

---

# **Nemaktis**

***Release 1.3.3***

**Guilhem Poy, Andrej Petelin**

**Oct 22, 2020**



# INTRODUCTION

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	License . . . . .	1
1.2	Contributors . . . . .	1
1.3	Highlights . . . . .	1
1.4	Limitations . . . . .	2
1.5	Contact . . . . .	2
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	With Conda (windows or linux) . . . . .	3
2.2	Developer method (only linux) . . . . .	4
<b>3</b>	<b>Microscopy model for Nemaktis</b>	<b>5</b>
3.1	1. General description . . . . .	5
3.2	2. Koehler illumination setup . . . . .	5
3.3	3. Transmission/Reflection of light inside the object . . . . .	7
3.4	4. Imaging of the object . . . . .	10
3.5	5. Optical elements for polarized optical micrographs . . . . .	11
<b>4</b>	<b>Tutorial</b>	<b>15</b>
4.1	Defining a DirectorField . . . . .	15
4.2	Defining a LCMaterial . . . . .	17
4.3	Propagating optical fields through the sample . . . . .	17
4.4	Visualising optical micrographs . . . . .	18
<b>5</b>	<b>API Reference</b>	<b>19</b>
5.1	DirectorField . . . . .	19
5.2	LCMaterial . . . . .	21
5.3	LightPropagator . . . . .	21
5.4	OpticalFields . . . . .	22
5.5	FieldViewer . . . . .	24
<b>6</b>	<b>Ray-tracing backend</b>	<b>27</b>
<b>7</b>	<b>Beam propagation backend</b>	<b>29</b>
<b>8</b>	<b>Diffraction transfer matrix backend</b>	<b>31</b>
	<b>Index</b>	<b>33</b>



## 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, which excludes high-NA microscope objectives.
- 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 developed 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

## INSTALLATION

Nemaktis is a mixture of C++ and python codes, and have been successfully tested both on Windows and Linux (note that it should be straightforward to adapt it for Mac, since it is a Unix-based OS). The next two subsections present the two possible ways of installing this package.

### 2.1 With Conda (windows or linux)

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

The first step is to install the software `miniconda`, which contains a barebone version of the package manager `conda`. If you already have the full python distribution `Anaconda` installed on your machine, this step is not necessary. The installation files for Windows/Linux are available at this address: <https://docs.conda.io/en/latest/miniconda.html>

If you are a Windows 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/v1.3/Install.Nemaktis.cmd>

This script will 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 conda environment `nm`).

If you are a Linux user or want to type the installation commands yourselves (they are not very complicated after all!), open a terminal (Windows: application “Conda terminal” installed with miniconda, Linux: any terminal) and type the following command:

```
conda create -n nm -c conda-forge -c warthan07 -y nemaktis=1.3.4
```

(Optional) If you want to use your favourite python editor when using Nemaktis, you have to install and run it from the same 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:

```
conda activate nm
conda install spyder
```

## 2.2 Developer method (only linux)

If you want to be able to modify the code, but still want to enjoy the simplicity of conda 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 activate any conda environment in which you have write access. If you don't have any conda environment yet, you can type:

```
conda create -n build
conda activate build
```

If necessary, install the conda-build tools:

```
conda install conda-build conda-verify
```

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-build . -c conda-forge
```

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

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



## 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 through the condenser lens 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 interfere 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):

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

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

$$q^{(k)} = \frac{k}{N_r - 1} \text{NA}_{\max}, \quad k = 0 \dots N_r - 1$$

$$\theta^{(k,l)} = \frac{\pi l}{3k}, \quad l = 0 \dots 6k$$

Here,  $\text{NA}_{\max} = \sin \psi_{\max}$  (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 Nemaktis, this mesh is fully characterized by the two parameters,  $\text{NA}_{\max}$  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  $\text{NA}_{\max}$  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 Nemaktis 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 exclude 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  $\vec{E}(x, y, z) \exp[-ik_0 z t]$  (with  $c$  the light speed and  $k_0$  the wavevector in empty space):

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

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.

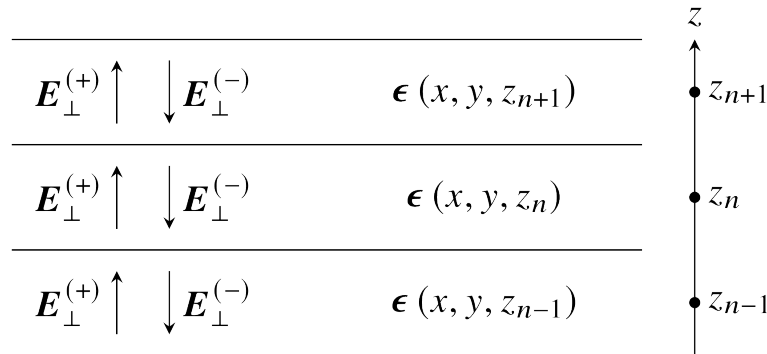
#### 3.3.1 3.1 The beam-propagation backend (bpm-solver)

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 (excluding 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  $\bar{\epsilon}(x, y, z)$ :



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-freedom 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-freedom 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 developed 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 inaccurate) simulations with faster running times than BPM on relatively small meshes (for big meshes, the logarithmic complexity of dtmm kicks in and DTMM is slower than BPM);
- high value of this parameter provide accurate simulations (computational errors slightly worse than the ones obtained with BPM, but still relatively good) with slower running times than with BPM.

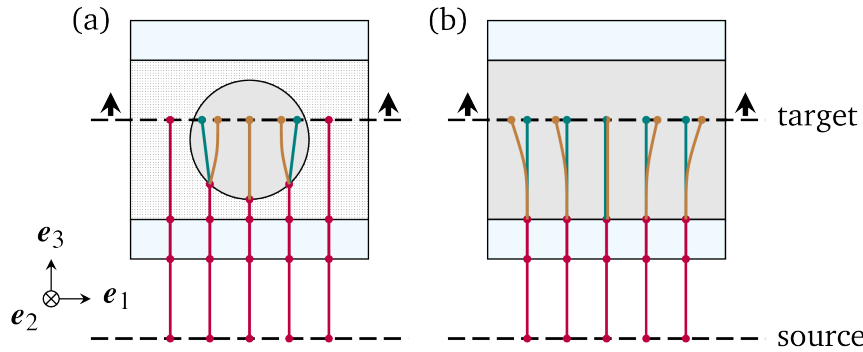
In short, DTMM is the perfect backend if you want to quickly try imaging simulations without worrying too much about the accuracy, whereas BPM is more suited for efficient accurate simulations on arbitrary big meshes (provided that enough random-access-memory is available!).

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):

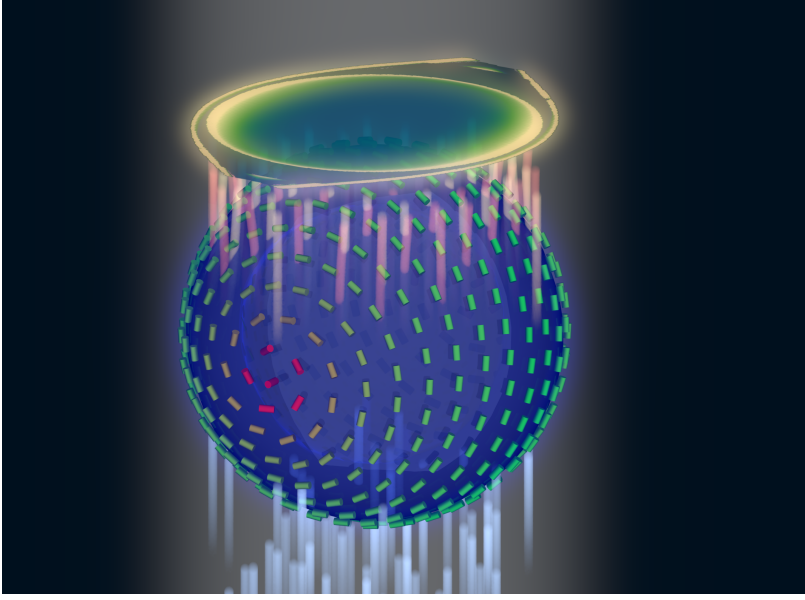
[DTMM: a Diffractive Transfer Matrix Method]

### 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  $\bar{\epsilon}(x, y, z)$  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:

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

where  $\vec{r}^{(\text{im})}$  ( $\vec{r}^{(\text{foc})}$ ) correspond to coordinates on the imaging (focusing) plane and  $\bar{\vec{G}}$  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)]):

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

where NA is the numerical aperture of the objective,  $\Pi$  is the rectangular function ( $\Pi(u)$  is equal to 1 if  $|u| < 0.5$ , else it is 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  $|\vec{k}_\perp| \geq k_0 \text{NA}$  are filtered out by the objective while Fourier components with  $|\vec{k}_\perp| < k_0 \text{NA}$  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 Helmholtz equation in Fourier space:

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

The final image on the imaging plane is defined as the squared amplitude of  $\vec{E}[\vec{r}^{(\text{im})}]$ , 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:

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

where  $\vec{E}_{\text{out}}^{(k,l,m)}$  correspond to the output transverse optical field of one of the backend in Sec. 3 for an incoming plane wave  $\vec{E}_{\text{in}}^{(k,l,m)}$ ,  $\vec{u}_m$  ( $m = 1, 2$ ) 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  $\bar{\vec{T}}_{\text{obj}}^{(k,l)}$  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):

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

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:

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

### 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  $\bar{T}_{\text{pol}}$ ,  $\bar{T}_{\text{an}}$  and  $\bar{T}_{\text{wp}}$  respectively associated with a polariser, analyser or waveplate. The expression of the transfer matrix for a polariser or analyser only depends on the angle  $\psi_{\text{pol,an}}$  of the optical element axis with respect to  $\vec{u}_1$  in the transverse plane (horizontal axis in Nemaktis):

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

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

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

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:  $\Gamma = 2\pi [0.54/\lambda]$ , where  $\lambda$  is the wavelength in  $\mu\text{m}$ ; the advantage of this waveplate is that it allows the visualization of in-plane molecular angular deviation as color shifts when illuminating an inhomogeneous 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  $\bar{T}_{\text{tot}}^{(k,l)}$ . 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:

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

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

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

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.



## TUTORIAL

This tutorial provides a hands-on introduction to the python package `nemaktis`. You will learn the different ways of creating director 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).

### 4.1 Defining a DirectorField

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 of the LC structure. Currently, only uniaxial media is supported in the high-level interface (which means we only need to specify the director field associated with the privileged axis of the birefringence medium), but support for arbitrary permittivity tensor should be added soon (the low-level backends `dtmm` and `bpm-solver` are already fully compatible with biaxial media). In `nemaktis`, any vector field is represented internally on a cartesian regular mesh as a numpy array of shape  $(N_z, N_y, N_x, N_v)$ , where  $N_v$  is the dimension of the vector data (3 for a director field, 6 for a symmetric tensor) and  $N_x$ ,  $N_y$  and  $N_z$  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  $L_x$ ,  $L_y$  and  $L_z$  in the following. All lengths are in micrometer in `nemaktis`, and the mesh for the director field is always centered on the origin (which means that the spatial coordinate  $u=x, y, z$  is always running from  $-L_u/2$  to  $L_u/2$ ).

Here, we will start by defining an empty *DirectorField* object on a mesh of dimensions  $80 \times 80 \times 80$  and lengths  $10 \times 10 \times 10$ :

```
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  $(N_z, N_y, N_x, 3)$ ):

```
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, you can apply geometric transformation to the director field with the methods *rotate()*, *rotate\_90deg()*, *rotate\_180deg()*, *rescale\_mesh()*, *extend()*, as well as specify a non-trivial domain for the LC phase with the method *set\_mask()*. All these methods are documented in the API section of this wiki. Here, we will simply demonstrate the capabilities of the director field object 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.

## 4.2 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 extraordinary and ordinary indices for the LC droplet and the refractive index of the host fluid. All these informations are stored in the class `LCMaterial`:

```
mat = nm.LCMaterial(
    director_field=nfield, ne=1.5, no=1.7, nhost=1.55)
```

Note that you can also specify refractive indices with a string expression depending on the wavelength variable “lambda” (in  $\mu\text{m}$ ), 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  $\mu\text{m}$  space between the droplet_
↪and glass plate
mat.add_isotropic_layer(nlayer=1.51, thickness=1000) # 1mm-thick glass plate
```

We don’t specify isotropic layers below the sample because in `nemaktis` the incident optical fields always correspond to a set of plane waves whose wavevectors 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  $z$ ).

## 4.3 Propagating optical fields through the sample

Now that the sample geometry is fully characterized, 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 wavevectors 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  $N_r$  of incident wavevectors in the radial direction of the condenser (the total number of wavevectors will be  $1+3*N_r*(N_r-1)$ , 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 appropriately 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 characterize the transmission matrix of the sample and reconstruct any type of micrographs (bright field, crossed polariser...), as explained in *Microscopy model for Nemaktis*. Similarly 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")
```

## 4.4 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  $\mu\text{m}$  and optical element angles in  $^\circ$  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 approximated 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 Fourier space and using the exact propagator for the Helmholtz equation in free space. The unit for the `z-focus` parameter is again micrometers.

## API REFERENCE

## 5.1 DirectorField

**class** nemaktis.lc\_material.DirectorField(\*\*kwargs)

The DirectorField class stores the director field and mesh informations. It is initialized given either the lengths and dimensions of the associated 3D mesh or a path to a vti file containing the director field and mesh details. In the first version of this constructor: .. code-block:: python

```
nfield = DirectorField( mesh_lengths=(Lx,Ly,Lz), mesh_dimensions=(Nx,Ny,Nz))
```

the actual values of the director field needs to be provided later using the “init\_from\_funcs” method or via the setter method “vals” (numpy array of shape (Nz,Ny,Nx,3)). The mesh lengths needs to be specified in micrometer. In the second version of this constructor: .. code-block:: python

```
nfield = DirectorField(vti_file="path to vti file")
```

the values of the director field and the details of the mesh are automatically assigned from the vti file.

**set\_mask** (\*, mask\_type, mask\_formula=None, mask\_ndarray=None)

Set a mask for the LC domain. This method allows to specify complex shape for the LC domain inside the regular cartesian mesh specified at construction. Positive mask value are associated with the LC domain, while negative values are associated with the embedding fluid. 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: .. code-block:: python

```
nfield.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  $\geq 0$  if the associated point is inside the LC domain, else to a value  $\leq 0$ : .. code-block:: python

```
nfield.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  $\geq 0$  if the associated mesh point is inside the LC domain, else  $\leq 0$ :

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

**delete\_mask** ()

Delete the current LC mask.

**property mask\_type**

“droplet”, “formula” or “raw”.

**Type** Returns the mask type

**property mask\_formula**

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

**property mask\_vals**

Returns the LC 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. These new points are initialized with the edge value of the director field on the  $x$  and  $y$  boundaries. :param *scale\_x*: The mesh length in the  $x$ -direction will be scaled by this factor. :type *scale\_x*: float :param *scale\_y*: The mesh length in the  $y$ -direction will be scaled by this factor. :type *scale\_y*: float

**rotate\_90deg** (*axis*)

Rotate the director field by 90 degrees around the specified axis.

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

**rotate\_180deg** (*axis*)

Rotate the director field by 180 degrees around the specified axis.

**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*)

Rotate the director field by an arbitrary angle around the specified axis.

**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*)

Scale the mesh using the given scaling factor.

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

**property vals**

Numpy array for the director values, of shape (Nz,Ny,Nx,3).

**init\_from\_funcs** (*nx\_func*, *ny\_func*, *nz\_func*)

Initialize the director field from three functions for each of its component. The functions must depends on the space variables  $x$ ,  $y$  and  $z$ . We recall that the mesh is centered on the origin. If the given function are numpy-vectorizable, this function should be pretty fast. If not, a warning will be printed and the given function 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 1.

**save\_to\_vti** (*filename*)

Save the director field 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*, *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

```

## 5.2 LCMaterial

```
class nemaktis.lc_material.LCMaterial(*, director_field, ne, no, nhost=1, nin=1)
```

A class containing the director field data, simulation mesh, and physics constants.

### Parameters

- **director\_field** (*DirectorField* object) –
- **ne** (*float or math string depending on the wavelength variable "lambda" ( $\mu\text{m}$ )*) – The extraordinary refractive index associated with the LC material.
- **no** (*float or math string depending on the wavelength variable "lambda" ( $\mu\text{m}$ )*) – The ordinary refractive index associated with the LC material.
- **nhost** (*optional, float or math string depending on the wavelength variable "lambda" ( $\mu\text{m}$ )*) – 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" ( $\mu\text{m}$ )*) – The refractive index associated with the input medium below the LC layer. A default value of 1 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

- **nlayers** (*float*) – Refractive index of the new isotropic layer
- **thickness** (*float*) – Thickness ( $\mu\text{m}$ ) of the new isotropic layer

## 5.3 LightPropagator

```
class nemaktis.light_propagator.LightPropagator(*, material, wave-
lengths, max_NA_objective,
max_NA_condenser=0,
N_radial_wavevectors=1)
```

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 (chosen 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 wavevectors of the incident plane waves. Conversely, an almost closed condenser aperture is associated with a single plane wave incident normally on the sample.

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 characterize 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  $1+3*N_r*(N_r-1)$ , where  $N_r$  correspond to the value of this parameter.

#### **property 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*") – If equal to “bpm”, the beam propagation backend will be used. Should be used if accuracy is privileged over speed.  
If equal to “dtmm”, the diffractive transfer matrix backend will be used (in its simplest version). Should be used if speed is privileged over accuracy.
- **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.

## 5.4 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/wavevectors 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  $(N\_wavelengths, N\_wavevectors, 4, N_y, N_x)$ , 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

**property focused\_vals**

Numpy array for the optical fields values after focalisation by the microscope objective, of shape  $(N\_wavelengths, N\_wavevectors, 4, N_y, N_x)$ .

**property vals**

Numpy array for the optical fields values, of shape  $(N\_wavelengths, N\_wavevectors, 4, N_y, N_x)$ .

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 wavevectors.

**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

## 5.5 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.

### 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?

**compensator = 'No'**

If “No”, remove the compensator from the optical setup. Other values set the type of compensator:

- “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

**compensator\_angle = 0**

Angle (in degree) between the fast axis of the compensator and the x-axis

**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)

**NA\_objective = 1**

Numerical aperture of the microscope's objective

**plot ()**

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

**property z\_focus**

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).

**update\_image ()**

Recompute the micrograph from the optical fields data



## 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:

```
rt-solver -c [Name of the configuration file]
```

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 adjusting 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:

```
rt-solver -x [Name of the configuration file]
```





## 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:

```
bpm-solver -c [Name of the configuration file]
```

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 adjusting 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:

```
bpm-solver -x [Name of the configuration file]
```



## DIFFRACTIVE TRANSFER MATRIX BACKEND

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

<https://dtmm.readthedocs.io>



## A

`add_isotropic_layer()` (*nemaktis.lc\_material.LCMaterial* method), 21

`analyser` (*nemaktis.field\_viewer.FieldViewer* attribute), 24

`analyser_angle` (*nemaktis.field\_viewer.FieldViewer* attribute), 24

## C

`compensator` (*nemaktis.field\_viewer.FieldViewer* attribute), 24

`compensator_angle` (*nemaktis.field\_viewer.FieldViewer* attribute), 24

`copy()` (*nemaktis.light\_propagator.OpticalFields* method), 23

## D

`delete_mask()` (*nemaktis.lc\_material.DirectorField* method), 19

`DirectorField` (class in *nemaktis.lc\_material*), 19

## E

`extend()` (*nemaktis.lc\_material.DirectorField* method), 20

## F

`FieldViewer` (class in *nemaktis.field\_viewer*), 24

`focus_fields()` (*nemaktis.light\_propagator.OpticalFields* method), 23

`focused_vals()` (*nemaktis.light\_propagator.OpticalFields* property), 23

## G

`get_delta_qr()` (*nemaktis.light\_propagator.OpticalFields* method), 24

`get_image()` (*nemaktis.field\_viewer.FieldViewer* method), 25

`get_mesh_dimensions()` (*nemaktis.lc\_material.DirectorField* method), 20

`get_mesh_dimensions()` (*nemaktis.light\_propagator.OpticalFields* method), 24

`get_mesh_lengths()` (*nemaktis.lc\_material.DirectorField* method), 20

`get_mesh_lengths()` (*nemaktis.light\_propagator.OpticalFields* method), 24

`get_mesh_spacings()` (*nemaktis.lc\_material.DirectorField* method), 21

`get_mesh_spacings()` (*nemaktis.light\_propagator.OpticalFields* method), 24

`get_n_vertices()` (*nemaktis.lc\_material.DirectorField* method), 21

`get_n_vertices()` (*nemaktis.light\_propagator.OpticalFields* method), 24

`get_pos()` (*nemaktis.lc\_material.DirectorField* method), 20

`get_pos()` (*nemaktis.light\_propagator.OpticalFields* method), 23

`get_qr_index()` (*nemaktis.light\_propagator.OpticalFields* method), 23

`get_wavelengths()` (*nemaktis.light\_propagator.OpticalFields* method), 23

`get_wavevectors()` (*nemaktis.light\_propagator.OpticalFields* method), 23

`grayscale` (*nemaktis.field\_viewer.FieldViewer* attribute), 25

## I

`init_from_funcs()` (*nemaktis.lc\_material.DirectorField* method), 20

`intensity` (*nemaktis.field\_viewer.FieldViewer* attribute), 24

## L

`LCMaterial` (class in *nemaktis.lc\_material*), 21

`LightPropagator` (class in `nemaktis.light_propagator`), 21

## M

`mask_formula()` (`nemaktis.lc_material.DirectorField` property), 19

`mask_type()` (`nemaktis.lc_material.DirectorField` property), 19

`mask_vals()` (`nemaktis.lc_material.DirectorField` property), 19

`material()` (`nemaktis.light_propagator.LightPropagator` property), 22

## N

`n_tiles_x` (`nemaktis.field_viewer.FieldViewer` attribute), 24

`n_tiles_y` (`nemaktis.field_viewer.FieldViewer` attribute), 25

`NA_condenser` (`nemaktis.field_viewer.FieldViewer` attribute), 24

`NA_objective` (`nemaktis.field_viewer.FieldViewer` attribute), 25

`normalize()` (`nemaktis.lc_material.DirectorField` method), 20

## O

`OpticalFields` (class in `nemaktis.light_propagator`), 22

## P

`plot()` (`nemaktis.field_viewer.FieldViewer` method), 25

`polariser` (`nemaktis.field_viewer.FieldViewer` attribute), 24

`polariser_angle` (`nemaktis.field_viewer.FieldViewer` attribute), 24

`propagate_fields()` (`nemaktis.light_propagator.LightPropagator` method), 22

## R

`rescale_mesh()` (`nemaktis.lc_material.DirectorField` method), 20

`rotate()` (`nemaktis.lc_material.DirectorField` method), 20

`rotate_180deg()` (`nemaktis.lc_material.DirectorField` method), 20

`rotate_90deg()` (`nemaktis.lc_material.DirectorField` method), 20

## S

`save_to_vti()` (`nemaktis.lc_material.DirectorField` method), 20

`save_to_vti()` (`nemaktis.light_propagator.OpticalFields` method), 23

`set_mask()` (`nemaktis.lc_material.DirectorField` method), 19

## U

`update_image()` (`nemaktis.field_viewer.FieldViewer` method), 25

## V

`vals()` (`nemaktis.lc_material.DirectorField` property), 20

`vals()` (`nemaktis.light_propagator.OpticalFields` property), 23

## Z

`z_focus()` (`nemaktis.field_viewer.FieldViewer` property), 25