metabox Documentation

Release 0.0.3

luochenghuang

Sep 13, 2023

CONTENTS

This is the documentation of **metabox**.

Note: This is the main page of your project's Sphinx documentation. It is formatted in reStructuredText. Add additional pages by creating rst-files in `docs` and adding them to the toctree below. Use then references in order to link them from this page, e.g. *Contributors* and *Changelog*.

It is also possible to refer to the documentation of other Python packages with the Python domain syntax. By default you can reference the documentation of Sphinx, Python, NumPy, SciPy, matplotlib, Pandas, Scikit-Learn. You can add more by extending the `intersphinx_mapping` in your Sphinx's `conf.py`.

The pretty useful extension autodoc is activated by default and lets you include documentation from docstrings. Docstrings can be written in Google style (recommended!), NumPy style and classical style.

CHAPTER

ONE

CONTENTS

1.1 `metabox`: A High-Level Python API for Diffractive Optical System Design

> metabox is a Python package built on TensorFlow, enabling the design, evaluation and optimization of complex diffractive optical systems with ease, flexibility, and high performance.

`metabox` is a high-level Python package specifically designed for the creation, evaluation, and inverse optimization of diffractive optical systems. Leaning on the robust capabilities of TensorFlow, `metabox` offers a comprehensive and user-friendly API for optical system design.

The package is built with flexibility at its core, making it easy to add new components, define custom merit functions, and employ various optimization algorithms. It's designed to be highly performant and scalable, capable of managing systems with millions of degrees of freedom. With its intuitive structure, `metabox` facilitates the design of intricate diffractive optical systems with minimal lines of code.

Key features of `metabox` include:

- A `rcwa` solver, derived from rcwa_tf, for direct computation of meta-atoms' diffraction efficiency.
- A built-in `raster` module for parameterizing meta-atoms' features.
- An easy-to-use sampling system for features, which can train a metamodel to replace the `rcwa` solver, thus significantly speeding up simulations and optimization processes.
- A module for sequential optics to model light propagation through the optical system.
- An `assembly` module offering a suite of tools for building the optical system from meta-atoms, apertures, and other optical components.
- A `merit` module for evaluating and inverse-designing the performance of the optical system.
- An `rcwa.Material` class for accessing pre-defined materials and their optical properties.
- An `export` module that allows for the export of the diffractive optical design to a `.gds` file for fabrication.

Overall, `metabox` is a powerful tool for both beginners and experienced users in the field of optical system design. By simplifying and accelerating the design process, it paves the way for innovative developments in the optical industry.

1.2 Install

Install `metabox` via `pip`:

```
pip install metabox
```

1.3 Getting Started

Try out `metabox` for free on Google Colab. Here are some tutorials on Colab. You can find the local versions here.

Tutorial 1: Metamodeling

Tutorial 2: Lens Optimization and Exporting

Tutorial 3: Optimization Serialization

Tutorial 4: Zemax Binary2 Import

Tutorial 5: Refractive Surfaces Simulation

Tutorial 6: Refractive Surfaces Optimization

Tutorial 7: Hologram Optimization

1.4 Documentation

Module Reference

Home Page

1.5 Contributors

- Luocheng Huang: luocheng@uw.edu, https://github.com/Luochenghuang

1.6 Citing `metabox`

The manuscript is in preparation.

1.7 Making Changes & Contributing

This project uses *pre-commit*, please make sure to install it before making any changes:

```
pip install pre-commit
cd metabox
pre-commit install
```

It is a good idea to update the hooks to the latest version:

```
pre-commit autoupdate
```

To make an editable installation, run the following commands:

```
git clone https://github.com/Luochenghuang/metabox.git
cd metabox
pip install -e .
```

1.8 Contributing

Welcome to `metabox` contributor's guide.

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp[1].

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

1.8.1 Issue Reports

If you experience bugs or general issues with `metabox`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

[1] Even though, these resources focus on open source projects and communities, the general ideas behind collaborating with other developers to collectively create software are general and can be applied to all sorts of environments, including private companies and proprietary code bases.

1.8.2 Documentation Improvements

You can help improve `metabox` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`metabox` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

1.8.3 Code Contributions

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n metabox python=3 six virtualenv pytest pytest-cov
conda activate metabox
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/metabox.git
   cd metabox
   ```

4. You should run:

   ```
   pip install -U pip setuptools -e .
   ```

   to be able to import the package under development in the Python REPL.

5. Install `pre-commit`:

```
pip install pre-commit
pre-commit install
```

`metabox` comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

```
git checkout -b my-feature
```

and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.
3. Add yourself to the list of contributors in `AUTHORS.rst`.
4. When you're done editing, do:

```
git add <MODIFIED FILES>
git commit
```

to record your changes in git.

Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with the project.

Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

```
git log --graph --decorate --pretty=oneline --abbrev-commit --all
```

to look for recurring communication patterns.

---

5. Please check that your changes don't break any unit tests with:

```
tox
```

(after having installed `tox` with `pip install tox` or `pipx`).

You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

```
git push -u origin my-feature
```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

```
tox -e docs
```

   Try running:

```
tox -r -e docs
```

3. Make sure to have a reliable `tox` installation that uses the correct Python version (e.g., 3.7+). When in doubt you can run:

```
tox --version
# OR
which tox
```

   If you have trouble and are seeing weird errors upon running `tox`, you can also try to create a dedicated virtual environment with a `tox` binary freshly installed. For example:

```
virtualenv .venv
source .venv/bin/activate
.venv/bin/pip install tox
.venv/bin/tox -e all
```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

1.8.4 Maintainer tasks

Releases

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `metabox`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.
6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

1.9 License

MIT License

Copyright (c) 2023 Luocheng Huang

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.10 Contributors

- Luocheng Huang <luocheng@uw.edu>

1.11 Changelog

1.11.1 Version 0.0.1

- Initial alpha release.

1.11.2 Version 0.0.2

- Fix import error in *assembly*

1.12 metabox

1.12.1 metabox package

Subpackages

metabox.rcwa_tf package

Submodules

metabox.rcwa_tf.shane_rcwa_tf module

metabox.rcwa_tf.shane_rcwa_tf.**convmat**(*A*, *P*, *Q*)

This function computes a convolution matrix for a real space matrix *A* that represents either a relative permittivity or permeability distribution for a set of pixels, layers, and batch. :param A: A *tf.Tensor* of dtype *complex* and shape *(batchSize, pixelsX, :param pixelsY: :param n_layersers: :param Nx: :param Ny)* specifying real space values on a Cartesian: :param grid.: :param P: A positive and odd *int* specifying the number of spatial harmonics :param along *T1*.: :param Q: A positive and odd *int* specifying the number of spatial harmonics :param along *T2*.:

Returns

A *tf.Tensor* of dtype *complex* and shape *(batchSize, pixelsX, pixelsY, n_layersers, P * Q, P * Q)* representing a stack of convolution matrices based on *A*.

metabox.rcwa_tf.shane_rcwa_tf.**eig_general**(*A*, *eps=1e-06*)

Computes the eigendecomposition of a batch of matrices, the same as *tf.eig()* but assumes the input shape also has extra dimensions for pixels and layers. This function also provides the reverse mode gradient of the eigendecomposition as derived in 10.1109/ICASSP.2017.7952140. This applies for general, complex matrices that do not have to be self adjoint. This result gives the exact reverse mode gradient for nondegenerate eigenvalue problems. To extend to the case of degenerate eigenvalues common in RCWA, we approximate the gradient by a Lorentzian broadening technique that introduces a small error but stabilizes the calculation. This is based on 10.1103/PhysRevX.9.031041. :param A: A *tf.Tensor* of shape *(batchSize, pixelsX, pixelsY, n_layersers, Nx, :param Ny)* and dtype *tf.complex64* where the last two dimensions define: :param matrices for which we will calculate the eigendecomposition of their: :param reverse mode gradients.: :param eps: A *float* defining a regularization parameter used in the :param denominator of the Lorentzian broadening calculation to enable reverse: :param mode gradients for degenerate eigenvalues.:

Returns

A *Tuple(List[tf.Tensor, tf.Tensor], tf.Tensor)*, where the *List* specifies the eigendecomposition as

computed by *tf.eig()* and the second element of the *Tuple* gives the reverse mode gradient of the eigendecompostion of the input argument *A*.

metabox.rcwa_tf.shane_rcwa_tf.**expand_and_tile_np**(*array*, *batchSize*, *pixelsX*, *pixelsY*)

Expands and tile a numpy array for a given batchSize and number of pixels. :param array: A *np.ndarray* of shape *(Nx, Ny)*.

Returns

A *np.ndarray* of shape *(batchSize, pixelsX, pixelsY, Nx, Ny)* with the values from *array* tiled over the new dimensions.

metabox.rcwa_tf.shane_rcwa_tf.**expand_and_tile_tf**(*tensor*, *batchSize*, *pixelsX*, *pixelsY*)

Expands and tile a *tf.Tensor* for a given batchSize and number of pixels. :param tensor: A *tf.Tensor* of shape *(Nx, Ny)*.

Returns

A *tf.Tensor* of shape *(batchSize, pixelsX, pixelsY, Nx, Ny)* with the values from *tensor* tiled over the new dimensions.

metabox.rcwa_tf.shane_rcwa_tf.**redheffer_star_product**(*SA*, *SB*)

This function computes the redheffer star product of two block matrices, which is the result of combining the S-parameter of two systems. :param SA: A *dict* of *tf.Tensor* values specifying the block matrix :param corresponding to the S-parameters of a system. *SA* needs to have the: :param keys: :type keys: 'S11', 'S12', 'S21', 'S22' :param of shape *: :type of shape `: batchSize, pixelsX, pixelsY, 2*NH, 2*NH :param total number of spatial harmonics.: :param SB: A `dict* of *tf.Tensor* values specifying the block matrix :param corresponding to the S-parameters of a second system. *SB* needs to have: :param the keys: :type the keys: 'S11', 'S12', 'S21', 'S22' :param *tf.Tensor* of shape *: :type `tf.Tensor* of shape `: batchSize, pixelsX, pixelsY, 2*NH, 2*NH :param NH is the total number of spatial harmonics.:

Returns

A *dict* of *tf.Tensor* values specifying the block matrix corresponding to the S-parameters of the combined system. *SA* needs to have the keys ('S11', 'S12', 'S21', 'S22'), where each key maps to a *tf.Tensor* of shape `(batchSize, pixelsX, pixelsY, 2*NH, 2*NH), where NH is the total number of spatial harmonics.

metabox.rcwa_tf.shane_rcwa_tf.**simulate_rcwa**(*incidence_dict: dict*, *PQ: Tuple[int, int]*, *n_cells: int*, *n_layers: int*, *layer_thicknesses: Tensor*, *L_xy: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *er1:* Feature | *float* | *Tensor*, *er2:* Feature | *float* | *Tensor*, *ER_t: Tensor*, *UR_t: Tensor*, *refl_n:* Feature | *float* | *Tensor*)

Simulates the periodic unit cell using RCWA.

Parameters

- **incidence_dict** – A *dict* of *tf.Tensor* values specifying the incidence parameters. *incidence_dict* needs to have the keys ('wavelength', 'theta', 'phi', 'polarization')
- **PQ** – A *tuple* of *int* values specifying the number of spatial harmonics in the x and y directions.
- **n_cells** – An *int* specifying the number of unit cells in the x direction.
- **n_layers** – An *int* specifying the number of layers in the unit cell.
- **layer_thicknesses** – A *tf.Tensor* of shape *(n_layers,)* specifying the thickness of each layer in the unit cell.
- **L_xy** – A *tuple* of *ParameterType* values specifying the size of the unit cell in the x and y directions.
- **er1** – A *ParameterType* specifying the relative permittivity of the transmission region in the unit cell.

- **er2** – A *ParameterType* specifying the relative permittivity of the reflection region in the unit cell.
- **ER_t** – A *tf.Tensor* specifying the relative permittivity of each layer in the unit cell.
- **UR_t** – A *tf.Tensor* specifying the relative permeability of each layer in the unit cell.
- **refl_n** – A *ParameterType* specifying the refractive index of the reflection region in the unit cell.

Returns
The diffraction coefficients of the unit cell.

Module contents

Submodules

metabox.assembly module

Defines a lens assembly and functionalities for simualting the performances of the lens assembly.

class metabox.assembly.**AmplitudeMask**(*diameter: float*, *refractive_index: float*, *thickness: float*, *periodicity: float*, *threshold_param: float*, *use_circular_expansions: bool = True*, *enable_propagator_cache: bool = False*, *set_mask_variable: bool = False*, *threshold_param_increment: float = 0.0*, *store_end_field: bool = False*)

Bases: *Surface*

Defines an amplitude modulation mask.

Parameters

- **diameter** – the diameter of the lens in meters.
- **refractive_index** – the refractive index of the lens.
- **thickness** – the thickness of the lens in meters.
- **periodicity** – the period of the pixels in meters.
- **threshold_param** – the threshold parameter for the amplitude modulation. This param multiples the amplitude coefficient before the sigmoid function. The larger the value, the more "black and white" the thresholding is.
- **threshold_param_increment** – the increment of the threshold parameter when the optimizer_hook() is called.
- **enable_propagator_cache** – whether to enable the propagator cache. If enabled, the propagator will be cached for the *Incidence*, The propagation would be a lot faster however at the cost of memory usage. Note that this is not recommended if the *Incidence* is not fixed.
- **set_mask_variable** – whether to make the mask variable.
- **store_end_field** – whether to store the end field.

enable_propagator_cache: bool = False

get_end_field(*incidence: Incidence*, *incident_field: Field2D*, *previous_refractive_index: float*, *lateral_shift: None | Tuple[float, float] = None*, *use_padding: bool = True*, *use_x_pol: bool = True*) → *Field2D*

Computes the field at the end of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **incident_field** – the incident field.
- **previous_refractive_index** – the refractive index of the previous
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple, in meters.
- **last_surface** – whether this is the last surface in the optical system.
- **use_padding** – whether to use padding for the field.

get_modulation_2d(*incidence:* Incidence) → *Field2D*

Computes the field modulation of the metasurface.

Parameters
incidence – the *Incidence* of the light.

Returns
the modulation field with shape (batch_size, n_pixels, n_pixels)

Return type
tf.Tensor

optimizer_hook()

Hook for the optimizer to modify the surface.

periodicity: float

set_mask_variable: bool = False

store_end_field: bool = False

threshold_param: float

threshold_param_increment: float = 0.0

use_circular_expansions: bool = True

class metabox.assembly.**Aperture**(*diameter: float*, *refractive_index: float*, *thickness: float*, *periodicity: float*, *enable_propagator_cache: bool = False*, *store_end_field: bool = False*)

Bases: *Surface*

Defines an aperture.

Parameters

- **diameter** – the diameter of the aperture in meters.
- **refractive_index** – the refractive index of the aperture.
- **thickness** – the thickness of the aperture in meters.
- **periodicity** – the period of the pixels in meters.
- **enable_propagator_cache** – whether to enable the propagator cache. If enabled, the propagator will be cached for the *Incidence*, The propagation would be a lot faster however at the cost of memory usage. Note that this is only useful when the aperture is not moving.
- **store_end_field** – whether to store the end field of the aperture.

enable_propagator_cache: bool = False

get_end_field(*incidence:* Incidence, *incident_field:* Field2D, *previous_refractive_index:* *float*, *lateral_shift:* *None* | *Tuple[float, float]* = *None*, *use_padding:* *bool* = *True*, *use_x_pol:* *bool* = *True*) → *Field2D*

Computes the field at the end of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **incident_field** – the incident field.
- **previous_refractive_index** – the refractive index of the previous
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple, in meters.
- **last_surface** – whether this is the last surface in the optical system.
- **use_padding** – whether to use padding for the field.

get_modulation_2d(*incidence:* Incidence) → *Field2D*

Computes the field modulation of the metasurface.

Parameters
incidence – the *Incidence* of the light.

Returns
the modulation field with shape (batch_size, n_pixels, n_pixels)

Return type
tf.Tensor

optimizer_hook()

Hook for the optimizer to modify the surface.

periodicity: float

store_end_field: bool = False

class metabox.assembly.**AtomArray1D**(*tensor: Tensor*, *period:* *float*, *mmodel:* Metamodel | *None* = *None*, *proto_unit_cell:* ProtoUnitCell | *None* = *None*)

Bases: object

Class to store the 1D atom array data and its metadata.

Parameters

- **tensor** – the atom structure array tensor with shape (n_features, n_atoms)
- **period** – the period of the atom array in meters.
- **mmodel** – the *MetaModel* used to generate the atom array. The *MetaModel* stores the trained model and the structure of the atom.
- **proto_unit_cell** – the proto unit cell (rcwa.ProtoUnitCell)

expand_to_2d(*basis_dir='basis_data'*) → *AtomArray2D*

Function to expand a 1d atom array to a 2d atom array.

Args:
basis_dir: the directory where the basis is saved.

"vim.normalModeKeyBindingsNonRecursive": [

The default directory is "basis_data".

Attributes:

tensor: the atom structure array tensor.
The outmost dimension is the feature dimension.

Returns:
AtomArray2D: a 2d atom array

`find_feature_index`(*feature_str: str*)

Returns the index of the feature in the structure tensor.

Parameters
feature_str – the name of the feature.

Raises
ValueError – if the feature is not found in the atom array.

`get_atom_array`(*incidence:* Incidence) → List[*UnitCell*]

Returns the batched atom array with shape (n_batch, n_atoms).

`get_feature_map`(*feature: str*) → ndarray

Returns the 2D feature array.

Parameters
feature – the feature to return the array of.

Returns
the feature array.

Return type
np.ndarray

`get_feature_map_1d`(*feature_str: str*) → ndarray

Returns the 1D feature array.

Parameters
feature_str – the feature string to return the array of.

Returns
the feature array.

Return type
np.ndarray

`mmodel: Metamodel = None`

`period: float`

`proto_unit_cell: ProtoUnitCell = None`

`set_feature_map`(*feature: str*, *feature_array: ndarray*)

Sets the 2D feature array.

Parameters

- **feature** – the feature to set the array of.
- **feature_array** – the feature array to set.

set_to_use_rcwa()

Skips the metamodel and directly simulate the atom array using RCWA.

Note that this method will change the atom array permanently. This method is useful for verifying the performance of the metamodel.

show_feature_map(*only_feature: str | None = None*)

Shows the structure of the atom array.

Parameters
only_feature – the only feature to show the structure of if not None. Shows all features if None.

tensor: Tensor

class metabox.assembly.**AtomArray2D**(*tensor: Tensor*, *period: float*, *mmodel:* Metamodel | *None = None*, *proto_unit_cell:* ProtoUnitCell | *None = None*, *cached_fields: List[Tensor] | None = None*)

Bases: object

Class to store the 2D atom array data and its metadata.

Parameters

- **tensor** – the atom structure array tensor with shape (n_features, n_atoms)
- **period** – the period of the atom array in meters.
- **mmodel** – the *MetaModel* used to generate the atom array. The *MetaModel* stores the trained model and the structure of the atom.
- **proto_unit_cell** – the proto unit cell (rcwa.ProtoUnitCell)
- **cached_fields** – the cached transmission coefficients for the atom array.

cached_fields: List[Tensor] = None

find_feature_index(*feature_str: str*)

Returns the index of the feature in the structure tensor.

Parameters
feature_str – the name of the feature.

Raises
ValueError – if the feature is not found in the atom array.

get_atom_array(*incidence:* Incidence) → List[*UnitCell*]

get_feature_map(*feature: str*) → Tensor

Returns the structure of the atom array.

Parameters
feature – the feature string to get the structure of.

Returns
the structure of the atom array.

Return type
tf.Tensor

mmodel: *Metamodel* = None

period: *float*

proto_unit_cell: *ProtoUnitCell* = None

set_feature_map(*feature: str*, *new_values: ndarray | Tensor*) → Tensor

Change the structure feature of the atom array to the given values.

Parameters
feature – the feature to get the structure of.

Returns
the structure of the atom array.

Return type
tf.Tensor

set_to_use_rcwa()

Skips the metamodel and directly simulate the atom array using RCWA.

Note that this method will change the atom array permanently. This method is useful for verifying the performance of the metamodel.

show_feature_map(*only_feature: str | None = None*)

Shows the structure of the atom array.

Parameters
only_feature – the only feature to show the structure of if not None. Shows all features if None.

tensor: Tensor

class metabox.assembly.**Binary2**(*diameter: float*, *refractive_index: float*, *thickness: float*, *periodicity: float*, *init_coeff: List[float] | None = None*, *set_coeff_variable: bool = True*, *enable_propagator_cache: bool = False*, *diffraction_order: int = 1*, *store_end_field: bool = False*, *previous_refractive_index: float = 1.0*)

Bases: *Surface*

Defines a binary 2 surface comparable to the namesake surface in Zemax.

The binary 2 surface use polynomial terms to express the phase delay of the

incident field. The phase delay is given by:
= M * {i=1; N} (A_i * **(2 * i))

Where M is the diffraction order, N is the maximum number of terms, A_i is the coefficient of the ith term, and is the normalized radial coordinate of the aperture.

Parameters

- **diameter** – the diameter of the surface in meters.
- **refractive_index** – the refractive index of the surface.
- **thickness** – the thickness of the surface in meters.
- **periodicity** – the period of the surface in meters.
- **diffraction_order** – the diffraction order of the surface.
- **store_end_field** – whether to store the end field of the surface.

- **previous_refractive_index** – the refractive index of the previous surface.

diffraction_order: int = 1

enable_propagator_cache: bool = False

get_end_field(*incidence:* Incidence, *incident_field:* Field2D, *previous_refractive_index:* *float*, *lateral_shift:* *None* | *Tuple[float, float]* = *None*, *use_padding:* *bool* = *True*, *use_x_pol:* *bool* = *True*) → *Field2D*

Computes the field at the end of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **incident_field** – the incident field.
- **previous_refractive_index** – the refractive index of the previous surface.
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple, in meters.
- **use_padding** – whether to use padding.

get_modulation_2d(*incidence:* Incidence, *use_padding:* *bool* = *True*) → *Field2D*

Computes the field modulation of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **use_padding** – whether to use padding.

init_coeff: List[float] = None

periodicity: float

previous_refractive_index: float = 1.0

set_coeff_variable: bool = True

store_end_field: bool = False

class metabox.assembly.**CustomFigureOfMerit**(*expression: str*, *data: Dict[str*, *tensorflow.python.framework.ops.Tensor]* = <*factory*>)

Bases: object

data: Dict[str, Tensor]

expression: str

get_validation_errors()

is_valid_expression(*user_expression*)

class metabox.assembly.**FigureOfMerit**(*value*)

Bases: Enum

Defines the types of figure of merit functions.

STREHL_RATIO

the Strehl ratio.

LOG_STREHL_RATIO

the log of the Strehl ratio.

CENTER_INTENSITY = 5

LOG_CENTER_INTENSITY = 6

LOG_MAX_INTENSITY = 4

LOG_STREHL_RATIO = 2

MAX_INTENSITY = 3

STREHL_RATIO = 1

class metabox.assembly.**IntensityTarget**(*intensity: tensorflow.python.framework.ops.Tensor*, *crop_factor: float = 1.0*)

Bases: object

crop_factor: float = 1.0

dist(*psf: Tensor*) → Tensor

Computes the loss between the target intensity and the intensity of the field.

Args:

intensity: Tensor

class metabox.assembly.**LensAssembly**(*surfaces: List[Surface]*, *incidence: Incidence*, *aperture_stop_index: int = -1*, *figure_of_merit: FigureOfMerit | CustomFigureOfMerit | None = None*, *use_antialiasing: bool = True*, *use_padding: bool = True*, *use_x_pol: bool = True*)

Bases: object

Defines a lens assembly.

Parameters

- **surfaces** – a list of surfaces in the lens assembly.
- **focal_length** – the focal length of the lens assembly in meters.
- **aperture_stop_index** – the index of the aperture stop in the lens assembly.
- **figure_of_merit** – the figure of merit of the lens assembly. Options can be found in the *FigureOfMerit* enum.
- **use_antialiasing** – whether to use antialiasing for propagations.
- **use_padding** – whether to use padding for propagations. If True, the sampling window is padded to avoid aliasing at the cost of ~4x memory usage.
- **use_x_pol** – whether the lens assembly is sensitive to the x polarization. if True, the x polarization is used. Otherwise, the y polarization is used.

aperture_stop_index: int = -1

clear_cache()

Clears saved fields.

compute_FOM() → Tensor

Computes the figure of merit of the lens assembly.

Parameters
tf.Tensor – The figure of merit.

compute_center_intensity()

Computes the center intensity of the lens assembly.

compute_custom_FOM(*custom_FOM:* CustomFigureOfMerit) → Tensor

compute_field_on_sensor()

Computes the Strehl ratio of the lens assembly.

compute_max_intensity()

Computes the maximum intensity of the lens assembly.

compute_penalty() → Tensor

Computes the penalty of the lens assembly.

Parameters
tf.Tensor – The penalty.

compute_strehl_ratio()

Computes the Strehl ratio of the lens assembly.

copy() → *LensAssembly*

Returns a copy of the lens assembly.

Returns
The copy of the lens assembly.

Return type
LensAssembly

figure_of_merit: *FigureOfMerit* | *CustomFigureOfMerit* | None = None

get_variables()

Returns the trainable variables.

incidence: *Incidence*

optimizer_hook()

Hook for the optimizer iteration.

save(*name: str*, *save_dir: str = './saved_lens_assemblies'*, *overwrite: bool = False*)

Saves the lens assembly to disk.

Parameters

- **name** – the name of the lens assembly.
- **save_dir** – the directory to save the lens assembly to.

set_to_use_rcwa()

Use RCWA simulation for all the metasurfaces, permanently.

Note that this function will permanently change the metasurfaces to use RCWA simulation. It's wise to save the lens assembly before calling this function. Or make a copy of the lens assembly before calling this.

show_color_psf(*crop_factor: float = 1.0*) → None

show_psf(*use_wavelength_average: bool = False*, *crop_factor: float = 1.0*) → None

Displays the point spread function of the lens assembly.

Parameters

- **use_wavelength_averaging** – whether to use wavelength averaging.
- **crop_factor** – the crop factor of the image.

surfaces: List[*Surface*]

use_antialiasing: bool = True

use_padding: bool = True

use_x_pol: bool = True

wavelength_average_psf()

Displays the wavelength averaged point spread function of the lens assembly.

class metabox.assembly.**Metasurface**(*diameter: float*, *refractive_index: float*, *thickness: float*, *metamodel:* Metamodel | *None = None*, *proto_unit_cell:* ProtoUnitCell | *None = None*, *use_circular_expansions: bool = True*, *enable_propagator_cache: bool = False*, *set_structures_variable: bool = False*, *store_end_field: bool = False*, *xy_harmonics: Tuple[int, int] = (3, 3)*, *unit_cell_spatial_res: int = 128*, *minibatch_size: int = 100*)

Bases: *Surface*

Defines a flat lens.

Parameters

- **diameter** – the diameter of the lens in meters.
- **refractive_index** – the refractive index of the lens.
- **thickness** – the thickness of the lens in meters.
- **metamodel** – the metamodel used to map the structure to the field modulation for each meta-atom.
- **enable_propagator_cache** – whether to enable the propagator cache. If enabled, the propagator will be cached for the *Incidence*, The propagation would be a lot faster however at the cost of memory usage. Note that this is not recommended if the *Incidence* is not fixed.
- **set_structures_variable** – whether to make the structures variable.
- **store_end_field** – whether to store the end field.
- **xy_harmonics** – the number of harmonics in the x and y directions for the field modulation.
- **unit_cell_spatial_res** – the spatial resolution of the unit cell.
- **minibatch_size** – the minibatch size for the rcwa simulation.

clear_cache()

Clears the saved fields.

enable_propagator_cache: bool = False

get_atom_arry(*incidence:* Incidence) → List[*UnitCell*]

Gets the atom array for the given incidence.

get_atom_positions() → ndarray

Gets the positions of the atoms.

get_end_field(*incidence:* Incidence, *incident_field:* Field2D, *previous_refractive_index:* *float*, *lateral_shift:* *None* | *Tuple[float, float]* = *None*, *use_padding:* *bool* = *True*, *use_x_pol:* *bool* = *True*) → *Field2D*

Computes the field at the end of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **incident_field** – the incident field.
- **previous_refractive_index** – the refractive index of the previous medium.
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple, in meters.
- **use_padding** – whether to use padding.

get_feature_map()

Gets the feature map of the metasurface.

get_modulation_2d(*incidence:* Incidence, *use_padding:* *bool* = *True*, *use_x_pol:* *bool* = *True*) → *Field2D*

Computes the field modulation of the metasurface.

metamodel: *Metamodel* **= None**

minibatch_size: int **= 100**

optimizer_hook()

Hook for the optimizer to modify the surface.

proto_unit_cell: *ProtoUnitCell* **= None**

set_feature_map(*feature_str:* *str*, *new_value:* *ndarray* | *Tensor*)

Sets the feature map of the metasurface.

Parameters

- **feature_str** – the feature to set.
- **new_value** – the new value of the feature.

set_structures_variable: bool **= False**

set_to_use_rcwa()

Set to use RCWA for the metasurface, permanently.

show_feature_map(*only_feature:* *str* | *None* = *None*)

Shows the feature map of the metasurface.

Parameters

only_feature – if not None, only shows the feature map of the given feature. Otherwise, shows the featue map of all features.

store_end_field: bool **= False**

unit_cell_spatial_res: *int* = 128

use_circular_expansions: *bool* = True

xy_harmonics: *Tuple[int, int]* = (3, 3)

class metabox.assembly.**RefractiveEvenAsphere**(*diameter: float*, *refractive_index: float*, *thickness: float*, *periodicity: float*, *init_coeff: List[float] | None = None*, *set_coeff_variable: bool = True*, *enable_propagator_cache: bool = False*, *store_end_field: bool = False*, *thickness_penalty_coeff: float = 0.001*)

Bases: *Surface*

Defines an even asphere surface comparable to Zemax.

The even asphere surfaces use polynomial terms to express the sag surface.

z = {i=1; N} (A_i * **(2 * i))

N is the maximum number of terms, we don't have restrictions here, but Zemax limits the number of terms to 8. The extended asphere supports up to 480 terms. A_i is the coefficient of the ith term, and is the normalized radial coordinate of the aperture.

Parameters

- **diameter** – the diameter of the surface in meters.
- **refractive_index** – the refractive index of the surface.
- **thickness** – the thickness of the surface in meters.
- **periodicity** – the period of the surface in meters.
- **unit** – the unit used in Zemax. Can be "m" or "mm". Defaults to "m".
- **set_coeff_variable** – whether to set the coefficients as variables.
- **enable_propagator_cache** – whether to enable the propagator cache.
- **store_end_field** – whether to store the end field of the surface.
- **thickness_penalty_coeff** – the coefficient of the thickness penalty term. Multiplied to the maximum thickness of the sag as the penalty.

enable_propagator_cache: *bool* = False

get_end_field(*incidence: Incidence*, *incident_field: Field2D*, *previous_refractive_index: float*, *lateral_shift: None | Tuple[float, float] = None*, *use_padding: bool = True*, *use_x_pol: bool = True*) → *Field2D*

Computes the field at the end of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **incident_field** – the incident field.
- **previous_refractive_index** – the refractive index of the previous surface.
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple, in meters.

- **use_padding** – whether to use padding to avoid aliasing.

get_modulation_2d(*incidence:* Incidence, *previous_refractive_index: float*, *use_padding: bool = True*) → *Field2D*

Computes the field modulation of the metasurface.

Parameters

- **incidence** – the *Incidence* of the light.
- **previous_refractive_index** – the refractive index of the previous
- **use_padding** – whether to use padding.

Returns
the field modulation of the metasurface.

Return type
propagation.Field2D

get_penalty()

Returns the penalty of the surface. This is used for the optimizer.

get_sag()

Returns the sag surface

init_coeff: List[float] = None

periodicity: float

set_coeff_variable: bool = True

show_sag()

store_end_field: bool = False

thickness_penalty_coeff: float = 0.001

class metabox.assembly.**SphericalLens**(*diameter: float*, *refractive_index: float*, *thickness: float*, *perioidicity: float*, *radius_or_curvature: float*)

Bases: *Surface*

get_end_field(*incidence:* Incidence, *incident_field:* Field2D, *previous_refractive_index: float*, *lateral_shift: None | Tuple[float, float] = None*, *use_padding: bool = True*, *use_x_pol: bool = True*)

get_modulation_2d()

perioidicity: float

radius_or_curvature: float

Defines a spherical lens.

Parameters

- **periodicity** – the period of the lens in meters.
- **radius_or_curvature** – the radius of curvature of the lens in meters.

Returns
description

Return type
type

class metabox.assembly.**Surface**(*diameter: float*, *refractive_index: float*, *thickness: float*)

Bases: object

Defines an optical surface.

Parameters

- **diameter** – the diameter of the surface in meters.
- **refractive_index** – the refractive index of the surface.
- **thickness** – the thickness of the surface in meters.

diameter: float

get_penalty()

Returns the penalty of the surface. This is used for the optimizer.

optimizer_hook()

Hook for the optimizer to modify the surface.

refractive_index: float

thickness: float

metabox.assembly.**cartesian_distance**(*intensity_target:* IntensityTarget, *psf: Tensor*)

Calculates the distance between the target intensity and the intensity of the field.

Parameters

- **intensity_target** – the target intensity.
- **psf** – the point spread function.

Returns

the distance between the target intensity and the intensity of
the field.

Return type
tf.Tensor

metabox.assembly.**copy_lens_assembly**(*lens_assembly:* LensAssembly) → *LensAssembly*

Returns a copy of the lens assembly.

Parameters
lens_assembly – the lens assembly to copy.

Returns
the copy of the lens assembly.

Return type
LensAssembly

metabox.assembly.**initialize_1d_atom_array_metamodel**(*n_pixels_radial: int*, *period: float*, *mmodel:* Metamodel, *set_structures_variable: bool = False*) → *AtomArray1D*

Initializes a 1D atom array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **period** – the period of the structure in meters.
- **mmmodel** – the metamodel to use for the initialization.
- **set_structures_variable** – whether to set the structure as a variable or not.

Returns
The initialized atom array.

metabox.assembly.**initialize_1d_atom_array_proto_unit_cell**(*n_pixels_radial: int*, *proto_unit_cell: ProtoUnitCell*, *set_structures_variable: bool = False*) → *AtomArray1D*

Initializes a 1D atom array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **proto_unit_cell** – the proto unit cell to use for the initialization.
- **set_structures_variable** – whether to set the structure as a variable or not.

Returns
The initialized atom array.

metabox.assembly.**initialize_1d_mask_array**(*n_pixels_radial: int*, *set_mask_variable: bool = False*, *init_bound: Tuple[float, float] = (0, 0)*) → Tensor

Initializes a 1D mask array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **period** – the period of the structure in meters.
- **set_structures_variable** – whether to set the structure as a variable or not.
- **init_bound** – the lower and upper bounds for the initialization.

Returns
The initialized amplitude modulation coefficients.

metabox.assembly.**initialize_2d_atom_array_metamodel**(*n_pixels_radial: int*, *period: float*, *mmodel: Metamodel*, *set_structures_variable: bool = False*, *exclude_wavelength: bool = True*) → *AtomArray2D*

Initializes a 2D atom array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **period** – the period of the structure in meters.
- **mmmodel** – the metamodel to use for the initialization.
- **set_structures_variable** – whether to set the structure as a variable or not.
- **exclude_wavelength** – whether to exclude the wavelength from the feature initialization.

Returns
The initialized atom array with shape (feature_0, feature_1, . . . , n_pixels_x, n_pixels_y)

metabox.assembly.**initialize_2d_atom_array_proto_unit_cell**(*n_pixels_radial: int*, *proto_unit_cell:* ProtoUnitCell, *set_structures_variable: bool = False*) → *AtomArray2D*

Initializes a 2D atom array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **proto_unit_cell** – the proto unit cell to use for the initialization.
- **set_structures_variable** – whether to set the structure as a variable or not.

Returns
The initialized atom array with shape (feature_0, feature_1, ..., n_pixels_x, n_pixels_y)

metabox.assembly.**initialize_2d_mask_array**(*n_pixels_radial: int*, *set_structures_variable: bool = False*) → Tensor

Initializes a 2D atom array.

Parameters

- **n_pixels_radial** – the number of pixels in the radial direction.
- **set_structures_variable** – whether to set the structure as a variable or not.

Returns
The initialized atom array.

metabox.assembly.**load_lens_assembly**(*name: str*, *save_dir: str = './saved_lens_assemblies'*) → *LensAssembly*

Loads a lens assembly from disk.

Parameters

- **name** (*str*) – the name of the lens assembly (folder name)
- **save_dir** (*str, optional*) – The parent folder where the lens assembly is saved to. Defaults to "./saved_lens_assemblies".

Returns
The loaded lens assembly.

Return type
LensAssembly

metabox.assembly.**optimize_multiple_lens_assemblies**(*lens_assembly_arr: List[*LensAssembly*]*, *optimizer: Optimizer*, *n_iter: int*, *verbose: int = 0*, *keep_best: bool = True*) → Tuple[*LensAssembly*, List[float]]

Optimizes multple lens assemblies.

The gradient is accumulated across all lens assemblies siquentially. Then the graident is applied to all lens assemblies for each optimization iteration.

Parameters

- **lens_assembly_arr** – array of lens assembles to optimize.
- **optimizer** – the optimizer to use.
- **n_iter** – the number of iterations to optimize.
- **verbose** – the verbosity level.

- **keep_best** – whether to keep the best lens assembly.

Returns

the optimized lens assembly and the history of the FOM.

Return type
Tuple[*LensAssembly*, List[float]]

metabox.assembly.**optimize_single_lens_assembly**(*lens_assembly: LensAssembly*, *optimizer: Optimizer*, *n_iter: int*, *verbose: int = 0*, *keep_best: bool = True*) → Tuple[*LensAssembly*, List[float]]

Optimizes a single lens assembly.

Parameters

- **lens_assembly** – the lens assembly to optimize.
- **optimizer** – the optimizer to use.
- **n_iter** – the number of iterations to optimize.
- **keep_best** – whether to keep the best lens assembly.

Returns

the optimized lens assembly and the history of the FOM.

Return type
Tuple[*LensAssembly*, List[float]]

metabox.assembly.**save_lens_assembly**(*lens_assembly: LensAssembly*, *name: str*, *save_dir: str = './saved_lens_assemblies'*, *overwrite: bool = False*) → None

Saves the lens assembly to disk.

Parameters

- **name** – the name of the lens assembly.
- **save_dir** – the directory to save the lens assembly to.
- **overwrite** – whether to overwrite the lens assembly if it already exists.

metabox.assembly.**structure_to_field_1d**(*structure: AtomArray1D*, *incidence: Incidence*, *feature_order: List[str] | None = None*, *use_padding: bool = True*) → *Field1D*

metabox.assembly.**structure_to_field_1d_mmodel**(*structure: AtomArray1D*, *incidence: Incidence*, *feature_order: List[str] | None = None*, *use_padding: bool = True*) → *Field1D*

Converts a structure to a 1D field.

Parameters

- **structure** – the structure to convert.
- **incidence** – the incidence of the light.
- **mmodel** – the metamodel to use for the conversion.
- **feature_order** – the order of the features in the structure tensor columns. The first feature has to be wavelength for chromatic optimizations. If None, the order of the features will be the same as the *features_attrs* in the metamodel.

- **use_padding** – whether to use padding for the field.

Returns
The converted field.

metabox.assembly.**structure_to_field_1d_proto_unit_cell**(*structure:* AtomArray1D, *incidence:* Incidence, *feature_order:* *List[str]* | *None* = *None*, *use_padding:* *bool* = *True*) → *Field1D*

Converts a structure to a 1D field.

Parameters

- **structure** – the structure to convert.
- **incidence** – the incidence of the light.
- **feature_order** – unused.
- **use_padding** – whether to use padding for the field.

Returns
The converted field.

metabox.assembly.**structure_to_field_2d**(*structure:* AtomArray2D, *incidence:* Incidence, *feature_order:* *List[str]* | *None* = *None*, *use_padding:* *bool* = *True*) → *Field2D*

Converts a structure to a 2D field.

Parameters

- **structure** – the structure to convert.
- **incidence** – the incidence of the light.
- **mmodel** – the metamodel to use for the conversion.
- **feature_order** – the order of the features in the structure tensor columns. The first feature has to be wavelength for chromatic optimizations. If None, the order of the features will be the same as the *features_attrs* in the metamodel.
- **use_padding** – whether to use padding for the field.

Returns
The converted field.

metabox.assembly.**unbatch_incidence**(*incidence:* Incidence) → List[*Incidence*]

Unbatches an incidence by the incident angles and wavelengths.

Parameters
incidence – the incidence to unbatch.

Returns
The unbacthed incidences.

metabox.assembly.**unbatch_lens_assembley**(*lens_assembly:* LensAssembly) → List[*LensAssembly*]

Unbatches a lens assembly by the incident angles and wavelengths.

Parameters
lens_assembly – the lens assembly to unbatch.

Returns
The unbacthed lens assemblies.

metabox.expansion module

Defines functions to expand a 1d field to a 2d field.

metabox.expansion.**expand_to_2d**(*tensor: Tensor*, *basis_dir='basis_data'*) → Tensor

Function to expand a 1d field to a 2d field.

Parameters

- **tensor** (*tf.tensor*) – the 1d field to expand
- **basis_dir** – the directory where the basis is saved. The default directory is "basis_data".

Returns
the expanded field tensor.

Return type
tf.Tensor

metabox.expansion.**load_basis**(*n_pix*, *basis_dir=None*) → Tensor

retrieve the 1d to 2d basis from the basis_dir.

If the basis is not found, it is created and saved to the basis_dir.

Parameters

- **n_pix_radial** (*int*) – number of pixels per axis
- **basis_dir** (*str*) – path to the directory where the basis is saved. if None, the basis is generated and not loaded or saved.

Returns
the 1d to 2d basis.

Return type
tf.Tensor

metabox.expansion.**radius_to_circle_basis**(*radius_size*) → Tensor

Create a basis to map a 1d field to a 2d field.

Parameters
radius_size (*int*) – number of pixels in the radius

Returns
the 1d to 2d basis

Return type
tf.Tensor

metabox.export module

This module contains functions to generate the GDSII and other visualization files.

metabox.export.**gds_shape_force_4fold_symmetry**(*shape: PolygonSet*) → PolygonSet

Generates a 4-fold symmetric shape from a polygon

metabox.export.**generate_gds**(*metasurface:* Metasurface, *layer: int*, *export_name: str*, *export_directory: str* | *None = None*, *inverted: bool = False*) → None

Function to generate the GDSII file for the metasurface.

Parameters

- **metasurface** (assembly.Metasurface) – the metasurface.
- **layer** – the layer in UnitCell to use for the shapes.
- **export_directory** (*str*) – the directory to export the GDSII file to.
- **inverted** (*bool, optional*) – whether to invert the metasurface. Defaults to False.

metabox.export.**generate_noncircular_gds**(*metasurface: Metasurface*, *layer: int*, *export_name: str*, *export_directory: str | None = None*, *inverted: bool = False*) → None

Function to generate the GDSII file for a noncirculat metasurface.

metabox.export.**get_loc_along_circle**(*radius_in_meters*, *radius_size*, *query_radius*, *period*, *r2c_basis*)

inputs radius in micron, radius size, and the index of which radius you want to draw a circle using, outputs the x and y locations of the elements

metabox.export.**unit_cell_to_gds_shape**(*cell: UnitCell*, *layer: int = 0*)

Converts a unit cell to a GDSII shape.

Parameters
cell (rcwa.UnitCell) – the unit cell.

Returns
the GDSII shape.

Return type
gdspy.PolygonSet

metabox.metrics module

This file contains the functions to calculate the metrics of the optical system.

metabox.metrics.**get_center_intensity**(*field: Field2D*, *use_log: bool = False*) → Tensor

Returns the center intensity of the field.

Parameters

- **field** (propagation.Field2D) – the field to calculate the Strehl ratio of.
- **use_log** (*bool*) – use the log of the intensity for each sampling.

Returns
the center intensity of the field.

Return type
tf.Tensor

metabox.metrics.**get_ideal_mtf_volume**(*field_props: FieldProperties*, *focal_length: float*) → ndarray

Function to calculate the volume under the ideal MTF surface.

Parameters

- **field_props** (propagation.FieldProperties) – the field properties.
- **focal_length** – the focal length of the lens in meters.

Returns
the volume under the ideal MTF surface.

Return type
np.ndarray

metabox.metrics.**get_max_intensity**(*field:* Field2D) → Tensor

Defines function that returns the maximum intensity of the field.

Parameters
field (propagation.Field2D) – the field to calculate the maximum intensity of.

Returns
The maximum intensity of the field.

Return type
tf.Tensor

metabox.metrics.**get_mtf_volume**(*field:* Field2D, *use_log:* *bool* = *False*) → Tensor

Returns the volume under the MTF surface.

Parameters

- **field** (propagation.Field2D) – the field to calculate the Strehl ratio of.
- **use_log** (*bool*) – use the log of the Strehl ratio for each sampling. By enabling this, the Strehl ratio will be more uniform after the optimization.

Returns
the Strehl ratio of the field.

Return type
tf.Tensor

metabox.modeling module

class metabox.modeling.**ComplexLayer**(**args*, ***kwargs*)

Bases: Layer

A layer that converts two features into a complex feature column.

call(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.

  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns
A tensor or list/tuple of tensors.

class metabox.modeling.**Metamodel**(*model: keras.src.engine.sequential.Sequential*, *history: keras.src.callbacks.History*, *protocell:* metabox.rcwa.ProtoUnitCell, *sim_config: dict*)

Bases: object

history: History

model: Sequential

plot_training_history() → None
Plots the training history.

protocell: *ProtoUnitCell*

save(*name: str*, *path: str = './saved_metamodels'*, *overwrite: bool = False*) → None
Saves the metamodel to a file.

Parameters
- **filename** (*str*) – The name of the file to save the metamodel to.
- **path** (*str*) – The path to the file.

set_feature_constraint(*feature_str: str*, *vmin: float | None*, *vmax: float | None*) → None
Sets the value of a feature.

Parameters
- **feature** (*str*) – The attribute name of the feature to set.
- **vmin** – The minimum value of the feature.
- **vmax** – The maximum value of the feature.

sim_config: dict

class metabox.modeling.**NormComplexLayer**(**args*, ***kwargs*)

Bases: Layer

A layer that converts two features into a complex feature column.

call(*inputs*)

This is where the layer's logic lives.

The *call()* method may not create state (except in its first invocation, wrapping the creation of variables or other resources in *tf.init_scope()*). It is recommended to create state, including *tf.Variable* instances and nested *Layer* instances,

in *__init__()*, or in the *build()* method that is

called automatically before *call()* executes for the first time.

Parameters

- **inputs** – Input tensor, or dict/list/tuple of input tensors. The first positional *inputs* argument is subject to special rules: - *inputs* must be explicitly passed. A layer cannot have zero

  arguments, and *inputs* cannot be provided via the default value of a keyword argument.

  - NumPy array or Python scalar values in *inputs* get cast as tensors.
  - Keras mask metadata is only collected from *inputs*.
  - Layers are built (*build(input_shape)* method) using shape info from *inputs* only.
  - *input_spec* compatibility is only checked against *inputs*.
  - Mixed precision input casting is only applied to *inputs*. If a layer has tensor arguments in **args* or ***kwargs*, their casting behavior in mixed precision should be handled manually.
  - The SavedModel input specification is generated using *inputs* only.
  - Integration with various ecosystem packages like TFMOT, TFLite, TF.js, etc is only supported for *inputs* and not for tensors in positional and keyword arguments.

- ***args** – Additional positional arguments. May contain tensors, although this is not recommended, for the reasons above.
- ****kwargs** – Additional keyword arguments. May contain tensors, although this is not recommended, for the reasons above. The following optional keyword arguments are reserved: - *training*: Boolean scalar tensor of Python boolean indicating

  whether the *call* is meant for training or inference.

  - *mask*: Boolean input mask. If the layer's *call()* method takes a *mask* argument, its default value will be set to the mask generated for *inputs* by the previous layer (if *input* did come from a layer that generated a corresponding mask, i.e. if it came from a Keras layer with masking support).

Returns

A tensor or list/tuple of tensors.

class metabox.modeling.**SimulationLibrary**(*protocell:* metabox.rcwa.ProtoUnitCell, *incidence:* metabox.utils.Incidence, *sim_config:* *dict*, *feature_values:* *numpy.ndarray*, *simulation_output:* *tensorflow.python.framework.ops.Tensor*)

Bases: object

feature_values: ndarray

get_training_x() → ndarray

Returns the training input.

First dim is the wavelength, the second dim is the first parameterized feature, the third dim is the second parameterized feature, etc.

get_training_y() → ndarray

Returns the training output.

incidence: *Incidence*

protocell: *ProtoUnitCell*

save(*name: str*, *path: str*, *overwrite: bool = False*)

Saves the simulation library

Parameters

- **name** (*str*) – the name of the library
- **path** (*str*) – a path to save the library to
- **overwrite** (*bool*) – Whether or not to overwrite exisiting library.

sim_config: dict

simulation_output: Tensor

Stores the simulation parameters and the simulation output.

protocell

the simulated protocell.

incidence

the incidence of the light.

sim_config

the simulation configuration.

feature_values

the sampled feature values.

simulation_output

the simulation output.

metabox.modeling.**create_and_train_model**(*sim_lib: SimulationLibrary*, *n_epochs: int = 100*, *optimizer: Optimizer | None = None*, *hidden_layer_units_list: List[int] = [64, 128, 256, 64]*, *activation_list: List[str] = ['relu', 'relu', 'relu', 'relu']*, *limit_output_to_unity: bool = False*, *train_batch_size: None | int = None*, *validation_split: float = 0.05*, *verbose: int = 0*) → Tuple[Sequential, History]

Creates and fits a given model to the atom library.

Fits a fully connected network to the atom library. The network is a simple fully connected network with a normalization layer. Returns the trained model and the history of the training.

Parameters

- **sim_lib** (*SimulationLibrary*) – the simulation library.
- **n_epochs** (*int*) – the number of epochs.
- **optimizer** (*tf.keras.optimizers.Optimizer*) – the optimizer.
- **hidden_layer_units** (*List[int]*) – the number of units in each hidden layer.
- **activation** (*List[str]*) – the activation function for each hidden layer.
- **limit_output_to_unity** (*bool*) – whether to limit the intensity to unity.
- **train_batch_size** (*Union[None, int]*) – the batch size for training.
- **validation_split** (*float*) – the fraction of the training data to use for validation.
- **verbose** (*int*) – the verbosity level.

Returns

Contains the trained model, the history of the training,
and the proto-atom used to train the model.

Return type
MetaModel

metabox.modeling.**create_fcc_model**(*normalizer: Normalization*, *optimizer: Optimizer*, *hidden_layer_units_list: List[int]*, *activation_list: List[str]*, *limit_output_to_unity: bool = False*) → Sequential

Creates a simple fully connected network with a normalization layer.

Parameters

- **normalizer** (*tf.keras.layers.Normalization*) – the normalization layer.
- **optimizer** (*tf.keras.optimizers.Optimizer*) – the optimizer.
- **hidden_layer_units** (*List[int]*) – the number of units in each hidden layer.
- **activation** (*List[str]*) – the activation function for each hidden layer.
- **limit_output_to_unity** (*bool*) – whether to limit the intensity to unity.

metabox.modeling.**euclidian_distance**(*y_true: Tensor*, *y_pred: Tensor*)

Calculates the euclidian distance between two complex numbers.

Parameters

- **y_true** (*tf.Tensor*) – true value.
- **y_pred** (*tf.Tensor*) – predicted value.

Returns
the euclidian distance between the two complex numbers.

Return type
tf.Tensor

metabox.modeling.**load_metamodel**(*name: str*, *save_dir: str = './saved_metamodels'*) → *Metamodel*

Loads a metamodel from a file.

Parameters

- **name** (*str*) – The name of the file to load the metamodel from.
- **path** (*str*) – The path to the file.

Returns
The loaded metamodel.

Return type
MetaModel

metabox.modeling.**load_simulation_library**(*name: str*, *path: str*) → *SimulationLibrary*

Loads a AtomLibrary from a file.

Parameters

- **name** (*str*) – The name of the file to load the atom library from.
- **path** (*str*) – The path to the file.

Returns
The loaded atom library.

Return type
AtomLibrary

metabox.modeling.**sample_protocell**(*protocell*, *incidence*, *sim_config*) → None

Sample a protocell with a given incidence.

For a given protocell, each unique *Feature* is sampled given its sampling
number (see *Feature.sampling*). Then the permutation of the sampled features is simulated with the given incidence. The output is a

Parameters

- **protocell** (*Protocell*) – The protocell to sample.
- **incidence** (*float*) – The incidence to sample the protocell with.
- **sim_config** (*dict*) – The simulation configuration.

Returns
The simulation library.

Return type
SimulationLibrary

metabox.modeling.**save_simulation_library**(*sim_lib: SimulationLibrary*, *name: str*, *path: str*, *overwrite: bool = False*) → None

Saves the metamodel to a file.

Parameters

- **filename** (*str*) – The name of the file to save the metamodel to.
- **path** (*str*) – The path to the file.

metabox.propagation module

This file contains the classes and functions to simulate the propagation of light.

class metabox.propagation.**Field1D**(*n_pixels: int*, *wavelength: List[float]*, *theta: List[float]*, *phi: List[float]*, *period: float*, *upsampling: int*, *use_padding: bool*, *use_antialiasing: bool*, *tensor: Tensor*)

Bases: *FieldProperties*

Class to store the field data and its metadata.

Parameters

- **wavelength** – the wavelength of the light in meters.
- **theta** – the angle of the light in degrees.
- **phi** – the phase of the light in degrees.
- **period** – the period of the pixels in meters.
- **upsampling** – the upsampling factor.
- **use_padding** – whether to use padding or not.
- **use_antialiasing** – whether to use antialiasing or not.

expand_to_2d(*basis_dir='basis_data'*) → *Field2D*

Function to expand a 1d field to a 2d field.

Parameters
basis_dir – the directory where the basis is saved. The default directory is "basis_data".

Returns
a 2d field

Return type
propagation2d.Field2D

get_intensity()

tensor: Tensor

class metabox.propagation.**Field2D**(*n_pixels: int*, *wavelength: List[float]*, *theta: List[float]*, *phi: List[float]*, *period: float*, *upsampling: int*, *use_padding: bool*, *use_antialiasing: bool*, *tensor: Tensor*)

Bases: *FieldProperties*

Class to store the field data and its metadata.

Parameters

- **tensor** – the tensor of the field.
- **wavelength** – the wavelength of the light in meters.
- **theta** – the angle of the light in degrees.
- **phi** – the phase of the light in degrees.
- **period** – the period of the pixels in meters.
- **upsampling** – the upsampling factor.
- **use_padding** – whether to use padding or not.

- **use_antialiasing** – whether to use antialiasing or not.

get_intensity()

Returns the intensity tensor of the tensor.

get_phase()

Returns the phase tensor of the tensor.

modulated_by(*other:* Field2D) → *Field2D*

Modulate this field by another field.

Parameters
other (Field2D) – the field to modulate by.

Returns
the modulated field.

Return type
Field2D

show_color_intensity(*crop_factor=1.0*)

Shows the intensity of the field.

Parameters
crop_factor (*float*) – The crop factor. Must be less than or equal to 1.0.

show_intensity(*crop_factor=1.0*)

Shows the intensity of the field.

Parameters
crop_factor (*float*) – The crop factor. Must be less than or equal to 1.0.

show_phase()

Shows the phase of the field.

tensor: Tensor

to_rgb_intensity()

Return RGB image of the intensity.

wavelength_average()

Returns the wavelength averaged field.

class metabox.propagation.**FieldProperties**(*n_pixels: int*, *wavelength: List[float]*, *theta: List[float]*, *phi: List[float]*, *period: float*, *upsampling: int*, *use_padding: bool*, *use_antialiasing: bool*)

Bases: object

Defines the properties of a 2d field.

Parameters

- **n_pixels** – the number of pixels per dim after 2D expansion.
- **wavelength** – the wavelength of the light in meters.
- **theta** – the angle of the light in degrees.
- **phi** – the phase of the light in degrees.
- **period** – the period of the pixels in meters.
- **upsampling** – the upsampling factor.

- **use_padding** – whether to use padding or not.
- **use_antialiasing** – whether to use antialiasing or not.

copy()

Returns a copy of the field properties.

n_pixels: *int*

period: *float*

phi: *List[float]*

theta: *List[float]*

upsampling: *int*

use_antialiasing: *bool*

use_padding: *bool*

wavelength: *List[float]*

metabox.propagation.**get_incident_field_2d**(*field_props:* FieldProperties) → Tensor

Defines the input electric fields for the given wavelengths and field angles.

Parameters
field_props – the field properties.

Returns
The incident field.

Return type
Field2d

metabox.propagation.**get_intensity_1d**(*field_1d:* Field1D)

returns the intensity tensor of the field.

Returns
intensity tensor of shape [wavelengths, angles, pixelsX]

Return type
tf.Tensor

metabox.propagation.**get_intensity_2d**(*field_2d:* Field2D)

returns the intensity tensor of the field.

Returns
intensity tensor of shape [wavelengths, angles, pixelsX, pixelsY]

Return type
tf.Tensor

metabox.propagation.**get_phase_2d**(*field_2d:* Field2D)

returns the phase tensor of the field.

Returns
intensity tensor of shape [wavelengths, angles, pixelsX, pixelsY]

Return type
tf.Tensor

metabox.propagation.**get_propagator_batched**(*ref_idx: float*, *prop_dist: float*, *n_pix: int*, *period: float*, *wavelength_sampling: List[float]*, *theta_sampling: List[float]*, *phi_sampling: List[float]*, *upsampling=1*, *use_padding=True*, *use_antialiasing=True*, *lateral_shift: None | Tuple[float, float] = None*) → Tensor

Returns the transfer function for a given propagation distance.

Parameters

- **ref_idx** – refractive index of the medium
- **prop_dist** – propagation distance in meters
- **n_pix** – number of pixels in each dimension
- **period** – pixel size in meters
- **wavelength** – wavelength in meters
- **theta_sampling** – list of angles to sample in degrees
- **phi_sampling** – list of angles to sample in degrees
- **upsampling** – upsampling factor
- **use_padding** – whether to pad the transfer function to prevent aliasing
- **use_antialiasing** – whether to limit the transfer function bandwidth to prevent aliasing
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple.

Returns
transfer function

Return type
propagator

metabox.propagation.**get_transfer_function**(*field_like:* Field2D, *ref_idx: float*, *prop_dist: float*, *lateral_shift: None | Tuple[float, float] = None*) → Tensor

Get the Propagator object given the propagation information.

Parameters

- **ref_idx** (*float*) – refractive index of the medium
- **prop_dist** (*float*) – propagation distance in meters
- **upsampling** (*int*) – upsampling factor
- **use_padding** (*bool*) – whether or not to use padding
- **use_anti_aliasing** (*bool*) – whether or not to use anti-aliasing
- **lateral_shift** – the lateral shift of the sampling window on the detector in meters. If None, the shift is set so that the Chief Ray is at the center of the detector. If a tuple of two floats, the shift is set according to the first element (x shift) and the second element (y shift) of the input tuple.

Returns
the complex field on the final plane

Return type
tf.Tensor

metabox.propagation.**propagate**(*field:* Field2D, *transfer_function: Tensor*) → *Field2D*

Propagate a field through a given transfer function.

Parameters

- **field** (*Field*) – the field to propagate
- **transfer_function** (*tf.Tensor*) – the transfer function

Returns
the complex field on the final plane

Return type
End field (Field)

metabox.propagation.**propagate_with_propagator_batched**(*field: Tensor*, *propagator: Tensor*, *use_padding=True*, *upsampling=1*) → Tensor

metabox.propagation.**wavelength_average_2d**(*field:* Field2D) → *Field2D*

Function to average the field over the wavelengths.

Parameters
field (Field2D) – the field to average over the wavelengths.

Returns
the averaged field.

Return type
Field2D

metabox.raster module

class metabox.raster.**Canvas**(*x_width:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor*, *y_width:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor*, *spacing:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor* = *1.0*, *background_value:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor* = *0.0*, *enforce_4fold_symmetry:* *bool* = *False*)

Bases: object

add_circle(*center: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *radius:* Feature | *float* | *Tensor*) → None

Adds a circle to the canvas.

Parameters

- **center** – The center of the circle.
- **radius** – The radius of the circle.
- **keep_positive** – Keeps the values positive. Defaults to True.
- **apply_threshold** – Applies threshold from 0 to 1. Defaults to True.

add_point(*p: Tuple[float, float]*, *radius=0.5*) → None

Adds a point to the canvas.

Parameters

- **p** – The point to add.
- **radius** – The radius of the point.

add_polygon(*points: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *keep_positive: bool = True*) → None

Adds a polygon to the canvas.

Parameters

- **points** (*List[Tuple[float, float]]*) – _description_
- **keep_positive** (*bool, optional*) – keeps the values positive. Defaults to True.

add_rectangle(*center: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *x_width:* Feature | *float* | *Tensor*, *y_width:* Feature | *float* | *Tensor*, *rotation_deg:* Feature | *float* | *Tensor = 0.0*) → None

Adds a rectangle to the canvas.

Parameters

- **center** – The center of the rectangle.
- **x_width** – The width of the rectangle in the x direction.
- **y_width** – The width of the rectangle in the y direction.
- **rotation_deg** – The rotation of the rectangle in degrees. Defaults to 0.

add_regular_polygon(*center: Tuple[float, float]*, *radius: float*, *n: int*, *keep_positive: bool = True*, *apply_threshold: bool = True*) → None

Adds a regular polygon to the canvas.

Parameters

- **center** – The center of the polygon.
- **radius** – The radius of the polygon.
- **n** – The number of sides of the polygon.
- **keep_positive** – Keeps the values positive. Defaults to True.
- **apply_threshold** – Applies threshold from 0 to 1. Defaults to True.

add_regular_star(*center: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *radius:* Feature | *float* | *Tensor*, *n: int*, *keep_positive: bool = True*, *apply_threshold: bool = True*) → None

Addes a regular star to the canvas.

Parameters

- **center** – The center of the star.
- **radius** – The radius of the star.
- **n** – The number of points of the star.
- **keep_positive** – Keeps the values positive. Defaults to True.
- **apply_threshold** – Applies threshold from 0 to 1. Defaults to True.

add_triangle(*p0: Tuple[float, float]*, *p1: Tuple[float, float]*, *p2: Tuple[float, float]*) → None

Adds a triangle to the canvas.

Parameters

- **p0** – The first point of the triangle.
- **p1** – The second point of the triangle.
- **p2** – The third point of the triangle.

If the points are in clockwise order, the triangle will be rasterized as a positive shape. If the points are in counter-clockwise order, the triangle will be rasterized as a negative shape.

background_value: *Feature* | float | Tensor = 0.0

draw()

Draws the canvas.

enforce_4fold_symmetry: bool = False

A class for drawing on a canvas.

x_width

The width of the canvas in the x direction.

y_width

The width of the canvas in the y direction.

spacing

The spacing between pixels. Defaults to 1.

background_value

The value of the background. Defaults to 0.

enforce_4fold_symmetry

whether to make the layer 4-fold symmetric. If true, then the layer will be mirrored along the x and y axes, then added to its transpose.

merge_shape(*shape:* Shape, *enforce_4fold_symmetry:* *bool*) → Tensor

Adds a shape onto the canvas.

Parameters
shape – The shape to rasterize.

merge_with(*other:* Canvas) → None

Merges the canvas with another canvas.

Parameters
other – The other canvas to merge with.

rasterize(*shape_list:* *List*[Shape]) → Tensor

Rasterizes a list of shapes.

High level API for rasterizing a list of shapes.

Parameters
shape_list – The list of shapes to rasterize.

spacing: *Feature* | float | Tensor = 1.0

x_width: *Feature* | float | Tensor

y_width: *Feature* | float | Tensor

class metabox.raster.**Circle**(*value:* Feature | *float* | *Tensor*, *center:* *Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *radius:* Feature | *float* | *Tensor*)

Bases: *Shape*

Defines a circle.

center

The center of the circle.

Type
Tuple[*metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor, *metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor]

radius

The radius of the circle.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

center: `Tuple[Feature | float | Tensor, Feature | float | Tensor]`

radius: `Feature | float | Tensor`

class metabox.raster.**Polygon**(*value:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor*, *points: List[Tuple[*metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor,* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor]]*)

Bases: *Shape*

points: `List[Tuple[Feature | float | Tensor, Feature | float | Tensor]]`

class metabox.raster.**Rectangle**(*value:* Feature | *float* | *Tensor*, *center:* *Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *x_width:* Feature | *float* | *Tensor*, *y_width:* Feature | *float* | *Tensor*, *rotation_deg:* Feature | *float* | *Tensor = 0.0*)

Bases: *Shape*

Defines a rectangle.

center

The center of the rectangle.

Type
Tuple[*metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor, *metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor]

x_width

The width of the rectangle in the x direction.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

y_width

The width of the rectangle in the y direction.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

rotation_deg

The rotation of the rectangle in degrees.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

center: Tuple[*Feature* | float | Tensor, *Feature* | float | Tensor]

rotation_deg: *Feature* | float | Tensor = 0.0

x_width: *Feature* | float | Tensor

y_width: *Feature* | float | Tensor

class metabox.raster.**Shape**(*value:* metabox.utils.Feature | *float* | *tensorflow.python.framework.ops.Tensor*)

Bases: object

value: *Feature* | float | Tensor

metabox.raster.**rectangle_to_vertices**(*center: Tuple[float, float]*, *x_width: float*, *y_width: float*, *rotation_deg: float = 0*) → *Canvas*

Adds a rectangle to the canvas.

Parameters

- **canvas** – The canvas to add the rectangle to.
- **center** – The center of the rectangle.
- **x_width** – The width of the rectangle in the x direction.
- **y_width** – The width of the rectangle in the y direction.
- **rotation_deg** – The rotation of the rectangle in degrees.

Returns
The points of the rotated rectangle.

metabox.rcwa module

class metabox.rcwa.**Circle**(*material:* Feature | *float* | *Tensor* | *None*, *radius:* Feature | *float* | *Tensor*, *x_pos:* Feature | *float* | *Tensor = 0*, *y_pos:* Feature | *float* | *Tensor = 0*)

Bases: *Shape*

Defines a circle.

Parameters

- **material** – the ref. index of the shape.
- **radius** – the radius of the circle.
- **center** – the center of the circle. A tuple of (x, y) coordinates. Default: (0, 0)

get_shape(*wavelength: float* | *None = None*)

get_vertices(*num_of_vertices: int = 21*)

Returns the vertices of the circle.

radius: *Feature* | float | Tensor

x_pos: *Feature* | float | Tensor = 0

y_pos: *Feature* | float | Tensor = 0

class metabox.rcwa.**Layer**(*material:* Feature | *float* | Material, *thickness:* Feature | *float*, *shapes:* *Tuple*[Shape] = *()*, *enforce_4fold_symmetry:* *bool* = *False*)

Bases: *Parameterizable*

Defines a layer.

Parameters

- **material** – the ref. index of the shape.
- **thickness** – the thickness of the layer in meters.
- **shapes** – the shapes of the layer. Tuple of Shape objects. Default: ()

enforce_4fold_symmetry: `bool = False`

get_layer_unique_features() → List[*Feature*]

Returns the unique features of the layer.

get_shapes(*wavelength:* *float* | *None* = *None*)

Returns the shapes of the layer.

get_variables() → List[Variable]

Returns the variables of the layer.

initialize_values(*value_assignment:* *None* | *Tuple*[*List*[Feature], *List*[*float*]] = *None*) → None

Initializes the layer variables.

material: *Feature* | `float` | *Material*

shapes: `Tuple`[*Shape*] = ()

thickness: *Feature* | `float`

class metabox.rcwa.**Material**(*name:* *str*, *custom_csv_dir:* *str* | *None* = *None*)

Bases: `object`

Defines a material class.

A material provides a way to define the refractive index of a material given the wavelength of the simulation.

name

the name of the material. The .csv file name must be [name].csv The file extension i.e. *.csv* must be in lowercase.

Type
str

custom_csv_dir

the path to the csv file folder that contains the refractive index data. The data can be downloaded from refractiveindex.info. Just search for the material and download the csv file, under the "Data" section. Save the [CSV - comma separated] file as *csv_file_dir*.

Type
str | None

custom_csv_dir: `str | None = None`

index_at(*wavelength*)

Returns the refractive index at the given wavelength.

name: str

class metabox.rcwa.**Parameterizable**

Bases: object

Defines a parameterizable object.

get_features() → List[*Feature*]

Returns the features of the shape.

get_unique_features() → List[*Feature*]

Returns the unique features of the shape (non-recursively).

get_variables() → List[Variable]

Returns the variables of the shape.

initialize_values(*value_assignment: None | Tuple[List[*Feature*], List[float]] = None*) → None

Initializes the variables.

replace_feature_with_value() → None

class metabox.rcwa.**Polygon**(*material:* Feature | *float* | *Tensor* | *None*, *vertices: List[Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]]*)

Bases: *Shape*

Defines a polygon.

Parameters

- **material** – the *Material* or the ref. index of the shape.
- **vertices** – the vertices of the polygon. List of (x, y) coordinates. Example_0: [(0, 0), (1, 0), (1, 1), (0, 1)] Example_1:

  var = Feature(vmin=0, vmax=1, name="var") [(0, 0), (var, 0), (var, 1), (0, 1)]

get_shape(*wavelength: float | None = None*)

get_vertices()

Returns the vertices of the polygon.

vertices: List[Tuple[*Feature* | float | Tensor, *Feature* | float | Tensor]]

class metabox.rcwa.**ProtoUnitCell**(*proto_unit_cell:* UnitCell)

Bases: object

Defines an archetype of UnitCell (i.e. parameterized by `Feature`s).

Provides an interface to generate an array of unit cells from a tensor with shape (n_features, n_unit_cells).

proto_unit_cell

the unit cell that the children units cells are based of. The children unit cells will share the same Features as the parent unit cell.

Type
metabox.rcwa.UnitCell

generate_cells_from_parameter_tensor(*tensor: Tensor*) → List[*UnitCell*]

Returns an array of unit cells from a tensor shape: (n_cell, n_feat).

Parameters
tensor (*tf.Tensor*) – a tensor with shape (n_unit_cells, n_features).

Raises
ValueError – when the tensor does not have the correct shape.

generate_initial_variables(*n_cells: int*) → Tensor

Returns a tensor of initial variable parameters for the unit cells.

The tensor has shape (n_features, n_unit_cells).

Parameters
n_cell – the number of unit cells to generate.

Returns
A tensor of initial parameters for the unit cells.

proto_unit_cell: *UnitCell*

class metabox.rcwa.**Rectangle**(*material:* Feature | *float* | *Tensor* | *None*, *x_width:* Feature | *float* | *Tensor*, *y_width:* Feature | *float* | *Tensor*, *x_pos:* Feature | *float* | *Tensor* = *0*, *y_pos:* Feature | *float* | *Tensor* = *0*, *rotation_deg:* Feature | *float* | *Tensor* = *0*)

Bases: *Shape*

Defines a rectangle.

Parameters

- **material** – the ref. index of the shape.
- **x_width** – the width of the rectangle in the x direction.
- **y_width** – the width of the rectangle in the y direction.
- **x_pos** – the x position of the rectangle. Default: 0
- **y_pos** – the y position of the rectangle. Default: 0
- **rotation_deg** – the rotation of the rectangle in degrees. Default: 0
- **use_4_fold_symmetry** – whether to use 4-fold symmetry. Default: False If True, the rectangle will be rotated by 0, 90, 180, 270 degrees. Then

get_shape(*wavelength: float* | *None* = *None*)

get_vertices()

Returns the vertices of the rectangle.

rotation_deg: *Feature* | **float** | **Tensor** = **0**

x_pos: *Feature* | **float** | **Tensor** = **0**

x_width: *Feature* | **float** | **Tensor**

y_pos: *Feature* | **float** | **Tensor** = **0**

y_width: *Feature* | **float** | **Tensor**

class metabox.rcwa.**Shape**(*material:* Feature | *float* | *Tensor* | *None*)

Bases: *Parameterizable*

Defines a shape.

Parameters
material – the *Material* or the ref. index of the shape.

material: *Feature* | float | Tensor | None

class metabox.rcwa.**SimConfig**(*xy_harmonics: Tuple[int, int]*, *resolution: int*, *minibatch_size: int = 100*, *return_tensor: bool = False*, *return_zeroth_order: bool | None = None*, *use_transmission: bool | None = None*, *include_z_comp: bool | None = None*)

Bases: object

Defines a simulation configuration.

The SimConfig class is an immutable dataclass that provides the configuration and precomuputed data for the RCWA simulation. TODO: add fast convolution matrix

xy_harmonics

a tuple of (x, y) positive odd ints of Fourier harmonics.

Type
Tuple[int, int]

x_resolution

the grid x resolution of the simulation of the real space. Note that the grid y resolution is determined by the aspect ratio of the unit cell.

minibatch_size

the minibatch size of the simulation.

Type
int

return_tensor

whether to use tensor as the output. If False, then the following arguments are ignored:

return_zeroth_order, use_transmission, include_tz. And the output is the in the form of *SimResult*.

If True, the output is in the form of a tf.Tensor, and the following
arguments are used: return_zeroth_order=True, use_transmission=True, include_z_comp=False.

Type
bool

return_zeroth_order

whether to use zeroth order diffraction as the output.

Type
bool | None

use_transmission

whether to use transmission as the output.

Type
bool | None

include_z

whether to include the z component of the electric field.

include_z_comp: bool | None = None

minibatch_size: int = 100

resolution: int

return_tensor: bool = False

return_zeroth_order: bool | None = None

use_transmission: bool | None = None

xy_harmonics: Tuple[int, int]

class metabox.rcwa.**SimInstance**(*unit_cell_array: List[*UnitCell*]*, *incidence:* Incidence, *sim_config:* SimConfig)

Bases: object

Defines a simulation instance.

unit_cell_array

an array of unit cells to be simulated.

Type
List[*metabox.rcwa.UnitCell*]

incidence

the incidence of the simulation.

Type
metabox.utils.Incidence

sim_config

the simulation configuration.

Type
metabox.rcwa.SimConfig

get_variables() → List[Variable]

Returns the variables of the simulation instance.

incidence: *Incidence*

sim_config: *SimConfig*

unit_cell_array: List[*UnitCell*]

class metabox.rcwa.**SimResult**(*rx: Tensor*, *ry: Tensor*, *rz: Tensor*, *r_eff: Tensor*, *r_power: Tensor*, *tx: Tensor*, *ty: Tensor*, *tz: Tensor*, *t_eff: Tensor*, *t_power: Tensor*, *xy_harmonics: Tuple[int, int]*)

Bases: object

The result of an RCWA simulation.

rx

the x component of the reflected diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

ry

the y component of the reflected diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

rz

the z component of the reflected diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

r_eff

the reflective efficiency.

Type
tensorflow.python.framework.ops.Tensor

r_power

the total reflected power.

Type
tensorflow.python.framework.ops.Tensor

tx

the x component of the transmitted diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

ty

the y component of the transmitted diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

tz

the z component of the transmitted diffraction coeff.

Type
tensorflow.python.framework.ops.Tensor

t_eff

the transmissive efficiency.

Type
tensorflow.python.framework.ops.Tensor

t_power

the total transmitted power.

Type
tensorflow.python.framework.ops.Tensor

get_result_using_config(*config:* SimConfig) → *SimResult* | Tensor

Returns the result according to the simulation configuation.

Parameters
config – the simulation configuration.

Returns
The result according to the simulation configuation.

r_eff: Tensor

r_power: Tensor

ref_field(*config:* SimConfig) → Tensor

Returns the reflected diffraction coefficients.

Returns
The reflected field according to the simulation configuation.

rx: Tensor

ry: Tensor

rz: Tensor

t_eff: Tensor

t_power: Tensor

trn_field(*config:* SimConfig) → Tensor

Returns the transmitted diffraction coefficients.

Parameters
config – the simulation configuration.

Returns
The transmitted field according to the simulation configuation.

tx: Tensor

ty: Tensor

tz: Tensor

xy_harmonics: Tuple[int, int]

class metabox.rcwa.**UnitCell**(*layers: List[*Layer*]*, *periodicity: Tuple[*Feature | *float* | *Tensor,* Feature | *float* | *Tensor]*, *refl_index:* Feature | *float* | *Tensor = 1.0*, *tran_index:* Feature | *float* | *Tensor = 1.0*)

Bases: *Parameterizable*

Defines a unit cell.

layers

the layers of the unit cell.

Type
List[*metabox.rcwa.Layer*]

periodicity

a tuple of (x, y) in meters that define the periodicity of the unit cell in the x and y direction.

Type
Tuple[*metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor, *metabox.utils.Feature* | float | tensorflow.python.framework.ops.Tensor]

refl_index

the ref. index in the reflection region.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

tran_index

the ref. index in the transmission region.

Type
metabox.utils.Feature | float | tensorflow.python.framework.ops.Tensor

find_feature_index(*feature_str*)

Returns the index of the feature with the given name.

get_cell_unique_features() → List[*Feature*]

Returns the unique features of the unit cell.

get_epsilon(*x_resolution: int*, *wavelength: float*) → Tensor

Returns the permittivity of the unit cell.

Parameters
x_resolution – the resolution of the permittivity in the x direction.

Returns
The permittivity of the unit cell as a tf.Tensor.

get_thickness() → Tensor

Returns the thickness of the unit cell as a tf.Tensor.

get_variables() → List[Variable]

Returns the variables of the shape.

initialize_values(*value_assignment: None | Tuple[List[*Feature*], List[float]] = None*) → None

Initializes the layer variables.

layers: List[*Layer*]

periodicity: Tuple[*Feature* | float | Tensor, *Feature* | float | Tensor]

refl_index: *Feature* | float | Tensor = 1.0

replace_features()

Replaces the features with the given values.

tran_index: *Feature* | float | Tensor = 1.0

metabox.rcwa.**combine_sim_results**(*sim_results: List[*SimResult*] | List[Tensor]*) → *SimResult* | Tensor

Combines a list of simulation results into one

Parameters
sim_results – the list of simulation results.

Returns
The combined simulation result.

metabox.rcwa.**duplicate_shape**(*shape:* Shape, *num_of_duplicates: int*) → List[*Shape*]

Generates a list of duplicate parameterized shapes.

The returned shapes share the same parameters as the input shape. But with different unique names.

Parameters

- **shape** – the shape to be duplicated.
- **num_of_duplicates** – the number of duplicates.

Returns
A list of duplicated shapes.

metabox.rcwa.**get_avaliable_materials**(*custom_csv_dir: str | None = None*) → List[str]

Returns a list of avaliable material strings in the given directory.

metabox.rcwa.**minibatch_sim_instance**(*sim_instance:* SimInstance, *minibatch_size: int*) → List[*SimInstance*]

Generates a list of minibatch simulation instances.

Parameters

- **sim_instance** – the simulation instance.
- **batch_size** – the batch size.

Returns
A list of minibatch simulation instances.

metabox.rcwa.**simulate**(*sim_instance:* SimInstance) → *SimResult*

Simulates the periodic unit cell using RCWA.

Calculates the transmission/reflection coefficients for a unit cell with a given a simulation instance (SimInstance), which contains the unit cell, incidence, and simulation configuration.

Parameters
sim_instance – the simulation instance.

Returns
The simulation result.

metabox.rcwa.**simulate_batch**(*sim_instances: List[*SimInstance*]*) → List[*SimResult*]

Simulates a batch of periodic unit cells using RCWA.

Calculates the transmission/reflection coefficients for a batch of unit cells with a given a list of simulation instances (SimInstance), which contains the unit cell, incidence, and simulation configuration.

TODO: parallelize this function.

Parameters
sim_instances – list of simulation instances.

Returns
The list of simulation results.

metabox.rcwa.**simulate_one**(*sim_instance:* SimInstance) → *SimResult*

Simulates the periodic unit cell using RCWA.

Calculates the transmission/reflection coefficients for a unit cell with a given a simulation instance (SimInstance), which contains the unit cell, incidence, and simulation configuration.

Parameters
sim_instance – the simulation instance.

Returns
The simulation result.

metabox.rcwa.**simulate_parameterized_unit_cells**(*parameter_tensor: Tensor*, *proto_cell:* ProtoUnitCell, *incidence:* Incidence, *sim_config:* SimConfig) → Tensor

Simulate RCWA and precompute the JVP with better memory efficiency.

This method computes the zeroth order diffraction coefficients and the Jacobian of the diffraction coefficients with respect to the unit cell parameters.

Parameters

- **parameter_tensor** – the tensor of unit cell parameters, in the shape (num_features, num_unit_cells).
- **proto_cell** – a parameterized unit cell.
- **incidence** – the incidence data.
- **sim_config** – the simulation configuration.

Returns

The 0th order diffraction coefficients. In the form of a tf.Tensor of shape (batch_size, num_unit_cells, 2).

metabox.rcwa.**simulate_parameterized_unit_cells_one_batch**(*parameter_tensor: Tensor*, *proto_cell:* ProtoUnitCell, *incidence:* Incidence, *sim_config:* SimConfig) → Tensor

metabox.utils module

class metabox.utils.**Feature**(*vmin: float*, *vmax: float*, *name: str*, *initial_value: float | None = None*, *sampling: None | int = None*, *value: Variable | None | float = None*)

Bases: object

Defines a feature variable.

Parameters

- **vmin** – the minimum value of the feature.
- **vmax** – the maximum value of the feature.
- **name** – the name of the feature.
- **sampling** – the number of samples to take between vmin and vmax. If None, the sampling is undefined.

initial_value: float | None = None

initialize_value() → None

Initializes the variables of the feature.

name: str

sampling: int | None = None

set_value(*value: Any*) → None

Set the value of the feature to value

set_variable() → None

Convert self.value to a variable

value: Variable | None | float = None

vmax: float

vmin: float

class metabox.utils.**Incidence**(*wavelength: Tuple[float]*, *theta: Tuple[float] = (0,)*, *phi: Tuple[float] = (0,)*, *jones_vector: Tuple[float] = (1, 0)*)

Bases: object

Defines the physical properties of the incident light.

Parameters

- **wavelength** – the wavelengths of the light in meters.
- **theta** – tuple of the angles of incidence in degrees on the xz plane. Defaults to (0).
- **phi** – tuple of the angles of incidence in degrees on the yz plane. Defaults to (0).
- **jones_vector** – the Jones vector of the incident light. Defaults to (1, 0) which corresponds to a linearly polarized light with the electric field vector parallel to the x axis.

jones_vector: Tuple[float] = (1, 0)

phi: Tuple[float] = (0,)

theta: Tuple[float] = (0,)

wavelength: Tuple[float]

metabox.utils.**recursively_convert_ndarray_in_dict_to_list**(*item: Any*)

Recursively converts ndarray item in dict to list

metabox.utils.**suppress_stdout_stderr**() → None

A context manager that redirects stdout and stderr to devnull

metabox.utils.**unravel_incidence**(*incidence:* Incidence) → Dict[str, Any]

Serializes an incidence data into lists.

metabox.utils.**unravel_wavelength_theta_phi**(*wavelength: List[float]*, *theta: List[float]*, *phi: List[float]*) → Tuple[Tensor, Tensor, Tensor]

Unravels the wavelength, theta, and phi lists into tensors.

Parameters

- **wavelength** – a list of wavelengths in meters.
- **theta** – a list of angles of incidence in degrees on the xz plane.
- **phi** – a list of angles of incidence in degrees on the yz plane.

Returns

A tuple of tensors of shape (batch_size,).

metabox.utils.**wavelength_to_rgb**(*wavelength*)

Convert a wavelength in the visible spectrum to RGB Input: wavelength (in m) Output: RGB tuple

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y