layer_vectors`(**args*)

Returns three lists with the numeric index of all layers in a structure given by the get_unique_layers() method and additionally vectors with the ids and Handles of the corresponding layer instances. The list and order of the unique layers can be either handed as an input parameter or is requested at the beginning.

Parameters
`layers` (*Optional[list]*) – list of unique layers including ids and handles

Returns

(tuple) –

- *indices (list[int])* - numeric index of all layers in a structure.
- *layer_ids (list[str])* - ids of all unique layers instances in the structure.
- *layer_handles (list[AmorphousLayer, UnitCell, Structure])* - handles of all unique layers instances in the structure.

`get_all_positions_per_unique_layer`()

Determines the position indices for each unique layer in the structure.

Returns
pos (dict{ndarray[int]}) – position indices for each unique layer in the structure.

`get_distances_of_layers`(*units=True*)

Returns a vector of the distance from the surface for each layer starting at 0 (dStart) and starting at the end of the first layer (dEnd) and from the center of each layer (dMid).

Parameters
`units` (*boolean, optional*) – whether units should be returned or not. Defaults to True.

Returns

(tuple) –

- *d_start (ndarray[float, Quantity])* - distances from the surface of each layer starting at 0.
- *d_end (ndarray[float, Quantity])* - distances from the bottom of each layer.
- *d_mid (ndarray[float, Quantity])*: distance from the middle of each layer.

`get_distances_of_interfaces`(*units=True*)

Calculates the distances of the interafaces of the structure.

Parameters
`units` (*boolean, optional*) – whether units should be returned or not. Defaults to True.

Returns
res (ndarray[float, Quantity]) – distances from the surface of each interface of the structure.

`interp_distance_at_interfaces`(*N*, *units=True*)

Interpolates the distances at the layer interfaces by an odd number N.

Parameters

- `N` (*int*) – number of point of interpolation at interface
- `units` (*boolean, optional*) – whether units should be returned or not. Defaults to True.

Returns

(tuple) –

- *dist_interp (ndarray[float, Quantity])* - distance array of the middle of each layer interpolated by an odd number N at the interfaces
- *original_indicies (ndarray[int])* - indicies of the original distances in the interpolated array

get_layer_property_vector(*property_name*)

Returns a vector for a property of all layers in the structure. The property is determined by the propertyName and returns a scalar value or a function handle.

Parameters
property_name (*str*) – name of property to return as array

Returns
prop (ndarray[float, @lambda]) – array of a property for all layers in the structure.

get_layer_handle(*i*)

Returns the handle to a layer at a given position index.

Parameters
i (*int*) – index of the layer to return.

Returns
handle (AmorphousLayer, UnitCell) – handle to the layer at position *i* in the structure.

reverse()

Returns a reversed structure also reversing all nested sub_structure.

Returns
reversed (Structure) – reversed structure.

reverse_sub_structures(*structure*)

Reverse a *Structure* and recursively call itself if a sub_structure is a *Structure* itself.

Parameters
structure (Structure) – structure to be reversed.

Returns
structure (Structure) – reversed structure.

5.4.4 simulations.simulation

Classes:

Simulation(S, force_recalc, **kwargs)	Base class for all simulations.

class udkm1Dsim.simulations.simulation.**Simulation**(*S*, *force_recalc*, ***kwargs*)

Bases: object

Base class for all simulations.

Handles the caching and some displaying option.

Parameters

- **S** (Structure) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Methods:

disp_message(message)	Wrapper to display messages for that class.
save(full_filename, data, *args)	Save data to file.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns
y_conv (ndarray[float]) – convoluted data.

5.4.5 simulations.heat

Classes:

Heat(S, force_recalc, **kwargs)	Heat simulations including laser excitation and N-temperature model.

class udkm1Dsim.simulations.heat.**Heat**(*S*, *force_recalc*, ***kwargs*)

Bases: *Simulation*

Heat simulations including laser excitation and N-temperature model.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **heat_diffusion** (*boolean*) – true when including heat diffusion in the calculations.
- **intp_at_interface** (*int*) – number of additional spacial points at the interface of each layer.
- **backend** (*str*) – pde solver backend - either default scipy or matlab.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **heat_diffusion** (*boolean*) – true when including heat diffusion in the calculations.
- **intp_at_interface** (*int*) – number of additional spacial points at the interface of each layer.
- **backend** (*str*) – pde solver backend - either default scipy or matlab.
- **excitation** (*dict{ndarray[float, Quantity]}*) – excitation parameters fluence, delay_pump, pulse_width, wavelength, theta, polarization, multilayer_absorption, backside
- **boundary_conditions** (*dict{str, float, Quantity}*) – boundary conditions of the top and bottom boundary for the heat diffusion calculation. `top_type` or `bottom_type` must be one of `boundary_types`. For the last two types the corresponding value, `top_value` and `bottom_value` have to be set as $K \times 1$ array, where K is the number of sub-systems.
- **boundary_types** (*list[str]*) – description of boundary types.
  - isolator

  - temperature
  - flux
- **distances** (*ndarray[float, Quantity]*) – spatial grid for heat diffusion [m]
- **ode_options** (*dict*) – options for scipy solve_ivp ode solver
- **ode_options_matlab** (*dict*) – dict with options for the MATLAB pdepe solver.
- **matlab_engine** (*module*) – MATLAB to Python API engine required for calculating heat diffusion.

Methods:

get_hash(delays, init_temp, **kwargs)	Returns a unique hash given by the `delays` and `init_temp` as well as the sample structure hash for relevant thermal parameters.
check_initial_temperature(init_temp[, distances])	Initial temperature for heat simulations can be either a scalar temperature which is assumed to be valid for all layers in the structure or a temperature profile is given with one temperature for each layer in the structure and for each subsystem.
check_excitation(delays)	The optical excitation is a dictionary with fluence F [J/m^2], delays t [s] of the pump events, and pulse width τ [s].
get_absorption_profile([distances, backside])	Returns the differential absorption profile dA/dz.
get_Lambert_Beer_absorption_profile([...])	The transmission is given by:
get_multilayers_absorption_profile([...])	Calculates the intensity, differential absorption and temperature increase profiles in each layer of a multilayers structure for p -polarized light.
get_temperature_after_delta_excitation(...)	Calculate the final temperature and temperature change for each layer of the sample structure after an optical excitation with a fluence F [J/m^2] and an initial temperature T_1 [K]:
get_temp_map(delays, init_temp)	Returns a temperature profile for the sample structure after optical excitation.
calc_temp_map(delays, input_init_temp)	Calculates the temperature profile for the sample structure after optical excitation.
calc_heat_diffusion(init_temp, distances, ...)	Calculates a temperature profile including heat diffusion for a given delay array and initial temperature profile.
odefunc(t, u, N, K, d_x_grid, x, ...)	Ordinary differential equation that is solved for 1D heat diffusion.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
save(full_filename, data, *args)	Save data to file.

get_hash(*delays*, *init_temp*, ***kwargs*)

Returns a unique hash given by the `delays` and `init_temp` as well as the sample structure hash for relevant thermal parameters.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.

- **init_temp** (*ndarray[float]*) – initial spatial temperature profile.
- ****kwargs** (*float, optional*) – optional parameters.

Returns
hash (str) – unique hash.

check_initial_temperature(*init_temp*, *distances=[]*)

Initial temperature for heat simulations can be either a scalar temperature which is assumed to be valid for all layers in the structure or a temperature profile is given with one temperature for each layer in the structure and for each subsystem. Alternatively a spatial grid can be provided.

Parameters

- **init_temp** (*float, Quantity, ndarray[float, Quantity]*) – initial temperature scalar or array [K].
- **distances** (*ndarray[float, Quantity], optional*) – spatial grid of the initial temperatrue.

Returns
init_temp (ndarray[float]) – checked initial temperature as array on the according spatial grid.

check_excitation(*delays*)

The optical excitation is a dictionary with fluence F [J/m^2], delays t [s] of the pump events, and pulse width τ [s]. N is the number of pump events.

Traverse excitation vector to update the `delay_pump` t_p vector for finite pulse durations $w(i)$ as follows

$$[t_p(i) - \text{window} \cdot w(i)] : [t_p(i) + \text{window} \cdot w(i)] : [w(i)/\text{intp}]$$

and to combine excitations which have overlapping intervals.

Moreover the incidence angle ϑ is taken into account for the user-defined incidence fluence in order to project the laser footprint onto the sample surface.

Parameters
delays (*ndarray[Quantity]*) – delays range of simulation [s].

Returns

(tuple) –

- *res (list[ndarray[float]])* - resulting list of excitations with interpolated delay density around excitations.
- *fluence (ndarray[float])* - projected excitation fluences.
- *delay_pump (ndarray[float])* - delays of the excitations.
- *pulse_width (ndarray[float])* - pulse widths of the excitations.

get_absorption_profile(*distances=[]*, *backside=False*)

Returns the differential absorption profile $\mathrm{d}A/\mathrm{d}z$.

Parameters

- **distances** (*ndarray[float], optional*) – spatial grid for calculation.
- **backside** (*boolean, optional*) – backside or frontside excitation.

Returns
dAdz (ndarray[float]) – differential absorption within each layer calculated either by Lambert-Beers law or by a multilayer absorption formalism.

get_Lambert_Beer_absorption_profile(*distances=[]*, *backside=False*)

The transmission is given by:

$$\tau = \frac{I}{I_0} = \exp(-z/\zeta)$$

and the absorption by:

$$A = 1 - \tau = 1 - \exp(-z/\zeta)$$

The absorption profile can be derived from the spatial derivative:

$$\frac{\mathrm{d}A(z)}{\mathrm{d}z} = \frac{1}{\zeta}\exp(-z/\zeta)$$

Parameters

- **distances** (*ndarray[float], optional*) – spatial grid for calculation.
- **backside** (*boolean, optional*) – backside or frontside excitation.

Returns

dAdz (ndarray[float]) – differential absorption within each layer calculated by Lambert-Beers law.

get_multilayers_absorption_profile(*distances=[]*, *backside=False*)

Calculates the intensity, differential absorption and temperature increase profiles in each layer of a multilayers structure for p -polarized light.

Calculation based on the method in Ref[5] and code developed Matlab by L. Le Guyader, see Ref[6].

Copyright (2012-2014) Loïc Le Guyader <loic.le_guyader@helmholtz-berlin.de>

Parameters

- **distances** (*ndarray[float], optional*) – spatial grid for calculation.
- **backside** (*boolean, optional*) – backside or frontside excitation.

Returns

(tuple) –

- *dAdz (ndarray[float])* - differential absorption within each layer.
- *Ints (ndarray[float])* - intensity profiles within each layer.
- *R_total (float)* - total amount of reflection from the multilayer.
- *T_total (float)* - total transmission in the last layer of the multilayer.

References:

get_temperature_after_delta_excitation(*fluence*, *init_temp*, *distances=[]*)

Calculate the final temperature and temperature change for each layer of the sample structure after an optical excitation with a fluence F [J/m^2] and an initial temperature T_1 [K]:

$$\Delta E = \int_{T_1}^{T_2} m\, c(T)\, \mathrm{d}T$$

[5] K. Ohta & H. Ishida, *Matrix formalism for calculation of the light beam intensity in stratified multilayered films, and its use in the analysis of emission spectra*, Appl. Opt. 29, 2466 (1990).

[6] L. Le Guyader, A. Kleibert, F. Nolting, L. Joly, P.M. Derlet, R.V. Pisarev, A. Kirilyuk, Th. Rasing & A.V. Kimel, *Dynamics of laser-induced spin reorientation in Co/SmFeO_3 heterostructure*, Phys. Rev. B 87, 054437 (2013).

where ΔE is the absorbed energy in each layer and $c(T)$ is the temperature-dependent heat capacity [J/kg K] and m is the mass [kg].

The absorbed energy per layer can be linearized from the absorption profile $\mathrm{d}\alpha/\mathrm{d}z$ as

$$\Delta E = \frac{\mathrm{d}\alpha}{\mathrm{d}z} E_0 \Delta z$$

where E_0 is the initial energy impinging on the first layer given by the fluence $F = E/A$. Δz is equal to the thickness of each layer.

Finally, one has to minimize the following modulus to obtain the final temperature T_2 of each layer:

$$\left| \int_{T_1}^{T_2} mc(T)\,\mathrm{d}T - \frac{\mathrm{d}\alpha}{\mathrm{d}z} E_0 \Delta z \right| \stackrel{!}{=} 0$$

Parameters

- **fluence** (*float, Quantity*) – incident fluence [J/m^2].
- **init_temp** (*float, Quantity, ndarray[float, Quantity]*) – initial temperature scalar or array [K].
- **distances** (*ndarray[float], optional*) – spatial grid for calculation.

Returns

(tuple) –

- *final_temp (ndarray[float])* - final temperature after delta excitation.
- *delta_T (ndarray[float])* - temperature change.

get_temp_map(*delays*, *init_temp*)

Returns a temperature profile for the sample structure after optical excitation. The result can be saved using an unique hash of the sample and the simulation parameters in order to reuse it.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **init_temp** (*float, Quantity, ndarray[float, Quantity]*) – initial temperature scalar or array [K].

Returns

(tuple) –

- *temp_map (ndarray[float])* - spatio-temporal temperature map.
- *delta_temp_map (ndarray[float])* - spatio-temporal temperature change map.

calc_temp_map(*delays*, *input_init_temp*)

Calculates the temperature profile for the sample structure after optical excitation. Heat diffusion can be included if `heat_diffusion = true`.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **input_init_temp** (*float, Quantity, ndarray[float, Quantity]*) – initial temperature scalar or array [K].

Returns

(tuple) –

- *temp_map (ndarray[float])* - spatio-temporal temperature map.
- *delta_temp_map (ndarray[float])* - spatio-temporal temperature change map.
- *checked_excitations (list[ndarray[float]])* - resulting list of checked excitations.

calc_heat_diffusion(*init_temp*, *distances*, *delays*, *delay_pump*, *pulse_width*, *fluence*)

Calculates a temperature profile including heat diffusion for a given delay array and initial temperature profile. Here we have to solve the 1D heat diffusion equation for N subsystems:

$$
\begin{aligned}
c_1(T_1)\rho\frac{\partial T_1}{\partial t} = \\
\frac{\partial}{\partial z}\left(k_1(T_1)\frac{\partial T_1}{\partial z}\right) + G_1(T_1, ..., T_N) + S(z,t) \\
\vdots \\
c_N(T_N)\rho\frac{\partial T_N}{\partial t} = \\
\frac{\partial}{\partial z}\left(k_N(T_N)\frac{\partial T_N}{\partial z}\right) + G_N(T_1, ..., T_N)
\end{aligned}
$$

where T_i is the temperature [K], z the distance [m], t the delay [s], $c_i(T)$ the temperature dependent heat capacity [J/kg K], ρ the density [kg/m^3] and $k_i(T)$ is the temperature-dependent thermal conductivity [W/m K] and $S(z,t)$ is a source term [W/m^3]. The energy flow between the subsystems is given by the `sub_system_coupling` parameter $G_i(T_1, ..., T_N)$ of the individual layers. The index i refers to the i-th subsystem.

The 1D heat diffusion equation can be either solved with SciPy or Matlab as backend.

Parameters

- **init_temp** (*float, Quantity, ndarray[float, Quantity]*) – initial temperature scalar or array [K].
- **distances** (*ndarray[float]*) – spatial grid for calculation.
- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **delay_pump** (*ndarray[float]*) – delays of the excitations.
- **pulse_width** (*ndarray[float]*) – pulse widths of the excitations.
- **fluence** (*ndarray[float]*) – excitation fluences.

Returns

temp_map (ndarray[float]) – spatio-temporal temperature map.

static odefunc(*t*, *u*, *N*, *K*, *d_x_grid*, *x*, *thermal_conds*, *heat_capacities*, *sub_system_coupling*, *densities*, *indices*, *dAdz*, *fluence*, *delay_pump*, *pulse_length*, *bc_top_type*, *bc_top_value*, *bc_bottom_type*, *bc_bottom_value*, *pbar*, *state*)

Ordinary differential equation that is solved for 1D heat diffusion.

Parameters

- **t** (*ndarray[float]*) – internal time steps of the ode solver.
- **u** (*ndarray[float]*) – internal variable of the ode solver.
- **N** (*int*) – number of spatial grid points.
- **K** (*int*) – number of sub-systems.

- **d_x_grid** (*ndarray[float]*) – derivative of spatial grid.
- **x** (*ndarray[float]*) – start point of actual layers.
- **thermal_conds** (*ndarray[@lambda]*) – T-dependent thermal conductivity function handles.
- **heat_capacities** (*ndarray[@lambda]*) – T-dependent heat capacity function handles.
- **sub_system_coupling** (*ndarray[@lambda]*) – T-dependent sub-system coupling.
- **densities** (*ndarray[float]*) – density of layers.
- **indices** (*ndarray[int]*) – indices of actual layers in respect to interpolated spatial grid.
- **dAdz** (*ndarray[float]*) – differential absorption profile.
- **fluence** (*ndarray[float]*) – excitation fluences.
- **delay_pump** (*ndarray[float]*) – delay of excitations.
- **pulse_length** (*ndarray[float]*) – pulse widths of excitations.
- **bc_top_type** (*int*) – top boundary type.
- **bc_top_value** (*ndarray[float]*) – top boundary value.
- **bc_bottom_type** (*int*) – bottom boundary type.
- **bc_bottom_value** (*ndarray[float]*) – bottom boundary value.
- **pbar** (*tqdm*) – tqdm progressbar.
- **state** (*list[float]*) – state variables for progress bar.

Returns
dudt (ndarray[float]) – temporal derivative of internal variable.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns
y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

5.4.6 simulations.phonons

Classes:

Phonon(S, force_recalc, **kwargs)	Base class for phonon simulations in a linear chain of masses and springs.
PhononNum(S, force_recalc, **kwargs)	Numerical model to simulate coherent acoustic phonons.
PhononAna(S, force_recalc, **kwargs)	Analytical model to simulate coherent acoustic phonons.

class udkm1Dsim.simulations.phonons.**Phonon**(*S*, *force_recalc*, ***kwargs*)

Bases: *Simulation*

Base class for phonon simulations in a linear chain of masses and springs.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **only_heat** (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **only_heat** (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.

Methods:

get_hash(delays, temp_map, delta_temp_map, ...)	Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.
get_all_strains_per_unique_layer(strain_map	Determines all values of the strain per unique layer.
get_reduced_strains_per_unique_layer(strain_	Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number.
check_temp_maps(temp_map, delta_temp_map, delays)	Check temperature profiles for correct dimensions.
calc_sticks_from_temp_map(temp_map, ...)	Calculates the sticks to insert into the layer springs which model the external force (thermal stress).
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
save(full_filename, data, *args)	Save data to file.

get_hash(*delays*, *temp_map*, *delta_temp_map*, ***kwargs*)

Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- ****kwargs** (*float, optional*) – optional parameters.

Returns

hash (str) – unique hash.

get_all_strains_per_unique_layer(*strain_map*)

Determines all values of the strain per unique layer.

Parameters

strain_map (*ndarray[float]*) – spatio-temporal strain profile.

Returns

strains (list[ndarray[float]] – all strains per unique layer.

get_reduced_strains_per_unique_layer(*strain_map*, *N=100*)

Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number. The reduction is done by equally spacing the strains between the `min` and `max` strain with a given number N, which can be also an array of the $len(N) = L$, where L is the number of unique layers.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **N** (*int, optional*) – number of reduced strains. Defaults to 100.

Returns

strains (list[ndarray[float]] – reduced strains per unique layer.

check_temp_maps(*temp_map*, *delta_temp_map*, *delays*)

Check temperature profiles for correct dimensions.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- **delays** (*ndarray[float]*) – delay grid for the simulation.

Returns

(tuple) –

- *temp_map (ndarray[float])* - checked spatio-temporal temperature profile.
- *delta_temp_map (ndarray[float])* - checked spatio-temporal differential temperature profile.

calc_sticks_from_temp_map(*temp_map*, *delta_temp_map*)

Calculates the sticks to insert into the layer springs which model the external force (thermal stress). The length of l_i of the i-th spacer stick is calculated from the temperature-dependent linear thermal expansion $\alpha(T)$ of the layer:

$$\alpha(T) = \frac{1}{L}\frac{dL}{dT}$$

which results after integration in

$$l = \Delta L = L_1 \exp(A(T_2) - A(T_1)) - L_1$$

where $A(T)$ is the integrated lin. therm. expansion coefficient in respect to the temperature T. The indices 1 and 2 indicate the initial and final state.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.

Returns

(tuple) –

- *sticks (ndarray[float])* - summed spacer sticks.
- *sticks_sub_systems (ndarray[float])* - spacer sticks per sub-system.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters

message (*str*) – message to display.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

class udkm1Dsim.simulations.phonons.**PhononNum**(*S*, *force_recalc*, ***kwargs*)

Bases: *Phonon*

Numerical model to simulate coherent acoustic phonons.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **only_heat** (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **only_heat** (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.
- **ode_options** (*dict*) – options for scipy solve_ivp ode solver.

References

Methods:

get_strain_map(delays, temp_map, delta_temp_map)	Returns a strain profile for the sample structure for given temperature profile.
calc_strain_map(delays, temp_map, delta_temp_map)	Calculates the `strain_map` of the sample structure for a given `temp_map` and `delta_temp_map` and `delay` array.
ode_func(t, X, delays, force_from_heat, ...)	Provides the according ode function for the ode solver which has to be solved.
calc_force_from_spring(d_X1, d_X2, spring_consts)	Calculates the force F_i^{spring} acting on each mass due to the displacement between the left and right site of that mass.
calc_force_from_heat(sticks, spring_consts)	Calculates the force acting on each mass due to the heat expansion, which is modeled by spacer sticks.
calc_force_from_damping(v, damping, masses)	Calculates the force acting on each mass in a linear spring due to damping (γ_i) according to the shift velocity difference $v_i - v_{i-1}$ with $v_i(t) = \dot{x}_i(t)$:
calc_sticks_from_temp_map(temp_map, ...)	Calculates the sticks to insert into the layer springs which model the external force (thermal stress).
check_temp_maps(temp_map, delta_temp_map, delays)	Check temperature profiles for correct dimensions.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_all_strains_per_unique_layer(strain_map	Determines all values of the strain per unique layer.
get_hash(delays, temp_map, delta_temp_map, ...)	Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.
get_reduced_strains_per_unique_layer(strain_	Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number.
save(full_filename, data, *args)	Save data to file.

get_strain_map(*delays*, *temp_map*, *delta_temp_map*)

Returns a strain profile for the sample structure for given temperature profile. The result can be saved using an unique hash of the sample and the simulation parameters in order to reuse it.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal differential
- **map.** (*temperature*) –

Returns

strain_map (ndarray[float]) – spatio-temporal strain profile.

calc_strain_map(*delays*, *temp_map*, *delta_temp_map*)

Calculates the `strain_map` of the sample structure for a given `temp_map` and `delta_temp_map` and `delay` array. Further details are given in Ref.[7]. The coupled differential equations are solved for each oscillator in a linear chain of masses and springs:

$$m_i\ddot{x}_i = -k_i(x_i - x_{i-1}) - k_{i+1}(x_i - x_{i+1}) + m_i\gamma_i(\dot{x}_i - \dot{x}_{i-1}) + F_i^{heat}(t)$$

[7] A. Bojahr, M. Herzog, D. Schick, I. Vrejoiu, & M. Bargheer, *Calibrated real-time detection of nonlinearly propagating strain waves*, Phys. Rev. B, 86(14), 144306 (2012).

where $x_i(t) = z_i(t) - z_i^0$ is the shift of each layer. m_i is the mass and $k_i = m_i\, v_i^2/c_i^2$ is the spring constant of each layer. Furthermore an empirical damping term $F_i^{damp} = \gamma_i(\dot{x}_i - \dot{x}_{i-1})$ is introduced and the external force (thermal stress) $F_i^{heat}(t)$. The thermal stresses are modelled as spacer sticks which are calculated from the linear thermal expansion coefficients. The equation of motion can be reformulated as:

$$m_i\ddot{x}_i = F_i^{spring} + F_i^{damp} + F_i^{heat}(t)$$

The numerical solution also allows for non-harmonic inter-atomic potentials of up to the order M. Accordingly $k_i = (k_i^1 \dots k_i^{M-1})$ can be an array accounting for higher orders of the potential which is in the harmonic case purely quadratic ($k_i = k_i^1$). The resulting force from the displacement of the springs

$$F_i^{spring} = -k_i(x_i - x_{i-1}) - k_{i+1}(x_i - x_{i+1})$$

includes:

$$k_i(x_i - x_{i-1}) = \sum_{j=1}^{M-1} k_i^j (x_i - x_{i-1})^j$$

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal differential temperature map.

Returns

(tuple) –

- *strain_map (ndarray[float])* - spatio-temporal strain profile.
- *sticks_sub_systems (ndarray[float])* - spacer sticks per sub-system.
- *velocities (ndarray[float])* - spatio-temporal velocity profile.

static ode_func(*t*, *X*, *delays*, *force_from_heat*, *damping*, *spring_consts*, *masses*, *L*, *pbar=None*, *state=None*)

Provides the according ode function for the ode solver which has to be solved. The ode function has the input t and $X(t)$ and calculates the temporal derivative $\dot{X}(t)$ where the vector

$$X(t) = [x(t)\ \dot{x}(t)] \quad \text{and} \quad \dot{X}(t) = [\dot{x}(t)\ \ddot{x}(t)].$$

$x(t)$ is the actual shift of each layer.

Parameters

- **t** (*ndarray[float]*) – internal time steps of the ode solver.
- **X** (*ndarray[float]*) – internal variable of the ode solver.
- **delays** (*ndarray[float]*) – delays range of simulation [s].
- **force_from_heat** (*ndarray[float]*) – force due to thermal expansion.
- **damping** (*ndarray[float]*) – phonon damping.
- **spring_consts** (*ndarray[float]*) – spring constants of masses.
- **masses** (*ndarray[float]*) – masses of layers.
- **L** (*int*) – number of layers.
- **pbar** (*tqdm*) – tqdm progressbar.

- **state** (*list[float]*) – state variables for progress bar.

Returns
X_prime (ndarray[float]) – velocities and accelerations of masses.

static calc_force_from_spring(*d_X1*, *d_X2*, *spring_consts*)

Calculates the force F_i^{spring} acting on each mass due to the displacement between the left and right site of that mass.

$$F_i^{spring} = -k_i(x_i - x_{i-1}) - k_{i+1}(x_i - x_{i+1})$$

We introduce-higher order inter-atomic potentials by

$$k_i(x_i - x_{i-1}) = \sum_{j=1}^{M} k_i^j (x_i - x_{i-1})^j$$

where M is the order of the spring constants.

Parameters

- **d_X1** (*ndarray[float]*) – left displacements.
- **d_X2** (*ndarray[float]*) – right displacements.
- **spring_consts** (*ndarray[float]*) – spring constants of masses.

Returns
F (ndarray[float]) – force from springs.

static calc_force_from_heat(*sticks*, *spring_consts*)

Calculates the force acting on each mass due to the heat expansion, which is modeled by spacer sticks.

Parameters

- **sticks** (*ndarray[float]*) – spacer sticks.
- **spring_consts** (*ndarray[float]*) – spring constants of masses.

Returns
F (ndarray[float]) – force from thermal expansion.

static calc_force_from_damping(*v*, *damping*, *masses*)

Calculates the force acting on each mass in a linear spring due to damping (γ_i) according to the shift velocity difference $v_i - v_{i-1}$ with $v_i(t) = \dot{x}_i(t)$:

$$F_i^{damp} = \gamma_i(\dot{x}_i - \dot{x}_{i-1})$$

Parameters

- **v** (*ndarray[float]*) – velocity of masses.
- **damping** (*ndarray[float]*) – phonon damping.
- **masses** (*ndarray[float]*) – masses.

Returns
F (ndarray[float]) – force from damping.

calc_sticks_from_temp_map(*temp_map*, *delta_temp_map*)

Calculates the sticks to insert into the layer springs which model the external force (thermal stress). The length of l_i of the i-th spacer stick is calculated from the temperature-dependent linear thermal expansion $\alpha(T)$ of the layer:

$$\alpha(T) = \frac{1}{L}\frac{dL}{dT}$$

which results after integration in

$$l = \Delta L = L_1 \exp(A(T_2) - A(T_1)) - L_1$$

where $A(T)$ is the integrated lin. therm. expansion coefficient in respect to the temperature T. The indices 1 and 2 indicate the initial and final state.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.

Returns

(tuple) –

- *sticks (ndarray[float])* - summed spacer sticks.
- *sticks_sub_systems (ndarray[float])* - spacer sticks per sub-system.

check_temp_maps(*temp_map*, *delta_temp_map*, *delays*)

Check temperature profiles for correct dimensions.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- **delays** (*ndarray[float]*) – delay grid for the simulation.

Returns

(tuple) –

- *temp_map (ndarray[float])* - checked spatio-temporal temperature profile.
- *delta_temp_map (ndarray[float])* - checked spatio-temporal differential temperature profile.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

get_all_strains_per_unique_layer(*strain_map*)

Determines all values of the strain per unique layer.

Parameters
strain_map (*ndarray[float]*) – spatio-temporal strain profile.

Returns
strains (list[ndarray[float]] – all strains per unique layer.

get_hash(*delays*, *temp_map*, *delta_temp_map*, ***kwargs*)

Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- ****kwargs** (*float, optional*) – optional parameters.

Returns
hash (str) – unique hash.

get_reduced_strains_per_unique_layer(*strain_map*, *N=100*)

Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number. The reduction is done by equally spacing the strains between the `min` and `max` strain with a given number N, which can be also an array of the $len(N) = L$, where L is the number of unique layers.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **N** (*int, optional*) – number of reduced strains. Defaults to 100.

Returns
strains (list[ndarray[float]] – reduced strains per unique layer.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

class udkm1Dsim.simulations.phonons.**PhononAna**(*S*, *force_recalc*, ***kwargs*)

Bases: *Phonon*

Analytical model to simulate coherent acoustic phonons.

Parameters

- **S** (Structure) – sample to do simulations with.

- `force_recalc` (*boolean*) – force recalculation of results.

Keyword Arguments

- `save_data` (*boolean*) – true to save simulation results.
- `cache_dir` (*str*) – path to cached data.
- `disp_messages` (*boolean*) – true to display messages from within the simulations.
- `progress_bar` (*boolean*) – enable tqdm progress bar.
- `only_heat` (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **only_heat** (*boolean*) – true when including only thermal expansion without coherent phonon dynamics.

References

Methods:

get_strain_map(delays, temp_map, delta_temp_map)	Returns a strain profile for the sample structure for given temperature profile.
calc_strain_map(delays, temp_map, delta_temp_map)	Calculates the `strain_map` of the sample structure for a given `temp_map` and `delta_temp_map` and `delay` array.
solve_eigenproblem()	Creates the real and symmetric K matrix ($L \times L$) of spring constants k_i and masses m_i and calculates the eigenvectors Ξ_j and eigenfrequencies ω_j for the matrix which are used to calculate the `strain_map` of the structure.
get_energy_per_eigenmode(A, B)	Returns the sorted energy per Eigenmode of the coherent phonons of the 1D sample.
calc_sticks_from_temp_map(temp_map, ...)	Calculates the sticks to insert into the layer springs which model the external force (thermal stress).
check_temp_maps(temp_map, delta_temp_map, delays)	Check temperature profiles for correct dimensions.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_all_strains_per_unique_layer(strain_map	Determines all values of the strain per unique layer.
get_hash(delays, temp_map, delta_temp_map, ...)	Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.
get_reduced_strains_per_unique_layer(strain_	Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number.
save(full_filename, data, *args)	Save data to file.

get_strain_map(*delays*, *temp_map*, *delta_temp_map*)

Returns a strain profile for the sample structure for given temperature profile. The result can be saved using an unique hash of the sample and the simulation parameters in order to reuse it.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal differential
- **map.** (*temperature*) –

Returns

(tuple) –

- *strain_map (ndarray[float])* - spatio-temporal strain profile.
- *A (ndarray[float])* - coefficient vector A of general solution.
- *B (ndarray[float])* - coefficient vector B of general solution.

calc_strain_map(*delays*, *temp_map*, *delta_temp_map*)

Calculates the `strain_map` of the sample structure for a given `temp_map` and `delta_temp_map` and `delay` array. Further details are given in Ref.[8]. Within the linear chain of N masses (m_i) at position z_i coupled

[8] M. Herzog, D. Schick, P. Gaal, R. Shayduk, C. von Korff Schmising & M. Bargheer, *Analysis of ultrafast X-ray diffraction data in a linear-chain model of the lattice dynamics*, Applied Physics A, 106(3), 489-499 (2011).

with spring constants k_i one can formulate the differential equation of motion as follow:

$$m_i \ddot{x}_i = -k_i(x_i - x_{i-1}) - k_{i+1}(x_i - x_{i+1}) + F_i^{heat}(t)$$

Since we only consider nearest-neighbor interaction one can write:

$$\ddot{x}_i = \sum_{n=1}^{N} \kappa_{i,n} x_n = \Delta_i(t)$$

Here $x_i(t) = z_i(t) - z_i^0$ is the shift of each layer, $F_i^{heat}(t)$ is the external force (thermal stress) of each layer and $\kappa_{i,i} = -(k_i + k_{i+1})/m_i$, and $\kappa_{i,i+1} = \kappa_{i+1,i} = k_{i+1}/m_i$.

$k_i = m_i\, v_i^2/c_i^2$ is the spring constant and c_i and v_i are the thickness and longitudinal sound velocity of each layer respectively. One can rewrite the homogeneous differential equation in matrix form to obtain the general solution

$$\frac{d^2}{dt^2} X = K\, X$$

Here $X = (x_1 \dots x_N)$ and K is the tri-diagonal matrix of κ which is real and symmetric. The differential equation can be solved with the ansatz:

$$X(t) = \sum_j \Xi_j \left(A_j \cos(\omega_j\, t) + B_j \sin(\omega_j\, t)\right)$$

where $\Xi_j = (\xi_1^j \dots \xi_N^j)$ are the eigenvectors of the matrix K. Thus by solving the Eigenproblem for K one gets the eigenvecotrs Ξ_j and the eigenfrequencies ω_j. From the initial conditions

$$X(0) = \sum_j \Xi_j\, A_j = \Xi\, A \qquad V(0) = \dot{X}(0) = \sum_j \Xi_j\, \omega_j\, B_j = \Xi\, \omega\, B$$

one can determine the real coefficient vectors A and B in order to calculate $X(t)$ and $V(t)$ using the ansatz:

$$A = \Xi \setminus X(0) \qquad B = (\Xi \setminus V(0))/\omega$$

The external force is implemented as spacer sticks which are inserted into the springs and hence the layers have a new equilibrium positions $z_i(\infty) = z_i^\infty$. Thus we can do a coordination transformation:

$$z_i(t) = z_i^0 + x_i(t) = z_i^\infty + x_i^\infty(t)$$

and

$$x_i^\infty(t) = z_i^0 - z_i^\infty + x_i(t)$$

with the initial condition $x_i(0) = 0$ it becomes

$$x_i^\infty(0) = z_i^0 - z_i^\infty = \sum_{j=i+1}^{N} l_j$$

$x_i^\infty(0)$ is the new initial condition after the excitation where l_i is the length of the i-th spacer stick. The spacer sticks are calculated from the temperature change and the linear thermal expansion coefficients. The actual strain $\epsilon_i(t)$ of each layer is calculates as follows:

$$\epsilon_i(t) = [\Delta x_i(t) + l_i)]/c_i$$

with $\Delta x_i = x_i - x_{i-1}$. The stick l_i have to be added here, because x_i has been transformed into the new coordinate system x_i^∞.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal differential temperature map.

Returns

(tuple) –

- *strain_map (ndarray[float])* - spatio-temporal strain profile.
- *A (ndarray[float])* - coefficient vector A of general solution.
- *B (ndarray[float])* - coefficient vector B of general solution.

`solve_eigenproblem()`

Creates the real and symmetric K matrix ($L \times L$) of spring constants k_i and masses m_i and calculates the eigenvectors Ξ_j and eigenfrequencies ω_j for the matrix which are used to calculate the `strain_map` of the structure. If the result has been save to file, load it from there.

Returns

(tuple) –

- *Xi (ndarray[float])* - eigenvectors.
- *omega (ndarray[float])* - eigenfrequencies.

`get_energy_per_eigenmode(A, B)`

Returns the sorted energy per Eigenmode of the coherent phonons of the 1D sample.

$$E_j = \frac{1}{2}(A_j^2 + B_j^2)\,\omega_j^2\,m_j\,\|\Xi_j\|^2$$

Frequencies are in [Hz] and energy per mode in [J].

Parameters

- **A** (*ndarray[float]*) – coefficient vector A of general solution.
- **B** (*ndarray[float]*) – coefficient vector B of general solution.

Returns

(tuple) –

- *omega (ndarray[float])* - eigenfrequencies.
- *E (ndarray[float])* - energy per eigenmode.

`calc_sticks_from_temp_map(temp_map, delta_temp_map)`

Calculates the sticks to insert into the layer springs which model the external force (thermal stress). The length of l_i of the i-th spacer stick is calculated from the temperature-dependent linear thermal expansion $\alpha(T)$ of the layer:

$$\alpha(T) = \frac{1}{L}\frac{dL}{dT}$$

which results after integration in

$$l = \Delta L = L_1 \exp(A(T_2) - A(T_1)) - L_1$$

where $A(T)$ is the integrated lin. therm. expansion coefficient in respect to the temperature T. The indices 1 and 2 indicate the initial and final state.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.

Returns

(tuple) –

- *sticks (ndarray[float])* - summed spacer sticks.
- *sticks_sub_systems (ndarray[float])* - spacer sticks per sub-system.

check_temp_maps(*temp_map*, *delta_temp_map*, *delays*)

Check temperature profiles for correct dimensions.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- **delays** (*ndarray[float]*) – delay grid for the simulation.

Returns

(tuple) –

- *temp_map (ndarray[float])* - checked spatio-temporal temperature profile.
- *delta_temp_map (ndarray[float])* - checked spatio-temporal differential temperature profile.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters

message (*str*) – message to display.

get_all_strains_per_unique_layer(*strain_map*)

Determines all values of the strain per unique layer.

Parameters

strain_map (*ndarray[float]*) – spatio-temporal strain profile.

Returns

strains (list[ndarray[float]] – all strains per unique layer.

get_hash(*delays*, *temp_map*, *delta_temp_map*, ***kwargs*)

Calculates an unique hash given by the `delays`, and `temp_map` and `delta_temp_map` as well as the sample structure hash for relevant lattice parameters.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature profile.
- **delta_temp_map** (*ndarray[float]*) – spatio-temporal temperature difference profile.
- ****kwargs** (*float, optional*) – optional parameters.

Returns

hash (str) – unique hash.

get_reduced_strains_per_unique_layer(*strain_map*, *N=100*)

Calculates all strains per unique layer that are given by the input `strain_map`, but with a reduced number. The reduction is done by equally spacing the strains between the `min` and `max` strain with a given number N, which can be also an array of the $len(N) = L$, where L is the number of unique layers.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **N** (*int, optional*) – number of reduced strains. Defaults to 100.

Returns

strains (list[ndarray[float]] – reduced strains per unique layer.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

5.4.7 simulations.magnetization

Classes:

Magnetization(S, force_recalc, **kwargs)	Base class for all magnetization simulations.
LLB(S, force_recalc, **kwargs)	Mean-field quantum Landau-Lifshitz-Bloch simulations.

class udkm1Dsim.simulations.magnetization.**Magnetization**(*S*, *force_recalc*, ***kwargs*)

Bases: *Simulation*

Base class for all magnetization simulations.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **ode_options** (*dict*) – options for scipy solve_ivp ode solver

Methods:

get_hash(**kwargs)	Calculates an unique hash given by the delays as well as the sample structure hash for relevant magnetic parameters.
check_initial_magnetization(init_mag[, ...])	Check if a given initial magnetization profile is valid.
get_magnetization_map(delays, **kwargs)	Returns an absolute `magnetization_map` for the sample structure with the dimension $M \times N \times 3$ with M being the number of delays and N the number of layers in the structure or the length of the given spatial grid.
calc_magnetization_map(delays, **kwargs)	Calculates an absolute `magnetization_map` - see *get_magnetization_map()* for details.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
save(full_filename, data, *args)	Save data to file.

get_hash(***kwargs*)

Calculates an unique hash given by the delays as well as the sample structure hash for relevant magnetic parameters. Optionally, part of the `strain_map` and `temp_map` are used.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.
- ****kwargs** (*ndarray[float], optional*) – optional delays, strain and temperature profile as well as external magnetic field and initial magnetization.

Returns

hash (str) – unique hash.

check_initial_magnetization(*init_mag*, *distances=[]*)

Check if a given initial magnetization profile is valid. The profile must be either a vector *[amplitude, phi, gamma]* describing a global initial magnetization or an array of shape *[Nx3]* with N being the number of layers or the length of the specific spatial grid. If no initial magnetization is given, the initial profile is determined from the magnetization of the layers on creation. In addition, a spatial grid can be provided.

Parameters

- **init_mag** (*ndarray[float]*) – initial global or local magnetization profile.
- **distances** (*ndarray[float, Quantity], optional*) – spatial grid of the initial magnetization.

Returns

init_mag (ndarray[float]) –

checked initial magnetization as array on
the according spatial grid.

get_magnetization_map(*delays*, ***kwargs*)

Returns an absolute `magnetization_map` for the sample structure with the dimension $M \times N \times 3$ with M being the number of delays and N the number of layers in the structure or the length of the given spatial grid. Each element of the map contains the three magnetization components `[amplitude, phi, gamma]`. The angles `phi` and `gamma` must be returned in radians as pure numpy arrays. The `magnetization_map` can depend on the `temp_map` and `strain_map` that can be also calculated for the sample structure. More over an external magnetic field `H_ext` and initial magnetization profile `init_mag` can be provided.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- ****kwargs** (*ndarray[float], optional*) – optional strain and temperature profile as well external magnetic field in [T] and initial magnetization.

Returns

magnetization_map (ndarray[float]) –

spatio-temporal absolute
magnetization profile.

calc_magnetization_map(*delays*, ***kwargs*)

Calculates an absolute `magnetization_map` - see *get_magnetization_map()* for details.

This method is just an interface and should be overwritten for the actual simulations.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- ****kwargs** (*ndarray[float], optional*) – optional strain and temperature profile as well external magnetic field and initial magnetization.

Returns

magnetization_map (ndarray[float]) –

spatio-temporal absolute
magnetization profile.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns
y_conv (ndarray[float]) – convoluted data.

`disp_message`(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

`save`(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

`class udkm1Dsim.simulations.magnetization.LLB`(*S*, *force_recalc*, ***kwargs*)

Bases: *Magnetization*

Mean-field quantum Landau-Lifshitz-Bloch simulations.

Please find a detailed review on the Landau-Lifshitz-Bloch equation by Unai Atxitia et al.[11].

In collaboration with Theodor Griepe (@Nilodirf) from the group of Unai Atxitia at Instituto de Ciencia de Materiales de Madrid (ICMM-CSIC).

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

References

Methods:

[11] U. Atxitia, D. Hinzke, and U. Nowak, *Fundamentals and Applications of the Landau-Lifshitz-Bloch Equation*, J. Phys. D. Appl. Phys. 50, (2017).

calc_magnetization_map(delays, temp_map[, ...])	Calculates the magnetization map using the mean-field quantum Landau-Lifshitz-Bloch equation (LLB) for a given delay range and according temperature map:
get_mean_field_mag_map(temp_map)	Returns the mean-field magnetization map see *calc_mean_field_mag_map()* for details.
calc_mean_field_mag_map(temp_map)	Calculate the mean-field magnetization map m_{eq} by solving the self consistent equation
get_directional_exchange_stiffnesses()	Returns the directional exchange stiffnesses with size $N \times 2$, with N being the number of layers, between the $(i-1)^{\mathrm{th}}$ and i^{th} layers as well as between the i^{th} and $(i+1)^{\mathrm{th}}$ layers in the structure.
odefunc(t, m, delays, N, H_ext, temp_map, ...)	Ordinary differential equation that is solved for 1D LLB.
calc_uniaxial_anisotropy_field(mag_map, ...)	Calculate the uniaxial anisotropy component of the effective field.
calc_exchange_field(mag_map, ...)	Calculate the exchange component of the effective field, which is defined as for each i^{th} layer in the structure as
calc_thermal_field(mag_map, mag_map_squared, ...)	Calculate the thermal component of the effective field.
calc_Brillouin(mag, temp, eff_spin, ...)	
calc_dBrillouin_dx(temp_map, ...)	Calculate the derivative of the Brillouin function B_x at $m = m_{\mathrm{eq}}$:
calc_transverse_damping(temp_map, ...)	Calculate the transverse damping parameter:
calc_longitudinal_damping(temp_map, ...)	calc_transverse_damping
calc_qs(temp_map, mf_magnetizations, ...)	Calculate the q_s parameter:
calc_long_susceptibility(temp_map, ...)	Calculate the the longitudinal susceptibility
check_initial_magnetization(init_mag[, ...])	Check if a given initial magnetization profile is valid.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_hash(**kwargs)	Calculates an unique hash given by the delays as well as the sample structure hash for relevant magnetic parameters.
get_magnetization_map(delays, **kwargs)	Returns an absolute `magnetization_map` for the sample structure with the dimension $M \times N \times 3$ with M being the number of delays and N the number of layers in the structure or the length of the given spatial grid.
save(full_filename, data, *args)	Save data to file.

calc_magnetization_map(*delays*, *temp_map*, *H_ext=array([0, 0, 0])*, *init_mag=[]*)

Calculates the magnetization map using the mean-field quantum Landau-Lifshitz-Bloch equation (LLB) for a given delay range and according temperature map:

$$\frac{d\mathbf{m}}{dt} = \gamma_e \left(\mathbf{m} \times \mathbf{H}_{\mathrm{eff}} + \frac{\alpha_\perp}{m^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{H}_{\mathrm{eff}}) - \frac{\alpha_\parallel}{m^2} (\mathbf{m} \cdot \mathbf{H}_{\mathrm{eff}}) \cdot \mathbf{m} \right)$$

The three terms describe

1. **precession** at Larmor frequency,
2. **transversal damping** (conserving the macrospin length), and

3. **longitudinal damping** (changing macrospin length due to incoherent atomistic spin excitations within the layer the macrospin is defined on).

$\alpha_\parallel$ and $\alpha_\perp$ are the *longitudinal damping* and *transverse damping* parameters, respectively. $\gamma_e = -1.761 \times 10^{11}\,\mathrm{rad\,s^{-1}\,T^{-1}}$ is the gyromagnetic ratio of an electron.

The effective magnetic field is the sum of all relevant magnetic interactions:

$$\mathbf{H}_{\mathrm{eff}} = \mathbf{H}_{\mathrm{ext}} + \mathbf{H}_{\mathrm{A}} + \mathbf{H}_{\mathrm{ex}} + \mathbf{H}_{\mathrm{th}}$$

where

- $\mathbf{H}_{\mathrm{ext}}$ is the external magnetic field
- $\mathbf{H}_{\mathrm{A}}$ is the *uniaxial anisotropy field*
- $\mathbf{H}_{\mathrm{ex}}$ is the *exchange field*
- $\mathbf{H}_{\mathrm{th}}$ is the *thermal field*

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map.
- **H_ext** (*ndarray[float], optional*) – external magnetic field (H_x, H_y, H_z) [T].

Returns

magnetization_map (ndarray[float]) – spatio-temporal absolute magnetization profile.

get_mean_field_mag_map(*temp_map*)

Returns the mean-field magnetization map see *calc_mean_field_mag_map()* for details. The dimension of the map are $M \times N$ with M being the number of delays and N the number of layers in the structure.

Parameters

temp_map (*ndarray[float]*) – spatio-temporal electron temperature map.

Returns

mf_mag_map (ndarray[float]) – spatio-temporal mean_field magnetization map.

calc_mean_field_mag_map(*temp_map*)

Calculate the mean-field magnetization map m_{eq} by solving the self consistent equation

$$m_{\mathrm{eq}}(T) = B_S(m_{\mathrm{eq}}, T)$$

where B_S is the Brillouin function.

Parameters

temp_map (*ndarray[float]*) – spatio-temporal electron temperature map.

Returns

mf_mag_map (ndarray[float]) – spatio-temporal mean_field magnetization map.

get_directional_exchange_stiffnesses()

Returns the directional exchange stiffnesses with size $N \times 2$, with N being the number of layers, between the $(i-1)^{\mathrm{th}}$ and i^{th} layers as well as between the i^{th} and $(i+1)^{\mathrm{th}}$ layers in the structure. In case two neighboring layers are identical the center entry of the 3-component *exchange_stiffness* is used. In case of an interface to the top $(i-1) \rightarrow i$ it will be the first entry and for an interface to the bottom $i \rightarrow (i+1)$ it will be the third entry.

Returns

A (ndarray[float]) – directional exchange stiffnesses.

static odefunc(*t*, *m*, *delays*, *N*, *H_ext*, *temp_map*, *mean_mag_map*, *curie_temps*, *eff_spins*, *lambdas*, *mf_exch_couplings*, *mag_moments*, *aniso_exponents*, *anisotropies*, *mag_saturations*, *exch_stiffnesses*, *thicknesses*, *pbar*, *state*)

Ordinary differential equation that is solved for 1D LLB.

Parameters

- **t** (*ndarray[float]*) – internal time steps of the ode solver.
- **m** (*ndarray[float]*) – internal variable of the ode solver.
- **delays** (*ndarray[float]*) – delays range of simulation [s].
- **N** (*int*) – number of spatial grid points.
- **H_ext** (*ndarray[float]*) – external magnetic field (H_x, H_y, H_z) [T].
- **temp_map** (*ndarray[float]*) – spatio-temporal electron temperature map.
- **mean_mag_map** (*ndarray[float]*) – spatio-temporal mean-field magnetization map.
- **curie_temps** (*ndarray[float]*) – Curie temperatures of layers.
- **eff_spins** (*ndarray[float]*) – effective spins of layers.
- **lambdas** (*ndarray[float]*) – coupling-to-bath parameter of layers.
- **mf_exch_couplings** (*ndarray[float]*) – mean-field exchange couplings of layers.
- **mag_moments** (*ndarray[float]*) – atomic magnetic moments of layers.
- **aniso_exponents** (*ndarray[float]*) – exponent of uniaxial anisotropy of layers.
- **anisotropies** (*ndarray[float]*) – anisotropy vectors of layers.
- **mag_saturations** (*ndarray[float]*) – saturation magnetization of layers.
- **exch_stiffnesses** (*ndarray[float]*) – exchange stiffness of layers towards the upper and lower layer.
- **thicknesses** (*ndarray[float]*) – thicknesses of layers.
- **pbar** (*tqdm*) – tqdm progressbar.
- **state** (*list[float]*) – state variables for progress bar.

Returns

dmdt (ndarray[float]) – temporal derivative of internal variable.

static calc_uniaxial_anisotropy_field(*mag_map*, *mf_magnetizations*, *aniso_exponents*, *anisotropies*, *mag_saturations*)

Calculate the uniaxial anisotropy component of the effective field.

$$\mathbf{H}_\mathrm{A} = -\frac{2}{M_s}\left(K_x\, m_\mathrm{eq}(T)^{\kappa-2} \begin{bmatrix} 0 \\ m_y \\ m_z \end{bmatrix} + K_y\, m_\mathrm{eq}(T)^{\kappa-2} \begin{bmatrix} m_x \\ 0 \\ m_z \end{bmatrix} + K_z\, m_\mathrm{eq}(T)^{\kappa-2} \begin{bmatrix} m_x \\ m_y \\ 0 \end{bmatrix} \right)$$

with $K = (K_x, K_y, K_z)$ as the anisotropy and κ as the uniaxial anisotropy exponent.

Parameters

- **mag_map** (*ndarray[float]*) – spatio-temporal magnetization map - possibly for a single delay.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **aniso_exponents** (*ndarray[float]*) – exponent of uniaxial anisotropy of layers.

- **anisotropies** (*ndarray[float]*) – anisotropy vectors of layers.
- **mag_saturations** (*ndarray[float]*) – saturation magnetization of layers.

Returns
H_A (ndarray[float]) – uniaxial anisotropy field.

static calc_exchange_field(*mag_map*, *exch_stiffnesses*, *mag_saturations*, *thicknesses*)

Calculate the exchange component of the effective field, which is defined as for each i^{th} layer in the structure as

$$H_{\mathrm{ex},i} = \frac{2}{M_{s,i}\Delta z_i^2}\left(A_i^{i-1}\left(\mathbf{m}_{i-1}-\mathbf{m}_i\right)+A_i^{i+1}\left(\mathbf{m}_{i+1}-\mathbf{m}_i\right)\right),$$

where Δz is the thickness of the layers or magnetic grains and M_s is the saturation magnetization. A_i^{i-1} and A_i^{i+1} describe the exchange stiffness between the nearest neighboring layers provided by *get_directional_exchange_stiffnesses()*.

Parameters

- **mag_map** (*ndarray[float]*) – spatio-temporal magnetization map - possibly for a single delay.
- **exch_stiffnesses** (*ndarray[float]*) – exchange stiffness of layers towards the upper and lower layer.
- **mag_saturations** (*ndarray[float]*) – saturation magnetization of layers.

Returns
H_ex (ndarray[float]) – exchange field.

static calc_thermal_field(*mag_map*, *mag_map_squared*, *temp_map*, *mf_magnetizations*, *eff_spins*, *curie_temps*, *mf_exch_couplings*, *mag_moments*, *under_tc*, *over_tc*)

Calculate the thermal component of the effective field.

$$\mathbf{H}_{\mathrm{th}} = \begin{cases} \frac{1}{2\chi_{\parallel}}\left(1-\frac{m^2}{m_{\mathrm{eq}}^2}\right)\mathbf{m} & \text{for } T < T_{\mathrm{C}} \\ -\frac{1}{\chi_{\parallel}}\left(1+\frac{3}{5}\frac{T_{\mathrm{C}}}{T-T_{\mathrm{C}}}m^2\right)\mathbf{m} & \text{for } T \geq T_{\mathrm{C}} \end{cases}$$

with $\chi_{\parallel}$ being the *longitudinal susceptibility*.

Parameters

- **mag_map** (*ndarray[float]*) – spatio-temporal magnetization map - possibly for a single delay.
- **mag_map_squared** (*ndarray[float]*) – spatio-temporal magnetization map squared- possibly for a single delay.
- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **eff_spins** (*ndarray[float]*) – effective spin of layers.
- **curie_temps** (*ndarray[float]*) – Curie temperature of layers.
- **mf_exch_couplings** (*ndarray[float]*) – mean-field exch. coupling of layers.
- **mag_moments** (*ndarray[float]*) – atomic magnetic moments of layers.
- **under_tc** (*ndarray[boolean]*) – mask temperatures under the Curie temperature.
- **over_tc** (*ndarray[boolean]*) – mask temperatures over the Curie temperature.

Returns
H_th (ndarray[float]) – thermal field.

static calc_Brillouin(*mag*, *temp*, *eff_spin*, *mf_exch_coupling*, *curie_temp*)

$$B_S(m,T) = \frac{2S+1}{2S}\coth\left(\frac{2S+1}{2S}\frac{J\,m}{k_\mathrm{B}\,T}\right) - \frac{1}{2S}\coth\left(\frac{1}{2S}\frac{J\,m}{k_\mathrm{B}\,T}\right)$$

where

$$J = 3\frac{S}{S+1}k_\mathrm{B}\,T_\mathrm{C}$$

is the mean field exchange coupling constant for effective spin S and Curie temperature T_C.

Parameters

- **mag** (*ndarray[float]*) – magnetization of layer.
- **temp** (*ndarray[float]*) – electron temperature of layer.
- **eff_spin** (*ndarray[float]*) – effective spin of layer.
- **mf_exch_coupling** (*ndarray[float]*) – mean-field exch. coupling of layers.
- **curie_temp** (*ndarray[float]*) – Curie temperature of layer.

Returns
brillouin (ndarray[float]) – brillouin function.

static calc_dBrillouin_dx(*temp_map*, *mf_magnetizations*, *eff_spins*, *mf_exch_couplings*)

Calculate the derivative of the Brillouin function B_x at $m = m_\mathrm{eq}$:

$$B_x = \frac{dB}{dx} = \frac{1}{4S^2\sinh^2(x/2S)} - \frac{(2S+1)^2}{4S^2\sinh^2\left(\frac{(2S+1)x}{2S}\right)}$$

with $x = \frac{J\,m}{k_\mathrm{B}\,T}$.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **eff_spins** (*ndarray[float]*) – effective spin of layers.
- **mf_exch_couplings** (*ndarray[float]*) – mean-field exchange couplings of layers.

Returns
dBdx (ndarray[float]) – derivative of Brillouin function.

static calc_transverse_damping(*temp_map*, *curie_temps*, *lambdas*, *qs*, *mf_magnetizations*, *under_tc*, *over_tc*)

Calculate the transverse damping parameter:

$$\alpha_\perp = \begin{cases} \frac{\lambda}{m_\mathrm{eq}(T)}\left(\frac{\tanh(q_s)}{q_s} - \frac{T}{3T_\mathrm{C}}\right) & \text{for } T < T_\mathrm{C} \\ \frac{2\lambda}{3}\frac{T}{T_\mathrm{C}} & \text{for } T \geq T_\mathrm{C} \end{cases}$$

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.

- **curie_temps** (*ndarray[float]*) – Curie temperatures of layers.
- **lambdas** (*ndarray[float]*) – coupling-to-bath parameter of layers.
- **qs** (*ndarray[float]*) – qs parameter.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **under_tc** (*ndarray[boolean]*) – mask temperatures under the Curie temperature.
- **over_tc** (*ndarray[boolean]*) – mask temperatures over the Curie temperature.

Returns
alpha_trans (ndarray[float]) – transverse damping parameter.

static calc_longitudinal_damping(*temp_map*, *curie_temps*, *eff_spins*, *lambdas*, *qs*, *under_tc*, *over_tc*)

calc_transverse_damping

Calculate the transverse damping parameter:

$$\alpha_{\parallel} = \begin{cases} \frac{2\lambda}{S+1}\frac{1}{\sinh(2q_s)} & \text{for } T < T_{\mathrm{C}} \\ \frac{2\lambda}{3}\frac{T}{T_{\mathrm{C}}} & \text{for } T \geq T_{\mathrm{C}} \end{cases}$$

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.
- **curie_temps** (*ndarray[float]*) – Curie temperatures of layers.
- **eff_spins** (*ndarray[float]*) – effective spins of layers.
- **lambdas** (*ndarray[float]*) – coupling-to-bath parameter of layers.
- **qs** (*ndarray[float]*) – qs parameter.
- **under_tc** (*ndarray[boolean]*) – mask temperatures under the Curie temperature.
- **over_tc** (*ndarray[boolean]*) – mask temperatures over the Curie temperature.

Returns
alpha_long (ndarray[float]) – transverse damping parameter.

static calc_qs(*temp_map*, *mf_magnetizations*, *curie_temps*, *eff_spins*, *under_tc*)

Calculate the q_s parameter:

$$q_s = \frac{3T_{\mathrm{C}}m_{\mathrm{eq}}(T)}{(2S+1)T}$$

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **curie_temps** (*ndarray[float]*) – Curie temperatures of layers.
- **eff_spins** (*ndarray[float]*) – effective spins of layers.
- **under_tc** (*ndarray[boolean]*) – mask temperatures below the Curie temperature.

Returns
qs (ndarray[float]) – qs parameter.

static calc_long_susceptibility(*temp_map*, *mf_magnetizations*, *curie_temps*, *eff_spins*, *mf_exch_couplings*, *mag_moments*, *under_tc*, *over_tc*)

Calculate the the longitudinal susceptibility

$$\chi_{\parallel} = \begin{cases} \frac{\mu_{\mathrm{B}} B_x(m_{eq},T)}{T k_{\mathrm{B}} - J B_x(m_{eq},T)} & \text{for } T < T_{\mathrm{C}} \\ \frac{\mu_{\mathrm{B}} T_{\mathrm{C}}}{J(T-T_{\mathrm{C}})} & \text{for } T \geq T_{\mathrm{C}} \end{cases}$$

with $B_x(m_{eq}, T)$ being the *derivative of the Brillouin function*.

Parameters

- **temp_map** (*ndarray[float]*) – spatio-temporal temperature map - possibly for a single delay.
- **mf_magnetizations** (*ndarray[float]*) – mean-field magnetization of layers.
- **curie_temps** (*ndarray[float]*) – Curie temperatures of layers.
- **eff_spins** (*ndarray[float]*) – effective spins of layers.
- **mf_exch_couplings** (*ndarray[float]*) – mean-field exchange couplings of layers.
- **mag_moments** (*ndarray[float]*) – atomic magnetic moments of layers.
- **under_tc** (*ndarray[boolean]*) – mask temperatures below the Curie temperature.
- **over_tc** (*ndarray[boolean]*) – mask temperatures over the Curie temperature.

Returns

chi_long (ndarray[float]) – longitudinal susceptibility.

check_initial_magnetization(*init_mag*, *distances=[]*)

Check if a given initial magnetization profile is valid. The profile must be either a vector *[amplitude, phi, gamma]* describing a global initial magnetization or an array of shape *[Nx3]* with N being the number of layers or the length of the specific spatial grid. If no initial magnetization is given, the initial profile is determined from the magnetization of the layers on creation. In addition, a spatial grid can be provided.

Parameters

- **init_mag** (*ndarray[float]*) – initial global or local magnetization profile.
- **distances** (*ndarray[float, Quantity], optional*) – spatial grid of the initial magnetization.

Returns

init_mag (ndarray[float]) –

checked initial magnetization as array on
the according spatial grid.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

get_hash(***kwargs*)

Calculates an unique hash given by the delays as well as the sample structure hash for relevant magnetic parameters. Optionally, part of the `strain_map` and `temp_map` are used.

Parameters

- **delays** (*ndarray[float]*) – delay grid for the simulation.
- ****kwargs** (*ndarray[float], optional*) – optional delays, strain and temperature profile as well as external magnetic field and initial magnetization.

Returns
hash (str) – unique hash.

get_magnetization_map(*delays*, ***kwargs*)

Returns an absolute `magnetization_map` for the sample structure with the dimension $M \times N \times 3$ with M being the number of delays and N the number of layers in the structure or the length of the given spatial grid. Each element of the map contains the three magnetization components `[amplitude, phi, gamma]`. The angles `phi` and `gamma` must be returned in radians as pure numpy arrays. The `magnetization_map` can depend on the `temp_map` and `strain_map` that can be also calculated for the sample structure. More over an external magnetic field `H_ext` and initial magnetization profile `init_mag` can be provided.

Parameters

- **delays** (*ndarray[Quantity]*) – delays range of simulation [s].
- ****kwargs** (*ndarray[float], optional*) – optional strain and temperature profile as well external magnetic field in [T] and initial magnetization.

Returns

magnetization_map (ndarray[float]) –

spatio-temporal absolute
magnetization profile.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

5.4.8 simulations.xrays

Classes:

Xray(S, force_recalc, **kwargs)	Base class for all X-ray scattering simulations.
XrayKin(S, force_recalc, **kwargs)	Kinetic X-ray scattering simulations.
XrayDyn(S, force_recalc, **kwargs)	Dynamical X-ray scattering simulations.
XrayDynMag(S, force_recalc, **kwargs)	Dynamical magnetic X-ray scattering simulations.

class udkm1Dsim.simulations.xrays.**Xray**(*S*, *force_recalc*, ***kwargs*)

Bases: *Simulation*

Base class for all X-ray scattering simulations.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **energy** (*ndarray[float]*) – photon energies E of scattering light
- **wl** (*ndarray[float]*) – wavelengths λ of scattering light
- **k** (*ndarray[float]*) – wavenumber k of scattering light
- **theta** (*ndarray[float]*) – incidence angles θ of scattering light
- **qz** (*ndarray[float]*) – scattering vector q_z of scattering light
- **polarizations** (*dict*) – polarization states and according names.
- **pol_in_state** (*int*) – incoming polarization state as defined in polarizations dict.
- **pol_out_state** (*int*) – outgoing polarization state as defined in polarizations dict.
- **pol_in** (*float*) – incoming polarization factor (can be a complex ndarray).
- **pol_out** (*float*) – outgoing polarization factor (can be a complex ndarray).

Methods:

set_incoming_polarization(pol_in_state)	Must be overwritten by child classes.
set_outgoing_polarization(pol_out_state)	Must be overwritten by child classes.
set_polarization(pol_in_state, pol_out_state)	Sets the incoming and analyzer (outgoing) polarization.
get_hash(strain_vectors, **kwargs)	Calculates an unique hash given by the energy E, q_z range, polarization states and the `strain_vectors` as well as the sample structure hash for relevant x-ray parameters.
get_polarization_factor(theta)	Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of *s*-polarization (pol = 0), or *p*-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):
update_experiment(caller)	Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
save(full_filename, data, *args)	Save data to file.

set_incoming_polarization(*pol_in_state*)

Must be overwritten by child classes.

Parameters
pol_in_state (*int*) – incoming polarization state id.

set_outgoing_polarization(*pol_out_state*)

Must be overwritten by child classes.

Parameters
pol_out_state (*int*) – outgoing polarization state id.

set_polarization(*pol_in_state*, *pol_out_state*)

Sets the incoming and analyzer (outgoing) polarization.

Parameters

- **pol_in_state** (*int*) – incoming polarization state id.
- **pol_out_state** (*int*) – outgoing polarization state id.

get_hash(*strain_vectors*, ***kwargs*)

Calculates an unique hash given by the energy E, q_z range, polarization states and the `strain_vectors` as well as the sample structure hash for relevant x-ray parameters. Optionally, part of the strain_map is used.

Parameters

- **strain_vectors** (*dict{ndarray[float]}*) – reduced strains per unique layer.
- ****kwargs** (*ndarray[float]*) – spatio-temporal strain profile.

Returns
hash (str) – unique hash.

get_polarization_factor(*theta*)

Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of s-polarization (pol = 0), or p-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):

$$P(\vartheta) = \sqrt{(1 - \text{pol}) + \text{pol} \cdot \cos(2\vartheta)}$$

Parameters
theta (*ndarray[float]*) – incidence angle.

Returns
P (ndarray[float]) – polarization factor.

update_experiment(*caller*)

Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.

$$\lambda = \frac{hc}{E}$$
$$E = \frac{hc}{\lambda}$$
$$k = \frac{2\pi}{\lambda}$$
$$\vartheta = \arcsin\frac{\lambda q_z}{4\pi}$$
$$q_z = 2k\sin\vartheta$$

Parameters
caller (*str*) – name of calling method.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns
y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

class udkm1Dsim.simulations.xrays.**XrayKin**(*S*, *force_recalc*, ***kwargs*)

Bases: *Xray*

Kinetic X-ray scattering simulations.

Parameters

- **S** (Structure) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **energy** (*ndarray[float]*) – photon energies E of scattering light
- **wl** (*ndarray[float]*) – wavelengths λ of scattering light
- **k** (*ndarray[float]*) – wavenumber k of scattering light
- **theta** (*ndarray[float]*) – incidence angles θ of scattering light
- **qz** (*ndarray[float]*) – scattering vector q_z of scattering light
- **polarizations** (*dict*) – polarization states and according names.
- **pol_in_state** (*int*) – incoming polarization state as defined in polarizations dict.
- **pol_out_state** (*int*) – outgoing polarization state as defined in polarizations dict.
- **pol_in** (*float*) – incoming polarization factor (can be a complex ndarray).
- **pol_out** (*float*) – outgoing polarization factor (can be a complex ndarray).

References

Methods:

set_incoming_polarization(pol_in_state)	Sets the incoming polarization factor for sigma, pi, and unpolarized polarization.
set_outgoing_polarization(pol_out_state)	For kinematical X-ray simulation only "no analyzer polarization" is allowed.
get_uc_atomic_form_factors(energy, qz, uc)	Returns the energy- and angle-dependent atomic form factors :math: *f(q_z, E)* of all atoms in the unit cell as a vector.
get_uc_structure_factor(energy, qz, uc[, strain])	Calculates the energy-, angle-, and strain-dependent structure factor .
homogeneous_reflectivity([strains])	Calculates the reflectivity $R = E_p^t \, (E_p^t)^*$ of a homogeneous sample structure as well as the reflected field E_p^N of all substructures.
homogeneous_reflected_field(S, energy, qz, theta)	Calculates the reflected field E_p^t of the whole sample structure as well as for each sub-structure (E_p^N).
get_interference_function(qz, z, N)	Calculates the interference function for N repetitions of the structure with the length z:
get_Ep(energy, qz, theta, uc, strain)	Calculates the reflected field E_p for one unit cell with a given strain ϵ:
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_hash(strain_vectors, **kwargs)	Calculates an unique hash given by the energy E, q_z range, polarization states and the `strain_vectors` as well as the sample structure hash for relevant x-ray parameters.
get_polarization_factor(theta)	Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of *s*-polarization (pol = 0), or *p*-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):
save(full_filename, data, *args)	Save data to file.
set_polarization(pol_in_state, pol_out_state)	Sets the incoming and analyzer (outgoing) polarization.
update_experiment(caller)	Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.

set_incoming_polarization(*pol_in_state*)

Sets the incoming polarization factor for sigma, pi, and unpolarized polarization.

Parameters
pol_in_state (*int*) – incoming polarization state id.

set_outgoing_polarization(*pol_out_state*)

For kinematical X-ray simulation only "no analyzer polarization" is allowed.

Parameters
pol_out_state (*int*) – outgoing polarization state id.

get_uc_atomic_form_factors(*energy*, *qz*, *uc*)

Returns the energy- and angle-dependent atomic form factors :math: *f(q_z, E)* of all atoms in the unit cell as a vector.

Parameters

- **energy** (*float, Quantity*) – photon energy.

- **qz** (*ndarray[float, Quantity]*) – scattering vectors.
- **uc** (UnitCell) – unit cell object.

Returns
f (ndarray[complex]) – unit cell atomic form factors.

get_uc_structure_factor(*energy*, *qz*, *uc*, *strain=0*)

Calculates the energy-, angle-, and strain-dependent structure factor .. math: *S(E,q_z,epsilon)* of the unit cell:

$$S(E, q_z, \epsilon) = \sum_i^N f_i \, \exp(-iq_z z_i(\epsilon))$$

Parameters

- **energy** (*float, Quantity*) – photon energy.
- **qz** (*ndarray[float, Quantity]*) – scattering vectors.
- **uc** (UnitCell) – unit cell object.
- **strain** (*float, optional*) – strain of the unit cell 0 .. 1. Defaults to 0.

Returns
S (ndarray[complex]) – unit cell structure factor.

homogeneous_reflectivity(*strains=0*)

Calculates the reflectivity $R = E_p^t \, (E_p^t)^*$ of a homogeneous sample structure as well as the reflected field E_p^N of all substructures.

Parameters
strains (*ndarray[float], optional*) – strains of each sub-structure 0 .. 1. Defaults to 0.

Returns

(tuple) –

- *R (ndarray[complex])* - homogeneous reflectivity.
- *A (ndarray[complex])* - reflected fields of sub-structures.

homogeneous_reflected_field(*S*, *energy*, *qz*, *theta*, *strains=0*)

Calculates the reflected field E_p^t of the whole sample structure as well as for each sub-structure (E_p^N). The reflected wave field E_p from a single layer of unit cells at the detector is calculated according to Ref.[9]:

$$E_p = \frac{i}{\varepsilon_0} \frac{e^2}{m_e c_0^2} \frac{P(\vartheta) S(E, q_z, \epsilon)}{A q_z}$$

For the case of N similar planes of unit cells one can write:

$$E_p^N = \sum_{n=0}^{N-1} E_p \exp(iq_z z n)$$

where z is the distance between the planes (c-axis). The above equation can be simplified to:

$$E_p^N = E_p \psi(q_z, z, N)$$

[9] B. E. Warren (1990). *X-ray diffraction.* New York: Dover Publications

introducing the interference function

$$\psi(q_z, z, N) = \sum_{n=0}^{N-1} \exp(iq_z z n) = \frac{1 - \exp(iq_z z N)}{1 - \exp(iq_z z)}$$

The total reflected wave field of all $i = 1 \dots M$ homogeneous layers (E_p^t) is the phase-correct summation of all individual $E_p^{N,i}$:

$$E_p^t = \sum_{i=1}^{M} E_p^{N,i} \exp(iq_z Z_i)$$

where $Z_i = \sum_{j=1}^{i-1} N_j z_j$ is the distance of the i-th layer from the surface.

Parameters

- **S** (*Structure, UnitCell*) – structure or sub-structure to calculate on.
- **energy** (*float, Quantity*) – photon energy.
- **qz** (*ndarray[float, Quantity]*) – scattering vectors.
- **theta** (*ndarray[float, Quantity]*) – scattering incidence angle.
- **strains** (*ndarray[float], optional*) – strains of each sub-structure 0 .. 1. Defaults to 0.

Returns

(tuple) –

- *Ept (ndarray[complex])* - reflected field.
- *A (ndarray[complex])* - reflected fields of substructures.

get_interference_function(*qz*, *z*, *N*)

Calculates the interference function for N repetitions of the structure with the length z:

$$\psi(q_z, z, N) = \sum_{n=0}^{N-1} \exp(iq_z z n) = \frac{1 - \exp(iq_z z N)}{1 - \exp(iq_z z)}$$

Parameters

- **qz** (*ndarray[float, Quantity]*) – scattering vectors.
- **z** (*float*) – thickness/length of the structure.
- **N** (*int*) – repetitions of the structure.

Returns

psi (ndarray[complex]) – interference function.

get_Ep(*energy*, *qz*, *theta*, *uc*, *strain*)

Calculates the reflected field E_p for one unit cell with a given strain ϵ:

$$E_p = \frac{i}{\varepsilon_0} \frac{e^2}{m_e c_0^2} \frac{PS(E, q_z, \epsilon)}{Aq_z}$$

with e as electron charge, m_e as electron mass, c_0 as vacuum light velocity, ε_0 as vacuum permittivity, P as polarization factor and $S(E, q_z, \sigma)$ as energy-, angle-, and strain-dependent unit cell structure factor.

Parameters

- **energy** (*float, Quantity*) – photon energy.
- **qz** (*ndarray[float, Quantity]*) – scattering vectors.
- **theta** (*ndarray[float, Quantity]*) – scattering incidence angle.
- **uc** (UnitCell) – unit cell object.
- **strain** (*float, optional*) – strain of the unit cell 0 .. 1. Defaults to 0.

Returns

Ep (ndarray[complex]) – reflected field.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters

message (*str*) – message to display.

get_hash(*strain_vectors*, ***kwargs*)

Calculates an unique hash given by the energy E, q_z range, polarization states and the `strain_vectors` as well as the sample structure hash for relevant x-ray parameters. Optionally, part of the strain_map is used.

Parameters

- **strain_vectors** (*dict{ndarray[float]}*) – reduced strains per unique layer.
- ****kwargs** (*ndarray[float]*) – spatio-temporal strain profile.

Returns

hash (str) – unique hash.

get_polarization_factor(*theta*)

Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of *s*-polarization (pol = 0), or *p*-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):

$$P(\vartheta) = \sqrt{(1 - \text{pol}) + \text{pol} \cdot \cos(2\vartheta)}$$

Parameters

theta (*ndarray[float]*) – incidence angle.

Returns

P (ndarray[float]) – polarization factor.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

set_polarization(*pol_in_state*, *pol_out_state*)

Sets the incoming and analyzer (outgoing) polarization.

Parameters

- **pol_in_state** (*int*) – incoming polarization state id.
- **pol_out_state** (*int*) – outgoing polarization state id.

update_experiment(*caller*)

Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.

$$\lambda = \frac{hc}{E}$$
$$E = \frac{hc}{\lambda}$$
$$k = \frac{2\pi}{\lambda}$$
$$\vartheta = \arcsin\frac{\lambda q_z}{4\pi}$$
$$q_z = 2k\sin\vartheta$$

Parameters

caller (*str*) – name of calling method.

class udkm1Dsim.simulations.xrays.**XrayDyn**(*S*, *force_recalc*, ***kwargs*)

Bases: *Xray*

Dynamical X-ray scattering simulations.

Parameters

- **S** (*Structure*) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

- **energy** (*ndarray[float]*) – photon energies E of scattering light
- **wl** (*ndarray[float]*) – wavelengths λ of scattering light
- **k** (*ndarray[float]*) – wavenumber k of scattering light
- **theta** (*ndarray[float]*) – incidence angles θ of scattering light
- **qz** (*ndarray[float]*) – scattering vector q_z of scattering light
- **polarizations** (*dict*) – polarization states and according names.
- **pol_in_state** (*int*) – incoming polarization state as defined in polarizations dict.
- **pol_out_state** (*int*) – outgoing polarization state as defined in polarizations dict.
- **pol_in** (*float*) – incoming polarization factor (can be a complex ndarray).
- **pol_out** (*float*) – outgoing polarization factor (can be a complex ndarray).
- **last_atom_ref_trans_matrices** (*list*) – remember last result of atom ref_trans_matrices to speed up calculation.

Methods:

set_incoming_polarization(pol_in_state)	Sets the incoming polarization factor for sigma, pi, and unpolarized polarization.
set_outgoing_polarization(pol_out_state)	For dynamical X-ray simulation only "no analyzer polarization" is allowed.
homogeneous_reflectivity(*args)	Calculates the reflectivity R of the whole sample structure and the reflectivity-transmission matrices M_{RT} for each substructure.
homogeneous_ref_trans_matrix(S, *args)	Calculates the reflectivity-transmission matrices M_{RT} of the whole sample structure as well as for each sub-structure.
inhomogeneous_reflectivity(strain_map, ...)	Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`).
sequential_inhomogeneous_reflectivity(...)	Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`).
parallel_inhomogeneous_reflectivity(...)	Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`).
distributed_inhomogeneous_reflectivity(...)	This is a stub.
calc_inhomogeneous_reflectivity(strains, ...)	Calculates the reflectivity of a inhomogeneous sample structure for given `strain_vectors` for a single time step.
calc_inhomogeneous_ref_trans_matrix(...)	Sub-function of *calc_inhomogeneous_reflectivity()* and for parallel computing (needs to be static) only for calculating the total reflection-transmission matrix M^t_{RT}:
get_all_ref_trans_matrices(*args)	Returns a list of all reflection-transmission matrices for each unique unit cell in the sample structure for a given set of applied strains for each unique unit cell given by the `strain_vectors` input.
calc_all_ref_trans_matrices(*args)	Calculates a list of all reflection-transmission matrices for each unique unit cell in the sample structure for a given set of applied strains to each unique unit cell given by the `strain_vectors` input.
get_uc_ref_trans_matrix(uc, *args)	Returns the reflection-transmission matrix of a unit cell:
get_atom_ref_trans_matrix(atom, area, ...)	Calculates the reflection-transmission matrix of an atom from dynamical x-ray theory:
get_atom_reflection_factor(atom, area, ...)	Calculates the reflection factor from dynamical x-ray theory:
get_atom_transmission_factor(atom, area, ...)	Calculates the transmission factor from dynamical x-ray theory:
get_atom_phase_matrix(distance)	Calculates the phase matrix from dynamical x-ray theory:
get_atom_phase_factor(distance)	Calculates the phase factor ϕ for a distance d from dynamical x-ray theory:
calc_reflectivity_from_matrix(M)	Calculates the reflectivity from an 2×2 matrix of transmission and reflectivity factors:
convolve_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_hash(strain_vectors, **kwargs)	Calculates an unique hash given by the energy E, q_z

set_incoming_polarization(*pol_in_state*)

Sets the incoming polarization factor for sigma, pi, and unpolarized polarization.

Parameters
pol_in_state (*int*) – incoming polarization state id.

set_outgoing_polarization(*pol_out_state*)

For dynamical X-ray simulation only "no analyzer polarization" is allowed.

Parameters
pol_out_state (*int*) – outgoing polarization state id.

homogeneous_reflectivity(**args*)

Calculates the reflectivity R of the whole sample structure and the reflectivity-transmission matrices M_{RT} for each substructure. The reflectivity of the 2×2 matrices for each q_z is calculates as follow:

$$R = \left| M^t_{RT}(0,1) / M^t_{RT}(1,1) \right|^2$$

Parameters
***args** (*ndarray[float], optional*) – strains for each substructure.

Returns

(tuple) –

- *R (ndarray[float])* - homogeneous reflectivity.
- *A (ndarray[complex])* - reflectivity-transmission matrices of sub-structures.

homogeneous_ref_trans_matrix(*S*, **args*)

Calculates the reflectivity-transmission matrices M_{RT} of the whole sample structure as well as for each sub-structure. The reflectivity-transmission matrix of a single unit cell is calculated from the reflection-transmission matrices H_i of each atom and the phase matrices between the atoms L_i:

$$M_{RT} = \prod_i H_i \, L_i$$

For N similar layers of unit cells one can calculate the N-th power of the unit cell $(M_{RT})^N$. The reflection-transmission matrix for the whole sample M^t_{RT} consisting of $j = 1 \ldots M$ sub-structures is then again:

$$M^t_{RT} = \prod_{j=1}^{M} \left(M_{RT\cdot j}\right)^{N_j}$$

Parameters

- **S** (Structure, UnitCell) – structure or sub-structure to calculate on.
- ***args** (*ndarray[float], optional*) – strains for each substructure.

Returns

(tuple) –

- *RT (ndarray[complex])* - reflectivity-transmission matrix.
- *A (ndarray[complex])* - reflectivity-transmission matrices of sub-structures.

inhomogeneous_reflectivity(*strain_map*, *strain_vectors*, ***kwargs*)

Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`). If no reflectivity is saved in the cache it is caluclated. Providing the `calc_type` for the calculation the corresponding sub-routines for the reflectivity computation are called:

- `parallel` parallelization over the time steps utilizing Dask
- `distributed` not implemented in Python, but should be possible with Dask as well
- `sequential` no parallelization at all

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- ****kwargs** –
  - *calc_type (str)* - type of calculation.
  - *dask_client (Dask.Client)* - Dask client.
  - *job (Dask.job)* - Dask job.
  - *num_workers (int)* - Dask number of workers.

Returns

R (ndarray[float]) – inhomogeneous reflectivity.

sequential_inhomogeneous_reflectivity(*strain_map*, *strain_vectors*, *RTM*)

Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`). The function calculates the results sequentially without parallelization.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- **RTM** (*list[ndarray[complex]]*) – reflection-transmission matrices for all given strains per unique layer.

Returns

R (ndarray[float]) – inhomogeneous reflectivity.

parallel_inhomogeneous_reflectivity(*strain_map*, *strain_vectors*, *RTM*, *dask_client*)

Returns the reflectivity of an inhomogeneously strained sample structure for a given `strain_map` in position and time, as well as for a given set of possible strains for each unit cell in the sample structure (`strain_vectors`). The function parallelizes the calculation over the time steps, since the results do not depend on each other.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- **RTM** (*list[ndarray[complex]]*) – reflection-transmission matrices for all given strains per unique layer.
- **dask_client** (*Dask.Client*) – Dask client.

Returns

R (ndarray[float]) – inhomogeneous reflectivity.

distributed_inhomogeneous_reflectivity(*strain_map*, *strain_vectors*, *RTM*, *job*, *num_worker*)

This is a stub. Not yet implemented in python.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- **RTM** (*list[ndarray[complex]]*) – reflection-transmission matrices for all given strains per unique layer.
- **job** (*Dask.job*) – Dask job.
- **num_workers** (*int*) – Dask number of workers.

Returns

R (ndarray[float]) – inhomogeneous reflectivity.

calc_inhomogeneous_reflectivity(*strains*, *strain_vectors*, *RTM*)

Calculates the reflectivity of a inhomogeneous sample structure for given `strain_vectors` for a single time step. Similar to the homogeneous sample structure, the reflectivity of an unit cell is calculated from the reflection-transmission matrices H_i of each atom and the phase matrices between the atoms L_i in the unit cell:

$$M_{RT} = \prod_i H_i \, L_i$$

Since all layers are generally inhomogeneously strained we have to traverse all individual unit cells ($j = 1 \dots M$) in the sample to calculate the total reflection-transmission matrix M_{RT}^t:

$$M_{RT}^t = \prod_{j=1}^{M} M_{RT,j}$$

The reflectivity of the 2×2 matrices for each q_z is calculates as follow:

$$R = \left| M_{RT}^t(1,2) / M_{RT}^t(2,2) \right|^2$$

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- **RTM** (*list[ndarray[complex]]*) – reflection-transmission matrices for all given strains per unique layer.

Returns

R (ndarray[float]) – inhomogeneous reflectivity.

static calc_inhomogeneous_ref_trans_matrix(*uc_indices*, *RT*, *strains*, *strain_vectors*, *RTM*)

Sub-function of *calc_inhomogeneous_reflectivity()* and for parallel computing (needs to be static) only for calculating the total reflection-transmission matrix M_{RT}^t:

$$M_{RT}^t = \prod_{j=1}^{M} M_{RT,j}$$

Parameters

- **uc_indices** (*ndarray[float]*) – unit cell indices.

- **RT** (*ndarray[complex]*) – reflection-transmission matrix.
- **strains** (*ndarray[float]*) – spatial strain profile for single time step.
- **strain_vectors** (*list[ndarray[float]]*) – reduced strains per unique layer.
- **RTM** (*list[ndarray[complex]]*) – reflection-transmission matrices for all given strains per unique layer.

Returns
RT (ndarray[complex]) – reflection-transmission matrix.

`get_all_ref_trans_matrices`(**args*)

Returns a list of all reflection-transmission matrices for each unique unit cell in the sample structure for a given set of applied strains for each unique unit cell given by the `strain_vectors` input. If this data was saved on disk before, it is loaded, otherwise it is calculated.

Parameters
args (*list[ndarray[float]], optional*) – reduced strains per unique layer.

Returns
RTM (list[ndarray[complex]]) – reflection-transmission matrices for all given strains per unique layer.

`calc_all_ref_trans_matrices`(**args*)

Calculates a list of all reflection-transmission matrices for each unique unit cell in the sample structure for a given set of applied strains to each unique unit cell given by the `strain_vectors` input.

Args::

args (list[ndarray[float]], optional): reduced strains per unique
layer.

Returns
RTM (list[ndarray[complex]]) – reflection-transmission matrices for all given strains per unique layer.

`get_uc_ref_trans_matrix`(*uc*, **args*)

Returns the reflection-transmission matrix of a unit cell:

$$M_{RT} = \prod_i H_i \, L_i$$

where H_i and L_i are the atomic reflection- transmission matrix and the phase matrix for the atomic distances, respectively.

Parameters

- **uc** (`UnitCell`) – unit cell object.
- **args** (*float, optional*) – strain of unit cell.

Returns

RTM (list[ndarray[complex]]) –

reflection-transmission matrices for
all given strains per unique layer.

get_atom_ref_trans_matrix(*atom*, *area*, *deb_wal_fac*)

Calculates the reflection-transmission matrix of an atom from dynamical x-ray theory:

$$H = \frac{1}{\tau}\begin{bmatrix}(\tau^2 - \rho^2) & \rho \\ -\rho & 1\end{bmatrix}$$

Parameters

- **atom** (Atom, AtomMixed) – atom or mixed atom
- **area** (*float*) – area of the unit cell [m^2]
- **deb_wal_fac** (*float*) – Debye-Waller factor for unit cell

Returns

H (ndarray[complex]) – reflection-transmission matrix

get_atom_reflection_factor(*atom*, *area*, *deb_wal_fac*)

Calculates the reflection factor from dynamical x-ray theory:

$$\rho = \frac{-i4\pi\ r_e\ f(E, q_z)\ P(\theta) \exp(-M)}{q_z\ A}$$

- r_e is the electron radius
- $f(E, q_z)$ is the energy and angle dispersive atomic form factor
- $P(q_z)$ is the polarization factor
- A is the area in $x - y$ plane on which the atom is placed
- $M = 0.5(\text{dbf}\ q_z)^2)$ where $\text{dbf}^2 = \langle u^2 \rangle$ is the average thermal vibration of the atoms - Debye-Waller factor

Parameters

- **atom** (Atom, AtomMixed) – atom or mixed atom
- **area** (*float*) – area of the unit cell [m^2]
- **deb_wal_fac** (*float*) – Debye-Waller factor for unit cell

Returns

rho (complex) – reflection factor

get_atom_transmission_factor(*atom*, *area*, *deb_wal_fac*)

Calculates the transmission factor from dynamical x-ray theory:

$$\tau = 1 - \frac{i4\pi r_e f(E, 0) \exp(-M)}{q_z A}$$

- r_e is the electron radius
- $f(E, 0)$ is the energy dispersive atomic form factor (no angle correction)
- A is the area in $x - y$ plane on which the atom is placed
- $M = 0.5(\text{dbf}\ q_z)^2$ where $\text{dbf}^2 = \langle u^2 \rangle$ is the average thermal vibration of the atoms - Debye-Waller factor

Parameters

- **atom** (Atom, AtomMixed) – atom or mixed atom

- **area** (*float*) – area of the unit cell [m^2]
- **deb_wal_fac** (*float*) – Debye-Waller factor for unit cell

Returns
tau (complex) – transmission factor

get_atom_phase_matrix(*distance*)

Calculates the phase matrix from dynamical x-ray theory:

$$L = \begin{bmatrix} \exp(i\phi) & 0 \\ 0 & \exp(-i\phi) \end{bmatrix}$$

Parameters
distance (*float*) – distance between atomic planes

Returns
L (ndarray[complex]) – phase matrix

get_atom_phase_factor(*distance*)

Calculates the phase factor ϕ for a distance d from dynamical x-ray theory:

$$\phi = \frac{d\, q_z}{2}$$

Parameters
distance (*float*) – distance between atomic planes

Returns
phi (float) – phase factor

static calc_reflectivity_from_matrix(*M*)

Calculates the reflectivity from an 2×2 matrix of transmission and reflectivity factors:

$$R = |M(0,1)/M(1,1)|^2$$

Parameters
M (*ndarray[complex]*) – reflection-transmission matrix

Returns
R (ndarray[float]) – reflectivity

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns
y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters
message (*str*) – message to display.

get_hash(*strain_vectors*, ***kwargs*)

Calculates an unique hash given by the energy E, q_z range, polarization states and the `strain_vectors` as well as the sample structure hash for relevant x-ray parameters. Optionally, part of the strain_map is used.

Parameters

- **strain_vectors** (*dict{ndarray[float]}*) – reduced strains per unique layer.
- ****kwargs** (*ndarray[float]*) – spatio-temporal strain profile.

Returns

hash (str) – unique hash.

get_polarization_factor(*theta*)

Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of *s*-polarization (pol = 0), or *p*-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):

$$P(\vartheta) = \sqrt{(1 - \text{pol}) + \text{pol} \cdot \cos(2\vartheta)}$$

Parameters

theta (*ndarray[float]*) – incidence angle.

Returns

P (ndarray[float]) – polarization factor.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

set_polarization(*pol_in_state*, *pol_out_state*)

Sets the incoming and analyzer (outgoing) polarization.

Parameters

- **pol_in_state** (*int*) – incoming polarization state id.
- **pol_out_state** (*int*) – outgoing polarization state id.

update_experiment(*caller*)

Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.

$$\lambda = \frac{hc}{E}$$
$$E = \frac{hc}{\lambda}$$
$$k = \frac{2\pi}{\lambda}$$
$$\vartheta = \arcsin\frac{\lambda q_z}{4\pi}$$
$$q_z = 2k\sin\vartheta$$

Parameters

caller (*str*) – name of calling method.

class udkm1Dsim.simulations.xrays.**XrayDynMag**(*S*, *force_recalc*, ***kwargs*)

Bases: *Xray*

Dynamical magnetic X-ray scattering simulations.

Adapted from Elzo et.al.[10] and initially realized in Project Dyna.

Original copyright notice:

Copyright Institut Neel, CNRS, Grenoble, France

Project Collaborators:

- Stéphane Grenier, stephane.grenier@neel.cnrs.fr
- Marta Elzo (PhD, 2009-2012)
- Nicolas Jaouen Sextants beamline, Synchrotron Soleil, nicolas.jaouen@synchrotron-soleil.fr
- Emmanuelle Jal (PhD, 2010-2013) now at LCPMR CNRS, Paris
- Jean-Marc Tonnerre, jean-marc.tonnerre@neel.cnrs.fr
- Ingrid Hallsteinsen - Padraic Shaffer's group - Berkeley Nat. Lab.

Questions to:

- Stéphane Grenier, stephane.grenier@neel.cnrs.fr

Parameters

- **S** (Structure) – sample to do simulations with.
- **force_recalc** (*boolean*) – force recalculation of results.

Keyword Arguments

- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.

Attributes

- **S** (*Structure*) – sample structure to calculate simulations on.
- **force_recalc** (*boolean*) – force recalculation of results.
- **save_data** (*boolean*) – true to save simulation results.
- **cache_dir** (*str*) – path to cached data.
- **disp_messages** (*boolean*) – true to display messages from within the simulations.
- **progress_bar** (*boolean*) – enable tqdm progress bar.
- **energy** (*ndarray[float]*) – photon energies E of scattering light
- **wl** (*ndarray[float]*) – wavelengths λ of scattering light
- **k** (*ndarray[float]*) – wavenumber k of scattering light
- **theta** (*ndarray[float]*) – incidence angles θ of scattering light

[10] M. Elzo, E. Jal, O. Bunau, S. Grenier, Y. Joly, A. Y. Ramos, H. C. N. Tolentino, J. M. Tonnerre & N. Jaouen, *X-ray resonant magnetic reflectivity of stratified magnetic structures: Eigenwave formalism and application to a W/Fe/W trilayer*, J. Magn. Magn. Mater. 324, 105 (2012).

- **qz** (*ndarray[float]*) – scattering vector q_z of scattering light
- **polarizations** (*dict*) – polarization states and according names.
- **pol_in_state** (*int*) – incoming polarization state as defined in polarizations dict.
- **pol_out_state** (*int*) – outgoing polarization state as defined in polarizations dict.
- **pol_in** (*float*) – incoming polarization factor (can be a complex ndarray).
- **pol_out** (*float*) – outgoing polarization factor (can be a complex ndarray).
- **last_atom_ref_trans_matrices** (*list*) – remember last result of atom ref_trans_matrices to speed up calculation.

References

Methods:

get_hash(**kwargs)	Calculates an unique hash given by the energy E, q_z range, polarization states as well as the sample structure hash for relevant x-ray and magnetic parameters.
set_incoming_polarization(pol_in_state)	Sets the incoming polarization factor for circular +, circular -, sigma, pi, and unpolarized polarization.
set_outgoing_polarization(pol_out_state)	Sets the outgoing polarization factor for circular +, circular -, sigma, pi, and unpolarized polarization.
homogeneous_reflectivity(*args)	Calculates the reflectivity R of the whole sample structure allowing only for homogeneous strain and magnetization.
calc_homogeneous_matrix(S, last_A, ...)	Calculates the product of all reflection-transmission matrices of the sample structure
inhomogeneous_reflectivity([strain_map, ...])	Returns the reflectivity and transmissivity of an inhomogeneously strained and magnetized sample structure for a given _strain_map_ and _magnetization_map_ in space and time for each unit cell or amorphous layer in the sample structure.
sequential_inhomogeneous_reflectivity(...)	Returns the reflectivity and transmission of an inhomogeneously strained sample structure for a given `strain_map` and `magnetization_map` in space and time.
parallel_inhomogeneous_reflectivity(...)	Returns the reflectivity and transmission of an inhomogeneously strained sample structure for a given `strain_map` and `magnetization_map` in space and time.
distributed_inhomogeneous_reflectivity(...)	This is a stub.
calc_inhomogeneous_matrix(last_A, ...)	Calculates the product of all reflection-transmission matrices of the sample structure for every atomic layer.
calc_uc_boundary_phase_matrix(uc, last_A, ...)	Calculates the product of all reflection-transmission matrices of a single unit cell for a given strain:
conv_with_function(y, x, handle)	Convolutes the array $y(x)$ with a function given by the handle on the argument array x.
disp_message(message)	Wrapper to display messages for that class.
get_atom_boundary_phase_matrix(atom, ...[, ...])	Returns the boundary and phase matrices of an atom from Elzo formalism[Page 189, 10].
get_polarization_factor(theta)	Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of s-polarization (pol = 0), or p-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):
save(full_filename, data, *args)	Save data to file.
set_polarization(pol_in_state, pol_out_state)	Sets the incoming and analyzer (outgoing) polarization.
update_experiment(caller)	Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.
calc_atom_boundary_phase_matrix(atom, ...)	Calculates the boundary and phase matrices of an atom from Elzo formalism[Page 189, 10].
calc_reflectivity_transmissivity_from_mat... ...)	Calculates the actual reflectivity and transmissivity from the reflectivity-transmission matrix for a given incoming and analyzer polarization from Elzo formalism[Page 189, 10].
calc_kerr_effect_from_matrix(RT)	Calculates the Kerr rotation and ellipticity for sigma and pi incident polarization from the reflectivity-transmission matrix independent of the given incoming and analyzer polarization from Elzo formalism[Page 189, 10].
calc_roughness_matrix(roughness, k_z, last_k_z)	Calculates the roughness matrix for an interface with a gaussian roughness for the Elzo formal-

get_hash(***kwargs*)

Calculates an unique hash given by the energy E, q_z range, polarization states as well as the sample structure hash for relevant x-ray and magnetic parameters. Optionally, part of the `strain_map` and `magnetization_map` are used.

Parameters
****kwargs** (*ndarray[float]*) – spatio-temporal strain and magnetization profile.

Returns
hash (str) – unique hash.

set_incoming_polarization(*pol_in_state*)

Sets the incoming polarization factor for circular +, circular -, sigma, pi, and unpolarized polarization.

Parameters
pol_in_state (*int*) – incoming polarization state id.

set_outgoing_polarization(*pol_out_state*)

Sets the outgoing polarization factor for circular +, circular -, sigma, pi, and unpolarized polarization.

Parameters
pol_out_state (*int*) – outgoing polarization state id.

homogeneous_reflectivity(**args*)

Calculates the reflectivity R of the whole sample structure allowing only for homogeneous strain and magnetization.

The reflection-transmission matrices

$$RT = A_f^{-1} \prod_m \left(A_m P_m A_m^{-1} \right) A_0$$

are calculated for every substructure m before post-processing the incoming and analyzer polarizations and calculating the actual reflectivities as function of energy and q_z.

Parameters
args (*ndarray[float], optional*) – strains and magnetization for each sub-structure.

Returns

(tuple) –

- *R (ndarray[float])* - homogeneous reflectivity.
- *R_phi (ndarray[float])* - homogeneous reflectivity for opposite magnetization.

calc_homogeneous_matrix(*S*, *last_A*, *last_A_phi*, *last_k_z*, **args*)

Calculates the product of all reflection-transmission matrices of the sample structure

$$RT = \prod_m \left(P_m A_m^{-1} A_{m-1} \right)$$

If the sub-structure m consists of N unit cells the matrix exponential rule is applied:

$$RT_m = \left(P_{UC} A_{UC}^{-1} A_{UC} \right)^N$$

Roughness is also included by a gaussian width

Parameters

- **S** (`Structure`, `UnitCell`, `AmorphousLayer`) – structure, sub-structure, unit cell or amorphous layer to calculate on.

- **last_A** (*ndarray[complex]*) – last atom boundary matrix.
- **last_A_phi** (*ndarray[complex]*) – last atom boundary matrix for opposite magnetization.
- **last_k_z** (*ndarray[float]*) – last internal wave vector
- **args** (*ndarray[float], optional*) – strains and magnetization for each sub-structure.

Returns

(tuple) –

- *RT (ndarray[complex])* - reflection-transmission matrix.
- *RT_phi (ndarray[complex])* - reflection-transmission matrix for opposite magnetization.
- *A (ndarray[complex])* - atom boundary matrix.
- *A_phi (ndarray[complex])* - atom boundary matrix for opposite magnetization.
- *A_inv (ndarray[complex])* - inverted atom boundary matrix.
- *A_inv_phi (ndarray[complex])* - inverted atom boundary matrix for opposite magnetization.
- *k_z (ndarray[float])* - internal wave vector.

inhomogeneous_reflectivity(*strain_map=array([], dtype=float64)*, *magnetization_map=array([], dtype=float64)*, ***kwargs*)

Returns the reflectivity and transmissivity of an inhomogeneously strained and magnetized sample structure for a given _strain_map_ and _magnetization_map_ in space and time for each unit cell or amorphous layer in the sample structure. If no reflectivity is saved in the cache it is caluclated. Providing the `calc_type` for the calculation the corresponding sub-routines for the reflectivity computation are called:

- `parallel` parallelization over the time steps utilizing Dask
- `distributed` not implemented in Python, but should be possible with Dask as well
- `sequential` no parallelization at all

Parameters

- **strain_map** (*ndarray[float], optional*) – spatio-temporal strain profile.
- **magnetization_map** (*ndarray[float], optional*) – spatio-temporal magnetization profile.
- ****kwargs** –
  - *calc_type (str)* - type of calculation.
  - *dask_client (Dask.Client)* - Dask client.
  - *job (Dask.job)* - Dask job.
  - *num_workers (int)* - Dask number of workers.

Returns

(tuple) –

- *R (ndarray[float])* - inhomogeneous reflectivity.
- *R_phi (ndarray[float])* - inhomogeneous reflectivity for opposite magnetization.
- *T (ndarray[float])* - inhomogeneous transmissivity.

- *T_phi (ndarray[float])* - inhomogeneous transmissivity for opposite magnetization.

`sequential_inhomogeneous_reflectivity`(*strain_map*, *magnetization_map*)

Returns the reflectivity and transmission of an inhomogeneously strained sample structure for a given `strain_map` and `magnetization_map` in space and time. The function calculates the results sequentially for every layer without parallelization.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **magnetization_map** (*ndarray[float]*) – spatio-temporal magnetization profile.

Returns

(tuple) –

- *R (ndarray[float])* - inhomogeneous reflectivity.
- *R_phi (ndarray[float])* - inhomogeneous reflectivity for opposite magnetization.
- *T (ndarray[float])* - inhomogeneous transmission.
- *T_phi (ndarray[float])* - inhomogeneous transmission for opposite magnetization.

`parallel_inhomogeneous_reflectivity`(*strain_map*, *magnetization_map*, *dask_client*)

Returns the reflectivity and transmission of an inhomogeneously strained sample structure for a given `strain_map` and `magnetization_map` in space and time. The function tries to parallelize the calculation over the time steps, since the results do not depend on each other.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **magnetization_map** (*ndarray[float]*) – spatio-temporal magnetization profile.
- **dask_client** (*Dask.Client*) – Dask client.

Returns

(tuple) –

- *R (ndarray[float])* - inhomogeneous reflectivity.
- *R_phi (ndarray[float])* - inhomogeneous reflectivity for opposite magnetization.
- *T (ndarray[float])* - inhomogeneous transmission.
- *T_phi (ndarray[float])* - inhomogeneous transmission for opposite magnetization.

`distributed_inhomogeneous_reflectivity`(*strain_map*, *magnetization_map*, *job*, *num_worker*)

This is a stub. Not yet implemented in python.

Parameters

- **strain_map** (*ndarray[float]*) – spatio-temporal strain profile.
- **magnetization_map** (*ndarray[float]*) – spatio-temporal magnetization profile.
- **job** (*Dask.job*) – Dask job.
- **num_workers** (*int*) – Dask number of workers.

Returns

(tuple) –

- *R (ndarray[float])* - inhomogeneous reflectivity.

- *R_phi (ndarray[float])* - inhomogeneous reflectivity for opposite magnetization.

calc_inhomogeneous_matrix(*last_A*, *last_A_phi*, *last_k_z*, *strains*, *magnetizations*)

Calculates the product of all reflection-transmission matrices of the sample structure for every atomic layer.

$$RT = \prod_m \left(P_m A_m^{-1} A_{m-1} \right)$$

Parameters

- **last_A** (*ndarray[complex]*) – last atom boundary matrix.
- **last_A_phi** (*ndarray[complex]*) – last atom boundary matrix for opposite magnetization.
- **last_k_z** (*ndarray[float]*) – last internal wave vector
- **strains** (*ndarray[float]*) – spatial strain profile for single time step.
- **magnetizations** (*ndarray[float]*) – spatial magnetization profile for single time step.

Returns

(tuple) –

- *RT (ndarray[complex])* - reflection-transmission matrix.
- *RT_phi (ndarray[complex])* - reflection-transmission matrix for opposite magnetization.
- *A (ndarray[complex])* - atom boundary matrix.
- *A_phi (ndarray[complex])* - atom boundary matrix for opposite magnetization.
- *A_inv (ndarray[complex])* - inverted atom boundary matrix.
- *A_inv_phi (ndarray[complex])* - inverted atom boundary matrix for opposite magnetization.
- *k_z (ndarray[float])* - internal wave vector.

calc_uc_boundary_phase_matrix(*uc*, *last_A*, *last_A_phi*, *last_k_z*, *strain*, *magnetization*, *force_recalc=False*)

Calculates the product of all reflection-transmission matrices of a single unit cell for a given strain:

$$RT = \prod_m \left(P_m A_m^{-1} A_{m-1} \right)$$

and returns also the last matrices A, A^{-1}, k_z.

Parameters

- **uc** (UnitCell) – unit cell
- **last_A** (*ndarray[complex]*) – last atom boundary matrix.
- **last_A_phi** (*ndarray[complex]*) – last atom boundary matrix for opposite magnetization.
- **last_k_z** (*ndarray[float]*) – last internal wave vector
- **strain** (*float*) – strain of unit cell for a single time step.
- **magnetization** (*ndarray[float]*) – magnetization of unit cell for a single time step.
- **force_recalc** (*boolean, optional*) – force recalculation of boundary phase matrix if True. Defaults to False.

Returns

(tuple) –

- *RT (ndarray[complex])* - reflection-transmission matrix.
- *RT_phi (ndarray[complex])* - reflection-transmission matrix for opposite magnetization.
- *A (ndarray[complex])* - atom boundary matrix.
- *A_phi (ndarray[complex])* - atom boundary matrix for opposite magnetization.
- *A_inv (ndarray[complex])* - inverted atom boundary matrix.
- *A_inv_phi (ndarray[complex])* - inverted atom boundary matrix for opposite magnetization.
- *k_z (ndarray[float])* - internal wave vector.

static conv_with_function(*y*, *x*, *handle*)

Convolutes the array $y(x)$ with a function given by the handle on the argument array x.

Parameters

- **y** (*ndarray[float]*) – y data.
- **x** (*ndarray[float]*) – x data.
- **handle** (*@lamdba*) – convolution function.

Returns

y_conv (ndarray[float]) – convoluted data.

disp_message(*message*)

Wrapper to display messages for that class.

Parameters

message (*str*) – message to display.

get_atom_boundary_phase_matrix(*atom*, *density*, *distance*, *force_recalc=False*, **args*)

Returns the boundary and phase matrices of an atom from Elzo formalism[Page 189, 10]. The results for a given atom, energy, q_z, polarization, and magnetization are stored to RAM to avoid recalculation.

Parameters

- **atom** (*Atom, AtomMixed*) – atom or mixed atom.
- **density** (*float*) – density around the atom [kg/m^3].
- **distance** (*float*) – distance towards the next atomic [m].
- **force_recalc** (*boolean, optional*) – force recalculation of boundary phase matrix if True. Defaults to False.
- **args** (*ndarray[float]*) – magnetization vector.

Returns

(tuple) –

- *A (ndarray[complex])* - atom boundary matrix.
- *A_phi (ndarray[complex])* - atom boundary matrix for opposite magnetization.
- *P (ndarray[complex])* - atom phase matrix.
- *P_phi (ndarray[complex])* - atom phase matrix for opposite magnetization.

- *A_inv (ndarray[complex])* - inverted atom boundary matrix.
- *A_inv_phi (ndarray[complex])* - inverted atom boundary matrix for opposite magnetization.
- *k_z (ndarray[float])* - internal wave vector.

get_polarization_factor(*theta*)

Calculates the polarization factor $P(\vartheta)$ for a given incident angle ϑ for the case of *s*-polarization (pol = 0), or *p*-polarization (pol = 1), or unpolarized X-rays (pol = 0.5):

$$P(\vartheta) = \sqrt{(1 - \text{pol}) + \text{pol} \cdot \cos(2\vartheta)}$$

Parameters
theta (*ndarray[float]*) – incidence angle.

Returns
P (ndarray[float]) – polarization factor.

save(*full_filename*, *data*, **args*)

Save data to file. The variable name can be handed as variable argument.

Parameters

- **full_filename** (*str*) – full file name to data file.
- **data** (*ndarray*) – actual data to save.
- ***args** (*str, optional*) – variable name within the data file.

set_polarization(*pol_in_state*, *pol_out_state*)

Sets the incoming and analyzer (outgoing) polarization.

Parameters

- **pol_in_state** (*int*) – incoming polarization state id.
- **pol_out_state** (*int*) – outgoing polarization state id.

update_experiment(*caller*)

Recalculate energy, wavelength, and wavevector as well as theta and the scattering vector in case any of these has changed.

$$\lambda = \frac{hc}{E}$$
$$E = \frac{hc}{\lambda}$$
$$k = \frac{2\pi}{\lambda}$$
$$\vartheta = \arcsin\frac{\lambda q_z}{4\pi}$$
$$q_z = 2k\sin\vartheta$$

Parameters
caller (*str*) – name of calling method.

calc_atom_boundary_phase_matrix(*atom*, *density*, *distance*, **args*)

Calculates the boundary and phase matrices of an atom from Elzo formalism[Page 189, 10].

Parameters

- **atom** (Atom, AtomMixed) – atom or mixed atom.

- **density** (*float*) – density around the atom [kg/m^3].
- **distance** (*float*) – distance towards the next atomic [m].
- **args** (*ndarray[float]*) – magnetization vector.

Returns

(tuple) –

- *A (ndarray[complex])* - atom boundary matrix.
- *A_phi (ndarray[complex])* - atom boundary matrix for opposite magnetization.
- *P (ndarray[complex])* - atom phase matrix.
- *P_phi (ndarray[complex])* - atom phase matrix for opposite magnetization.
- *A_inv (ndarray[complex])* - inverted atom boundary matrix.
- *A_inv_phi (ndarray[complex])* - inverted atom boundary matrix for opposite magnetization.
- *k_z (ndarray[float])* - internal wave vector.

static calc_reflectivity_transmissivity_from_matrix(*RT*, *pol_in*, *pol_out*)

Calculates the actual reflectivity and transmissivity from the reflectivity-transmission matrix for a given incoming and analyzer polarization from Elzo formalism[Page 189, 10].

Parameters

- **RT** (*ndarray[complex]*) – reflection-transmission matrix.
- **pol_in** (*ndarray[complex]*) – incoming polarization factor.
- **pol_out** (*ndarray[complex]*) – outgoing polarization factor.

Returns

(tuple) –

- *R (ndarray[float])* - reflectivity.
- *T (ndarray[float])* - transmissivity.

static calc_kerr_effect_from_matrix(*RT*)

Calculates the Kerr rotation and ellipticity for sigma and pi incident polarization from the reflectivity-transmission matrix independent of the given incoming and analyzer polarization from Elzo formalism[Page 189, 10].

Parameters
RT (*ndarray[complex]*) – reflection-transmission matrix.

Returns
K (ndarray[float]) – kerr.

static calc_roughness_matrix(*roughness*, *k_z*, *last_k_z*)

Calculates the roughness matrix for an interface with a gaussian roughness for the Elzo formalism[Page 189, 10].

Parameters

- **roughness** (*float*) – gaussian roughness of the interface [m].
- **k_z** (*ndarray[float*) – internal wave vector.
- **last_k_z** (*ndarray[float*) – last internal wave vector.

Returns
W (ndarray[float]) – roughness matrix.

5.4.9 helpers

Functions:

make_hash_md5(obj)	**param obj** anything that can be hashed.
make_hashable(obj)	Recursive calls to elements of tuples, lists, dicts, set, and frozensets.
m_power_x(m, x)	Apply `numpy.linalg.matrix_power` to last 2 dimensions of 4-dimensional input matrix.
m_times_n(m, n)	Matrix multiplication of last 2 dimensions for two 4-dimensional input matrices.
finderb(key, array)	Binary search algorithm for sorted array.
multi_gauss(x[, s, x0, A])	Multiple gauss functions with width `s` given as FWHM and area normalized to input `A` and maximum of gauss at `x0`.
convert_cartesian_to_polar(cartesian)	Convert a vector or field from cartesian coordinates (x, y, z) to polar coordinates (r, ϕ, γ):
convert_polar_to_cartesian(polar)	Convert a vector or field from polar coordinates (r, ϕ, γ) to cartesian coordinates (x, y, z):

udkm1Dsim.helpers.**make_hash_md5**(*obj*)

Parameters
obj (*any*) – anything that can be hashed.

Returns
hash (str) – hash from object.

udkm1Dsim.helpers.**make_hashable**(*obj*)

Recursive calls to elements of tuples, lists, dicts, set, and frozensets.

Parameters
obj (*any*) – anything that can be hashed..

Returns
obj (tuple) – hashable object.

udkm1Dsim.helpers.**m_power_x**(*m*, *x*)

Apply `numpy.linalg.matrix_power` to last 2 dimensions of 4-dimensional input matrix.

Parameters

- **m** (*ndarray[float, complex]*) – 4-dimensional input matrix.
- **x** (*float*) – exponent.

Returns
m (ndarray[float, complex]) – resulting matrix.

udkm1Dsim.helpers.**m_times_n**(*m*, *n*)

Matrix multiplication of last 2 dimensions for two 4-dimensional input matrices.

Parameters

- **m** (*ndarray[float, complex]*) – 4-dimensional input matrix.
- **n** (*ndarray[float, complex]*) – 4-dimensional input matrix.

Returns

res (ndarray[float, complex]) – 4-dimensional multiplication result.

udkm1Dsim.helpers.**finderb**(*key*, *array*)

Binary search algorithm for sorted array. Searches for the first index `i` of array where `key >= array[i]`. `key` can be a scalar or a np.ndarray of keys. `array` must be a sorted np.ndarray.

Author: André Bojahr. Licence: BSD.

Parameters

- **key** (*float, ndarray[float]*) – single or multiple sorted keys.
- **array** (*ndarray[float]*) – sorted array.

Returns

i (ndarray[float]) – position indices for each key in the array.

udkm1Dsim.helpers.**multi_gauss**(*x*, *s=[1]*, *x0=[0]*, *A=[1]*)

Multiple gauss functions with width `s` given as FWHM and area normalized to input `A` and maximum of gauss at `x0`.

Parameters

- **x** (*ndarray[float]*) – argument of multi_gauss.
- **s** (*ndarray[float], optional*) – FWHM of Gaussians. Defaults to 1.
- **x0** (*ndarray[float], optional*) – centers of Gaussians. Defaults to 0.
- **A** (*ndarray[float], optional*) – amplitudes of Gaussians. Defaults to 1.

Returns

y (ndarray[float]) – multiple Gaussians.

udkm1Dsim.helpers.**convert_cartesian_to_polar**(*cartesian*)

Convert a vector or field from cartesian coordinates (x, y, z) to polar coordinates (r, ϕ, γ):

$$F_r = \sqrt{F_x^2 + F_y^2 + F_z^2}$$

$$F_\phi = \begin{cases} \arctan\left(\frac{F_y}{F_x}\right) & \text{for } F_x > 0 \\ \pi + \arctan\left(\frac{F_y}{F_x}\right) & \text{for } F_x < 0 \text{ and } F_y \geq 0 \\ \arctan\left(\frac{F_y}{F_x}\right) - \pi & \text{for } F_x < 0 \text{ and } F_y < 0 \\ 0 & \text{for } F_x = F_y = 0 \end{cases}$$

$$F_\gamma = \arccos\left(\frac{F_z}{F_r}\right)$$

where F_r, F_ϕ, F_γ are the radial (amplitude), azimuthal, and polar component of vector field $\mathbf{F}$, respectively.

Parameters

cartesian (*ndarray[float]*) – vector of field to convert.

Returns
polar (ndarray[float]) – converted vector or field.

udkm1Dsim.helpers.**convert_polar_to_cartesian**(*polar*)

Convert a vector or field from polar coordinates (r, ϕ, γ) to cartesian coordinates (x, y, z):

$$F_x = r\sin(\phi)\cos(\gamma)$$
$$F_y = r\sin(\phi)\sin(\gamma)$$
$$F_z = r\cos(\phi)$$

where r, ϕ, γ are the radius (amplitude), azimuthal, and polar angles of vector field $\mathbf{F}$, respectively.

Parameters
polar (*ndarray[float]*) – vector of field to convert.

Returns
cartesian (ndarray[float]) – converted vector or field.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

u

INDEX

D

F

G

H

I

L

M

O

P

R

S

U

V

X