PyLlama

Release 1.0

May 03, 2022

Contents:

CHAPTER 1

Getting started

1.1 Overview

PyLlama enables to calculate the reflection and transmission spectra of an arbitrary multilayer stack whose layers are made of dispersive or non-dispersive, absorbing or non absorbing, isotropic or anisotropic materials. The layers are assumed to be perpendicular to the z axis and homogeneous and infinite in the x and y directions. The stack is sandwiched between an entry and an exit semi-infinite isotropic media, such as air.

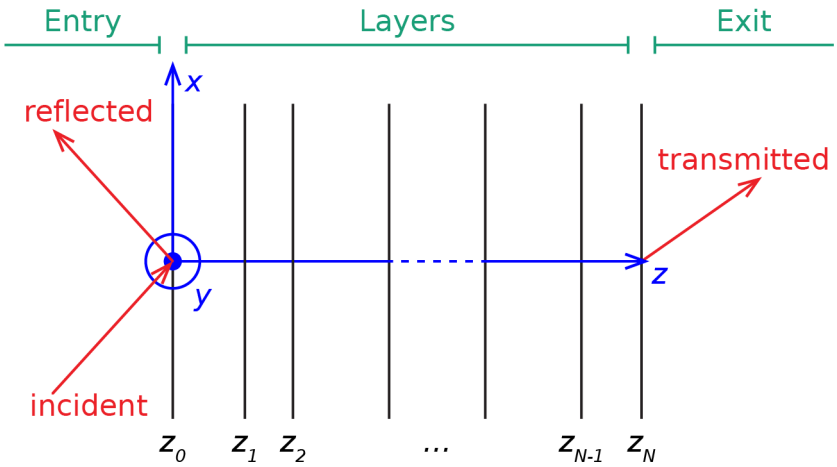


Fig. 1.1: Schematic of the axes and the multilayer stack between entry and exit semi-infinite media.

We use two different matrix method approaches to calculate the reflectance and transmittance of a multilayer stack: the transfer matrix method and the scattering method. Both methods are based on the same optical principles: continuity of the electric and magnetic fields at the interfaces between the layers (referred to as "transition") and phase build-up inside the layers (referred to as "propagation"). We describe the underlying theory in our paper that the user of the code is invited to read.

1.2 Code organisation

PyLlama is a code that can be used as a package (the user imports the code and uses the implemented classes and methods) **and** can be customised (the user writes their own routines to model specific multilayer stacks in form of children classes).

The code in `pyllama.py` is organised as follows:

- the three classes `Wave`, `Layer` (and its child `HalfSpace`) and `Structure` implement the optical calculations described in our paper. In principle, the user should not modify these classes.
- the class `Model` and its children contain routines that construct `Structure` instances through useful routines. The user should use these classes in scripts (some scripts are available on StackMat's GitHub repository) and may also add their own children classes to the code.
- the class `Spectrum` provides an extra level of automation to calculate full spectra in one command and to export results in Python-compatible (Pickles) or MATLAB format. The user should use this class in scripts and may also interface it with their custom child classes of `Model`.

1.3 How to install

`PyLlama` requires Python 3 to run. It has been tested with Python 3.6 from Python 3.8. It also requires the following packages (other versions may work too):

- Numpy version 1.18 to 1.19
- Sympy version 1.4 to 1.6
- Scypy version 1.2 to 1.5
- Matplotlib version 3.2

The file `pyllama.py` must be downloaded and placed in a location that is in Python's path. It contains the classes and function required to build multilayer stacks and calculate their reflectance. In each script, `PyLlama` must be imported with:

```
import pyllama
```

Custom libraries may be used in interaction with the class `Model` to construct `Structures`. The user should ensure that they have installed all the required libraries.

1.4 Library for cholesterics

The file `cholesteric.py` is required to work with the class `CholestericModel`, and the file `geometry.py` contains tools to represent cholesterics in 3D plots. If the used wishes to use the class `CholestericModel`, they need to import the `Cholesteric` class with:

```
import cholesteric
```

If the user does not wish to use the class `CholestericModel`, they do not need to download `cholesteric.py` nor `geometry.py`.

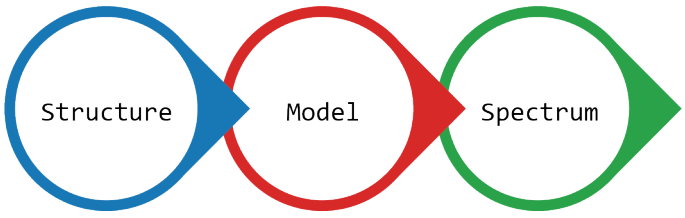
CHAPTER 2

Creating a multilayer stack

A multilayer stack and by extension a layer is defined for given incident conditions, which are:

- the wavelength of light λ, in nanometers
- the angle of incidence upon the stack θ_{in} defined in the entry isotropic semi-infinite medium in the (xz) plane, in degrees or in radians
- the associated wavevector $k = (k_x, k_y, k_z) = k_0(K_x, K_y, K_z)$:
  - k_0 is the normalised wavevector ($k_0 = 2\pi/\lambda$)
  - K_x is the x-component of the normalised wavevector that stays constant throughout the stack ($K_x = n_{entry}\sin(\theta_{in})$ where n_{entry} is the refractive index of the entry isotropic half-space)
  - K_y is its y-component (that equals to 0 by construction)
  - K_z is its z-component ($K_z = n_{entry}\cos(\theta_{in})$ in the entry semi-infinite isotropic medium)

There are three ways of creating a multilayer stack with `PyLlama`:



- creating a multilayer stack from scratch with the classes `Structure` and `Layer`, and working directly with classes that handle the optical calculations
- creating multilayer stacks with the abstract class `Model` and its children, and writing one's own child class when a different kind of stack is needed
- using the class `Spectrum` that provides a higher level of automation to automatically calculate reflection and transmission across a range of wavelengths

This section explains how to create a multilayer stack and the section *Calculating the reflection and transmission spectra of a stack* explains how to calculate their reflectance.

2.1 From scratch: the technical way

A multilayer stack consists in a series of layers sandwiched between an entry and an exit semi-infinite isotropic media. A layer is represented by the class `Layer` and the semi-infinite isotropic media are represented by the class `HalfSpace`.

The entry and exit semi-infinite isotropic media are instances of the class `HalfSpace` and are created with:

```
k0 = 2 * numpy.pi / wl_nm
Kx = n_entry * numpy.sin(theta_in_rad)
Kz_entry = n_entry * numpy.cos(theta_in_rad)
theta_out_rad = numpy.arcsin((n_entry / n_exit) * numpy.sin(self.theta_in_rad))
Kz_exit = n_exit * numpy.cos(theta_out_rad)
epsilon_entry = numpy.array([[n_entry ** 2, 0, 0],
                             [0, n_entry ** 2, 0],
                             [0, 0, n_entry ** 2]])
epsilon_exit = numpy.array([[n_exit ** 2, 0, 0],
                            [0, n_exit ** 2, 0],
                            [0, 0, n_exit ** 2]])
entry = HalfSpace(epsilon_entry, Kx, Kz_entry, k0)
exit = HalfSpace(epsilon_exit, Kx, Kz_exit, k0)
```

where:

- `n_entry` is the refractive index of the entry isotropic medium
- `n_exit` is the refractive index of the exit isotropic medium
- `theta_in_rad` is the angle of incidence in the entry isotropic medium in radians
- `wl_nm` is the wavelength of light, in nanometers
- `k0` is the wavevector magnitude
- `Kx` is the x-component of the normalised wavevector that stays constant through the stack
- `Kz_entry` and `Kz_exit` are the z-components of the normalised wavevector in the entry and exit isotropic half-spaces
- `epsilon_entry` and `epsilon_exit` are the permittivity tensors of the entry and exit isotropic half-spaces

A multilayer stack whose layers are embedded between the semi-infinite isotropic media `entry` and `exit` is then created with:

```
Ky = 0
Kz = n_entry * numpy.cos(theta_in_rad)
my_stack_structure = Structure(entry, exit, Kx, Ky, Kz_entry, Kz_exit, k0)
```

where:

- `Ky` is the y-component of the normalised wavevector that is equal to 0 throughout the stack

At this point, `my_stack_structure` represents two semi-infinite isotropic half-space that sandwich no layer. Layers are instances of the class `Layer` and are created with:

```
my_layer = Layer(epsilon, thickness_nm, Kx, k0)
```

where:

- `k0` is the normalised wavevector
- `Kx` is the x-component of the normalised wavevector

- `epsilon` is the permittivity tensor (3x3 Numpy array) of the layer, which can represent a material that is isotropic or anisotropic, absorbing or non-absorbing
- `thickness_nm` is the thickness of the layer in nanometers

The z-component of the normalised wavevector changes inside the stack and is not defined in the `Layer`: the partial waves will be calculated instead.

Then, the layer can be added to the stack with:

```
my_stack_structure.add_layer(my_layer)
```

The content of a stack can then be accessed with:

```
my_stack_structure.entry  # access the entry HalfSpace
my_stack_structure.exit   # access the exit HalfSpace
my_stack_structure.layers # access the list of Layers in the stack
```

The functions `add_layers()`, `remove_layer()` and `replace_layer()` also enable the user to construct the stacks that they want.

Note: k_0 and K_x stay constant throughout the stack and checks are carried out in the function `Structure.add_layer()` to ensure that the user only adds `Layers` that are compatible with the stack. The user should **not** add `Layers` with `my_stack.layers.append(my_layer)` as this may lead to impossible situations.

Lastly, the periodicity of the stack can be changed with:

```
my_stack_structure.N_periods = number_of_periods
```

The `Layers` in the list `my_stack.layers` represent one periodic pattern that is repeated `number_of_periods` times in the stack. A multilayer stack made of `N` repetitions of a periodic unit consisting in `layer_a` with a permittivity `eps_a` and a thickness `thick_a` (in nanometers) and `layer_b` with a permittivity `eps_b` and a thickness `thick_b` (in nanometers) is defined with:

```
my_stack_structure = Structure(entry, exit, Kx, Ky, Kz_entry, Kz_exit, k0)
layer_a = Layer(eps_a, thick_a, Kx, k0)
layer_b = Layer(eps_b, thick_b, Kx, k0)
my_stack_structure.add_layers([layer_a, layer_b])
my_stack_structure.N_per = N
```

The leftmost `Layer` in the list is located after the entry half-space and the rightmost `Layer` in the list is located before the exit half-space. The periodic pattern can include an arbitrary number of layers and we used two layers as an example.

2.2 With the `Model` class: the flexible way

Creating `Layers` from scratch to build up a multilayer stack can become constraining. Instead, pre-defined routines (`Models`) allow the user to create particular multilayer stacks such as single slabs, periodic Bragg stacks, isotropic stacks where each layer has the same optical thickness for a given wavelength, and cholesteric stacks more easily.

The class `Model` is a general class that gives a blueprint for all its specific children classes: `SlabModel`, `StackModel`, `StackOpticalThicknessModel`, `CholestericModel`, etc. "Giving the blueprint" means that parameters and functions that are common to all model classes are defined in the class `Model`, and then its children classes inherit them, in addition to having their own specific parameters and functions. The user should directly use the children classes.

Note: Classes that give the blueprint for their children classes are usually called "abstract classes" and cannot be instanciated but it is here possible to instanciate `Model`: it creates a stack with no layer.

A multilayer stack made of `N` repetitions of a periodic unit consisting in a first layer with a permittivity `eps_a` and a thickness `thick_a` (in nanometers) and a second layer with a permittivity `eps_b` and a thickness `thick_b` (in nanometers) can be represented with `StackModel` and is defined with:

> my_stack_model = StackModel([eps_a, eps_b], [thick_a, thick_b], n_entry, n_exit, wl_nm, theta_in_rad, N)

where the following parameters are required by all `Models`:

- `n_entry` is the refractive index of the entry isotropic medium
- `n_exit` is the refractive index of the exit isotropic medium
- `theta_in_rad` is the angle of incidence in the entry isotropic medium in radians
- `wl_nm` is the wavelength of light, in nanometers

while the lists `[eps_a, eps_b]` and `[thick_a, thick_b]` are required specifically by `StackModel`.

The documentation of the classes `SlabModel`, `StackOpticalThicknessModel` and `CholestericModel` provide information on which parameters are required to create stacks with these specific classes.

2.3 With the `Spectrum` class: the automated way

The `Spectrum` class provides a further level of automation for the user. It is meant for experimentalists who measure spectra from multilayered samples and want to quickly model their sample. The class `Spectrum` enables to get a full spectrum in one go and to export it for MATLAB or Python processing. The creation of a `Spectrum` is not more convenient than the creation of a `Model` but they will differ in their utilisation.

A `Spectrum` can be created the following way:

```
my_stack_spec = Spectrum(wl_nm_list, model_type, model_parameters)
```

where:

- `wl_nm_list` is a list of wavelengths
- `model_type` is a string that describes the type of model to use
- `model_parameters` is a dictionary that contains all parameters needed for the model

For example, to create a `Spectrum` in the visible for the periodic multilayer stack described in the previous examples, the input parameters are:

```
wl_nm_list = range(400, 800)
model_type = "StackModel"
model_parameters = {"eps_list": [eps_a, eps_b],
                    "thickness_nm_list": [thick_a, thick_b],
                    "n_entry": n_entry,
                    "n_exit": n_exit,
                    "theta_in_rad": theta_in_rad,
                    "N_per": N}
```

The documentation of the classes `SlabModel`, `StackOpticalThicknessModel` and `CholestericModel` provide information on which parameters are required to create the appropriate dictionaries for their associated `Spectra`.

Note: `Models` may have default parameters (for example, when the user does not specify a number of periods for `StackModel`, the value is set to 1 automatically), which is specified in their respective documentation. This is maintained in their associated `Spectra`: the dictionary of parameters created by the user is merged with a dictionary of default parameters.

CHAPTER 3

Calculating the reflection and transmission spectra of a stack

The section *Creating a multilayer stack* explains how to create the multilayer stack with three methods, which all enable to calculate the reflection and transmission spectra of the stack with different level of additional details:

- creating a multilayer stack from scratch with the classes `Structure` `Layer`, and working directly with classes that handle the optical calculations. **This method gives direct access to the partial waves inside each layer of the multilayer stack, to the transfer and scattering matrices, to the multilayer stack's reflection and transmission coefficients and to the multilayer stack's reflectance and transmittance for one wavelength.**
- creating multilayer stacks with the abstract class `Model` and its children, and writing one's own child class when a different kind of stack is needed. **This method gives direct access to the multilayer stack's reflectance and transmittance for one wavelength.** Additionally, the layer's partial waves, the transfer and scattering matrices and the reflection and transmission coefficients can also be obtained since the `Model` creates a `Structure`.
- using the class `Spectrum` that provides a higher level of automation. **This method gives direct access to the multilayer stack's reflectance and transmittance for a range of wavelength.**

This section explains how to get the reflection spectrum of a multilayer stack which has been created through one of there three methods, following the tutorials *Creating a multilayer stack*.

3.1 From scratch: the technical way

When the user follows the method "from scratch" with the class `Structure` to create a multilayer stack `my_stack_structure`, they interact directly with the classes that handle the optics calculation. The `Structure` that represents the multilayer stack contains a list of `Layers` and the entry and exit `HalfSpaces` (which are children of `Layer`). `Layers` and `HalfSpaces` implement the calculation of Berreman's matrix and of the layer's eigenvalues and eigenvectors, used to calculate the layer's partial waves. These are calculated immediately upon the creation of the `Layer` or `HalfSpace` and can be accessed through:

```
my_stack_structure.layers[k].D                # layer's Berreman's matrix
my_stack_structure.layers[k].eigenvalues      # layer's eigenvalues
my_stack_structure.layers[k].eigenvectors     # layer's eigenvectors
my_stack_structure.layers[k].partial_waves    # layer's partial waves
```

where `k` is the index of the `Layer` in the `Structure` `my_stack_structure`.

A `Structure` therefore automatically contains a series of four partial waves per layer, which are used to construct its transfer matrix or its scattering matrix (for the wavelength that was used to create the `Structure`). The transfer and scattering matrices can be calculated the following way:

```
my_stack_structure.build_transfer_matrix()     # transfer matrix
my_stack_structure.build_scattering_matrix()   # scattering matrix
```

Reflection and transmission coefficients in the linear polarisation basis (for the wavelength that was used to create the `Structure`) can be calculated with:

```
J_refl_lin, J_trans_lin = my_stack_structure.get_fresnel()
```

and converted to the circular polarisation basis with:

```
J_refl_circ, J_trans_circ = Structure.fresnel_to_fresnel_circ(J_lin)
```

The results are two 2×2 Numpy arrays of reflection coefficients (r) and transmission coefficients (t) organised the following way:

- in the linear polarisation basis:

$$r_{lin} = \begin{bmatrix} r_{p\,\text{to}\,p} & r_{s\,\text{to}\,p} \\ r_{p\,\text{to}\,s} & r_{s\,\text{to}\,s} \end{bmatrix}$$
$$t_{lin} = \begin{bmatrix} t_{p\,\text{to}\,p} & t_{s\,\text{to}\,p} \\ t_{p\,\text{to}\,s} & t_{s\,\text{to}\,s} \end{bmatrix}$$

- in the circular polarisation basis:

$$r_{circ} = \begin{bmatrix} r_{RCP\,\text{to}\,RCP} & r_{LCP\,\text{to}\,RCP} \\ r_{RCP\,\text{to}\,LCP} & r_{LCP\,\text{to}\,LCP} \end{bmatrix}$$
$$t_{circ} = \begin{bmatrix} t_{RCP\,\text{to}\,RCP} & t_{LCP\,\text{to}\,RCP} \\ t_{RCP\,\text{to}\,LCP} & t_{LCP\,\text{to}\,LCP} \end{bmatrix}$$

For example, the user can access the reflection coefficient for incoming s-polarised light reflected as p-polarised light of the multilayer stack represented by the `Structure` `my_stack_structure` with:

```
J_lin, _ = my_stack_structure.get_fresnel()
J_lin[0, 1]
```

The reflectance and transmittance of the multilayer stack (for the wavelength that was used to create the `Structure`) can be obtained with:

```
my_stack_structure.get_refl_trans(circ=<False|True>, method=<"SM"|"TM">)
```

where `method` defines the matrix method used (`"SM"` (default) for the scattering matrix method and `"TM"` for the transfer matrix method) and `circ=False` (default) calculates the reflectance and transmittance in the linear polarisation basis and `circ=True` calculates them in the circular polarisation basis.

The results are two 2×2 Numpy arrays of reflectances (R) organised the following way:

- in the linear polarisation basis:

$$R_{lin} = \begin{bmatrix} R_{p\,\text{to}\,p} & R_{s\,\text{to}\,p} \\ R_{p\,\text{to}\,s} & R_{s\,\text{to}\,s} \end{bmatrix}$$
$$T_{lin} = \begin{bmatrix} T_{p\,\text{to}\,p} & T_{s\,\text{to}\,p} \\ T_{p\,\text{to}\,s} & T_{s\,\text{to}\,s} \end{bmatrix}$$

- in the circular polarisation basis:

$$R_{circ} = \begin{bmatrix} R_{RCP\,\text{to}\,RCP} & R_{LCP\,\text{to}\,RCP} \\ R_{RCP\,\text{to}\,LCP} & R_{LCP\,\text{to}\,LCP} \end{bmatrix}$$
$$T_{circ} = \begin{bmatrix} T_{RCP\,\text{to}\,RCP} & T_{LCP\,\text{to}\,RCP} \\ T_{RCP\,\text{to}\,LCP} & T_{LCP\,\text{to}\,LCP} \end{bmatrix}$$

To calculate the reflection and transmission spectra of the stack over a range of wavelengths, the user must create a new `Structure` for each wavelength and recalculate the reflectance, for example with:

```
# Creation of an empty variable
reflection_s_to_p = []

# Creation of the wavelengths
wl_nm_list = range(400, 800)

# Calculation of the reflectance for each wavelength
for wl_nm in wl_nm_list:
    # Calculation of the wavevector
    k0 = 2 * numpy.pi / wl_nm
    Kx = n_entry * numpy.sin(theta_in_rad)
    Ky = 0
    Kz_entry = n_entry * numpy.cos(theta_in_rad)
    theta_out_rad = numpy.arcsin((n_entry / n_exit) * numpy.sin(self.theta_in_rad))
    Kz_exit = n_exit * numpy.cos(theta_out_rad)

    # Creation of the entry and exit half-spaces and of the two layers
    entry = HalfSpace(epsilon_entry, Kx, Kz_entry, k0)
    exit = HalfSpace(epsilon_exit, Kx, Kz_exit, k0)
    layer_a = Layer(eps_a, thick_a, Kx, k0)
    layer_b = Layer(eps_b, thick_b, Kx, k0)

    # Creation of the periodic stack
    my_stack_structure = Structure(entry, exit, Kx, Ky, Kz_entry, Kz_exit, k0)
    my_stack_structure.add_layers([layer_a, layer_b])
    my_stack_structure.N_periods = N

    # Calculation of the reflectance and storage
    J_refl_lin, _ = my_stack_structure.get_refl_trans()
    reflection_s_to_p.append(J_refl_lin[0, 1])

# Plotting
matplotlib.pyplot.plot(wl_nm_list, reflection_s_to_p)
```

where:

- `eps_a` and `eps_b` are the permittivity tensors (3x3 Numpy array) of the layer, which can represent a material that is isotropic or anisotropic, absorbing or non-absorbing
- `thick_a` and `thick_b` are the thicknesses of the two layers of the periodic pattern, in nanometers

- `N` is the number of periods
- ‘‘ theta_in_rad‘‘ is the angle of incidence upon the stack, in radians
- `eps_entry` and `eps_exit` are the permittivities of the two isotropic half-spaces; they can be defined differently for each wavelength if the materials are dispersive

3.2 With the `Model` class: the flexible way

When the user creates a multilayer stack `my_stack_model` through one of the `Model` children classes, the reflectance and transmittance of the multilayer stack (for the wavelength that was used to create the `Structure`) can be obtained with:

```
my_stack_model.get_refl_trans(circ=<False|True>, method=<"SM"|"TM">)
```

where `method` defines the matrix method used (`"SM"` (default) for the scattering matrix method and `"TM"` for the transfer matrix method) and `circ=False` (default) calculates the reflectance and transmittance in the linear polarisation basis and `circ=True` calculates them in the circular polarisation basis.

The results are two 2×2 Numpy arrays of reflectances (R) organised the following way:

- in the linear polarisation basis:

$$R_{lin} = \begin{bmatrix} R_{p\text{ to }p} & R_{s\text{ to }p} \\ R_{p\text{ to }s} & R_{s\text{ to }s} \end{bmatrix}$$
$$T_{lin} = \begin{bmatrix} T_{p\text{ to }p} & T_{s\text{ to }p} \\ T_{p\text{ to }s} & T_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis:

$$R_{circ} = \begin{bmatrix} R_{RCP\text{ to }RCP} & R_{LCP\text{ to }RCP} \\ R_{RCP\text{ to }LCP} & R_{LCP\text{ to }LCP} \end{bmatrix}$$
$$T_{circ} = \begin{bmatrix} T_{RCP\text{ to }RCP} & T_{LCP\text{ to }RCP} \\ T_{RCP\text{ to }LCP} & T_{LCP\text{ to }LCP} \end{bmatrix}$$

Note: Each children class of `Model` contains a `Structure` that can be accessed through `my_stack_model.structure` and the the previous part of this tutorial can be applied to `my_stack_model.structure` to access the partial waves, the transfer or scattering matrices and the reflection and transmission coefficients.

To calculate the reflection and transmission spectra of the stack over a range of wavelengths, the user must create a new `Model` for each wavelength and recalculate the reflectance and transmittance, for example with:

```
# Creation of an empty variable
reflection_s_to_p = []

# Creation of the wavelengths
wl_nm_list = range(400, 800)

# Calculation of the reflectance for each wavelength
for wl_nm in wl_nm_list:
```

(continues on next page)

(continued from previous page)

```
    # Creation of the periodic stack
    my_stack_model = StackModel([eps_a, eps_b],
                                [thick_a, thick_b],
                                n_entry,
                                n_exit,
                                wl_nm,
                                theta_in_rad,
                                N)

    # Calculation of the reflectance and storage
    J_refl_lin, _ = my_stack_model.get_refl_trans()
    reflection_s_to_p.append(J_refl_lin[0, 1])

# Plotting
matplotlib.pyplot.plot(wl_nm_list, reflection_s_to_p)
```

where:

- `eps_a` and `eps_b` are the permittivity tensors (3x3 Numpy array) of the layer, which can represent a material that is isotropic or anisotropic, absorbing or non-absorbing
- `thick_a` and `thick_b` are the thicknesses of the two layers of the periodic pattern, in nanometers
- `N` is the number of periods
- ‘‘ theta_in_rad‘‘ is the angle of incidence upon the stack, in radians
- `n_entry` and `n_exit` are the refractive indices of the two isotropic half-spaces; they can be defined differently for each wavelength if the materials are dispersive

3.3 With the `Spectrum` class: the automated way

When the user creates a multilayer stack `my_stack_spec` through the `Spectrum` class, the reflection and transmission spectra of the multilayer stack (for the range of wavelength that was inputted in the `Spectrum`) can be obtained with:

```
my_stack_spectrum.calculate_refl_trans(circ=<False|True>, method=<"SM"|"TM">, talk=
↪<False|True>)
```

where `method` defines the matrix method used (`"SM"` (default) for the scattering matrix method and `"TM"` for the transfer matrix method), `circ=False` (default) calculates the reflectance and transmittance in the linear polarisation basis and `circ=True` calculates them in the circular polarisation basis, and `talk=True` enables to display the calculation progress on the screen (default is `False`).

The calculated reflection spectra are stored into the dictionary `my_stack_spectrum.data` and can be accessed with:

- in the linear polarisation basis: `my_stack_spectrum.data["R_p_to_p_to_p"]`, `my_stack_spectrum.data["R_s_to_p"]`, `my_stack_spectrum.data["R_p_to_s"]`, `my_stack_spectrum.data["R_s_to_s"]`
- in the circular polarisation basis: `my_stack_spectrum.data["R_R_to_R"]`, `my_stack_spectrum.data["R_L_to_R"]`, `my_stack_spectrum.data["R_R_to_L"]`, `my_stack_spectrum.data["R_L_to_L"]`

and similarly for the transmission spectra: - in the linear polarisation basis: `my_stack_spectrum.data["T_p_to_p"]`, `my_stack_spectrum.data["T_s_to_p"]`, `my_stack_spectrum.data["T_p_to_s"]`,`my_stack_spectrum.data["T_s_to_s"]`

- in the circular polarisation basis: `my_stack_spectrum.data["T_R_to_R"]`, `my_stack_spectrum.data["T_L_to_R"]`, `my_stack_spectrum.data["T_R_to_L"]`, `my_stack_spectrum.data["T_L_to_L"]`

The calculated spectra (everything stored in `my_stack_spectrum.data`) can then be exported in MATLAB or Python-compatible format with:

```
my_stack_spectrum.export(path_out, with_param=<True|False>)
```

where:

- `path_out` is the name of the file. If it ends with `.mat`, the export will be in MATLAB-compatible format, and if it ends with `.pck`, the export will be in Python-compatible format (with Pickles)
- `with_param` is set to `True` (default) when the parameters user for the model are exported too and to `False` when they are not exported

---

Note: Some `Models` may take as input parameters objects that are created through the user's custom-made libraries (for example, `CholestericModel` requires an instance of a `Cholesteric` as a parameter). These objects will be stored in the `Model`'s parameters. MATLAB can import any unknown object in shape of MATLAB's type `struct` but Python can only import objects for whose it can load the libraries that created them. In this case, exporting the spectra without the parameters may be useful, but this is not the default option.

The calculation the reflection spectrum of the stack over a range of wavelengths is automatic, for example with:

```
# Creation of the wavelengths
wl_nm_list = range(400, 800)

# Parameters for the stack
model_type = "StackModel"
model_parameters = {"eps_list": [eps_a, eps_b],
                    "thickness_nm_list": [thick_a, thick_b],
                    "n_entry": n_entry,
                    "n_exit": n_exit,
                    "theta_in_rad": theta_in_rad,
                    "N_per": N}

# Creation of the periodic stack
my_stack_spec = Spectrum(wl_nm_list, model_type, model_parameters)

# Calculation of the reflectance spectrum in one go
my_stack_spec.calculate_refl_trans()

# Plotting
matplotlib.pyplot.plot(wl_nm_list, my_stack_spec.data["R_s_to_p"])

# Export for MATLAB
    # All polarisation combinations are exported (p to p, s to p, p to p, s to s)
my_stack_spec.export("my_file_name.mat")
```

where:

- `eps_a` and `eps_b` are the permittivity tensors (3x3 Numpy array) of the layer, which can represent a material that is isotropic or anisotropic, absorbing or non-absorbing; if the material is dispersive, a `Model` different than

`StackModel` must be used that is able to handle a list of permittivities

- `thick_a` and `thick_b` are the thicknesses of the two layers of the periodic pattern, in nanometers
- `N` is the number of periods
- `theta_in_rad` is the angle of incidence upon the stack, in radians
- `n_entry` and `n_exit` are the refractive indices of the two isotropic half-spaces; they can be defined differently for each wavelength if the materials are dispersive

CHAPTER 4

Quick cholesteric tutorial

In this section aimed at users who are not experienced in programming and/or in Python, we explain a code minimal working example to:

- create a cholesteric architecture with a chosen pitch, tilt and handedness
- obtain a 3D representation of the cholesteric architecture
- calculate the reflectance in transmittance in the circular polarisation basis for a chosen angle of incidence
- extract and plot the results in Python
- export the results for further processing in MATLAB

First, the user must import the required packages. We need `PyLLama` for the optical calculations, `Cholesteric` to create the cholesteric architecture, `NumPy` for basic numerical calculation and `Matplotlib.PyPlot` for plotting. In Python, all packages are imported in lowercaps at the beginning of the code and they are often given a shorter name (`ch` instead of `cholesteric` for example).

```
# Import the required packages
import pyllama as ll
import cholesteric as ch
import numpy as np
import matplotlib.pyplot as plt
```

Then, the user creates their `Cholesteric` object with `ch.Cholesteric`:

```
# Cholesteric object
pitch_nm = 500
tilt_rad = 10 * np.pi / 180
chole = ch.Cholesteric(pitch360=pitch_nm,
                       tilt_rad=tilt_rad,
                       handedness=1)
```

The user can obtain a 3D representation of the `Cholesteric` object with the function `plot_simple()`. The `view` parameters defines the viewing angle and the `type` parameters defines how the pseudolayers are represented (here, with arrows). The user should note that the x, y and z axes might not have exactly the same scale.

```
# 3D representation of the cholesteric object
fig_3D, ax_3D = chole.plot_simple(view="classic", type="arrow")
```

The user can then define the incident conditions upon the cholesteric and obtain their 3D representation. When the cholesteric is tilted, the z axis will be shifted to the helical axis for the calculations.

```
# Incident conditions and 3D representation
theta_in_deg = 50
theta_in_rad = theta_in_deg * np.pi / 180
chole.plot_add_optics(fig_3D, ax_3D, theta_in_rad)
```

The output is displayed on Figure Fig. 4.1.

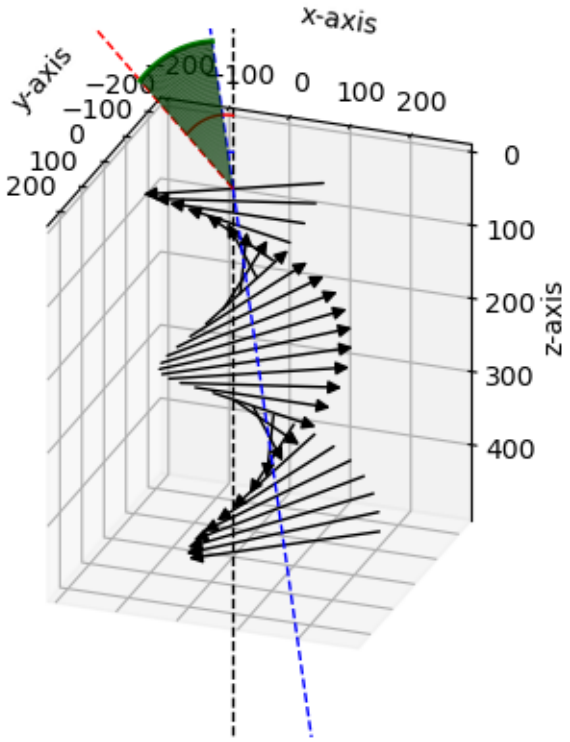


Fig. 4.1: 3D representation of `chole` obtained with the functions `plot_simple()` and `add_optics()`.

Then, the user can choose the optical parameters such as the average refractive index and the birefringence, and create a `Spectrum` object:

```
# Optical parameters
n_av = 1.433
biref = 0.04
n_e = n_av + 0.5 * biref
n_o = n_av - 0.5 * biref
n_entry = n_av
```

(continues on next page)

(continued from previous page)

```
n_exit = n_av
N_per = 20
wl_nm_list = np.arange(400, 800)

# Creation of the spectrum
spectrum = ll.Spectrum(wl_nm_list,
                       "CholestericModel",
                       dict(chole=chole,
                            n_e=n_e,
                            n_o=n_o,
                            n_entry=n_entry,
                            n_exit=n_exit,
                            N_per=N_per,
                            theta_in_rad=theta_in_rad))
```

The calculation of the reflectance and transmittance is done in one go; the parameter `circ=True` enables to calculate the reflectance and transmittance in the circular polarisation basis, the parameter `method="SM"` enables to choose the always-accurate scattering matrix method for the calculation and the parameter `talk=True` allows do display the calculation progress on the screen. The spectrum will take about 30 seconds to be calculated.

```
# Calculation of the reflectance and transmittance
spectrum.calculate_refl_trans(circ=True, method="SM", talk=True)
```

The results can be plotted:

```
# Plot the spectra
fig = plt.figure()

ax1 = fig.add_subplot(311)
ax1.plot(wl_nm_list, spectrum.data['R_R_to_R'], label="RCP to RCP")
ax1.plot(wl_nm_list, spectrum.data['R_R_to_L'], label="RCP to LCP")
plt.legend(loc=2)
plt.xlim([400, 800])
plt.ylim([0, 1])
plt.xlabel('Wavelength (nm)')
plt.ylabel('Reflectance')
ax1.set_title('Incoming RCP')

ax2 = fig.add_subplot(312)
ax2.plot(wl_nm_list, spectrum.data['R_L_to_R'], label="LCP to RCP")
ax2.plot(wl_nm_list, spectrum.data['R_L_to_L'], label="LCP to LCP")
plt.legend(loc=2)
plt.xlim([400, 800])
plt.ylim([0, 1])
plt.xlabel('Wavelength (nm)')
plt.ylabel('Reflectance')
ax2.set_title('Incoming LCP')
```

For incoming unpolarised light, the user should not forget to average the incoming RCP and incoming LCP:

```
ax3 = fig.add_subplot(313)
ax3.plot(wl_nm_list, 0.5 * (spectrum.data['R_R_to_R']
                            + spectrum.data['R_R_to_L']
                            + spectrum.data['R_L_to_R']
                            + spectrum.data['R_L_to_L']),
                            label="reflection")
```

(continues on next page)

(continued from previous page)

```
ax3.plot(wl_nm_list, 0.5 * (spectrum.data['T_R_to_R']
                            + spectrum.data['T_R_to_L']
                            + spectrum.data['T_L_to_R']
                            + spectrum.data['T_L_to_L']),
                            label="transmission")
plt.legend(loc=2)
plt.xlim([400, 800])
plt.ylim([0, 1])
plt.xlabel('Wavelength (nm)')
plt.ylabel('Intensity')
ax3.set_title('Incoming unpolarised')
```

The user can make the layout of the plots a bit nicer, before saving and displaying the figure (without `plt.show()`, the plots will not be shown on the computer screen):

```
plt.tight_layout()
fig.savefig("script_cholesteric_example.png", dpi=300)
plt.show()
```

The output is displayed on Figure Fig. 4.2.

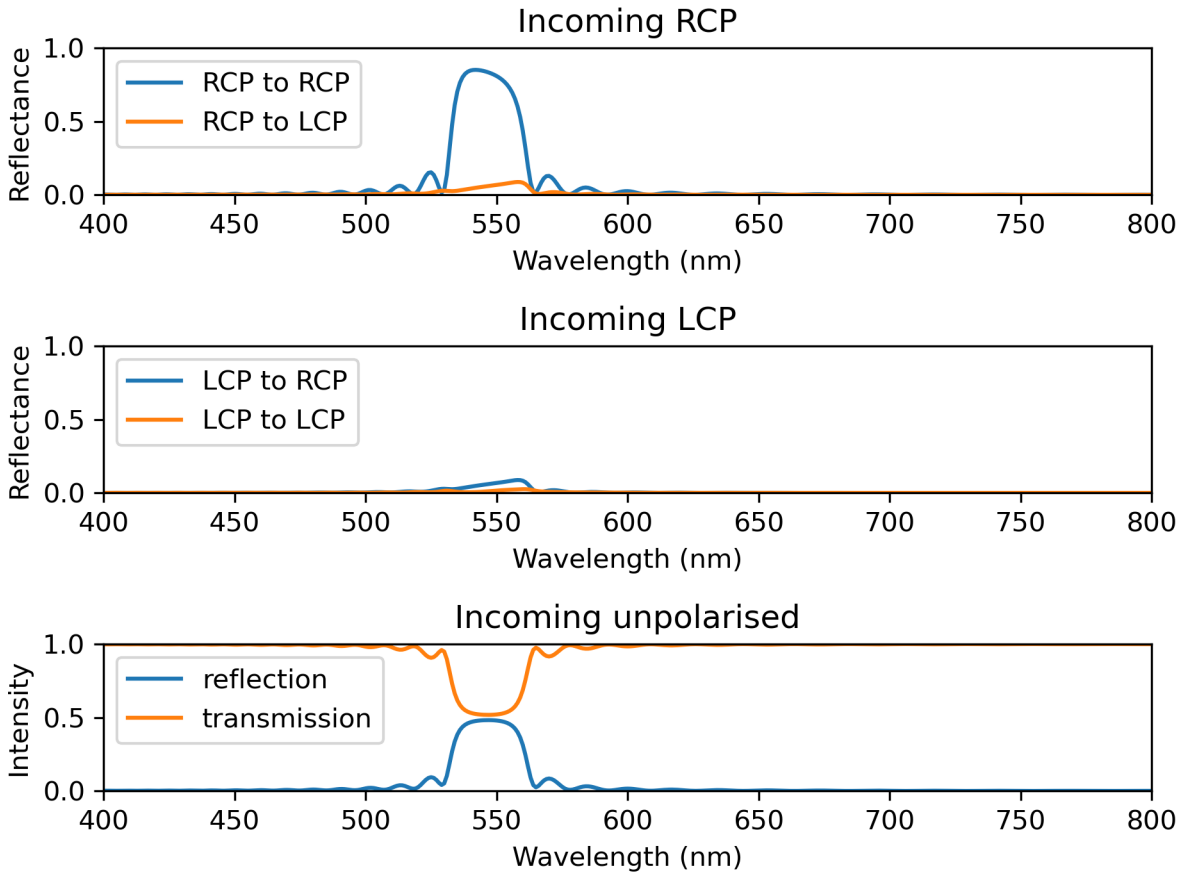


Fig. 4.2: Spectra obtained with the `Spectrum` and `CholestericModel`.

The results can be exported to MATLAB very simply for further processing. The parameters such as the refractive

indices are also exported, as well as all the fields of the cholesteric object (such as the directors of the pseudo-layers). To export with Pickles (for Python further processing), the user should replace the extension `.mat` by `.pck`.

```
# Export the spectra to an external file
path_out = "pyllama_cholesteric_spectrum.mat"
spectrum.export(path_out)
```

CHAPTER 5

Choosing which matrix method to use

Note: The theory that leads to the equations presented in this section is detailed in "PyLlama: a stable and flexible Python toolkit for the electromagnetic modeling of multilayered anisotropic media" (in preparation).

The transfer matrix and the scattering matrix both link the incoming, reflected and transmitted electric fields, but their equations are built differently. The notations are displayed on Figure Fig. 5.1.

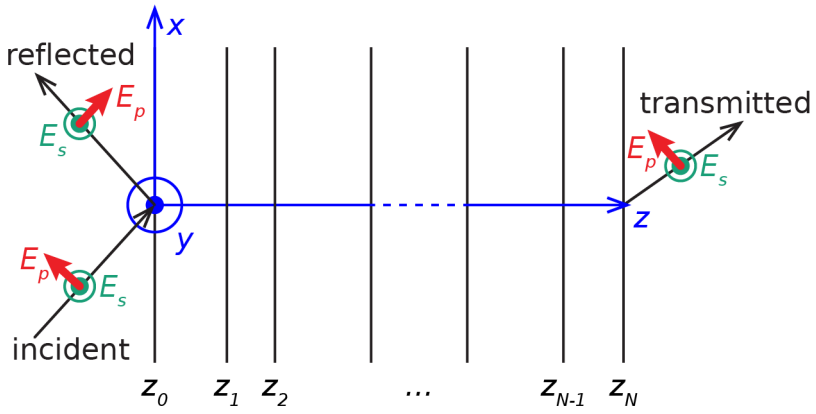


Fig. 5.1: Schematic of the electric field direction for p and s-polarisations.

The transfer matrix equation is:

$$\begin{bmatrix} E_p \text{ transmitted} \\ E_s \text{ transmitted} \\ 0 \\ 0 \end{bmatrix} = T \begin{bmatrix} E_p \text{ incident} \\ E_s \text{ incident} \\ E_p \text{ reflected} \\ E_s \text{ reflected} \end{bmatrix}$$

and the scattering matrix equation is:

$$\begin{bmatrix} E_p \text{ transmitted} \\ E_s \text{ transmitted} \\ E_p \text{ reflected} \\ E_s \text{ reflected} \end{bmatrix} = S \begin{bmatrix} E_p \text{ incident} \\ E_s \text{ incident} \\ 0 \\ 0 \end{bmatrix}$$

The reflection and transmission coefficients are extracted from either of these matrix equations to obtain:

$$r_{p\text{ to }p} = \frac{E_p\text{reflected}}{E_p\text{incident}} \quad r_{s\text{ to }p} = \frac{E_p\text{reflected}}{E_s\text{incident}}$$
$$r_{p\text{ to }s} = \frac{E_s\text{reflected}}{E_p\text{incident}} \quad r_{s\text{ to }s} = \frac{E_s\text{reflected}}{E_s\text{incident}}$$

and:

$$t_{p\text{ to }p} = \frac{E_p\text{transmitted}}{E_p\text{incident}} \quad t_{s\text{ to }p} = \frac{E_p\text{transmitted}}{E_s\text{incident}}$$
$$t_{p\text{ to }s} = \frac{E_s\text{transmitted}}{E_p\text{incident}} \quad t_{s\text{ to }s} = \frac{E_s\text{transmitted}}{E_s\text{incident}}$$

Mathematically, both equations lead to the same result and choosing which one to use has no impact on the reflection and transmission coefficients, but numerically, differences can be observed in the robustness and in the computation time.

As an example, we built two cholesterics with a different birefringence (`chole_1` with $\Delta n = 0.05$ and `chole_2` with $\Delta n = 0.2$) and a different number of periods (20 for `chole_1` and 200 for `chole_2`), and confronted the transfer and scattering matrix methods. A we can see on Figure Fig. 5.2, the two matrix methods give the exact same results when the birefringence is low and when the number of periods is low. However, for a higher birefringence and a larger number of periods, the transfer matrix (be it calculated with the eigenvalues and eigenvectors or with the direct exponential of Berreman's matrix) is numerically unstable.

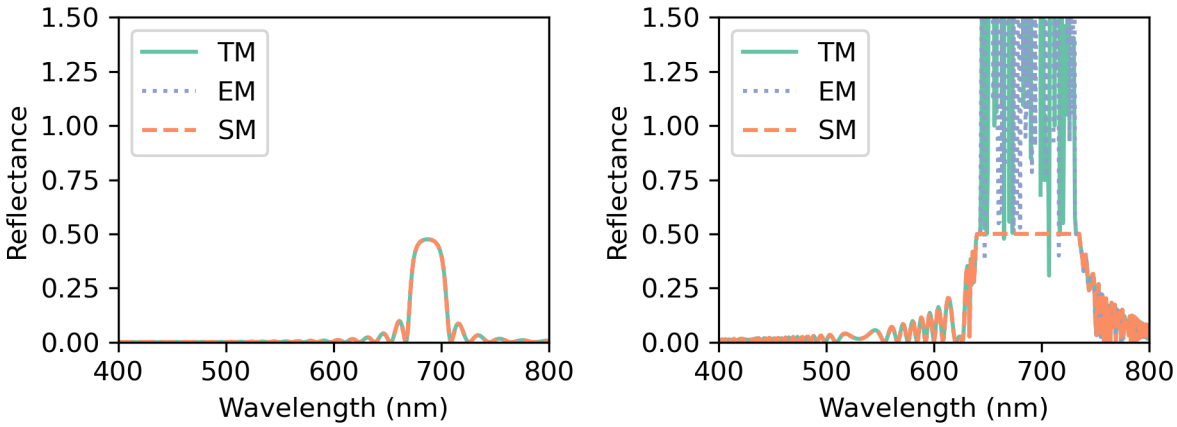


Fig. 5.2: Reflection spectra calculated with the transfer matrix method (`TM` and `EM`) and the scattering matrix method (`SM`) for `chole_1` (left) and `chole_2` (right).

With the scattering matrix method (`SM`), the spectra took about 40 seconds to be calculated on a laptop, against 20 seconds for the transfer matrix method with the eigenvalue and eigenvector calculation (`TM`) and 35 seconds for the transfer matrix method with the direct exponential of Berreman's matrix (`EM`).

The `SM` method always gives accurate results. However, for speed, the user may prefer using the `TM` transfer matrix method. The user should always check that the results calculated with `TM` match these calculated with `SM` in the most critical conditions of the user's range of parameters (higher birefringence, larger number of periods, larger angle of incidence).

CHAPTER 6

Creating a custom `Model` class

The `Model` class and its children enable the user to construct `Structures` automatically from given parameters. The class `Model` can be viewed as an abstract class that defines parameters and methods common to all its children; however, it is possible to create an instance of `Model`: it will have an empty `Structure`. In the tutorial, we explain how the class `Model` is constructed and how the user can write their own child class.

6.1 Anatomy of the `Model` class

The parameters of the `Model` class are:

- `n_entry`: the refractive index of the stack's entry isotropic semi-infinite medium
- `n_exit`: the refractive index of the stack's exit isotropic semi-infinite medium
- `wl_nm`: the wavelength in nanometers
- `theta_in_rad`: the angle of incidence in radians

and they are used to initialise the `Model` with the following fields:

```
def __init__(self, n_entry, n_exit, wl_nm, theta_in_rad):
    self.n_entry = n_entry
    self.n_exit = n_exit
    self.wl = wl_nm
    self.theta_in = theta_in_rad
    theta_out = np.arcsin((n_entry / n_exit) * np.sin(self.theta_in))
    self.k0 = 2 * np.pi / self.wl
    self.Kx = self.n_entry * np.sin(self.theta_in)          # kx = Kx * k0
    self.Ky = 0                                             # ky = Ky * k0
    self.Kz_entry = self.n_entry * np.cos(self.theta_in)    # kz_entry = Kz_entry * k0
    self.Kz_exit = self.n_exit * np.cos(theta_out)          # kz_entry = Kz_entry * k0
    self.structure = self._build_structure_total()
```

The `Model`'s `structure` is an instance of `Structure` created with the function `_build_structure_total()`:

```
def _build_structure_total(self):
    entry_space, exit_space = self._build_entry_exit()
    structure = self._build_structure(entry_space, exit_space)
    return structure
```

This function calls two sub-functions: `_build_entry_exit()` that creates the entry and exit isotropic `HalfSpaces`, and `_build_structure()` that creates a `Structure` containing the appropriate `Layers` to represent the multilayer stack.

The function `_build_entry_exit()` simply creates the entry and exit `HalfSpaces`:

```
def _build_entry_exit(self):
    epsilon_entry = np.array([[self.n_entry ** 2, 0, 0],
                              [0, self.n_entry ** 2, 0],
                              [0, 0, self.n_entry ** 2]])
    epsilon_exit = np.array([[self.n_exit ** 2, 0, 0],
                             [0, self.n_exit ** 2, 0],
                             [0, 0, self.n_exit ** 2]])
    entry_space = HalfSpace(epsilon_entry, self.Kx, self.Kz_entry, self.k0, category=
↪"isotropic")
    exit_space = HalfSpace(epsilon_exit, self.Kx, self.Kz_exit, self.k0, category=
↪"isotropic")
    return entry_space, exit_space
```

and the function `_build_structure()` creates the `Structure` representing the multilayer stack, without any `Layer` when the class `Model` is used:

```
def _build_structure(self, entry_space, exit_space):
    warnings.warn("The build_function method of the Model class is used.")
    return Structure(entry=entry_space, exit=exit_space, Kx=self.Kx, Ky=self.Ky, Kz_
↪entry=self.Kz_entry, Kz_exit=self.Kz_exit, k0=self.k0, N_periods=1)
```

When the `Structure` has been built, its reflectance can be calculated with `get_refl_trans()`:

```
def get_refl_trans(self, circ=False, method="SM"):
    return self.structure.get_refl_trans(circ=circ, method=method)
```

6.2 Creating a custom child

The core functions in the `Model` class are the following:

- `__init__` to create the `Model` instance
- `_build_entry_exit()` to create the entry and exit `HalfSpaces`
- `_build_structure()` to create the `Structure` with the `Layers`
- `_build_structure_total()` that calls `_build_entry_exit()` and `_build_structure()`
- `get_refl_trans()` that calculates the reflectance of the multilayer stack represented by the `Model`

This constitutes a blueprint for the children classes of `Model`, such as `StackModel` or `CholestericModel`. A child class of `Model` contains functions that can be divided into three categories:

- functions that are implemented in `Model` and that the child class inherits
- functions that are implemented in `Model` and that are overwritten in the child class
- functions that are specific to the child class

Typically, the user's new child class will be written as follows:

```
class ChildModel(Model):
    def __init__(self, parameter1, parameter2, parameter3, n_entry, n_exit, wl_nm,␣
↪theta_in_rad):
        # Initialisation with parameters that are specific to ChildModel:
        self.param1 = parameter1
        self.param2 = parameter2
        self.param3 = parameter3
        # Initialisation with the inherited method:
        # (ChildModel's parameters might be used to recalculate the parent's␣
↪parameters)
        super().__init__(n_entry, n_exit, wl_nm, theta_in_rad)

    def _build_structure(self, entry_space, exit_space):
        # Create an empty structure between isotropic half spaces
        my_structure = Structure(entry_space, exit_space, self.Kx, self.Ky, self.Kz_
↪entry, self.Kz_exit, self.k0, N_per=1)

        # A custom routine with self.param1, self.param2, self.param3 that creates␣
↪Layers and adds them to the Structure
        my_structure.add_layers(my_list_of_layers)

        # Return the Structure that contains the custom-made Layers
        return my_structure
```

When using `super().__init__` in the child class (`ChildModel`), it will call the `_build_structure_total()` method in the parent class (`Model`), which will then call both the `_build_structure()` method of the child (which overrides the parent one), and `_build_entry_exit()` of the parent (since it is not overridden by a child version). `CholestericModel`, `SlabModel`, `StackModel` and `StackOpticalThicknessModel` are built this way. They also inherit `get_refl_trans()` from `Model`.

The user simply needs to add their own `ChildModel` to the code by using the sample used as an example above with their own chosen parameters, and when calling `ChildModel.get_refl_trans()`, they will immedietaly benefit from the optical calculations that have been implemented.

Of course, when the user writes a new child class, they may overwrite as many functions as they want, and they may add as many specific functions as they want. For example, `MixedModel` overwrites most functions from `Model`.

`Model` also contains the function `copy_as_stack()` that creates a `StackModel` containing the same layers as a given `Model`. The user will need to overwrite this function too.

6.3 Pairing the custom child with `Spectrum`

The `Spectrum` class implements the modelling of a multilayer stack over a range of wavelength and provide tools for calculating reflection spectra with the choice of the polarisation basis and for exporting the data. Pairing the user's new child class of `Model` with `Spectrum` enables the user to have access to such functionalities.

A `Spectrum` is defined by the following function:

```
def __init__(self, wl_nm_list, model_type, model_parameters):
    self.wl_list = wl_nm_list          # list of wavelengths in nm
    self.mo_type = model_type          # name of model
    self.mo_param = model_parameters   # dictionary with model parameters
    self.data = {}                     # empty dictionary for storage
```

When the user calls the function `calculate_refl()`, the name of the `Model` (`mo_type`) is checked and this triggers the creation of the appropriate `Model`, from the parameters in the dictionary `mo_param`. The user needs to add their own `elif` case to identify the `ChildModel` and handle its parameters correctly. For example, for the following `ChildModel`'s input parameters:

```
def __init__(self, parameter1, parameter2, parameter3, n_entry, n_exit, wl_nm, theta_
↪in_rad, default4=value4, default5=value5)
```

the new `elif` case to add to `Spectrum`'s `calculate_refl()` corresponds to:

```
elif self.mo_type == "ChildModel":
    default_param = dict("default4"=value4, "default5"=value5)
    self.mo_param = {**default_param, **self.mo_param}  # self.mo_param is added to␣
↪default_param and overwrites the default parameters
    model = ChildModel(self.mo_param["parameter1"],
                       self.mo_param["parameter2"],
                       self.mo_param["parameter3"],
                       self.mo_param["n_entry"],
                       self.mo_param["n_exit"],
                       wl,
                       self.mo_param["theta_in_rad"],
                       self.mo_param["default4"],
                       self.mo_param["default5"])
```

This says that when `Spectrum` is instanciated with the parameter `mo_type` equal to the string `ChildModel`, an instance of `ChildModel` will be created with the parameters chosen by the user.

The keys `"parameter1"`, `"parameter2"`, etc, can have an arbitrary name, but for clarity it is easier if the keys match the parameter's name in the `__init__` function.

Once this is done, the user can create a `Spectrum` with `ChildModel` as usual, as well as calculate the reflectance and export the spectra:

```
# Creation of the wavelengths
wl_nm_list = range(400, 800)

# Parameters for the ChildModel
# There are two default parameters: default4 and default5
# The user sets a value for default5: this overwrites the default value
# The user doesn't set a value for default4: the default value will be used
model_type = "ChildModel"
model_parameters = {"parameter1": my_value_1,
                    "parameter2": my_value_2,
                    "parameter3": my_value_3,
                    "default5": my_value_5,
                    "n_entry": n_entry,
                    "n_exit": n_exit,
                    "theta_in_rad": theta_in_rad}

# Creation of the periodic stack
my_spec = Spectrum(wl_nm_list, model_type, model_parameters)

# The functions of the Spectrum class automatically work
my_stack_spec.calculate_refl_trans()
matplotlip.pyplot.plot(wl_nm_list, my_stack_spec.data["R_ps"])
my_stack_spec.export("my_file_name.mat")
```

CHAPTER 7

Acknowledgements

7.1 Authors

PyLlama has originally been developed by Mélanie M. Bay, under the supervision of Silvia Vignolini and Kevin Vynck, as part of her PhD project "The Interplay of between order and disorder in cholesteric hydroxypropyl cellulose films" (University of Cambridge, 2021).

7.2 Referencing

Please cite the following article in any work making use of PyLlama: M. M. Bay, S. Vignolini, and K. Vynck, "PyLlama: A stable and versatile Python toolkit for the electromagnetic modelling of multilayered anisotropic media", Comput. Phys. Commun. 273, 108256 (2022). DOI 10.1016/j.cpc.2021.108256

7.3 Financial support

This work was supported by ERC grant ERC-2014-STG H2020 639088 and Philip Leverhulme Prize (PLP-2019-271) for S.V. and M.M.B.

CHAPTER 8

Documentation

class pyllama.**CholestericModel**(*chole*, *n_e*, *n_o*, *n_entry*, *n_exit*, *wl_nm*, *N_per*, *theta_in_rad*)

This class represents a cholesteric liquid crystal with a multilayer stack of rotating nematic layers, constructed from a `Cholesteric` physical model.

Parameters

- **chole** (*Cholesteric*) – a `Cholesteric` object from the `Cholesteric` library
- **n_e** (*float*) – the extraordinary refractive index
- **n_o** (*float*) – the ordinary refractive index
- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **N_per** (*int*) – the number of periods. The cholesteric `chole` may already represent more than one helicoid: the layers created from the helicoid(s) in `chole` represent the periodic unit, which is repeated `N_per` times in `CholestericModel`.
- **wl_nm** (*float*) – the wavelength in nanometers
- **theta_in_rad** (*float*) – the angle of incidence in radians

class pyllama.**HalfSpace**(*epsilon*, *Kx*, *Kz*, *k0*, *category='isotropic'*)

This class represents an isotropic semi-infinite medium before or after a multilayer stack and enables to build the partial waves (eigenvalues, eigenvectors) of the medium. `HalfSpace` represents the physical layer for one specific wavelength (the material may be dispersive).

Parameters

- **epsilon** (*ndarray*) – permittivity tensor: a 3x3 Numpy array
- **Kx** (*float*) – x-component of the normalised wavevector
- **k0** (*float*) – normalisation factor of the wavevector: the x-component of the wavevector is equal to $k_x = k_0 K_x$
- **category** (*str*) – always `"isotropic"` in the code's 1.0 version

class pyllama.**Layer**(*epsilon*, *thickness_nm*, *Kx*, *k0*, *rot_angle_rad=0*, *rot_axis='z'*, *hold=False*, *numerical_method='numpy'*)

This class represents a homogeneous layer in a multilayer stack and enables to build Berreman's matrix as well as the partial waves (eigenvalues, eigenvectors) of the layer. The layer is made of a non-magnetic and non-optically acvive material. `Layer` represents the physical layer for one specific wavelength (the material may be dispersive). Its parameters are:

Parameters

- **epsilon** (*ndarray*) – permittivity tensor, a 3x3 Numpy array
- **thickness_nm** (*float*) – thickness of the `Layer` in nanometers
- **Kx** (*float*) – x-component of the normalised wavevector
- **k0** (*float*) – normalisation factor of the wavevector: the x-component of the wavevector is equal to $k_x = k_0 K_x$
- **rot_angle_rad** (*float*) – rotation angle of the layer (in radians) around the axis *rot_axis*
- **rot_axis** (*ndarray*) – rotation axis: a one-dimensional Numpy array of length 3 (or the string `'x'`, `'y'` or `'z'`)
- **hold** (*bool*) – when the user decides to hold (`hold=True`) the calculation of Berreman's matrix, the eigenvalues and eigenvectors, the user must then manually apply the functions to the `Layer` before calculating the transfer or scattering matrix. This is exceptional practice. The default is `hold=True`.
- **numerical_method** (*String*) – indicates the package to use to calculate the eigenvectors and eigenvalues of the layer; either `'numpy'` (default) or `'sympy'`

build_P_Q()

This function constructs the interface matrix P and the propagation matrix Q for one `Layer`.

- The interface matrix P describes the change of medium.
- The propagation matrix Q describes the propagation in the thickness of the medium and the phase build-up.

Return ndarray P interface matrix P, 3x3 Numpy array

Return ndarray Q propagation matrix Q, 3x3 Numpy array

static rotate_permittivity(*eps*, *angle_rad*, *axis='z'*)

This function calculates a rotated permittivity tensor.

Parameters

- **eps** (*ndarray*) – permittivity tensor: a 3x3 Numpy array
- **angle_rad** (*float*) – rotation angle (in radians) around the rotation axis `axis`
- **axis** (*ndarray*) – rotation axis: a one-dimensional Numpy array of length 3 (or the string `'x'`, `'y'` or `'z'`)

Returns rotated permittivity tensor: a 3x3 Numpy array

class pyllama.**MixedModel**(*models_list*, *n_entry*, *n_exit*, *wl_nm*, *theta_in_rad*)

This class represents the combination of several `Models` with their sub-periodicities, given in a list of `Models`. The entry and exit `HalfSpaces` of these `Models` are ignored and replaced by these of the `MixedModel`. `Kx` and `k0` must be identical throughout all stacked models, which is checked at the initialisation; the `Models` that don't fit will be discarded and a warning will be issued.

Parameters

- **models_list** (*list*) – a list of `Models` (`models_list[0]` is on top of the stack, after the entry `HalfSpace`)
- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **wl_nm** (*float*) – the wavelength in nanometers
- **theta_in_rad** (*float*) – the angle of incidence in radians

copy_as_stack()

This function retrieves the permittivity and the thickness of the `Structure` created by the `Model` and creates an identical non-periodic `StackModel` (if the `Model` was periodic, the `StackModel` contains multiple times the same layers, but no periodic pattern to repeat).

Returns a `StackModel`

get_refl_trans(*circ=False*, *method='SM'*)

This function calculates the `Model`'s reflectance in the linear or circular polarisation basis, with the method chosen by the user.

Parameters

- **circ** (*bool*) – `False` to express results in the linear polarisation basis, `True` to express results in the circular polarisation basis
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Returns

reflectance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{p\,\text{to}\,p} & R_{s\,\text{to}\,p} \\ R_{p\,\text{to}\,s} & R_{s\,\text{to}\,s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{RCP\,\text{to}\,RCP} & R_{LCP\,\text{to}\,RCP} \\ R_{RCP\,\text{to}\,LCP} & R_{LCP\,\text{to}\,LCP} \end{bmatrix}$$

Returns

transmittance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{p\,\text{to}\,p} & T_{s\,\text{to}\,p} \\ T_{p\,\text{to}\,s} & T_{s\,\text{to}\,s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{RCP\text{ to }RCP} & T_{LCP\text{ to }RCP} \\ T_{RCP\text{ to }LCP} & T_{LCP\text{ to }LCP} \end{bmatrix}$$

class pyllama.**Model**(*n_entry*, *n_exit*, *wl_nm*, *theta_in_rad*)

This class and its children enable the user to construct `Structures` automatically from given parameters. The class `Model` can be viewed as an abstract class that defines parameters and methods common to all its children; however, it is possible to create an instance of `Model`: it will have an empty `Structure` and its `Layers` (`Model.structure.layers`) can be added manually (`Structure.add_layer()`). The parameters of `Models` are:

Parameters

- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **wl_nm** (*float*) – the wavelength in nanometers
- **theta_in_rad** (*float*) – the angle of incidence in radians

copy_as_stack()

This function retrieves the permittivity and the thickness of the `Structure` created by the `Model` and creates an identical non-periodic `StackModel` (if the `Model` was periodic, the `StackModel` contains multiple times the same layers, but no periodic pattern to repeat).

Returns a `StackModel`

get_refl_trans(*circ=False*, *method='SM'*)

This function calculates the `Model`'s reflectance in the linear or circular polarisation basis, with the method chosen by the user.

Parameters

- **circ** (*bool*) – `False` to express results in the linear polarisation basis, `True` to express results in the circular polarisation basis
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Returns

reflectance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{p\text{ to }p} & R_{s\text{ to }p} \\ R_{p\text{ to }s} & R_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{RCP\text{ to }RCP} & R_{LCP\text{ to }RCP} \\ R_{RCP\text{ to }LCP} & R_{LCP\text{ to }LCP} \end{bmatrix}$$

Returns

transmittance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{p\,\text{to}\,p} & T_{s\,\text{to}\,p} \\ T_{p\,\text{to}\,s} & T_{s\,\text{to}\,s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{RCP\,\text{to}\,RCP} & T_{LCP\,\text{to}\,RCP} \\ T_{RCP\,\text{to}\,LCP} & T_{LCP\,\text{to}\,LCP} \end{bmatrix}$$

class pyllama.**SlabModel**(*eps*, *thickness_nm*, *n_entry*, *n_exit*, *wl_nm*, *theta_in_rad*, *rotangle_rad=0*, *rotaxis='z'*)

This class represents a homogeneous slab of arbitrary permittivity.

Parameters

- **eps** (*ndarray*) – permittivity tensor, 3x3 Numpy array
- **thickness_nm** (*float*) – thickness in nanometers
- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **wl_nm** (*float*) – the wavelength in nanometers
- **theta_in_rad** (*float*) – the angle of incidence in radians
- **rotangle_rad** (*float*) – the rotation angle to apply to the permittivity tensor, in radians
- **rotaxis** (*ndarray*) – the rotation axis, a one-dimensional Numpy array of length 3 (or the string `'x'`, `'y'` or `'z'`)

class pyllama.**Spectrum**(*wl_nm_list*, *model_type*, *model_parameters*)

This class implements the modelling of a multilayer stack over a range of wavelength and provide tools for calculating reflection spectra with the choice of the polarisation basis and for exporting the data. `Spectrum` contains an initially-empty dictionary `Spectrum.data` that will be filled with the calculated reflection spectra and additional data.

Parameters

- **wl_nm_list** (*ndarray*) – a list (or array or range) of wavelengths (integers or floats), for example `range(400, 800)`
- **model_type** (*string*) – the name of the model to use. These include:
  - `"CholestericModel"`
  - `"SlabModel"`
  - `"StackModel"`
  - `"StackOpticalThicknessModel"`
- **model_parameters** (*dict*) – a dictionary containing the list of parameters required to create the chosen `Model`,

except the wavelength. See the chosen `Model`'s documentation to know how to construct the dictionary.

add_result (*key*, *value*)
This function add an entry to the dictionary `Spectrum.data`. This entry will be included in the content that is saved by `Spectrum.export()`.

Parameters

- **key** (*string*) – the key for the value to add to the dictionary
- **value** – the value to add (any type)

calculate_refl_trans (*circ=False*, *method='SM'*, *talk=False*)
This function creates the required `Model` and calculates the reflection spectrum in the linear (default) or circular polarisation basis, usinq a chosen method (by default the scattering matrix method). The results are stored in the initially empty dictionary `Spectrum.data`. The values of the results correspond to:

- in the linear polarisation basis (`circ=False`):
  - `Spectrum.data["R_p_to_p"]`: reflection spectrum for incoming p-polarisation to outgoing p-polarisation, 1d Numpy array
  - `Spectrum.data["R_p_to_s"]`: reflection spectrum for incoming p-polarisation to outgoing s-polarisation, 1d Numpy array
  - `Spectrum.data["R_s_to_p"]`: reflection spectrum for incoming s-polarisation to outgoing p-polarisation, 1d Numpy array
  - `Spectrum.data["R_s_to_s"]`: reflection spectrum for incoming s-polarisation to outgoing s-polarisation, 1d Numpy array
- in the circular polarisation basis (`circ=False`):
  - `Spectrum.data["R_R_to_R"]`: reflection spectrum for incoming RCP-polarisation to outgoing RCP-polarisation, 1d Numpy array
  - `Spectrum.data["R_R_to_L"]`: reflection spectrum for incoming RCP-polarisation to outgoing LCP-polarisation, 1d Numpy array
  - `Spectrum.data["R_L_to_R"]`: reflection spectrum for incoming LCP-polarisation to outgoing RCP-polarisation, 1d Numpy array
  - `Spectrum.data["R_L_to_L"]`: reflection spectrum for incoming LCP-polarisation to outgoing LCP-polarisation, 1d Numpy array

as well as `time_elapsed` (float) which calculates the time that it took to compute the spectrum.

Parameters

- **circ** (*bool*) – `False` to express results in the linear polarisation basis, `True` to express results in the circular polarisation basis
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix
- **talk** (*bool*) – `True` (non-default) to display the computation progress, wavelength per wavelength

export (*path_out*, *with_param=True*)
This function exports the `Spectrum` for further processing in MATLAB or Python, and stores it to the specified path. The contents of `Spectrum.data` and `Spectrum.wl_list` are exported.

Parameters

- **path_out** (*string*) – path of the file to save the spectrum. It must end with ".mat" (to save in MATLAB-compatible format) or ".pck" (to save with Pickles in Python-compatible format).
- **with_param** (*bool*) – True to save the Spectrum's model parameters in addition to the content of Spectrum.data,

False (default) to only save the content of Spectrum.data.

rename_result(*old_key*, *new_key*)

This function enables to rename one of the elements contained in Spectrum.data.

For example, the user may calculate the reflection spectrum with the scattering matrix method:

```
my_spectrum.calculate(method="SM")
```

then rename the keys in Spectrum.data with:

```
my_spectrum.rename_result("R_R_to_R", "R_R_to_R_SM")
my_spectrum.rename_result("R_R_to_L", "R_R_to_L_SM")
my_spectrum.rename_result("R_L_to_R", "R_L_to_R_SM")
my_spectrum.rename_result("R_L_to_L", "R_L_to_L_SM")
my_spectrum.rename_result("time_elapsed", "time_elpased_SM")
```

then calculate the reflection spectrum with the transfer matrix method:

```
Spectrum.calculate(method="TM")
```

then rename the keys in Spectrum.data with:

```
my_spectrum.rename_result("R_R_to_R", "R_R_to_R_TM")
my_spectrum.rename_result("R_R_to_L", "R_R_to_L_TM")
my_spectrum.rename_result("R_L_to_R", "R_L_to_R_TM")
my_spectrum.rename_result("R_L_to_L", "R_L_to_L_TM")
my_spectrum.rename_result("time_elapsed", "time_elpased_TM")
```

in order to save results calculated with both matrix methods.

class pyllama.**StackModel**(*eps_list*, *thickness_nm_list*, *n_entry*, *n_exit*, *wl_nm*, *theta_in_rad*, *N_per=1*)

This class represents a periodic multilayer stack where each layer has a given permittivity and thickness.

Parameters

- **eps_list** (*list*) – list of permittivity tensors for each layer, each a 3x3 Numpy array
- **thickness_nm_list** (*list*) – list of thicknesses in nanometers for each layer, each a float
- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **wl_nm** (*float*) – the wavelength in nanometers
- **theta_in_rad** (*float*) – the angle of incidence in radians
- **N_per** (*int*) – the number of periods

add_layer(*new_layer*)
This function adds a Layer to the multilayer stack represented by the StackModel.

Parameters **new_layer** (Layer) – Layer to add

add_layers(*new_layers_list*)
This function adds a list of Layers to the multilayer stack represented by the StackModel.

Parameters **new_layers_list** (*list*) – list of Layers to add

change_N_per(*new_N_per*)
This function changes the number of periods of the Bragg stack.

Parameters **new_N_per** (*int*) – new number of periods

extract_stack(*index_first_layer*, *index_last_layer*)
This function extracts a sub-stack from the StackModel (and return a new instance of StackModel).

Parameters

- **index_first_layer** – index of the first Layer
- **index_last_layer** – index of the last Layer to extract + 1 (if index_first_layer = index_last_layer, the

sub-stack contains the Layer indexed index_first_layer) :return: a StackModel

rotate_layer(*layer_index*, *rot_angle_rad*, *rot_axis='z'*, *hold=False*)
This function rotates a given Layer's permittivity tensor: it creates the new rotated Layer and replaces the non-rotated Layer by the rotated Layer in the Structure.

Parameters

- **layer_index** (*int*) – the index of the Layer to rotate
- **rot_angle_rad** (*float*) – the rotation angle in radians
- **rot_axis** (*ndarray*) – the rotation axis
- **hold** (*bool*) – when the user decides to hold (hold=True) the calculation of Berreman's matrix, the eigenvalues

and eigenvectors, the user must then manually apply the functions to the Layer before calculating the transfer or scattering matrix. This is exceptional practice. The default is hold=True.

rotate_layers(*layer_number_list*, *rot_angle_rad_list*, *rot_axis='z'*)
This function applies the function rotate_layer on several Layers. See rotate_layer's documentation.

class pyllama.**StackOpticalThicknessModel**(*n_list*, *total_thickness_nm*, *n_entry*, *n_exit*, *wl_nm*, *theta_in_rad*, *N_per=1*)
This class represents a periodic multilayer stack where all layers are isotropic and have the same optical thickness.

Parameters

- **n_list** (*list*) – list of refractive indices for each Layer, each a float
- **total_thickness_nm** (*float*) – totat thickness of the stack, in nanometers
- **n_entry** (*float*) – the refractive index of the stack's entry isotropic semi-infinite medium
- **n_exit** (*float*) – the refractive index of the stack's exit isotropic semi-infinite medium
- **wl_nm** (*float*) – the wavelength in nanometers

- **theta_in_rad** (*float*) – the angle of incidence in radians
- **N_per** (*int*) – the number of periods

class pyllama.**Structure**(*entry*, *exit*, *Kx*, *Ky*, *Kz_entry*, *Kz_exit*, *k0*, *N_periods=1*)

This class represents a multilayer stack by:

- a list of layers (instances of `Layer`), initially an empty list
- an isotropic entry semi-infinite medium (instance of `HalfSpace`)
- an isotropic exit semi-infinite medium (instance of `HalfSpace`)
- the number of periods N, which means that the list of layers will be repeated N times

Parameters

- **Kx** (*float*) – x-component of the normalised wavevector (stays the same throughought the stack)
- **Ky** (*float*) – y-component of the normalised wavevector (equal to 0 by construction)
- **Kz** (*float*) – z-component of the normalised wavevector (changes in each layer)
- **k0** (*float*) – normalisation factor of the wavevector:

$$\begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} = k_0 \begin{bmatrix} K_x \\ K_y \\ K_z \end{bmatrix}$$

  which stays the same throughout the stack and depends on the wavelength
- **N_periods** (*int*) – the number of periods

add_layer(*new_layer*)

This function adds a `Layer` to the structure, provided it is compatible with the structure (see function `Structure.is_layer_compatible(layer)`).

Parameters **new_layer** – a `Layer` to add

add_layers(*new_layers_list*)

This function adds `Layers` to the structure, provided they are compatible with the structure (see function `Structure.is_layer_compatible(layer)`). Compatible `Layers` are added, non-compatible `Layers` are not added.

Parameters **new_layers_list** – a list of `Layers` to add

static are_structures_compatible(*structures_list*)

This function checks if several structures are compatible with each other: the x-component of the normalised wavevector and its normalisation factor k_0 must stay the same in all structures.

Parameters **structures_list** (*list*) – a list of `Structures`

Return bool `True` if all `Structures` from the list are compatible, `False` otherwise

build_exponential_matrix()

This function calculates the transfer matrix of the system with the exponential of Berreman's matrix.

Returns transfer matrix, a 4x4 Numpy array

static build_exponential_matrix_multi(*struct_list*, *entry*, *exit*)

This function calculates the transfer matrix with the direct exponential of Berreman's matrix for a system made of sub-stacks.

Returns transfer matrix, a 4x4 Numpy array

build_scattering_matrix()
This function calculates the scattering matrix of the system.

Returns scattering matrix, a 4x4 Numpy array

static build_scattering_matrix_multi(*struct_list*, *entry*, *exit*)
This function calculates the scattering matrix for a system made of sub-stacks.

Returns scattering matrix, a 4x4 Numpy array

static build_scattering_matrix_to_next(*layer_a*, *layer_b*)
This function constructs the scattering matrix S_{ab} between two successive layers a and b by taking into acount the following phenomena:

- the propagation through the first layer with the propagation matrix Q of `layer_a`
- the transition from the first layer (`layer_a`'s matrix P) and the second layer (`layer_b`'s matrix P)

Parameters

- **layer_a** (*ndarray*) – the first `Layer`
- **layer_b** (*ndarray*) – the second `Layer`

Returns partial scattering matrix from layer a to layer b, a 4x4 Numpy array

build_transfer_matrix()
This function calculates the transfer matrix of the system with the matrices of eigenvalues and eigenvectors.

Returns transfer matrix, a 4x4 Numpy array

static build_transfer_matrix_multi(*struct_list*, *entry*, *exit*)
This function calculates the transfer matrix with the eigenvectors and eigenvalues for a system made of sub-stacks.

Returns transfer matrix, a 4x4 Numpy array

static combine_scattering_matrices(*S_ab*, *S_bc*)
This function constructs the scattering matrix between three successive layers a, b and c by combining the scattering matrices S_{ab} from layer a to layer b and S_{bc} from layer b to layer c.

Parameters

- **S_ab** (*ndarray*) – the scattering matrix from layer a to layer b, a 4x4 Numpy array
- **S_bc** (*ndarray*) – the scattering matrix from layer b to layer c, a 4x4 Numpy array

Returns partial scattering matrix from layer a to layer c, a 4x4 Numpy array

static fresnel_to_fresnel_circ(*J_refl*, *J_trans*)
This function converts reflection and transmission coefficients in the linear polarisation basis to reflection and transmission coefficients in the circular polarisation bases.

Parameters

- **J_refl** (*ndarray*) – 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} r_{p\,\text{to}\,p} & r_{s\,\text{to}\,p} \\ r_{p\,\text{to}\,s} & r_{s\,\text{to}\,s} \end{bmatrix}$$

- **J_trans** (*ndarray*) – 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} t_{p\,\text{to}\,p} & t_{s\,\text{to}\,p} \\ t_{p\,\text{to}\,s} & t_{s\,\text{to}\,s} \end{bmatrix}$$

Return J_refl_c 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} r_{RCP\text{ to }RCP} & r_{LCP\text{ to }RCP} \\ r_{RCP\text{ to }LCP} & r_{LCP\text{ to }LCP} \end{bmatrix}$$

Return J_trans_c 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} t_{RCP\text{ to }RCP} & t_{LCP\text{ to }RCP} \\ t_{RCP\text{ to }LCP} & t_{LCP\text{ to }LCP} \end{bmatrix}$$

get_fresnel(*method='SM'*)

This function calculates the `Structure`'s reflection and transmission coefficients in the linear polarisation basis, with the method chosen by the user.

Parameters **method** (*string*) – the matrix method to use for the calculation:

- `"SM"` for the scattering matrix method
- `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
- `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Return J_refl 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} r_{p\text{ to }p} & r_{s\text{ to }p} \\ r_{p\text{ to }s} & r_{s\text{ to }s} \end{bmatrix}$$

Return J_trans 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} t_{p\text{ to }p} & t_{s\text{ to }p} \\ t_{p\text{ to }s} & t_{s\text{ to }s} \end{bmatrix}$$

static get_fresnel_multi(*structures_list*, *entry*, *exit*, *method='SM'*)

This function calculates reflection and transmission coefficients for a system made of a list of Structures in the linear polarisation basis, with the method chosen by the user.

Parameters

- **structures_list** (*list*) – list of multiple `Structures` that constitute the stack. Their respective entry and exit `HalfSpaces` will be ignored.
- **entry** – instance of `HalfSpace` that constitutes the stack's entry semi-infinite medium
- **exit** – instance of `HalfSpace` that constitutes the stack's exit semi-infinite medium
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Return J_refl 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} r_{p\text{ to }p} & r_{s\text{ to }p} \\ r_{p\text{ to }s} & r_{s\text{ to }s} \end{bmatrix}$$

Return J_trans 2x2 Numpy array whose values correspond to:

$$\begin{bmatrix} t_{p\text{ to }p} & t_{s\text{ to }p} \\ t_{p\text{ to }s} & t_{s\text{ to }s} \end{bmatrix}$$

get_refl_trans (*circ=False*, *method='SM'*)

This function calculates the `Structure`'s reflectance in the linear or circular polarisation basis, with the method chosen by the user.

Parameters

- **circ** (*bool*) – `False` to express results in the linear polarisation basis, `True` to express results in the circular polarisation basis
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Returns

reflectance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{p\text{ to }p} & R_{s\text{ to }p} \\ R_{p\text{ to }s} & R_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{RCP\text{ to }RCP} & R_{LCP\text{ to }RCP} \\ R_{RCP\text{ to }LCP} & R_{LCP\text{ to }LCP} \end{bmatrix}$$

Returns

transmittance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{p\text{ to }p} & T_{s\text{ to }p} \\ T_{p\text{ to }s} & T_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{RCP\text{ to }RCP} & T_{LCP\text{ to }RCP} \\ T_{RCP\text{ to }LCP} & T_{LCP\text{ to }LCP} \end{bmatrix}$$

static get_refl_trans_multi (*structures_list*, *entry*, *exit*, *circ=False*, *method='SM'*)

This function calculates reflectance of a system made of a list of Structures in the linear or circular polarisation basis, with the method chosen by the user.

Parameters

- **structures_list** (*list*) – list of multiple `Structures` that constitute the stack. Their respective entry and exit `HalfSpaces` will be ignored.
- **entry** – instance of `HalfSpace` that constitutes the stack's entry semi-infinite medium
- **exit** – instance of `HalfSpace` that constitutes the stack's exit semi-infinite medium

- **circ** (*bool*) – `False` to express results in the linear polarisation basis, `True` to express results in the circular polarisation basis
- **method** (*string*) – the matrix method to use for the calculation:
  - `"SM"` for the scattering matrix method
  - `"TM"` for the transfer matrix method with the eigenvectors and eigenvalues
  - `"EM"` for the transfer matrix method with the direct exponential of Berreman's matrix

Returns

reflectance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{p\text{ to }p} & R_{s\text{ to }p} \\ R_{p\text{ to }s} & R_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} R_{RCP\text{ to }RCP} & R_{LCP\text{ to }RCP} \\ R_{RCP\text{ to }LCP} & R_{LCP\text{ to }LCP} \end{bmatrix}$$

Returns

transmittance: 2x2 Numpy array whose values correspond to:

- in the linear polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{p\text{ to }p} & T_{s\text{ to }p} \\ T_{p\text{ to }s} & T_{s\text{ to }s} \end{bmatrix}$$

- in the circular polarisation basis (`circ=False`):

$$\begin{bmatrix} T_{RCP\text{ to }RCP} & T_{LCP\text{ to }RCP} \\ T_{RCP\text{ to }LCP} & T_{LCP\text{ to }LCP} \end{bmatrix}$$

is_layer_compatible (*layer*)

This function checks if the layer `layer` is compatible with the structure: the x-component of the normalised wavevector and its normalisation factor k_0 must stay the same throughout the stack.

Parameters **layer** – a `Layer`

Return bool `True` if the layer is compatible with the structure, `False` otherwise

remove_layer (*layer_index*)

This function removes from the `Structure` the `Layer` at the index `layer_index`.

Parameters **layer_index** (*int*) – index of the `Layer` to remove

replace_layer (*layer_index*, *new_layer*)

This function replaces the `Layer` at the index `layer_index` by the `Layer` `new_layer` provided it is compatible with the structure (see function `Structure.is_layer_compatible(layer)`).

Parameters

- **layer_index** (*int*) – index of the `Layer` to replace
- **new_layer** – `Layer` to add

class `pyllama.`**Wave**(*epsilon*, *Kx*, *Ex*, *Ey*, *Hx*, *Hy*)

This class represents a partial wave in a layer of the multilayer stack with:

- its electric field
- its magnetic field
- its Poynting vector
- the x- (tangential) component of its normalised wavevector

Parameters

- **epsilon** (*ndarray*) – permittivity tensor: a 3x3 Numpy array
- **Kx** (*float*) – x-component of the normalised wavevector
- **Ex** (*float*) – x-component of the electric field
- **Ey** (*float*) – y-component of the electric field
- **Hx** (*float*) – x-component of the magnetic field
- **Hy** (*float*) – y-component of the magnetic field

static calc_Ez_Hz(*epsilon*, *Kx*, *Ex*, *Ey*, *Hy*)

This function calculates the z-components of the electric and magnetic fields of the Wave

calc_cp_elec()

This function calculates the parameter Cp, used to sort a pair of partial waves between s and p polarisations. :return: Cp = |Ex| ** 2 / (|Ex|**2 + |Ey|**2)

calc_cp_poynting()

This function calculates the parameter Cp, used to sort a pair of partial waves between s-like and p-like polarisations. :return: Cp = |Sx| ** 2 / (|Sx|**2 + |Sy|**2)

calc_poynting()

This function calculates the Poynting vector of the Wave

static matrix_to_waves(*mat*, *epsilon*, *Kx*)

Given a layer's 4 eigenvectors in a 4x4 Numpy array where each column corresponds to a column vector $\psi = [E_x, H_y, E_y, -H_x]$, this function returns a list of the 4 corresponding Waves, which are easier to manipulate when electric fields, magnetic fields and Poynting vectors need to be accessed.

Parameters

- **mat** (*ndarray*) – 4x4 Numpy array of 4 eigenvectors ψ_0, ψ_1, ψ_2 and ψ_3 whose values correspond to:

$$\begin{bmatrix} E_{x,0} & E_{x,1} & E_{x,2} & E_{x,3} \\ H_{y,0} & H_{y,1} & H_{y,2} & H_{y,3} \\ E_{y,0} & E_{y,1} & E_{y,2} & E_{y,3} \\ -H_{x,0} & -H_{x,1} & -H_{x,2} & -H_{x,3} \end{bmatrix}$$

- **epsilon** (*ndarray*) – permittivity tensor: 3x3 Numpy array
- **Kx** (*float*) – the x- (tangential) component of its normalised wavevector

Returns list of 4 corresponding `Waves [w_0, w_1, w_2, w_3]`

static waves_to_matrix(*w_list*, *norm=False*)

Given a layer's 4 partial waves as `Waves` objects, this function returns a matrix where the components E_x, E_y, H_x and H_y are arranged so that the matrix can be used as a transition matrix for the layer. The function extracts a vector $\psi = [E_x, H_y, E_y, -H_x]$ for each of the 4 waves and formats them into a matrix where each column is a ψ.

Parameters

- **w_list** (*list*) – list of 4 partial waves `Waves [w_0, w_1, w_2, w_3]`
- **norm** (*bool*) – set to True to normalise each ψ to a modulus of 1, otherwive set to False. Must be set to False for the partial waves of the `HalfSpaces`.

Returns

4x4 Numpy array of 4 eigenvectors ψ_0, ψ_1, ψ_2 and ψ_3 whose values correspond to:

$$\begin{bmatrix} E_{x,0} & E_{x,1} & E_{x,2} & E_{x,3} \\ H_{y,0} & H_{y,1} & H_{y,2} & H_{y,3} \\ E_{y,0} & E_{y,1} & E_{y,2} & E_{y,3} \\ -H_{x,0} & -H_{x,1} & -H_{x,2} & -H_{x,3} \end{bmatrix}$$

pyllama.**rot_mat**(*axis=array([0, 0, 1])*, *theta_rad=0*)

This function builds a rotation matrix for a given angle around a given axis accordung to https://en.wikipedia.org/wiki/Rotation_matrix.

Parameters

- **axis** (*ndarray*) – rotation axis: a one-dimensional Numpy array of length 3 (or the string `'x'`, `'y'` or `'z'`)
- **theta_rad** (*float*) – rotation angle (in radians) around the rotation axis `axis`

Returns a 3x3 Numpy array rotation matrix

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

W