Nemaktis

Release 1.4.8

Apr 25, 2024

Introduction

CHAPTER 1

Overview

Nemaktis is an open-source platform including tools for propagating and visualising optical fields in complex birefringent media such as liquid crystal (LC) layers. It includes three backends implementing advanced numerical methods for light propagation, as well as an easy-to-use high level interface in python allowing to quickly setup a simulation and visualize optical microraphs of a LC structure as in a real microscope. It goes well beyond the Jones method usually used in LC research, by accurately modeling diffraction, walk-off, focusing effects, Koehler illumination...

If you want a platform for easily comparing experimental images and numerical micrographs from simulated or theoretical birefringent structures, you are in the right place!

1.1 License

Nemaktis is released under MIT license so you can use it freely. Please cite the following publications if you use Nemaktis to prepare a figure in a scientific paper:

- G. Poy, S. Žumer, Soft Matter 15, 3659-3670 (2019)
- G. Poy, S. Žumer, Optics Express 16, 24327-24342 (2020)

1.2 Contributors

- High-level interface, ray-tracing and beam propagation backends: Guilhem Poy, Slobodan Žumer.
- Diffraction transfer matrix backend: Andrej Petelin, Alex Vasile.

1.3 Highlights

- Easy-to-use scripting interface in python
- Support for Koehler illumination setup (multiple incoming plane waves)
- Support for arbitrary number of isotropic layers around the birefringent object (e.g. glass plates).
- Support for arbitrary uniaxial media (biaxial support coming soon).

- Graphical user interface to visualize optical fields, with interactive sliders for the parameters of the microscope.

1.4 Limitations

- Paraxial propagation is assumed for the BPM backend (but the PSF of the microscope can be set with a high NA).
- No support for reflection microscopy
- No support (yet) for biaxial media

The last two limitations should be lifted in the future since the associated theoretical framework is already ready (I just need to find students or the time to code everything!). If you want to implement new features that you think could benefit the whole software, please contact me (address below)!

Concerning the paraxial approximation, I can mention that I have also developped a closed-source wide-angle beam propagation method, which can model non-linear optical systems, wide-angle deflection of light beams by birefringent structures, waveguiding... If you are interested in starting a collaboration on this closed-source software, please send me a quick message explaining the optics problem that you want to solve.

1.5 Contact

- Personal website: https://www.syncpoint.fr
- Email: guilhempoy07 [*at*] gmail [*dot*] com

CHAPTER 2

Installation

`Nemaktis` is a mixture of C++ and python codes, and has been successfully tested both on Windows 10 and Linux. The recommended way of installing and using Nematkis is through the package manager `mamba` (see next subsection).

If you are an experienced Mac user and want to become maintainer of a MacOS version of Nematkis (using mamba), please contact me (https://nemaktis.readthedocs.io/en/latest/intro/overview.html#contact). Otherwise, it is probably possible to compile and use the software by hand on other OSs than the ones mentioned above (windows 7, macOS...), provided you know what you are doing :)

2.1 With Mamba (windows or linux)

2.1.1 a. Install Miniforge

The simplest way of installing the `Nemaktis` package is through the package manager `mamba` (no compilation and no dependency to be installed by hand). I provide precompiled packages for both **Linux** and **Windows 10**, which means the following method will work on these two operating systems.

`mamba` is a package manager based on the `conda` python package manager, but with far better performance since it is written in C++. Previous versions of Nemaktis were using `conda` (or `mamba` installed on top of `conda`) but this proved to be too difficult to maintain because of these poor performances. For this reason, any Nemaktis version >=1.4.7 requires the `Miniforge` distribution to be installed locally on your computer (i.e. on your user folder), and will not necessarily work with the `Anaconda` or `Miniconda` distributions. The installation files of `Miniforge` for Windows/Linux are available at this address (be careful to choose the `Miniforge` distribution, not the `Miniforge-pypy3` distribution): https://github.com/conda-forge/miniforge#miniforge3

2.1.2 b1. (Windows) Install Nemaktis automatically

If you are a Windows 10 user and do not want to copy-paste commands in a terminal, the next step is as simple as running the following installation script

https://github.com/warthan07/Nemaktis/releases/download/v1.4.8/Install_Nemaktis-1.4.8.cmd

This script will ask for the root path of the `Miniforge` distribution installed in step a, create a special environment for `Nemaktis` named *nm* and will install everything needed in it. It will also install the python editor `Spyder` and

create a shortcut named *Spyder (Nemakis environment)* for it on your Desktop (this is necessary even if you already installed Spyder, since it has to be run from inside the mamba environment *nm*).

2.1.3 b2. (Windows/Linux) Install Nemaktis on the command line

Alternatively, if you are a Linux user or want to type the installation commands yourself (they are not very complicated after all!), open a terminal (Windows: application "Miniforge prompt" installed with Miniforge, Linux: any terminal) and type the following command:

```
mamba create -n nm -c conda-forge -c warthan07 -c anaconda -y nemaktis=1.4.8
```

(Optional) If you want to use your favourite python editor when using `Nemaktis`, you have to install and run it from the same mamba environment. You can search https://anaconda.org/ to find the associated package and installation command. For example, to install `Spyder` you just need to type:

```
mamba activate nm
mamba install -c conda-forge spyder
```

Note that when you want to run python scripts using nemaktis, the installed python editor should always be run from inside the *nm* environment. For example, to run `Spyder`, you should type in the terminal:

```
mamba activate nm
spyder
```

The advantage of step b1 is that it creates a shortcut for Spyder which automatically does this activation step for you.

2.1.4 c. (Windows/Linux) How to update

I do not recommend updating nematkis using the command `mamba update`, since I do not compile Nematkis sufficiently often for a correct update of all dependencies. In other words, running `mamba update` has the risk of breaking your mamba environment! I apologize for this current limitation, which mostly stems from my inexperience at mamba packaging with complex dependencies.

In this case, how can you safely update Nemaktis? The simplest way is to fully remove the mamba environment *nm*, either by removing the folder "envs/nm" inside the root folder of Miniforge or by opening a terminal ("Miniforge prompt" for windows, any terminal for linux) and typing:

```
mamba remove --name nm --all
```

Then, simply repeat the installation step b1 or b2 above, eventually adjusting the version number of Nemaktis (it can be obtained from https://anaconda.org/warthan07/nemaktis/files) inside the commands or script if I forgot to do it :)

Another possibility is to repeat the installation step b1/b2 with a different environment name than *nm* (for the windows script method, you need to manually edit the script file), for example *nm[version number]*. Although this method is probably fine to test new features of the software without removing the old version , it is probably not very good in the long term since each mamba environment takes a non-negligible portion of disk space.

2.2 Developper method (only linux)

If you want to be able to modify the code, but still want to enjoy the simplicity of mamba packages (no relative paths to manage, everything works as with a system package), you can build yourselves the nemaktis package for Linux:

1. Get the code of the `Nemaktis` repository:

   ```
   git clone git@github.com:warthan07/Nemaktis.git
   ```

And implements the changes that you want. For the C++ codes, compilation steps are provided in the subfolders bpm-solver and rt-solver if you want to test them locally (in which case you will have to install yourselves the dependencies listed in each CMakeLists).

2. In a terminal, go to the subfolder **conda_recipe** of the `Nemaktis` repository and type the following (the first line is only needed if the build environment does not exist yet):

```
mamba create -n build -c conda-forge -c anaconda -y boa conda-verify anaconda-
↪client
mamba activate build
```

3. Run the following command, which will create a sub-environment, install all dependencies listed in meta.yaml, and compile/package everything (it should take between 5 and 10 minutes):

```
conda mambabuild . -c conda-forge -c anaconda
```

4. Once the package is built, you can install it in your current environment by typing:

```
conda install -c conda-forge -c anaconda -c ${CONDA_PREFIX}/conda-bld/ nemaktis
```

CHAPTER 3

Microscopy model for Nemaktis

We present here the theoretical model of microscopy that is at the core of Nemaktis. A few interactive graphics are provided in order to better understand important concepts. The javascript code for these interactive examples can be found in an ObservableHQ notebook.

3.1 1. General description

In a real transmission and/or reflection microscope, objects are imaged using a combination of lighting systems and lenses. The path of light in such microscopes can always be decomposed in three steps:

1. Light propagates from the illuminating source to the object through the illumination optical setup.
2. Light is transmitted through (or reflected from) the object.
3. Light transmitted or reflected from the object propagates through the microscope objective and form an observable image in a target imaging plane.

The case of spectrally-extended lighting (e.g. white light lamp) can be easily covered by projecting on an appropriate color space the final images formed by the different wavelengths of the lamp spectrum. In Nemaktis, this is done internally after repeating the imaging simulations for all the wavelengths in a user-defined array approximating the lamp spectrum. For more details on the color space projection method, see Color conversion in the documentation of `dtmm`, one of the backend used in Nemaktis. Here, we consider for simplicity's sake a simple microscopy model based on lighting with a single wavelength. We describe in the next sections the physical mechanisms behind the three steps introduced above, as schematized below in a simplified representation of our virtual microscope in transmission mode:

We also provide an additional Sec. 5 to explain the modeling of optical elements for polarized micrographs, which are mostly ignored in Sec. 2-4.

3.2 2. Koehler illumination setup

The first propagation step is the illumination of the object by the light source. The standard illumination setup used in most microscopes is called the Koehler illumination setup (introduced by August Koehler in 1893), and has the advantage of allowing a uniform lighting even with non-uniform light source. In short, it allows to map each point

of the light source to a single uniform plane wave incident on the object with a certain angle; the maximum angle of incidence for the plane waves is set by an aperture called the **condenser aperture**, thus the set of plane waves incident on the object all have wavevectors included inside a cone of illumination whose opening is set by the condenser aperture.

In addition, a **field aperture** allows to control the size of the lighting spot on the object. In practice, the mapping between points on the light source and incident plane waves is only approximate due the imperfection of the optical elements and the wave nature of light, especially near the boundary of the lighting spot on the object. However, this is not a problem in Nemaktis since we always assume that the lighting spot is much bigger than the typical size of the observed object, thus justifying the representation in terms of incoming plane waves.

In order to better understand how this illumination setup works, an interactive example is provided below, where the reader can dynamically adjust the sliders for opening/closing the field and condenser apertures:

A correctly assembled Koehler illumination setup has the following properties:

- The field aperture is in the back focal plane of the lamp collector lens.
- The condenser aperture is in the front focal plane of the condenser lens.
- The image of the lamp filament through the lamp collector lens is in the same plane as the condenser aperture.
- The image of the field aperture throught the condenser lens is is in the same plane as the object.

We emphasize that the lamp filament is always spatially incoherent, thus the different incident plane waves cannot interfer between themselves. This means that the final image in the microscope is always obtained by summing-by-intensity the individual images formed by each incident plane waves. In real life, there is always an infinite number of plane waves incident on the object, but in the computer one must choose an approximate discrete set of plane waves. In Nemaktis, the set of incoming plane waves is chosen to have the following wavevectors (assuming that the third coordinate correspond to the main propagation axis in the microscope):

$$\vec{k}^{(k,l)} = k_0 \begin{pmatrix} q^{(k)} \cos\theta^{(k,l)} \\ q^{(k)} \sin\theta^{(k,l)} \\ \sqrt{1 - \left[q^{(k)}\right]^2} \end{pmatrix}$$

where we defined $k_0 = 2\pi/\lambda$ with λ the wavelength in empty space and:

$$q^{(k)} = \frac{k}{N_r - 1}\mathrm{NA}_{\max}, \qquad k = 0 \cdots N_r - 1$$
$$\theta^{(k,l)} = \frac{\pi l}{3k}, \qquad l = 0 \cdots 6k$$

Here, $\mathrm{NA}_{\max} = \sin\psi_{\max}$ (with $\psi_{\max}$ the maximal angle of opening of the wavevectors) is the maximal numerical aperture of the Koehler illumination setup, and N_r correspond to the number of discretization steps in the radial direction. This choice of wavevectors correspond to a standard discretization of a circular aperture in the transverse plane, which can be interactively visualized below by adjusting the sliders for N_r and NA.

In Nematkis, this mesh is fully characterized by the two parameters, $\mathrm{NA}_{\max}$ and N_r, and has a total number of points of $1 + 3N_r(N_r - 1)$. Since this mesh is (weakly) nonuniform, we use a tailored integration rule to recombine the microscope images in the final imaging plane, which also have the benefit of being able to dynamically change the numerical aperture of the condenser between 0 and $\mathrm{NA}_{\max}$ in the microscope's user interface.

To conclude this section, we mention the final approximation made in Nemaktis for the illumination setup: we assume that all the incoming plane waves have the same intensity. This approximation is probably not true in a real microscope, but has the advantage of always yielding rotationally invariant images when observing rotationally invariant objects (e.g. isotropic droplets) with natural light, as empirically observed in most microscopes. In any case, the goal of our simple model of Koehler illumination setup for Nematkis is only to provide a qualitatively correct description of the "smoothing" effect (due to the increasing number of incident planewaves) of a real microscope when opening the condenser aperture.

3.3 3. Transmission/Reflection of light inside the object

The next step is the transmission or reflection of light inside the object. In Nemaktis, we exlude metallic surfaces and metamaterials, and assume that the object is fully transparent (no absorption), time-invariant (no fluctuations leading to light scattering), and can be represented by a permittivity tensor field $\epsilon_{ij}(x, y, z)$ which is discretized on a 3D cartesian mesh. The transmission and reflection of light in such objects is modeled by the well-known wave equation for the time-harmonic Maxwell electric field $\vec{E}(x, y, z) \exp[-ik_0 ct]$ (with c the light speed and k_0 the wavevector in empty space):

$$\vec{\nabla} \times \vec{\nabla} \times \vec{E} - k_0^2 \bar{\bar{\epsilon}} \vec{E} = 0$$

There exists general schemes for fully solving this equation (most notably the Finite-Difference-Time-Domain method), but they are computationally very intensive, which is why we resort to less expansive (but approximate) methods in Nemaktis. More specifically, we propose three "backends" which allows to propagate the optical fields inside the object and are described in the following subsections. As a general rule, each backend provides a set of mappings between each incoming plane waves (see Sec. 2) and output optical fields defined on the output object plane (see figure in Sec. 1). In the current version of Nemaktis, only transmitted optical fields are considered as "output", support for reflected fields will come later.

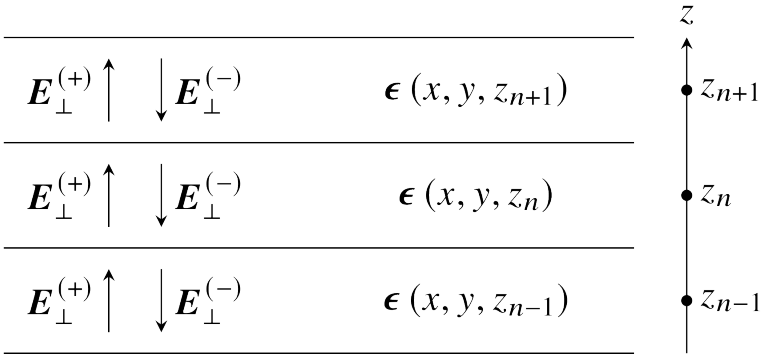
3.3.1 3.1 The beam-propagation backend (bpm-solver)

This backend correspond to a subset of a generalized framework for beam propagation in general birefringent medium that I developed (see citation below). This generalized framework relies on a minimal set of physical assumptions (most notably a relatively small refractive index contrast $\Delta n < 0.4$ inside the object) and admits two Beam Propagation Methods (BPM):

- Wide-angle BPM, which can accurately propagate optical fields up to deviation angles of 20-30°.
- Paraxial BPM, which can accurately propagate optical fields up to deviation angles of 5-10°.

The second version of BPM is especially suite for microscopy applications, since in most microscopes (exluding confocal microscopes with high numerical aperture objective) only the paraxial components of light contributes to the final image. In our open-source package Nemaktis, only paraxial BPM is included as a backend for microscopy, but we are open to new collaborations on our closed-source wide-angle BPM for advanced uses (nonlinear optics, modeling of complex photonics devices, steering of light using birefringent structures...).

At its core, the beam propagation works by decomposing the optical (electric) field $\vec{E}$ into forward and backward propagating fields inside a series of layers approximating the full permittivity profile $\bar{\bar{\epsilon}}(x, y, z)$:



The permittivity tensor is assumed to be stepwise constant along z (the main axis of propagation inside the microscope) but is allowed to have arbitrary variations in the transverse directions x and y. Our beam propagation framework correspond to a set of equations allowing to propagate the optical fields inside each layers (including diffraction and beam walk-off effects due to the nonuniformity of the optical and permittivity fields) and transfer fields through the discontinuity interface between each layer. In Nemaktis, we assume smooth variations of the permittivity along z and

therefore only propagates forward-propagating fields using the following formula:

$$\mathbf{E}(z_{n+1}) = \mathbf{U} \cdot \mathbf{E}(z_n),$$

where $\mathbf{E}(z_n)$ is a huge vector containing all degree-of-freedoms for the optical fields in the transverse plane $z = z_n$ and $\mathbf{U}$ is an evolution operator which can be written as an easy-to-compute product of exponential of sparse matrices representing differential operators on 2D meshes. The evolution operator $\mathbf{U}$ is directly derived from Maxwell equations with a few mathematical assumptions (small index contrast and paraxiality of fields) and can be applied in a very efficient way (complexity $O(N)$, with N the total number of degree-of-freedoms for the computational mesh).

Since we only take into account forward-propagating fields, reflection microscopy is currently not supported in Nemaktis, but we hope to implement this feature in the future since we already derived the associated theoretical framework. Note that internally, each imaging simulation includes a lot of different paraxial BPM sub-simulations for each incident plane-wave, source wavelength, and input polarisations. Using the same notation as in Sec. 2 and assuming a single input wavelength, the incident optical fields for all these sub-simulations are written as:

$$\vec{E}^{(k,l,m)}(x,y) = \exp\left\{ i \left[x k_x^{(k,l)} + y k_y^{(k,l)} \right] \right\} \vec{u}_m,$$

where k and l are the indices for the input wavevector $\vec{k}^{(k,l)}$ and $\vec{u}_m$ ($m = 1, 2$) is an orthogonal basis for the input polarisation. The use of repeated simulations based on orthogonal polarisations allows the caching of relevant data for efficiently simulating arbitrary polarized optical micrographs (using polariser, analyzer, waveplate...), with a dynamic real-time adjustment of the associated parameters (e.g. polariser and analyzer angle) in the graphical user interface.

Readers interested in our beam propagation framework can read the associated publication:

[G. Poy and S. Žumer, Optics Express 28, 24327 (2020)]

3.3.2 3.2 The diffraction transfer matrix backend (dtmm)

This backend correspond to a python package originally written by a colleague, Dr. Andrej Petelin, and that we decided to include in Nemaktis for easy comparison between different approaches of electromagnetic field propagation. At its core, the diffractive transfer matrix method (DTMM) of Dr. Petelin is conceptually very close to the beam propagation backend presented above in Sec. 3.1: the permittivity tensor field representing the object is also split in a series of birefringent slabs, evolution operators are similarly used to propagate the fields inside the slabs, and continuity equations are used to transfer the fields between the layers. The difference between DTMM and our BPM framework mainly lie in the way that the evolution operators are calculated: in DTMM, this evolution operator is calculated with a clever heuristic combination of the Berreman method and diffraction transfer matrix applied in Fourier space. The Berreman method was originally developped for the calculation of transmitted and reflected light in layered system (permittivity tensor field independent from x and y) and neglects diffraction (the redistribution of optical energy due to non-uniformity of the optical and permittivity fields); in DTMM, the evolution operators derived by Berreman are combined with a powerful treatment of diffraction in Fourier space based on local mode grouping, thus allowing to take into account variations of fields in the x and y directions.

Since this is a Fourier-based method, its complexity is $O(N \log [N/N_z])$ with N the total number of mesh points and N_z the number of layers. It is also based on a user-defined parameter allowing to define the accuracy of diffraction in the simulation:

- low value of this parameter provide quick (but inacurate) simulations with faster running times than BPM on relatively small meshes (for big meshes, the logarithmic complexity of dtmm kicks in and DTMM can be slower than BPM);
- high value of this parameter provide accurate simulations (computational errors similar than the ones obtained with BPM) with generally slower running times than with BPM.

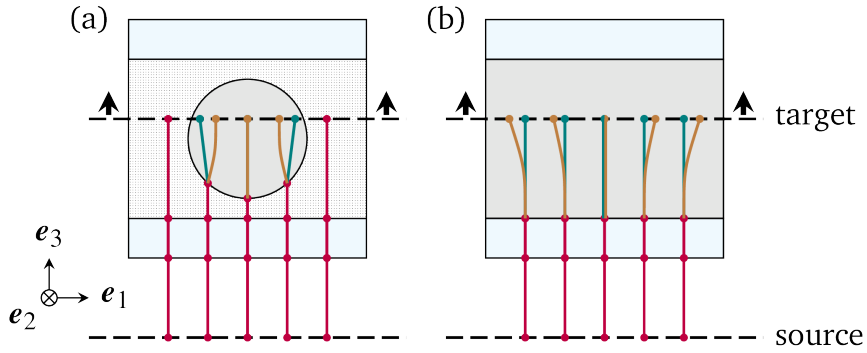
In short, DTMM is the perfect backend if you want to quickly try imaging simulations without worrying too much about the accuracy, whereas BPM is more suited for efficient accurate simulations on arbitrary big meshes (provided that enough random-access-memory is available!).

In Nemaktis, DTMM is closely integrated in the high-level python package allowing to run imaging simulations, but we emphasize that DTMM also has a dedicated python package with advanced features such as iterative algorithms for the calculation of reflected fields (a feature which is currently missing in the BPM backend). If you absolutely need these advanced features, please directly use the python package `dtmm` (which is automatically installed with Nematkis!):

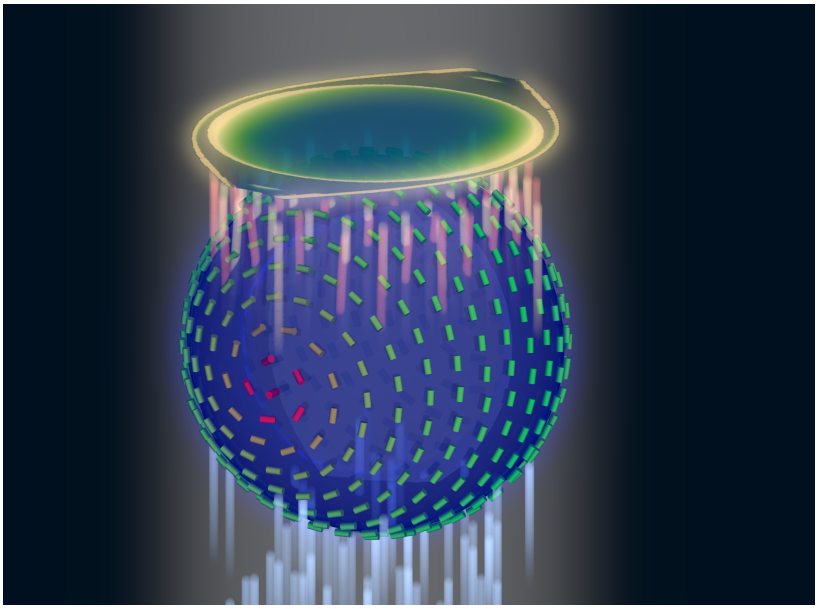
[DTMM: a Diffractive Transfer Matrix Method]

3.3.3 3.3 The ray-tracing backend (rt-solver)

This backend relies on the so-called geometrical optics approximation and works by decomposing the incoming plane wave in a series of light rays, which are propagated through the object using Hamiltonian ray-tracing equations. The validity of this method is quite restricted: the permittivity tensor field $\bar{\bar{\epsilon}}(x, y, z)$ must correspond to a uniaxial birefringent medium whose optical axis is smoothly varying in space, with typical variation lengths much bigger than the wavelength of light. It also necessitates some tweaking in order to correctly reconstruct the optical fields on a cartesian mesh (since the ray-tracing method only gives optical fields along rays, which can be deflected by the object).



Since this method cannot be really used as a "blackbox" simulation tool, it is provided as such (i.e. as a low-level C++ code) without any integration in the easy-to-use high-level python interface in Nemaktis. Nevertheless, this method can still be useful to get some physics insight on how light is deflected in particular systems (see for example [J. Hess, G. Poy, J.-S. B. Tai, S. Žumer and I. I. Smalyukh, Phys. Rev. X 10, 031042 (2020)] or to make attractive scientific visualizations like the image below (cover of the paper presenting our method, which is cited below):



Readers interesting with further details on our ray-tracing method can refer to the following publication:

[G. Poy and S. Žumer, Soft Matter 15, 3659 (2019)]

3.4 4. Imaging of the object

The final step of light propagation inside the microscope is the proper imaging of the object using the light coming from the object (i.e. the output of the backends presented in Sec. 3). In a real microscope, this is done by combining an objective with an eyepiece lens allowing to project on the user's retina the optical fields coming from a plane aligned inside the object. As a general rule, this system is always associated with two planes: the **focusing plane** which is roughly aligned with the object, and the **imaging plane** in which the final image is formed. Since this is a linear optical system, the optical fields on both planes are always related by a linear transform:

$$\vec{E}\left[\vec{r}^{(\mathrm{im})}\right] = \int \bar{\bar{G}}\left[\vec{r}^{(\mathrm{im})}, \vec{r}^{(\mathrm{foc})}\right] \vec{E}\left[\vec{r}^{(\mathrm{foc})}\right] \mathrm{d}^2\vec{r}^{(\mathrm{foc})}$$

where $\vec{r}^{(\mathrm{im})}$ ($\vec{r}^{(\mathrm{foc})}$) correspond to coordinates on the imaging (focusing) plane and $\bar{\bar{G}}$ is called the point-spread-function (PSF) of the imaging system. The actual expression of the PSF depends on the implementation of the imaging lens, but in general it acts as a low-pass filter because it is aperture-limited, i.e. one cannot observe details below the diffraction limit (typical width of a detail smaller than the wavelength). In Nemaktis, we use a very simple model of imaging system based on a single objective lens and the imaging/focusing planes placed at distance $2f$ on each side of the lens (with f the focal length of the objective). We assume that the objective is an ideal thin-lens, which allows us to obtain a very simple form of the linear transform above in the transverse Fourier space (see details in [J. W. Goodman, *Introduction to Fourier optics*, Roberts & Company Publishers (2005)]):

$$\tilde{\vec{E}}\left[\vec{k}_\perp, z^{(\mathrm{im})}\right] = \Pi\left[\frac{\left|\vec{k}_\perp\right|}{2k_0\mathrm{NA}}\right] \tilde{\vec{E}}\left[\vec{k}_\perp, z^{(\mathrm{foc})}\right]$$

where NA is the numerical aperture of the objective, Π is the rectangular function ($\Pi(u)$ is equal to 1 if $|u| < 0.5$, else it it equal to 0), and a tilde indicate a partial Fourier transform along the x and y coordinates (associated with a Fourier frequency $\vec{k}_\perp$). Note that this formula neglects the reversal of the image due to the negative magnification of a single converging lens; in practice, this can be easily remedied by adding a second lens (as in a real microscope) or by reversing the axes' orientations in the imaging plane, in which case the formula above is perfectly valid.

The formula above shows that Fourier components with $\left|\vec{k}_\perp\right| \geq k_0\mathrm{NA}$ are filtered out by the objective while Fourier components with $\left|\vec{k}_\perp\right| < k_0\mathrm{NA}$ are preserved as such, which indeed corresponds to a low-pass filter. However, this formula is insufficient to completely model our imaging system since the **object plane** (which we define as the output plane of the object, i.e. the output of the backends presented in Sec. 3) can be slightly shifted with respect to the focusing plane: in a real microscope, this shift is usually controled by a knob allowing to set the vertical position of the sample with respect to the objective lens. Therefore, we need to propagate the fields from the object plane to the focusing plane before applying the formula above. Since this propagation step happens in free space with $\epsilon = 1$, this can be done by exactly solving Helmoltz equation in Fourier space:

$$\tilde{\vec{E}}\left[\vec{k}_\perp, z^{(\mathrm{foc})}\right] = \exp\left\{i\left[z^{(\mathrm{foc})} - z^{(\mathrm{obj})}\right]\sqrt{k_0^2 - \vec{k}_\perp^2}\right\} \tilde{\vec{E}}\left[\vec{k}_\perp, z^{(\mathrm{obj})}\right]$$

The final image on the imaging plane is defined as the squared amplitude of $\vec{E}\left[\vec{r}^{(\mathrm{im})}\right]$, which can be calculated from the two formulas above via the Fast-Fourier-Transform algorithm. To get an idea on how the numerical aperture of the objective and the position of the object plane affect the final image, we provide a simple interactive example showing how the image of a perfect circular mask is distorted through the imaging system:

3.5 5. Optical elements for polarized optical micrographs

In Sec. 2-4, we mostly focused on the general principles of microscopy and neglected the presence of optical elements such as polarisers and waveplates, which play an important role in polarised optical microscopy. In this section,

we introduce the method used in Nemaktis for taking into account these optical elements in the calculation of the final images, which are usually called polarised optical micrographs (POM). Nemaktis support two classes of optical elements for polarised optical microscopy: polarisers/analysers which allows us to project the light polarisation on a single axis, and waveplates which introduce a given phase shift between two given orthogonal polarisation axes. The disposition of these elements in our virtual microscope model is schematized below. In a real microscope, the exact disposition of these elements may be a bit different (they are often directly embedded inside the illumination/imaging setups) but we will see in a moment that this does not change much for the calculation of POMs.

3.5.1 5.1 Calculation of natural light optical micrographs

Let us start with the simplest optical setup possible, without any polarisers or waveplates. Based on Sec. 2-4, the mapping between incoming plane waves and final optical fields on the imaging plane may be described with a set of special matrices:

$$\bar{\bar{T}}_{\mathrm{obj}}^{(k,l)} = \begin{pmatrix} \left[P \star \vec{E}_{\mathrm{out}}^{(k,l,1)}\right] \cdot \vec{u}_1 & \left[P \star \vec{E}_{\mathrm{out}}^{(k,l,2)}\right] \cdot \vec{u}_1 \\ \left[P \star \vec{E}_{\mathrm{out}}^{(k,l,1)}\right] \cdot \vec{u}_2 & \left[P \star \vec{E}_{\mathrm{out}}^{(k,l,2)}\right] \cdot \vec{u}_2 \end{pmatrix},$$

where $\vec{E}_{\mathrm{out}}^{(k,l,m)}$ correspond to the output transverse optical field of one of the backend in Sec. 3 for an incoming plane wave $\vec{E}_{\mathrm{in}}^{(k,l,m)}$, $\vec{u}_m$ ($m = 1, 2$) correspond to an orthogonal basis of polarisations in the transverse plane, and the operation $P\star$ correspond to a convolution with the linear filter P representing the full imaging system, including the point-spread-function of the objective and the propagation from the output object plane to the the focusing plane (see Sec. 4).

So why use this complicated representation in terms of matrices? The advantage is that it allows to easily calculate the final optical fields for an arbitrary input polarisation $\vec{v}$ (not simply $\vec{u}_1$ and $\vec{u}_2$) by multiplying the matrix $\bar{\bar{T}}_{\mathrm{obj}}^{(k,l)}$ with the vector $\vec{v}$. This is very similar to the classical Jones calculus, except here the entries of the 2x2 matrices are not scalars but rather scalar fields (which can depends on the transverse coordinates and can be submitted to convolution operation including diffraction effects). Note that this representation is accurate only when the same polarisation basis can be used for all incoming plane waves. This is true only in the paraxial regime of propagation, where the longitudinal components of the polarisation can be neglected (second order in the angle between the wavevector and the main propagation axis z). Since we already assumed paraxial propagation in Sec. 2-4, we can therefore assume that our transfer matrix representation is consistent and accurate.

We also assume that the illuminating source is unpolarised and that the detector in the imaging plane is polarisation-independent: it simply measures the squared amplitude of the transverse optical field. This means that the final image associated with the incoming wavevector $\vec{k}^{(k,l)}$ can be calculated as (up to a constant multiplicative factor):

$$I^{(k,l)} = \int_0^{2\pi} \left| \bar{\bar{T}}_{\mathrm{obj}}^{(k,l)} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \right|^2 \frac{\mathrm{d}\theta}{\pi}$$

A direct calculation shows that we do not even need to perform an integration, we can simply sum the squared amplitude of the transfer matrix entries:

$$I^{(k,l)} = \sum_{m=1}^{2} \sum_{n=1}^{2} \left| \left[\bar{\bar{T}}_{\mathrm{obj}}^{(k,l)}\right]_{mn} \right|^2$$

3.5.2 5.2 Calculation of polarised optical micrographs

Now, how do we generalize the calculation of the previous subsection by including any combination of optical elements for polarised microscopy? Let us introduce the usual Jones matrices $\bar{\bar{T}}_{\mathrm{pol}}$, $\bar{\bar{T}}_{\mathrm{an}}$ and $\bar{\bar{T}}_{\mathrm{wp}}$ respectively associated with a polariser, analyser or waveplate. The expression of the transfer matrix for a polariser or analyser only depends on the angle $\psi_{\mathrm{pol,an}}$ of the optical element axis with respect to $\vec{u}_1$ in the transverse plane (horizontal axis in Nemaktis):

$$\bar{\bar{T}}_{\mathrm{pol,an}} = \begin{bmatrix} \cos^2\psi_{\mathrm{pol,an}} & \cos\psi_{\mathrm{pol,an}} \sin\psi_{\mathrm{pol,an}} \\ \cos\psi_{\mathrm{pol,an}} \sin\psi_{\mathrm{pol,an}} & \sin^2\psi_{\mathrm{pol,an}} \end{bmatrix}$$

The expression of the transfer matrix for a waveplate depends on the angle ψ_{wp} of the fast axis of the optical element with respect to $\vec{u}_1$ and the phasor $\eta = \exp\left[i\Gamma/2\right]$, with Γ the retardance of the waveplate:

$$\bar{\bar{T}}_{\mathrm{wp}} = \begin{bmatrix} \eta^* \cos^2\psi_{\mathrm{wp}} + \eta \sin^2\psi_{\mathrm{wp}} & (\eta^* - \eta)\cos\psi_{\mathrm{wp}}\sin\psi_{\mathrm{wp}} \\ (\eta^* - \eta)\cos\psi_{\mathrm{wp}}\sin\psi_{\mathrm{wp}} & \eta^* \sin^2\psi_{\mathrm{wp}} + \eta\cos^2\psi_{\mathrm{wp}} \end{bmatrix}$$

Nemaktis supports three different kinds of waveplates:

- Achromatic quarter-wave plate: $\Gamma = \pi/2$ independently of the wavelength;
- Achromatic half-wave plate: $\Gamma = \pi$ independently of the wavelength;
- Tint-sensitive full-wave plate: $\Gamma = 2\pi\left[0.54/\lambda\right]$, where λ is the wavelength in μm; the advantage of this waveplate is that it allows the visualization of in-plane molecular angular deviation as color shifts when illuminating an inhomogenenous birefringent sample with white light.

Now that this set of transfer matrices is introduced, the calculation of the final POM images is really simple:

- Multiply right to left the transfer matrices associated by each elements of the microscope in the same order that they are crossed by the illumination beam, and store the result in a global transfer matrix $\bar{\bar{T}}_{\mathrm{tot}}^{(k,l)}$. For example, if the setup includes a polariser, the object, a waveplate and an analyser, the total transfer matrix associated with the wavevector $\vec{k}^{(k,l)}$ is:

$$\bar{\bar{T}}_{\mathrm{tot}}^{(k,l)} = \bar{\bar{T}}_{\mathrm{an}}\bar{\bar{T}}_{\mathrm{wp}}\bar{\bar{T}}_{\mathrm{obj}}^{(k,l)}\bar{\bar{T}}_{\mathrm{pol}}$$

- Calculate the final image as in the last subsection by summing the squared amplitude of each components of the total transfer matrix:

$$I^{(k,l)} = \sum_{m=1}^{2}\sum_{n=1}^{2}\left|\left[\bar{\bar{T}}_{\mathrm{tot}}^{(k,l)}\right]_{mn}\right|^2$$

The validity of our method is again ensured by our assumption of paraxial propagation:

- Since in this regime of propagation the operation $P\star$ representing the imaging setup is polarisation-independent, it can be commuted with any operation on the polarisation state (such as the transfer matrices introduced above); this is why the real position of the waveplate and analyser inside the imaging setup of a real microscope do not matter in our simple and ideal model of microscopy.
- The transfer matrices of the polariser/analyser and waveplate, as introduced above, do not depend on the wavevector of the incoming plane wave, which is not true for wide-angle incoming plane waves.

CHAPTER 4

Accuracy and efficiency of Nemaktis' backends

We present here accuracy and efficiency benchmarks for the main two backends of Nemaktis: DTMM and BPM. The javascript code for the interactive graphs of this page can be found in an ObservableHQ notebook

4.1 1. Introduction

As explained in [Transmission/Reflection of light inside the object], the two main backends of Nemaktis (DTMM and BPM) propagate optical fields through a series of inhomonogeneous birefringent layers based on different formulation of the evolution operator of Maxwell equations. Both schemes include diffraction effects due to inhomogeneities in the optical and/or permittivity fields. Since analytical solutions of Maxwell equations are readily available only on homogeneous birefringent media, we propose here to evaluate the accuracy of both schemes by simulating the diffraction of a highly-focused Gaussian beam in a uniform uniaxial crystal. This has the advantage of evaluating in one simulation the propagation accuracy of many different plane waves since a focused Gaussian beam spans a wide-area in the transverse Fourier plane.

However, the reader should keep in mind that "real-life" computational errors of simulations on inhomogeneous birefringent media can be higher than the ones presented here, since they can also include contributions from permittivity-based diffraction (e.g. the boundary of a birefringent droplet, or sharp variations of the optical axis). In general, one should always try to adapt the mesh resolution in order to resolve the finest details of the permittivity tensor field and get the best possible accuracy. Note that discontinuities of permittivity are accurately modeled in Nemaktis if they are positioned midway between two mesh points, which is why fine meshes are needed to model complex boundaries.

Our methodology to calculate the computational error is as follows:

1. Define an input optical field with polarisation $\vec{u}$:

$$\vec{E}(\vec{r}_\perp, z=0) = \exp\left[-\frac{|\vec{r}_\perp|^2}{2w^2}\right]\vec{u}$$

2. Propagate the optical field on a computational box with length L for each side.
3. Calculate the mean error associated with all Fourier components relevant for microscopy simulations (i.e. paraxial components):

$$\epsilon = \left\langle \frac{\left| \tilde{\vec{E}}\left[\vec{k}_\perp, z\right] - \tilde{\vec{E}}_{\text{exact}}\left[\vec{k}_\perp, z\right] \right|}{2\left| \tilde{\vec{E}}_{\text{exact}}\left[\vec{k}_\perp, z\right] \right|} \right\rangle_{\theta(\vec{k}_\perp)<\theta_m}$$

In the definition above, a tilde indicate a partial Fourier transform along the x and y coordinates (associated with a Fourier frequency $\vec{k}_\perp$), $\theta(\vec{k}_\perp)$ is the the angle between $\vec{k}_\perp$ and the main direction of propagation z, and the bracket indicate a constrained average on paraxial Fourier frequencies and z coordinates. The exact solution $\tilde{\vec{E}}_{\text{exact}}\left[\vec{k}_\perp, z\right]$ is obtained directly from an eigenmode decomposition of Maxwell equations in transverse Fourier space, which can be done analytically since we assumed a uniform optical axis (see python script propagate_fields.py). The maximum angle of propagation $\theta_m \approx 24$ was choosen based on the typical numerical aperture $\sin\theta_m = 0.4$ of microscope objectives.

Note that this definition of the error is roughly independent from the choice of input profile, since it basically measure the relative error made on phase evolution in Fourier space (hence the factor 2: a phase error of π is associated with a maximal error of 1). However, it still necessitates a focused profile in order to get non-negligible high-frequency components and avoid division by zero in the definition of ϵ.

In addition to the calculation of the error, we also sytematically save the computational time associated with one calculation. We chose a waist $w = 2$ µm and a mesh length $L = 10$ µm in all the numerical experiments of this page, and vary the mesh density and choice of backends (BPM or DTMM(D) schemes, where D represents the "diffraction" parameter). You can experimentally test the accuracy of our schemes using the scripts provided in the benchmarks folder of the GitHub repository

4.2 2. Results with 2D meshes

We present first our results on 2D meshes (i.e. mesh with one transverse direction and one longitudinal direction z associated with the main direction of propagation of the Gaussian beam). Below, we plot the computational error as a function of the mesh spacing for different choice of backends:‘

The saturation of the computational mesh on very fine mesh can be simply understood from the fact that all backends in Nemaktis are approximate and not exact, even with an infinite number of degree of freedom. However, they still allows to get very good computational error, which can be below 1% for the BPM, DTMM(5) and DTMM(7) backends. The inacurracy of the DTMM(1) and DTMM(3) backends is due to the fact that these backends are unable to correctly propagate tilted plane waves (i.e. high frequency transverse Fourier modes), so if diffraction is not negligible in the system you want to study (tight beam profile or birefringent structures with details of typical length 1-10 µm) you should avoid these schemes.

As can be observed, the BPM backend allows to get the best possible accuracy provided a sufficiently fine mesh is used. However, the DTMM schemes have a very interesting property: they are less sensitive to the mesh spacing than the BPM scheme. This means that you will always get a smaller computational error on small meshes by using high-order DTMM schemes instead of the BPM accuracy.

But accuracy is not the only interesting parameter here, since the running times of these simulations widely vary when changing the backend and/or mesh spacing. To understand how all these parameters are linked, we provide below an interactive bar chart allowing to quickly visualize the error and running times of each backend for a given mesh spacing. Since 2D simulations are computationally not very intensive, the running times were evaluated on a 6 years old laptop with a processor i7-4600M (4 threads).

Not very surprinsingly, the inacurate DTMM(1) and DTMM(3) backends are also the fastest. Basically, these low-order DTMM schemes correspond to Jones-like calculus with a fast-but-inacurate treatment of diffraction, which is why their computational error is high due to the presence of high-frequency Fourier modes in these simulations. But if you know in advance that diffraction in your system is negligible (for example if the optical axis vary over lengths much bigger than the wavelength along directions orthogonal to the main axis of propagation), these schemes are a really good choice since they are very fast and can still be reasonably accurate for propagating low-frequency Fourier modes.

As for the BPM, DTMM(3) and DTMM(5), it can be observed that the DTMM schemes wins the time race on small meshes, while the BPM schemes is the fastest (and most accurate) on big meshes. This can be interpreted from the complexity of the numerical algorithm of each backends: the DTMM(D) backend has a $O\left(D^{(d-1)}N\log\left[N/N_z\right]\right)$ complexity while the BPM backend has a better linear complexity $O(N)$, with d the dimensionality of the mesh, N the total number of mesh point and N_z the number of points along the z axis); therefore, it is not really surprising that the DTMM schemes gets penalized in terms of running times for high diffraction parameter D or points number N.

4.3 3. Results with 3D meshes

We now turns our focus to 3D meshes (i.e. meshes with two transverse directions and one longitudinal direction z associated with the main direction of propagation of the Gaussian beam). Results are qualitatively similar than for 2D meshes, except now the DTMM(7) backend is always the most accurate scheme, whatever the mesh spacing. Nevertheless, the DTMM(5) and BPM backends still manage to get relatively good computational error of $\sim 1\%$ on sufficiently fine mesh.‘

However the running times of DTMM backends vs BPM backend are vastly different than in the 2D case, as expected from the $D^{(d-1)}$ factor in the complexity of DTMM backends (see above). Since 3D simulations are computationally intensive, the results of the interactive bar chart below were obtained on a recent desktop computer with processor i7-7800X (12 threads).

This time, the BPM backend is practically always faster than DTMM schemes (only the DTMM(1) can be faster than BPM on fine meshes), while having a very good computational error for most mesh spacings. In particular, the very accurate DTMM(5) and DTMM(7) schemes necessitates 4-8 times longer running times than the BPM scheme. As a consequence, we recommend to use the DTMM schemes on 3D meshes only when you want a fast simulation method without accurate diffraction (DTMM(1) backend) or a very accurate but very slow simulation (DTMM(7) backend). For all other case of applications, the BPM backend provide a reliable and accurate simulation scheme whatever the size of the computational mesh.

CHAPTER 5

Tutorial

This tutorial provides a hands-on introduction to the python package `nemaktis`. You will learn the different ways of creating permittivity field data, how to define the sample geometry and material constants, and how to propagate and visualise optical fields.

First of all, open your favorite text/code editor and create a new python file (which we will call `script.py` in the following). The script can be tested at any moment in a terminal on condition that the conda environment in which you installed `nemaktis` is activated (`conda activate [environment name]`):

```
cd [path to your script]
python script.py
```

Alternatively, you can work interactively with ipython (which must be run from a terminal in which the conda environment for `nemaktis` is activated).

5.1 Defining the optical axes

Before starting using `nemaktis`, we of course need to import the associated python package. We will also import numpy, which will be needed to define arrays:

```
import nemaktis as nm
import numpy as np
```

Next, we need to define the permittivity tensor inside the sample, which is always defined on a regular cartesian mesh. As a general rule, the permittivity tensor in a non-absorbing medium can always be represented in terms of two (resp. one) optical axes and three (resp. two) refractive indices if the the medium is biaxial (resp. uniaxial). Currently, we only support initialization of the optical axes from a director or Q-tensor field defining a liquid crystal phase.

In the first case, the phase is unixial and the optical axis simply corresponds to the director field $\vec{n}$. The relation between permttivity field and unixial optical axis is as follows (with n_e and n_o the extraordinay and ordinary indices):

$$\epsilon_{ij} = n_o^2 \delta_{ij} + (n_e^2 - n_o^2) n_i n_j$$

In the second case, the Q-tensor fully defines the orientational order of the LC phase and therefore encompasses both biaxial and unixial media. The general definition of the Q-tensor is as follows:

$$Q_{ij} = \frac{\tilde{S}}{2}\left(n_i^{(1)}n_j^{(1)} - \delta_{ij}\right) + \frac{\tilde{P}}{2}\left(n_i^{(2)}n_j^{(2)} - n_i^{(3)}n_j^{(3)}\right)$$

with $\tilde{S}$ and $\tilde{P}$ the renormalized scalar order parameter and biaxiality parameter, $\vec{n}^{(1)}$ and $\vec{n}^{(2)}$ the two optical axes, and $\vec{n}^{(3)} = \vec{n}^{(1)} \times \vec{n}^{(2)}$. When $\tilde{S} \neq 0$ and $\tilde{P} = 0$ (resp. $\tilde{P} \neq 0$), the phase is unixial (resp. biaxial). An equilibrium (uniform) nematic liquid crystal phase is associated with $\tilde{S} = 1$ and $\tilde{P} = 0$, but both these parameters can have different values inside topological defects. Finally, the relation between the Q-tensor and permittivity tensor is as follows (with n_e and n_o the extraordinay and ordinary indices of the uniform nematic liquid crystal phase):

$$\epsilon_{ij} = \frac{2n_o^2 + n_e^2}{3}\delta_{ij} + \frac{2(n_e^2 - n_o^2)}{3}Q_{ij}$$

Note that despite the use of two refractive indices, optical biaxiality near the core of defects is still taken into account through the associated biaxiality of the Q-tensor.

In `nemaktis`, any tensor field (director or Q-tensor) is represented internally on a cartesian regular mesh as a numpy array of shape `(Nz,Ny,Nx,Nv)`, where `Nv` is the dimension of the tensor data (3 for vector fields such as the director, 6 for symmetric tensors such as the Q-tensor) and `Nx`, `Ny` and `Nz` are the number of mesh points in each spatial direction. In addition to these variables, one needs to specify the total lengths of the mesh in each spatial direction, which we will call `Lx`, `Ly` and `Lz` in the following. All lengths are in micrometer in `nemaktis`, and the mesh for the director field is always centerered on the origin (which means that the spatial coordinate `u=x,y,z` is always running from `-Lu/2` to `Lu/2`).

Here, we will focus on a simple director field structure and start by defining an empty *`DirectorField`* object on a mesh of dimensions `80x80x80` and lengths `10x10x10` (for q-tensor field, use instead *`QTensorField`*):

```
nfield = nm.DirectorField(
    mesh_lengths=(10,10,10), mesh_dimensions=(80,80,80))
```

Next, we need to specify numerical values for the director field. Two methods are possible: either you already have a numpy array containing the values of your director field, in which case you can directly give this array to the *`DirectorField`* object (remember, you need to make sure that this array is of shape `(Nz,Ny,Nx,3)`):

```
nfield.vals = my_director_vals_numpy_array
```

Or you have an analytical formula for the director field, in which case you can define three python functions and give these to the *`DirectorField`* object. In this tutorial, we will assume the latter option and define the director field of a double twist cylinder:

```
q = 2*np.pi/20
def nx(x,y,z):
    r = np.sqrt(x**2+y**2)
    return -q*y*np.sinc(q*r)
def ny(x,y,z):
    r = np.sqrt(x**2+y**2)
    return q*x*np.sinc(q*r)
def nz(x,y,z):
    r = np.sqrt(x**2+y**2)
    return np.cos(q*r)
nfield.init_from_funcs(nx,ny,nz)
```

If the analytical formula for the director components do not give normalized director values, you can still normalize manually the director values after importing them:

```
nfield.normalize()
```

Finally, we point out that both the *DirectorField* class used here and the more general *QTensorField* class derive from a common class *TensorField* which includes useful geometric transformation routines (*rotate()*, *rotate_90deg()*, *rotate_180deg()*, *rescale_mesh()*, *extend()*) and a routine *set_mask()* allowing the specification of non-trivial definition domain for the LC phase. All these methods are documented in the API section of this wiki and are inherited by the *DirectorField* and *QTensorField* classes. Here, we will simply demonstrate the capabilities of the tensor field class by applying a 90° rotation around the axis `x`, extending the mesh in the `xy` plane with a scale factor of 2, and defining a droplet mask centered on the mesh with a diameter equal to the mesh height:

```
nfield.rotate_90deg("x")
nfield.extend(2,2)
nfield.set_mask(mask_type="droplet")
```

Note that extending the mesh in the xy direction is essential if you define a non-trivial LC mask, because you need to leave enough room for the optical fields to propagate around the LC domain.

And that's it, we now have set-up the director field of a double-twist droplet with the polar axis oriented along the axis `y`! If you want to save this director file to a XML VTK file (the standard format used by the excellent visualisation software Paraview), you can add the following command to your script:

```
nfield.save_to_vti("double_twist_droplet")
```

You can import back the generated file in any script by directly constructing the DirectorField object with the path to this file:

```
nfield = nm.DirectorField(vti_file="double_twist_droplet.vti")
```

This functionality is especially useful if generating the director field values takes a lot of time. Of course, the same type of functionality can also be found in the *QTensorField* class.

5.2 Defining a LCMaterial

The next step is to define possible isotropic layers above the LC layer (which can distort the optical fields on the focal plane), as well as the refractive indices of all the materials in the sample. Since our system here consists of a droplet embedded in another fluid, we need to specify both extraordinay and ordinary indices for the LC droplet and the refractive index of the host fluid. To approximate the reflection loss at the entrance of the LC layer, the index `nin` of the medium just below the LC layer can also be set. Finally, the index `nout` of the medium between the sample and the microscope objective can also be set and allows to specify an objective's numerical aperture greater than one (e.g. in the case of an oil-immersion objective) All these informations are stored in the class *LCMaterial*:

```
mat = nm.LCMaterial(
    lc_field=nfield, ne=1.5, no=1.7, nhost=1.55, nin=1.51, nout=1)
```

Note that you can also specify refractive indices with a string expression depending on the wavelength variable "lambda" (in µm), in case you want to take into account the dispersivity of the materials of your sample.

We also want to add a glass plate above the sample and additional space for the host fluid between the droplet and the glass plate:

```
mat.add_isotropic_layer(nlayer=1.55, thickness=5) # 5 µm space between the droplet␣
↪and glass plate
mat.add_isotropic_layer(nlayer=1.51, thickness=1000) # 1mm-thick glass plate
```

Using all those informations, Fresnel reflections in the isotropic layers above the sample can be calculated exactly and also give the fields transmitted through the sample. The full details of the isotropic layers below the sample are not needed because in `nemaktis` the incident optical fields always correspond to a set of plane waves whose wavectors are weakly tilted with respect to the `z` direction (in which case the amplitude of the fields is uniformly affected by any isotropic layers orthogonal to `z`). However, anisotropic Fresnel boundary conditions at the entrance of the LC layer may affect the optical fields in a space-dependent way. These interface conditions are determined from the refractive indices given in the constructor of the LCMaterial, as explained above.

5.3 Propagating optical fields through the sample

Now that the sample geometry is fully caracterized, we can propagate fields through the sample and through an objective into the visualisation plane (which we initially assume to be conjugate to the center of the sample), as in a real microscope (see *Microscopy model for Nemaktis* for more details): a set of plane waves with different wavevectors and wavelengths are sent on the LC sample, and the associated transmitted optical fields are calculated using one of the backend.

The actual set of wavelengths for the plane waves approximate the relevant part of the spectrum of the illumination light, whereas the set of wavevectors is determined from the numerical aperture of the input condenser. The more open the condenser aperture is, the smoother the micrograph will look, since an open condenser aperture is associated with a wide range of angle for the wavectors of the mutually incoherent incident plane waves. Conversely, an almost closed condenser aperture is associated with a single plane wave incident normally on the sample.

With `nemaktis`, the propagation of optical field through a LC sample is as simple as defining an array of wavelengths defining the spectrum of the light source, creating a *`LightPropagator`* object, and calling the method *`propagate_fields`*:

```
wavelengths = np.linspace(0.4, 0.8, 11)
sim = nm.LightPropagator(
    material=mat, wavelengths=wavelengths, max_NA_objective=0.4,
    max_NA_condenser=0, N_radial_wavevectors=1)
output_fields = sim.propagate_fields(method="bpm")
```

The parameter `max_NA_objective` defined in this code snippet corresponds to the maximal numerical aperture of the microscope objective. The parameters `max_NA_condenser` and `N_radial_wavevectors` respectively sets the maximal numerical aperture of the input condenser aperture and the number `Nr` of incident wavevectors in the radial direction of the condenser (the total number of wavevectors will be `1+3*Nr*(Nr-1)`, so be carefull to not set a value too big to avoid memory overflow or long running time). Here, we assume an almost fully closed condenser aperture, so we set the numerical aperture to zero and the total number of wavevectors to 1. Note that omitting the two parameters `max_NA_objective` and `N_radial_wavevectors` during the construction of the *`LightPropagator`* object will default to these values, i.e. this class will assume that there is only one single plane wave incident normally on the sample. Finally, we mention that you will be able to dynamically set the actual values of the numerical aperture of the objective and condenser later on when visualizing the optical fields (with the constraints that these quantities must always be comprised between 0 and the max bounds set here).

The *`propagate_fields`* method uses the specified backend to propagate fields (here, `bpm-solver`) and returns an *`OpticalFields`* object containing the results of the simulation. Periodic boundary conditions in the `x` and `y` directions are systematically assumed, so you should always extend apropriately your director field in order to have a uniform field near the mesh boundaries.

Note that internally two simulations are run for each wavelength and wavevector, one with an input light source polarised along `x` and the other with an input light source polarised along `y`. This allows us to fully caracterize the transmission matrix of the sample and reconstruct any type of micrographs (bright field, crossed polariser...), as explained in *Microscopy model for Nemaktis*. Similaryly to the *`DirectorField`* object, you can save the output fields to a XML VTK file, and reimport them in other scripts:

```
# If you want to save the simulation results
output_fields.save_to_vti("optical_fields")

# If you want to reimport saved simulation results
output_fields = nm.OpticalFields(vti_file="optical_fields.vti")
```

5.4 Visualising optical micrographs

To help the user visualise optical micrographs as in a real microscope, `nemaktis` includes a graphical user interface allowing to generate any type of micrograph in real-time. Once you have generated/imported optical fields in you script, you can start using this interface with the following lines of code:

```
viewer = nm.FieldViewer(output_fields)
viewer.plot()
```

All parameters in this user interface should be pretty self-explanatory, with lengths expressed in µm and optical element angles in ° with respect to `x`. We will simply mention here that the quarter-wavelength and half-wavelength compensators are assumed to be achromatic, while the full-wave "tint sensitive" compensator is aproximated with a slab of wavelength-independent refractive index with a full-wave shift at a wavelength of 540 nm.

Concerning color management, we assume a D65 light source and project the output light spectrum first on the XYZ space, then on the sRGB color space, to finally obtain a usual RGB picture. For more details, see https://dtmm.readthedocs.io/en/latest/tutorial.html#color-conversion.

Finally, refocalisation of the optical micrographs is done by switching to Fourrier space and using the exact propagator for the Helmholtz equation in free space. The unit for the `z-focus` parameter is again micrometers.

CHAPTER 6

API Reference

6.1 TensorField

class nemaktis.lc_material.**TensorField**(***kwargs*)

The TensorField class stores the discrete data of a tensor field on a cartesian mesh. Currently, only first-order tensor (vector) fields and symmetric second-order tensor fields are supported by this class. The ordering of degree of freedoms for each mesh points is as follows:

- Vector field n: (nx,ny,nz)
- Symmetric second-order tensor field Q: (Qxx,Qyy,Qzz,Qxy,Qxz,Qyz)

This class is initialized given either the lengths and dimensions of the associated 3D mesh or a path to a vti file containing the tensor field and mesh details.

In the first version of this constructor:

```
field = TensorField(
    mesh_lengths=(Lx,Ly,Lz), mesh_dimensions=(Nx,Ny,Nz), tensor_order=m)
```

the actual values of the tensor field needs to be provided later via the setter method "vals", which should be given a numpy array of shape (Nz,Ny,Nx,Nv), where Nv=3 (Nv=6) if m=1 (m=2). The mesh lengths needs to be specified in micrometer.

In the second version of this constructor:

```
field = TensorField(vti_file="path to vti file", vti_array="name of tensor array")
```

the values of the tensor field and the details of the mesh are automatically assigned from the given vti file and array name.

set_mask(*, *mask_type*, *mask_formula=None*, *mask_ndarray=None*)

Set a mask for the definition domain of this tensor field. This method allows to specifify an arbitrary complex definition domain which is a subset of the regular cartesian mesh specified at construction. Positive mask values are associated with the definition domain, while negative values are associated with the "host"

embedding domain (for LC structure, this would be the host fluid or material which encases the LC domain). Three possible ways of initializing the mask are possible. If you simply want to specify a spherical domain for a droplet centered on the mesh and of diameter equal to the mesh length along z, call:

```
tensor_field.set_mask(mask_type="droplet")
```

You can also use a string formula depending on the space variables `x`, `y` and `z` and which must evaluates to a value >=0 if the associated point is inside the definition domain of the tensor field, else to a value <=0:

```
tensor_field.set_mask(mask_type="formula", mask_formula="your formula")
```

Finally, you can directly gives a numpy array of shape (Nz,Ny,Nx), where each value in this array must be >=0 if the associated mesh point is inside the definition domain, else <=0:

```
tensor_field.set_mask(mask_type="raw", mask_ndarray=your_mask_array)
```

delete_mask()
Delete the current LC mask.

mask_type
Returns the mask type – "droplet", "formula" or "raw".

mask_formula
Returns the LC mask formula if it was set, else returns None.

mask_vals
Returns the mask boolean array.

extend(*scale_x*, *scale_y*)
Extend the computational mesh in the `xy` plane by padding new points near the `x` and `y` boundaries. The associated new data points are initialized with the edge value of the tensor field on the `x` and `y` boundaries.

Parameters

- **scale_x** (*float*) – The mesh length in the x-direction will be scaled by this factor.
- **scale_y** (*float*) – The mesh length in the y-direction will be scaled by this factor.

rotate_90deg(*axis*)
Apply a solid rotation of the tensor field of 90 degrees around the specified axis. This is a lossless operation which does not rely on interpolation.

Parameters **axis** (*str*) – Axis around which to perform the rotation. Need to be under the form '[s]A' where the optional parameter 's'='+' or '-' decribes the sign of rotation and 'A'='x', 'y' or 'z' defines the rotation axis.

rotate_180deg(*axis*)
Apply a solid rotation of the tensor field of 180 degrees around the specified axis. This is a lossless operation which does not rely on interpolation.

Parameters **axis** (*str*) – Axis around which to perform the rotation. Need to be under the form 'A' where 'A'='x', 'y' or 'z' defines the rotation axis.

rotate(*axis*, *angle*, *fill_value=None*)
Apply a solid rotation of the tensor field of an arbitrary angle around the specified axis. This is a lossy operation that will rely on interpolation, so possible artefacts can appear if the tensor field data is not smooth enough.

Parameters

- **axis** (*str*) – Axis around which to perform the rotation. Need to be under the form 'A' where 'A'='x', 'y' or 'z' defines the rotation axis.

- **angle** (*float*) – Angle of rotation in degrees.

rescale_mesh(*scaling_factor*)
Uniformly scale the mesh using the given scaling factor.

Parameters **scaling_factor** (*factor*) – The mesh lengths and spacings will be multiplied by this factor.

vals
Numpy array for the tensor values, of shape (Nz,Ny,Nx,Nv), where Nv=3 for a vector field and Nv=6 for a symmetric second-order tensor field (we only store the [xx,yy,zz,xy,xz,yz] components for efficiency reasons).

save_to_vti(*file_name*, *array_name*)
Save the tensor field inside a vti file.

Parameters

- **file_name** (*string*) – Path to the exported vti file. The ".vti" extension is automatically appended, no need to include it in this parameter (but in case you do only one extension will be added).
- **array_name** (*string*) – Name of the vti array that will store the tensor field.

get_pos(*ix*, *iy*, *iz*)
Returns the spatial position associated with the mesh indices (ix,iy,iz) It is assumed that the mesh is centered on the origin (0,0,0).

get_mesh_dimensions()
Returns the dimensions (Nx,Ny,Nz) of the simulation mesh

get_mesh_lengths()
Returns the lengths (Lx,Ly,Lz) of the simulation mesh

get_mesh_spacings()
Returns the spacings (dx,dy,dz) of the simulation mesh

get_n_vertices()
Returns the number of vertices in the simulation mesh

6.2 DirectorField

class nemaktis.lc_material.**DirectorField**(***kwargs*)
A specialization of the TensorField class for director fields.

The two versions of the constructor of the parent class TensorField are simplified since we do not need the parameters 'tensor_order' (always 1 for a director field) or 'vti_array' (assumed to be "n"):

```
# First version of the constructor
nfield = DirectorField(
    mesh_lengths=(Lx,Ly,Lz), mesh_dimensions=(Nx,Ny,Nz))
# Second version of the constructor
nfield = DirectorField(
    vti_file="path to vti file", vti_array="name of tensor array")
```

In addition to all the methods of the parent class for initializing and manipulating the field values, we specialize the "save_to_vti" method (imposing that the exported vti array name is always "n") and provide additional methods for initializing the director field values from theoretical functions, exporting a q-tensor field from the director field, and normalizing the director field to unit norm.

init_from_funcs(*nx_func*, *ny_func*, *nz_func*)
Initialize the director field from three functions for each of its component. The functions must depend on the space variables `x`, `y` and `z`. We recall that the mesh is centered on the origin.

If the given functions are numpy-vectorizable, this function should be pretty fast. If not, a warning will be printed and the faulty function(s) will be vectorized with the numpy method `vectorize` (in which case you should expect a much slower execution time).

init_from_func(*n_func*)
Initialize the director field from a single function returning an array whose last axis is associated with each director components. The function must depend on the space variables `x`, `y` and `z`. We recall that the mesh is centered on the origin.

If the given functions are numpy-vectorizable, this function should be pretty fast. If not, a warning will be printed and the faulty function(s) will be vectorized with the numpy method `vectorize` (in which case you should expect a much slower execution time).

normalize()
Normalize the director field values to unit norm.

get_qtensor_field()
Returns a QTensorField object equivalent to the director field represented by this class, assuming a constant scalar order parameter (equal to its equilibrium value).

Since in Nematkis a q-tensor field is always renormalized by the equilibrium value of the order parameter, this method simply uses the formula Q_ij=(3*n_i*n_j-d_ij)/2 to convert a director value n to a q-tensor value Q, with d_ij the kronecker delta.

save_to_vti(*file_name*)
Save the director field into a vti file, assuming "n" for the vti array name.

Parameters **file_name** (*string*) – Path to the exported vti file. The ".vti" extension is automatically appended, no need to include it in this parameter (but in case you do only one extension will be added).

6.3 QTensorField

class nematkis.lc_material.**QTensorField**(***kwargs*)
A specialization of the TensorField class for Q-tensor fields (i.e. the full LC order parameter field).

In Nematkis, we always assume that the Q-tensor field is renormalized by the equilibrium value S_eq of the scalar order parameter S. This means for uniaxial LC, the Q-tensor far from topological defects can always be put under the following form:

$$Q_{ij} = \left(3n_i n_j - \delta_{ij}\right)/2$$

The two versions of the constructor of the parent class TensorField are simplified since we do not need the parameters 'tensor_order' (always 2 for Q-tensor) or 'vti_array' (assumed to be "Q"):

```
# First version of the constructor
qfield = QTensorField(
    mesh_lengths=(Lx,Ly,Lz), mesh_dimensions=(Nx,Ny,Nz))
# Second version of the constructor
qfield = QTensorField(
    vti_file="path to vti file", vti_array="name of tensor array")
```

In addition to all the methods of the parent class for initializing and manipulating the field values, we specialize the "save_to_vti" method (imposing that the exported vti array name is always "Q") and provide addional

methods for initializing the Q-tensor field values from theoretical functions, exporting a director field from a q-tensor field, and imposing the traceless constraint Tr(Q)=0.

init_from_funcs(*Qxx_func*, *Qyy_func*, *Qzz_func*, *Qxy_func*, *Qxz_func*, *Qyz_func*)
Initialize the Q-tensor field from six functions for each of its component (xx, yy, zz, xy, xz, yz). The functions must depend on the space variables `x`, `y` and `z`. We recall that the mesh is centered on the origin.

If the given functions are numpy-vectorizable, this function should be pretty fast. If not, a warning will be printed and the faulty function(s) will be vectorized with the numpy method `vectorize` (in which case you should expect a much slower execution time).

init_from_func(*Q_func*)
Initialize the Q-tensor field from a single function returning an array whose last axis is associated with each Q-tensor components (xx, yy, zz, xy, xz, yz). The function must depend on the space variables `x`, `y` and `z`. We recall that the mesh is centered on the origin.

If the given functions are numpy-vectorizable, this function should be pretty fast. If not, a warning will be printed and the faulty function(s) will be vectorized with the numpy method `vectorize` (in which case you should expect a much slower execution time).

apply_traceless_constraint()
Apply the traceless constraint Tr(Q)=0 to this q-tensor field.

get_director_field()
Returns a DirectorField object equivalent to the q-tensor field represented by this class, assuming a unixial medium and discarding any variations of the scalar order parameter S.

In practice this method simply calculate the eigenvectors of the Q-tensor field and initialize the director field from the eigenvectors with highest algebraic value. The returned director field is therefore not fully equivalent to the Q-tensor field if there are topological defects or biaxiality inside the LC structure.

save_to_vti(*file_name*)
Save the Q-tensor field into a vti file, assuming "Q" for the vti array name.

Parameters **file_name** (*string*) – Path to the exported vti file. The ".vti" extension is automatically appended, no need to include it in this parameter (but in case you do only one extension will be added).

6.4 LCMaterial

class nemaktis.lc_material.**LCMaterial**(*, *lc_field*, *ne*, *no*, *nhost=1*, *nin=1*, *nout=1*, *ne_imag=0*)
A class containing the LC orientational field data (director or q-tensor) and physics constants.

Parameters

- **lc_field** (*DirectorField* or *QTensorField* object) –
- **ne** (*float or math string depending on the wavelength variable "lambda" (µm)*) – The extraordinary refractive index associated with the LC material.
- **no** (*float or math string depending on the wavelength variable "lambda" (µm)*) – The ordinary refractive index associated with the LC material.
- **nhost** (*optional, float or math string depending on the wavelength variable "lambda" (µm)*) – The refractive index associated with an eventual host fluid in which the LC domain is embedded (see DirectorField.set_mask).
- **nin** (*optional, float or math string depending on the wavelength variable "lambda" (µm)*) – The refractive index associated with the input medium below the LC layer. A default value of 1 is assumed.

- **nout** (*optional, float or math string depending on the wavelength variable "lambda" (µm)*) – The refractive index associated with the output medium between the sample and objective. A default value of 1 is assumed.
- **ne_imag** (*optional, float or math string depending on the wavelength variable "lambda" (µm)*) – Imaginary part of the extroardinary index allowing to take into account absorption along the optical axis. A default value of 0 is assumed

add_isotropic_layer(*, *nlayer*, *thickness*)

Add an isotropic layer above the sample. Light is assumed to propagate in the z-direction, and will cross first the LC material, and then the isotropic layers specified with this function.

Parameters

- **nlayer** (*float*) – Refractive index of the new isotropic layer
- **thickness** (*float*) – Thickness (µm) of the new isotropic layer

6.5 LightPropagator

class nemaktis.light_propagator.**LightPropagator**(*, *material*, *wavelengths*, *max_NA_objective*, *max_NA_condenser=0*, *N_radial_wavevectors=1*, *koehler_1D=False*)

The LightPropagator class allows to propagate optical fields through a LC sample as in a real microscope: a set of plane waves with different wavevectors and wavelengths are sent on the LC sample, and the associated transmitted optical fields (which can now longer be represented as plane waves due to diffraction) are calculated using one of the backend.

The actual set of wavelengths for the plane waves (choosen at construction) approximate the relevant part of the spectrum of the illumination light, whereas the set of wavevectors (also calculated at construction) are determined from the numerical aperture of the input condenser. The more open the condenser aperture is, the smoother the micrograph will look, since an open condenser aperture is associated with a wide range of angle for the wavectors of the incident plane waves. Conversely, an almost closed condenser aperture is associated with a single plane wave incident normally on the sample. For more details, see [Koehler illumination setup].

Note that with the FieldViewer class, the transmitted optical fields calculated with this class can be projected on a visualisation screen through an objective of given numerical aperture. The numerical apertures of both the objective and condenser aperture can be set interactively in the FieldViewer class, whereas in this class we only specify the maximum value allowed for both quantities.

The simulation and choice of backend is done by calling the method `propagate_field`.

For each wavelength and wavevector of the incident plane wave, two simulations are done: one with a light source polarised along x, and one with a light source polarised along y. This allows us to fully caracterize the transmission of the LC sample and reconstruct any kind of optical micrograph.

Parameters

- **material** (*LCMaterial* object) –
- **wavelengths** (*array-like object*) – An array containing all the wavelengths of the spectrum for the light source.
- **max_NA_objective** (*float*) – Sets the maximal numerical aperture for the microscope objective (you can dynamically adjust this quantity later on with a FieldViewer).
- **max_NA_condenser** (*float*) – Sets the maximal numerical aperture for the microscope condenser (you can dynamically adjust this quantity later on with a FieldViewer).

- **N_radial_wavevectors** (*int*) – Sets the number of wavevectors in the radial direction for the illumination plane waves. The total number of plane waves for each wavelength is 1+3*Nr*(Nr-1), where Nr correspond to the value of this parameter.

material
Returns the current LC material

propagate_fields(*, *method*, *bulk_filename=None*)
Propagate optical fields through the LC sample using the specified backend.

Parameters

- **method** (*"bpm" | "dtmm(D)"*) – If equal to "bpm", the beam propagation backend will be used (see [The beam-propagation backend (bpm-solver)] for details). Should be used if accuracy is privileged over speed.

  If equal to "dtmm(D)" (with D a positive integer), the diffractive transfer matrix backend will be used with the "diffraction" parameter set to D (see [The diffraction transfer matrix backend (dtmm)] for details). Should be used with small values of D if speed is privileged over accuracy (D=0 correspond to the Jones method).

- **bulk_filename** (*None or string*) – If none, the backend will not export the bulk value of the optical fields in the LC layer. Else, the bulk fields values will be exported to a vti file whose basename is set by this parameter.

6.6 OpticalFields

class nemaktis.light_propagator.**OpticalFields**(**kwargs*)
The OpticalFields object stores the mesh information of the transverse mesh (plane mesh orthogonal to the z-direction, default altitude of 0) and the optical fields values on this mesh. Since this python package is mainly used to reconstruct micrographs, we only store internally the complex horizontal electric field for two simulation: one with a light source polarised along `x`, and the other with a light source polarised along `y`. In case multiple wavelengths/wavectors were used in the simulation, we store these quantities separately for each wavelength/wavevector.

This class is initialised either manually or with a path to a vti file containing previously calculated optical fields and mesh details.

In the first version of this constructor:

```
optical_fields = OpticalFields(
    wavelengths=[l0,l1,...,lN], max_NA_objective=NA_o,
    max_NA_condenser=NA_c, N_radial_wavevectors=Nr,
    mesh_lengths=(Lx,Ly), mesh_dimensions=(Nx,Ny))
```

the actual values of the transverse fields needs to be provided later using the raw setter method fields_vals (shape (N_wavelengths,N_wavevectors,4,Ny,Nx), with N_wavevectors=3*Nr*(Nr-1)+1).

In the second version of this constructor:

```
optical_fields = OpticalFields(vti_file="path to vti file")
```

the values of the wavelengths and transverse fields are automatically assigned from the vti file.

copy()
Returns a hard copy of this OpticalFields object

focused_vals
Numpy array for the optical fields values after focalisation by the microscope objective, of shape (N_wavelengths,N_wavevectors,4,Ny,Nx).

vals
Numpy array for the optical fields values, of shape (N_wavelengths,N_wavevectors,4,Ny,Nx).

If you want to initialize by hand the optical fields, the four components in the third dimension correspond to:

- complex Ex field for an input polarisation//x
- complex Ey field for an input polarisation//x
- complex Ex field for an input polarisation//y
- complex Ey field for an input polarisation//y

focus_fields(*z_focus=None*)
Propagate the optical fields through the objective lens to the screen conjugate to the focusing plane (whose altitude inside the sample is set with the parameter z_focus).

save_to_vti(*filename*)
Save the optical fields into a vti file.

The ".vti" extension is automatically appended, no need to include it in the filename parameter (but in case you do only one extension will be added)

get_pos(*ix*, *iy*)
Returns the position associated with the mesh indices (ix,iy)

It is assumed that the mesh is centered on the origin (0,0).

get_wavelengths()
Returns the wavelength array

get_wavevectors()
Returns the wavevectors array

get_qr_index(*NA_condenser*)
For internal use.

Allows to build sub-range of wavevector index for a given numerical aperture of the condenser, which must be smaller than the internal maximal numerical aperture set at construction.

get_delta_qr()
For internal use.

Allows to build integration rule with respect to the wavectors.

get_mesh_dimensions()
Returns the dimensions (Nx,Ny) of the transverse mesh

get_mesh_lengths()
Returns the lengths (Lx,Ly) of the transverse mesh

get_mesh_spacings()
Returns the spacings (dx,dy,dz) of the transverse mesh

get_n_vertices()
Returns the number of vertices in the transverse mesh

6.7 FieldViewer

class nemaktis.field_viewer.**FieldViewer**(*optical_fields*, *cmf=None*)
A class allowing to recombine optical fields to generate optical micrographs like in a real microscope. For more details, see [Imaging of the object] and [Optical elements for polarized optical micrographs]

Parameters

- **optical_fields** (*OpticalFields* object) – Can be created either by a LightPropagator or directly by importing a vti file exported in a previous simulation.
- **cmf** (*numpy ndarray*) – A color matching array created with the *dtmm* package, see https://dtmm.readthedocs.io/en/latest/reference.html#module-dtmm.color

polariser = True
Is there a polariser in the optical setup?

analyser = True
Is there an analyser in the optical setup?

upper_waveplate = 'No'
If "No", remove the upper waveplate from the optical setup. Other values set the type of waveplate:

- "Quarter-wave": An achromatic quarter-wave compensator
- "Half-wave": An achromatic half-wave compensator
- "Tint-sensitive": a full-wave compensator at 540 nm.

lower_waveplate = 'No'
If "No", remove the upper waveplate from the optical setup. Other values set the type of waveplate:

- "Quarter-wave": An achromatic quarter-wave compensator
- "Half-wave": An achromatic half-wave compensator
- "Tint-sensitive": a full-wave compensator at 540 nm.

polariser_angle = 0
Angle (in degree) between the privileged axis of the polariser and the x-axis

analyser_angle = 90
Angle (in degree) between the privileged axis of the analyser and the x-axis

upper_waveplate_angle = 0
Angle (in degree) between the fast axis of the upper waveplate and the x-axis

lower_waveplate_angle = 0
Angle (in degree) between the fast axis of the lower waveplate and the x-axis

angle_lock = False
Should the relative angles between optical elements be locked?

intensity = 1
Intensity factor of the micrograph

NA_condenser = 0
Numerical aperture of the microscope's condenser

n_tiles_x = 1
Number of repetitions of the micrograph in the x-direction

n_tiles_y = 1
Number of repetitions of the micrograph in the y-direction

grayscale = False
Should we calculate a grayscale micrograph (True) or a color micrograph (False)

plot()
Run a graphical user interface allowing to dynamically adjust the attributes of this class and visualize the associated micrographs in real-time.

z_focus
Current vertical position of the focal plane

NA_objective
Current vertical position of the focal plane

get_image()
Returns the current micrograph as a numpy array of shape (Ny,Nx,3|1), (last dim is 3 if in color mode, 1 if in grayscale mode).

get_spectrum()
Returns the q-averaged spectrum as a numpy array of shape (Ny,Nx,Nl), (last dim is 3 if in color mode, 1 if in grayscale mode).

update_image()
Recompute the micrograph from the optical fields data

CHAPTER 7

Ray-tracing backend

The executable of the ray tracing backend is named `rt-solver` and can be called in any terminal with the conda environment for nemaktis activated.

`rt-solver` relies on json-like configuration file, so you don't have to touch to any C++ code if you just want to use the method. To generate a default configuration file, run:

All parameters in this configuration file are fully documented, so you should be able to understand how to make the code working just by reading and ajusting the parameters in this file. More information on the subtleties of this code will be added later on this wiki, for now we will just mention that the input vti file for the director field can be created directly with the high-level interface (see *DirectorField*).

Once you are satisfied with the change you made to your configuration file, you can actually run the code by typing:

CHAPTER 8

Beam propagation backend

The executable of the beam propagation backend is named `bpm-solver` and can be called in any terminal with the conda environment for nemaktis activated.

`bpm-solver` relies on json-like configuration file, so you don't have to touch to any C++ code if you just want to use the method. To generate a default configuration file, run:

All parameters in this configuration file are fully documented, so you should be able to understand how to make the code working just by reading and ajusting the parameters in this file. More information on the subtleties of this code will be added later on this wiki, for now we will just mention that the input vti file for the director field can be created directly with the high-level interface (see *DirectorField*).

Once you are satisfied with the change you made to your configuration file, you can actually run the code by typing:

CHAPTER 9

Diffractive transfer matrix backend

The complete documentation of `dtmm` can be found in another wiki:

https://dtmm.readthedocs.io

Index