bmiptools

Release 0.5

Curcuraci Luca

Dec 06, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

bmiptools is a Python library equipped with a basic graphical interface useful for the processing of images, typically produced in some biological context (FIB-SEM and cryo FIB-SEM in particular).

1.1 Scope

The goals of this library are the following:

1. **Open code implementation of various image processing transformations**. Ideally the user should have access to the actual implementation of any tool is using, so that everything can be checked at low level if needed.
2. **Open documentation of various image processing transformations**. The user should have access to a documentation explaining the rationale behind the method implemented and how this was implemented in the code.
3. **User friendly interfaces**. The user should be able to use this Python library even with minimal experience with Python. Alternatively, non expert users should be able to use the majority of the library functionalities with a simple GUI.
4. **Objective parameter selection to reduce human bias in the pre-processing**. The user should specify just the parameters ranges rather than the parameters values. The best parameters are later selected, in automatic manner, according to reasonable objective criteria.
5. **Parameter tracking in a human understandable manner**. The user should be able to collect all the parameters describing a series of transformations, and they need to be stored in an ordered and a human understandable way.
6. **Easy result reproducibility**. Anyone should be able to reproduce the result of some post-processing pipeline on its computer with minimal efforts.
7. **Metadata management**. Metadata do not have to get lost during the post-processing.
8. **Open to external contribution**. The user with a minimum of coding experience should be able to implement their own custom extension integrable in bmiptools with minimal efforts.

1.2 Repository

The bmitools repository is available on the mpikg gitlab, in particular at

- https://gitlab.mpikg.mpg.de/curcuraci/bmiptools

1.3 How to cite this tool?

Bmiptools is released under the Apache 2.0 License. It can be freely used by anyone for whatever scope, provided that credits are recognized somewhere. Bmiptools can be cited as reported below…

> "bmiptools - BioMaterials Image Processing Tools" - The monochromatic bmiptools team - Patagonian temple for the Russell's teapot of the Flying Spaghetti Monster on Ceres, Asteroids belt, June 2022.

..but, those how do not have enough sense of humor and take themself too seriously can alternatively use

> Curcuraci et al., (2022). bmiptools: BioMaterials Image Processing Tools. Journal of Open Source Software, 7(79), 4859, https://doi.org/10.21105/joss.04859.

CHAPTER

TWO

INSTALLATION AND CONFIGURATION

2.1 Installation

bmiptools is available on PyPI and can be installed simply using pip. bmiptools has been developed using Python 3.8. To create and setup the python environment where bmiptools can run, one needs to install Anaconda on the computer.

2.1.1 CPU

To install bmiptools and use it on CPU only open the Anaconda prompt and write the instruction below.

```
$ conda create -n bmiptools_env python=3.8
$ conda activate bmiptools_env
$ (bmiptools_env) pip install bmiptools
```

2.1.2 GPU

To run bmiptools with GPU acceleration you need a CUDA-compatible GPU having compute capability of 3.5 or higher. To install bmiptools open the Anaconda prompt and write the instruction below.

```
$ conda create -n bmiptools_env python=3.8
$ conda activate bmiptools_env
$ (bmiptools_env) conda install cudatoolkit==10.1.243
$ (bmiptools_env) conda install cudnn==7.6.5
$ (bmiptools_env) pip install bmiptools
```

If all worked correctly, this package and the python environment which is necessary to make bmiptools works has been installed. To check the successful installation, one may simply get some basic information about bmiptools by plotting some information. This can be done by executing the 3 line of code below in a python terminal (e.g. in the anaconda propt which comes from free once anaconda is installed).

```
>>> import bmiptools
>>> print(bmiptools.__version__)
'v0.4.0'
>>> print(bmiptools.info())
'Developed by  Curcuraci Luca  @  MPICI - Max Planck Institute of Colloids and Interfaces
↪'
'Tool name:  BioMaterials Image Processing Tools - bmiptools'
'Version:  v0.4.0'
'Tool scope:  Image processing tools for typical FIB-SEM and micro-CT images acquired at␣
```

(continues on next page)

(continued from previous page)

```
↪the institute.'
'Contributors: '
' ['Bertinetti Luca']'
'Manual available @  https://bmiptools.readthedocs.io/en/latest/ '
```

Note: It is also possible to install bmiptools from its GitLab repository. To to that it is sufficient to replace the last line of the code snapshot above with the line below.

```
$ (bmiptools_env) pip install git+ssh:git@gitlab.mpikg.mpg.de:curcuraci/bmiptools.git
```

This way should be preferred, if one wants to have the most updated version of bmiptools.

2.1.3 Unit test

It is also possible to check that bmiptools has been correctly installed, by running the unit tests. See here to understand how to run them.

2.2 Configuration

bmiptools can be configured in order to limit the computational resources used. This operation is normally done the first time, if needed. The bmiptools configuration is contained in the *global_setting.txt* file, which can be found in the *./setting/files* folder (the path is specified with respect to the bmiptools library folder). An example of its content can be found below

```
verbosity = 1
use_multiprocessing = 1
multiprocessing_type = 1
cpu_buffer = 2
use_gpu = 0
```

Modifying the values in this file, one may change the global setting of bmiptools. What can be configured in bmiptools is the following:

- set the verbosity level of the library. Setting `verbosity = 0` will reduce the number of printed messages, while with `verbose=1 all the messages will be printed.
- enable/disable multiprocessing. Setting `use_multiprocessing=0` the code is executed on a single core, while with `use_multiprocessing=1` multiprocessing is enables.
- select the type of parallelization (currently not used).
- specify the number of cpu that are not used. Given a processor with N>1 core and M<N, setting `cpu_buffer = M` only N-M core are used by bmiptool.
- enable/disable gpu usage (currently under development: leave it equal to 0. In the acutal version the GPU is always used if available when needed).

Rather than modifying the *global_setting.txt* directly, one can modify this file using some function of bmiptools. For example the same configuration above can be obtained as follow:

```
import bmiptools
bmiptools.set_verbosity(1)
bmiptools.set_use_multiprocessing(1)
bmiptools.set_multiprocessing_type(1)
bmiptools.set_cpu_buffer(2)
bmiptools.set_use_gpu(0)
```

A detailed documentation about these function can be found in `bmiptools.setting.configure`. As can be seen from the documentation, it is possible to specify a different path for the *global_setting.txt*.

Attention: The *global_setting.txt* is read every time a plugin, a stack or a pipeline object is initialized. As such, changing the value in this file **after** the initialization of one of those object in your script, **does not** change the behavior of the library for that plugin/stack/pipeline object until it will be reinitialized.

CHAPTER

THREE

BASIC API USAGE

bmiptools can be used as Python API in your script. The two basic objects of this API are:

- *Stack objects*, which can be used to do all the I/O operations;
- *Pipeline object*, which can be used to apply a series of transformations to a Stack objects.

3.1 Stack

A Stack is the basic object of bmiptools where the data can be stored. A stack can be loaded from an multitiff image or from a folder containing a collection of tiff images, and saved in the same way. In addition a stack can also be saved as gif file, to have a rapid 3D preview of the content. Finally one can also initialize an empty Stack and fill it in a second time with a numpy array. Details on Stack objects and its methods can be found in `bmiptools.stack.Stack`.

3.1.1 Load a Stack

A single multitiff or a collection of tiff images in a folder can be loaded in stack object. For a multitiff one can use the code below

```
from bmiptools.stack import Stack

path_to_multitiff_image = r'PATH_TO_MULTITIFF'
stack = Stack(path = path_to_tiff_image)
```

In case the stack is contained in a folder (as collection of tiff) images, the code below show how to load it. Note that this time you have to specify the path of the folder containing the images.

```
from bmiptools.stack import Stack

path_to_tiff_folder = r'PATH_TO_FOLDER'
stack = Stack(path = path_to_tiff_image,from_folder = True)
```

Note: Alternatively one can always initialize an empty stack and later use the method `Stack.load_stack` or `Stack.load_stack_from_folder`, to load a multitiff stack or a stack contained in a folder. The code below shows as example, how to load a stack from a folder in this way.

```
from bmiptools.stack import Stack

path_to_tiff_folder = r'PATH_TO_FOLDER'
```

(continues on next page)

(continued from previous page)

```
stack = Stack()
stack.load_stack_from_folder(path=path_to_tiff_folder)
```

Slice loading order

The stack is reconstructed assuming that the alphabetic order of the single tiff images is equal to the ordering along the z-axis. Note that this may not always give the correct order of the slice. For example, if the slice name are

```
slice_01.tiff
slice_02.tiff
...
slice_09.tiff
slice_10.tiff
```

reading the slice in alphabetic order would give

```
slice_01.tiff
slice_10.tiff
slice_02.tiff
...
slice_09.tiff
```

which is clearly the wrong order if the enumeration of files have some sense. This is why the user can choose among different ordering possibility. This can be done by specifying the `image_type` field during the initialization of the stack.

```
from bmiptools.stack import Stack

path_to_tiff_folder = r'PATH_TO_FOLDER'
stack = Stack(path = path_to_tiff_image,from_folder = True,image_type = 'FIB-SEM')
```

At the moment two options are possible:

- `image_type = None`: the slices are loaded according to a simple alphabetic order.
- `image_type = 'FIB-SEM'`: it is the default value for this field. It assumes that each slice of the stack loaded has the following structure:

  ```
  [ARBITRARY_NAME]slice_[NUMBERS].tiff
  ```

  where `[ARBITRARY_NAME]` is an arbitrary string of character, while `[NUMBERS]` is an arbitrary number. With this option, the slice are orderd in increasing order with respect to the number specified in the `[NUMBER]` part of the name. In this way the problem with the example above disappear.

Note: How to specify the file extension. Sometimes one need to specify the file extension in order to correctly load the stack (both from folder or from multitiff). This can be done by specifying it in the variable `loading_extension` of a Stack when you initialize it (it is set equal to `tiff` as default.)

Load slices

Rather than loading the whole stack, one can load just subset of slices. This can be done initializing an empty stack al call later the method `Stack.load_slices` specifying the slice list. The code below show how this can be done for a multitiff image.

```
from bmiptools.stack import Stack

path_to_multitiff_image = r'PATH_TO_MULTITIFF'
stack = Stack()

slice_list = [0,3,10]                                    # list of slices to load␣
↪(enumeration start from 0)
stack.load_slices(path_to_multitiff_image,S=slice_list)
```

In case the stack is contained in a folder of tiff images, the last line need to be replaced as follow

```
stack.load_slices_from_folder(path=path_to_tiff_folder,S=slice_list)
```

Keep in mind that the number in the slice list are the position of the path to the slices ordered according to a given convention. As already explained, the ordering can be specified during the stack initialization via the variable `image_type` (see *above*).

3.1.2 Fill a stack

Finally a stack can be initialized empty, and filled later using a numpy array.

```
import numpy as np
from bmiptools.stack import Stack

# empty stack
stack = Stack(load_stack = False)

# generate some random content
x = np.random.uniform(0,1,size=(30,200,200))

# fill the stack
stack.from_array(x)
```

3.1.3 Save a Stack

The content of stack can be saved using the method `Stack.save`. The code below show that, producing a tiff file containing the stack.

```
import numpy as np

saving_path = 'PATH_TO_THE_FOLDER_WHERE THE_STACK_IS_SAVED'
saving_name = 'STACK_NAME'
stack.save(saving_path,saving_name,standardized_saving=True,data_type=np.uint8,mode='all_
↪stack')
```

This code will save the stack as a single multitiff image. To save the stack as a folder containing a tiff image for each slice, one have to set `mode = 'slice_by_slice'`. With this option

Another possibility to save a stack is via the method `Stack.save_as_gif`, which save the stack content as an animated gif. This may help to visualize the 3d features of the stack, but the resolution is limited by the feature of the GIF format. The code below show how this can be done.

```
import numpy as np

saving_path = 'PATH_TO_THE_FOLDER_WHERE THE_STACK_IS_SAVED'
saving_name = 'STACK_NAME'
stack.save_as_gif(saving_path,saving_name,standardized_saving=True,data_type=np.uint8)
```

Note: The options `standard_saving` and `data_type` present in both saving methods are particularly important, and deserve some discussion. In order produce images that can be open with the usual image reader, the images need to be saved in a specific way, depending on the data format chosen. In particular for an 8-bit integer representation (using `data_type = np.uint8`) the typical image viewer expect that in all the image channels the values are integers between 0 and 256. Similarly, for a 32-bit float representation (using `data_type = np.float32`) the typical image viewer expect that in all the image channels the values are 32 bit float between 0 and 1. Even if the input stack is in a viewer compatible format, this is not guaranteed anymore after the application of a plugin. The option `standard_saving = True` rescales the images in a suitable way (based on the data type chosen), so that the saved tiff are all viewer compatible.

3.1.4 Basic Stack operations

Slicing

The data in an a Stack object is stored in the attribute `.data`, but one can access to the data in a more natural why. Stack allow a numpy-like slicing, as the code below show

```
import numpy as np
from bmiptools.stack import Stack

# fill a stack with some data
content = np.random.uniform(0,1,size=(20,20,20))
stack = Stack(load_stack=False)
stack.from_array(content)

# get the first 5 slices
a1 = stack[:5]
print(a1 == content[:5])

# get the stack content in the top-left 10x10 square
a2 = stack[:,:10,:10]
print(a2 == content[:,:10,:10])

# get whole stack content and store in a numpy array
a3 = stack.data
print(a3 == content)

a4 = stack[:,:,:]
print(a4 == content)
```

Stack statistics

As soon as some data is loaded in stack, or a stack is filled, a series of simple statics are computed. In particular, they are:

- `.stack_mean`, contains the mean value of the *whole* stack;
- `.stack_std`, contains the standard deviation of the *whole* stack;
- `.slices_means`, contains a list of mean values *for each slice* of the stack;
- `.slices_stds`, contains a list of standard deviations *for each slice* of the stack;
- `.min_stack`, contain the smallest pixel/voxel value of the *whole* stack;
- `.max_stack`, contains the largest pixel/voxel value of the *whole* stack;
- `.min_slices`, contains a list of the smallest pixel values *for each slice* of the stack;
- `.max_slices`, contains a list of the largest pixel values *for each slice* of the stack.

The method `Stack.statistics` of a stack object returns a dictionary containing all these quantities.

```
import numpy as np
from bmiptools.stack import Stack

# fill a stack with some data
content = np.random.uniform(0,1,size=(20,20,20))
content[2,2,2] = 100                                    # set the maximum of the stack
stack = Stack(load_stack=False)
stack.from_array(content)

# get maximum of the stack
print(stack.stack_max)

# get statistics
print(stack.statistics())
```

Stack metadata

Sometimes tiff images contains relevant metadata. To load them when also the images are loaded just use the code below:

```
from bmiptools.stack import Stack

path_to_stack = r'PATH_TO_STACK'
stack = Stack(path_to_stack,load_metadata=True)
```

To load metadata, one have to specify `load_metada = True` during the stack initialization. There are 3 type of metadata that are loaded:

1. **image metadata**: are those metadata containing image information like image color depth, image dimension, image type, ecc… namely the basic metadata TAG of the tif format, (see here).

2. **experimental metadata**: are those metadata containing the information related to the image acquisition process. The experimental metadata reading and interpretation in bmiptool is done by `bmiptools.stack.ExperimentalMetadataInspector`.

3. **image processing metadata**: are those metadata containing the information relate to the image processing transformations done by bmiptools itself. They are produced at the end of the application of a bmiptools Pipeline (see *later*).

Attention: At the moment the automatic loading of the experimental metadata may work only in a restricted number of cases, due to the lack of standardization in the metadata organization.

To access to the metadata at later times one can use the attribute `.metadata`.

```
print(stack.metadata)
```

Metadata can be added also at later time, using the method `Stack.add_metadata`. The code below show how to add the metadata called 'added_metadata' having as content the string 'example content' can be added.

```
stack.add_metadata('added_metadata','example content')
print(stack.metadata)
```

The added content can be of any kind (e.g. int, list, dictionary,ecc..) and not only string. Finally, when a stack is saved and the option `save_metadata = True` is used, the metadata dictionary is saved as json file in the same path in which the stack is saved.

3.2 Pipeline

The second basic object of the library is the Pipeline object. A pipeline is an object which apply a series of image-processing transformation to a given input stack. Those image-processing transformation are the so called bmiptools plugins (see section *General information about plugins* to have a list and a description of the currently available plugins). The main features of a bmiptools pipeline are:

1. A pipeline keeps track automatically of all the parameters used, both the ones chosen by the user and the ones obtained at the end of an optimization process.

2. A pipeline can be saved and loaded in a later time reproducing exactly the same result.

3. A pipeline can save automatically a preview of a restricted number of slice of the input stack after the application of each plugin of the pipeline.

To use a pipeline of transformation on a stack one have to create and initialize a Pipeline object. After that the pipeline applied to a stack object and later can be saved. In general, this is the typical order that one need to follow to use pipeline objects in bmiptools. Alternatively, rather that create and initialize a pipeline, one can simply load an already existing one.

3.2.1 List available plugins

To have an idea on the kind of transformations that can be applied to a stack, one can list the available plugins. The list of the currently available plugin is contained in the `PLUGINS` dictionary of the `installed_plugins` module. This dictionary is imported in `bmiptool.pipeline` file, therefore `PLUGINS` is a global attribute of the pipeline module. Thus the list of installed plugins can be obtained as follow:

```
from bmiptools.pipeline import PLUGIN

print(PLUGINS.keys())
```

More information about currently installed plugins can be found in the section *General information about plugins*.

3.2.2 Pipeline creation

A pipeline can be created from scratch, with the method `Pipeline.create`. When calling this method, one need to:

1. specify a list of plugins writing the name of the plugins and their order of application in the list (plugins can be repeated multiple times);
2. specify a folder used to save all the pipeline information;
3. (optional) specify a pipeline name.

The name of the plugins are the one that can be seen when they are listed (see *above*). The code below is an example of how to create a pipeline.

```
from bmiptools.pipeline import Pipeline

operation_lists = ['Standardizer','Flatter','Decharger']
pipeline_path = r'PATH_TO_PIPELINE_FOLDER'
name = 'NAME'

pipeline = Pipeline(operation_lists = operation_lists,
                    pipeline_folder_path = pipeline_path,
                    pipeline_name = name)
```

The order given in `operation_list` is the order in which the plugins are applied to the stack. Once that the pipeline is created by executing the code above, a json file is created in the pipeline folder. This `pipeline_[PIPELINE_NAME].json` file is called *pipeline json* and contains all the information about the pipeline and represent the way the user can interact with all the plugins setting, when pipeline objects are used. In this json file, the field *pipeline_setting* contains a series of dictionary (one for each plugin) containing all the parameters of the plugins. The user have to set these parameter manually and save the json file once that this is done. The meaning of the various parameters for each plugins can be found in section *General information about plugins*.

Attention: Fit order. By default a plugin is fitted (if possible) just before the application of it on the stack. On the other hand the fit and application of the plugin may be done in different time. This can be done by specifying when the fit have to be done in the `operation_lists` by writing `fit_` before the name of the plugin. In the example below, the fit of the `Flatter` plugin happens before the application of the `Decharger` plugin, and only then the `Flatter` plugin is applied.

```
operation_lists = ['Standardizer','fit_Flatter','Decharger','Flatter']
```

How to configure the pipeline json setting

The pipeline json produced once a pipeline is created contains for each plugin of the pipeline all the parameters which can be set by the user. The structure of this json is in general the following.

```
{"pipeline_name": "pipeline__01011901_0000",
"pipeline_creation_date': "01/01/1901 at 00:00",
"bmiptools_version": "v0.5",
"plugins_list": ["Plugin_1",...,"Plugin_N"],
"true_operations_list": ["fit_Plugin_1","Plugin_1",...,"fit_Plugin_N","Plugin_N"],
"pipeline_setting":{"Plugin_1": {
                                  ...        # transformation dictionary Plugin_1
                                  },
```

(continues on next page)

(continued from previous page)

```
                    ...
                    "Plugin_N": {
                                 ...        # transformation dictionary Plugin_N
                                 }
                    }
}
```

As mentioned above, the parameters for each plugin can be found in the `"pipeline_setting"` field. In this field, the parameters for the plugin `Plugin_x` can be found in the dictionary in the field having the same plugin name. This dictionary is called *transfomation dictionary* and a description of its general structure and the explanation of some general parameters can be found *here*, while for each plugin in this documentation all the plugin specific parameters are explained in the corresponding plugin page.

Attention: Python dictionaries and json files. bmiptools is written in Python. As such the code snapshot in this documentation are generally written in python. By the way the pipeline json is written in json (as the code snapshot above show). The code snapshots of the transformation dictionary of the various plugins are written in Python (since they are Python dictionaries). Therefore if one want to use this documentation to fill the pipeline json, one need to *convert* the Python notation (data-structure to be precise) into the JSON one. Fortunately the two notations are already very similar, and only few things need to be changed. The table below should be sufficient for this scope.

Python	JSON
'	"
numpy.array([])	[]
True	true
False	false
None	"null"

See here for more details on the Python-JSON conversion. An example of conversion can be the following.

Python dictionary	JSON file
{'key1': numpy.array([[0,1],[1,0]]), 'key2': None, 'key3': False, 'key4': {'key41': None, 'key42': True } }	{"key1": [[0,1],[1,0]], "key2": "null", "key3": false, "key4": {"key41": "null", "key42": true } }

3.2.3 Load pipeline template

Rather than create a pipeline from zero, one can create them one time and save the pipeline json produced in some folder and use it different times. In order to create a pipeline object in this way, one has to first create an empty pipeline object, and load the template in a later time with the method `Pipeline.load_pipeline_template_from_json`. The code below show how this can be done.

```
from bmiptools.pipeline import Pipeline

path_to_pipeline_template = r'PATH TO PIPELINE TEMPLATE JSON'
path_to_pipeline_folder = r'PATH TO PIPELINE FOLDER'

pipeline = Pipeline()                                                    # initialize an␣
↪empty pipeline
pipeline.load_pipeline_template_from_json(pipeline_template_path = path_to_pipeline_
↪template,
                                          new_pipeline_folder_path = path_to_pipeline_
↪folder)
```

3.2.4 Pipeline initialization

Once that the pipeline is created and the pipeline json is filled, or once that a pipeline template is loaded, the pipeline object can be initialized. This operation initialize all the specified plugins, i.e. executing all the input independent operations for each plugin, so that the pipeline is ready for the application to a stack. The pipeline can be initialized using simply the code below.

```
pipeline.initialize()
```

3.2.5 Pipeline application

The application of the pipeline (with eventual fitting according to the order specified during the creation) on some stack called `stack`, can be done simply as follow:

```
pipeline.apply(stack)
```

Note: The order in which the plugins are fitted and applied can be seen from the pipeline json at the `true_operations_list` field. This field contains a list with the name of the true operations that are applied at a given step. For example given

```
true_operations_list = ['fit_Flatter','Flatter','fit_Registrator','fit_HistogramMatcher',
                        'HisogramMatcher','Registrator']
```

one can understand that the `Flatter` plugin is first fitted and then applied to the input stack, later the `Registrator` plugin is fitted *but not applied*. Indeed after this operation `HistogramMatcher` is fitted and then applied, and only at the end the (already fitted) `Registrator` plugin is applied.

Get a preview

There is the possibility to obtain a preview showing how the input stack is transformed at each step of the pipeline (i.e. after the application of each plugin of the pipeline). In order to do that, one have to call the method `Pipeline.setup_preview` *before* the application of the pipeline. In this method, one has to specify:

1. `slice_list`, namely the list of integer indicating slice used to produce the preview.
2. `plugin_to_exclude`, namely a list containing the name of the plugins which are not considered for the construction of the preview.

The code below should be used in order to get the preview *during* the execution of the pipeline, instead of the previous line of code.

```
pipeline.setup_preview(slice_list = [0,1,5,7],                                  # specify␣
↪preview setting.
                       plugin_to_exclude = ['Standardizer','Registrator'])
pipeline.apply(stack)                                                           # apply␣
↪pipeline on the stack.
```

As soon as the pipeline is applied, a folder inside the pipeline folder called `preview` is created, and inside the slices specified in `slice_list` as saved before the application of any transformation in the folder 'original'. During the pipeline application, after the application of each plugin a folder with the name of the plugin is created, provided that the plugin is not in the `plugin_to_exclude` list. In this folder the selected slice of the stack at that step of the pipeline are saved.

3.2.6 Pipeline saving

After the application of the pipeline two things happens:

1. The stack object now contains the result of all the plugin applied according the specified order.
2. The plugins forming the pipeline has been optimized (the ones that can be fitted) on the specific input (the input stack for the first plugin, the output of the plugins preceding the corresponding `fit_` methods for all the other).

The pipeline at this point can be saved with the `Pipeline.save` method.

```
pipeline.save()
```

Attention: The stack have to be saved separately using the methods described *before*. Saving the pipeline does not save the stack automatically!

The saving process produces two file in the pipeline folder chose in the beginning (during the creation or during the template loading):

1. A '.dill' file, which really contain the pipeline. This is the file that need to be used to load a pipeline. In case some plugin has dill incompatible component, an additional `undillable` folder is created. This folder contains the part of the pipeline that require a custom saving and loading operations. This folder need to be in the same folder of the dill file, in order to load the pipeline later.
2. A json file containing the pipeline json *update*, i.e. containing the parameters found during the optimization of the plugins (if any).

Note: Note that a pipeline can be saved also after the initialization (but before the application).

3.2.7 Pipeline loading

Once that the pipeline has been saved, it can be load with the method `Pipeline.load`. The code below, show how one can use it.

```
path_to_pipeline_file = r'PATH_TO_DILL_FILE'
pipeline.load(path_to_pipeline_file)
```

After the loading the pipeline, it can be applied using the code explained in the *apply subsection* above. However, in case the pipeline was saved after a fit, the application of the loaded pipeline does not execute a new fit, but uses the parameters found previously.

3.3 Further reading

Tutorials:

- *Basic operations with stacks and pipelines*

CHAPTER

FOUR

ADVANCED API USAGE

The `Pipeline` object is the most automatized way to apply a series of transformations to a stack. By the way it does not allow to have a low level interaction with the individual plugins. This can be important for example to control the result of a certain transformation on a stack, without applying a full pipeline of transformation. In bmiptools a 'low-level' interaction with the plugins is possible.

4.1 Initialize a plugin

In bmiptools the default plugins can be found in `bmiptools.transformation`. A list of the possible available plugins can be found in the `PLUGINS` dictionary of the `bmiptools.setting.installed_plugins` module (as explained *here*), which contains also the plugins installed locally by the user (see *here* for more details)

```
from bmiptools.setting.installed_plugins import PLUGIN

print(PLUGINS.keys())
```

Every plugin in bmiptools are initialized with a dictionary, called *transformation dictionary*, which contains all the plugin parameters. Typically a plugin has many parameters and the transformation dictionary need to have a specific structure. For this reason every plugin is equipped with a global attribute called `.empty_transformation_dictionary` which can be read without initializing the plugin. The value stored in this dictionary are the default parameters of the plugin, which can be used for generic tasks.

```
from bmiptools.transformation import Destriper

# get the default transformation dictionary
transformation_dictionary = Destriper.empty_transformation_dictionary
print(transformation_dictionary)
```

This system can be used also to set the parameters without the need to reproduce the transformation dictionary structure so that the plugin can be initialized with the desired setting easily.

```
# set a parameter in the transformation dictionary
transformation_dictionary['optimization_setting']['opt_bounding_box']['use_bounding_box
↪'] = False

# initialize the plugin
destriper = Destriper(transformation_dictionary)
```

The transformation dictionary has a common structure, whose explanation can be found in *General information about plugins* (together with general information about the plugins), while all the plugin-specific parameters are explained in the corresponding plugin documentation.

Note: Once that the plugin is initialized, the initialized plugin acquires a series of attributes having the *same name* of the parameter in the transformation dictionary. Therefore one can check the parameters value after the initialization simply by checking the corresponding class attributes.

```
# check the value of the parameter 'use_bounding_box'
print(destriper.use_bounding_box)
```

These attributes can also be used to change the parameters after the plugin initialization.

```
# set parameter value after initialization
destriper.use_bounding_box = True
print(destriper.use_bounding_box)
```

4.2 Plugin optimization

Many plugin can be optimized. The optimization can be run by calling the method `.fit` which is present in any plugin (but only when the plugin can be optimized, this method actually do something). This method takes always as input the stack on which the optimization have to be done (for the meaning of plugin optimization see *here*). The optimization of the plugin can be done with the following line of code

```
# load/create a stack on which the plugin is applied
# stack = ...

# fit the previously initialized plugin
destriper.fit(stack)
```

Attention: At the end of the optimization the result of the optimization (i.e. the parameters value found) are stored in the corresponding attributes of the plugin class, overwriting the initial value assigned during the optimization.

4.3 Apply a plugin

The application of the plugin can be done with the method `.transform`, which takes a stack input the stack on which the plugin is applied.

```
# in place application of a plugin
destriper.transform(stack)
```

It is important to know that the content of the *stack* is *overwritten* with the result of the plugin application, i.e. at the end of the transformation application, the content of the initial stack is updated with th result of the transformations. This is the default behavior, however this feature can be changed by setting `inplace=False`.

```
result = destriper.transform(stack,inplace=False)
```

In this case, *stack* is not overwritten and still contain the initial data. The transformation result is stored in the numpy array `result`.

Attention: If the field `auto_optimize = True` in the transformation dictionary of a plugin that can be optimized, when the `.transform` methods is called but the plugin was still not optimized (i.e. the `.fit` method was not called), the optimization is executed automatically (i.e. the `.fit` method is called internally before the application of the transformation).

4.4 Get plugin parameters

It is possible to get the transformation dictionary with the *actual* status of all the plugin parameters with a single command. this cna be done calling the `.get_transformation_dictionary`. This is particularly useful to get access to all the parameters of a plugin at the end of the optimization procedure.

```
current_transformation_dictionary = destriper.get_transformation_dictionary()
print(current_transformation_dictionary)
```

CHAPTER

FIVE

GUI USAGE

bmiptools is equipped also with a basic graphical interface. In order to use it, one need to have a python terminal installed on its PC. Here the one provided by Anaconda is used.

5.1 Run the GUI

To run the bmiptool GUI it open the Anaconda Prompt and activate the bmiptools environment. At this point one has to execute the `run_gui` module of bmiptools with the python interpreter using the `-m` option. More precisely, assuming bmiptools has been installed in the environment called `bmiptools_env`, the activation of the environment and the launch of the GUI can be done with the two following line

```
$ conda activate bmiptools_env
$ (bmiptools_env) python -m bmiptools.run_gui
```

The short clip below show how this can be done one a Windows 10 compute with anaconda already installed.

5.2 Load a stack

The clip below show how a stack of tiff images can be loaded from a folder. The loading operation terminates when `"Stack loaded!"` is printed on the terminal.

5.3 Create and initialize a pipeline

The creation and initialization of a pipeline with the bmiptool GUI is showed in the clip below. By clicking on the `create and initialize pipeline`, a windows where the pipeline can be created by adding the various plugins. In this window one has also to specify the working folder of the pipeline. To initialize each plugin click on the `setting` button and, once the plugin configuration windows opens, click on `ok`, after modifying the parameters if needed. When this operation is fully executed, `"Pipeline created and initialized!"` appears on the terminal.

Attention: In order to add and correctly initialize the plugins in the pipeline, always click on the setting `setting` and then click on `ok`, possibly after modifying the parameters.

5.4 Load a pipeline template

The creation and initialization of a pipeline can happens also ina different way: via a pipeline template. This is a json file containing all the pipeline parameters (see *here* to get more information about its structure). At the end of this operation `"Pipeline template loaded!"` is printed on the terminal.

5.5 Apply pipeline and previews

To apply the pipeline crated to the stack (running also the plugins optimizations if available) simply click on the button `Apply pipeline`. If a preview of the final result is needed, *before* to click on 'Apply pipeline' one needs to configure the preview. This can be done by clicking `Preview setting`, specify the preview setting and then apply the pipeline. Once that the pipeline has been applied on the stack, `Pipeline applied!` is printed on the terminal.

5.6 Save results

Once that pipeline has been applied to the stack, one can do two things: save the resulting stack and save the trained pipeline. To save stack simply click on `Save stack`: when the stack is saved in the selected folder `"Stack saved!"` is printed on the terminal. To save the pipeline click on `Save pipeline`: a .dill file will be produced in the pipeline folder selected, while the pipeline json will be updated with the parameters found during the optimization routine (if any). The clip below shows both these operations.

5.7 Load an existing pipeline

Once that a pipeline has been saved, a '.dill' file is created in the pipeline folder (plus eventually a folder called 'undillable', see *here* for more information). This file is the one containing all the pipeline parameters (possibly determined by the optimization procedure) and can be read by bmiptools to reproduce exactly the same pipeline later or on a different computer. the clip below show how to load a pipeline with the bmiptoos GUI. When the loading is terminated, `"Pipeline loaded!"` appears in the terminal window.

At this point, one can apply the pipeline simply by clicking on `Apply pipeline` button. In this case no optimization routines are executed, and the (already trained) plugins are simply applied to the stack.

5.8 Further reading

Tutorials:

- *Basic operations with stacks and pipelines*

CHAPTER

SIX

GENERAL INFORMATION ABOUT PLUGINS

Plugins are the object which in bmiptools allow to apply the image-processing transformations. They are designed so that certain operations can be executed in a standardized way, as showed in the *Advanced API usage* section. In particular, they can be alla initialized in the same way (via the *transformation dictionary*), they can be fitted all in the same way (if optimization is possible), applied to a stack in the same way, and finally the actual configuration of the plugin can be obtained all in the same way.

6.1 What is plugin optimization?

The word 'optimization' deserve some explanation in the context of bmiptools plugins. Plugins implement some image processing transformation, and many of them have some optimization procedure which need to be executed in order to produce the transformed image. Is that the meaning of the words 'plugin optimization', in the context of bmiptools? No. In bmiptools, the words 'plugin optimization' are referred to the automatic selection of the parameters (if any) of the image processing transformation.

An example may clarify better this point. An effective denoising technique is the so called total variational denoising. In this denoising technique, given a noisy image as input, the output is produced by finding the minimum of a suitable mathematical function (loss function) which depends on a series of parameters the user need to specify. No general analytical solution for this kind of minimization problem is possible, but it is possible to find algorithms that allow to find the minimum (or an output image close to it). Clearly, the minimum found depends not only on the input image, but also on the parameters specified in the loss function. From the point of view of the mathematical terminology, the minimization of the loss function is an optimization problem. By the way optimization of the Denoiser plugin in bmiptool (allowing to do also total variational denoising) *is not* referred to the solution of the minimization/optimization problem just explained. Rather 'plugin optimization' has to be understood as the mathematical procedure which selects the best parameter of the total variational denoising filter (in this case, this can be done by using the J-invariance criteria).

> **Attention:** The plugin optimization is done using (eventually part of) the stack in order to estimate the plugin parameters. Once that these parameters are found, the plugin are initialized with them and then applied to *all the slices with the same parameters*. In this sense the plugin optimization in bmiptools is **global** for a stack and is not done slice by

slice.

6.2 Transformation dictionary

All the plugins are equipped with a dictionary which contains all the parameters that the user may set. This dictionary, called *transformation dictionary* is the one that appears after the corresponding plugin name in the pipeline dictionary generated by `pipeline.Pipeline` when a pipeline is created (see *here* and *here*).

The transformation dictionary has a common structure for all the plugins having an optimization method, which is briefly described below.

- **auto_optimize**: The input is a boolean, i.e. `true` or `false` (in the json file, corresponding to `True` or `False` in python). When true, the auto-optimization routine of the plugin is executed, according to the content found in the `optimization_setting` field.
- **optimization_setting**: It contains a dictionary with all the setting associated to the optimization process (typically the definition of the parameter space, in which the optimal parameters are searched). Many times it makes sense to reduce the optimization time, it can be useful to restrict a the plugin optimization routine to a smaller region of the input stack, called *bounding box*. The bounding box is specified using the two objects below
  - **opt_bounding_box**: It contains a dictionary which specify the region in the YX-plane used for the plugin optimization routine. Typically is a dictionary with the following structure

```
{'use_bounding_box': True,
'y_limits_bbox': [None,None],
'x_limits_bbox': [None,None]
},
```

    The meaning should be evident.

    * `use_bounding_box`: if `True` the bounding box is used, otherwise not;
    * `y_limits_bbox`: list used to express the limits in the Y-direction;
    * `x_limits_bbox`: list used to express the limits in the X-direction.

    The convention used to express the Y/X-direction limits are discussed *here*.

  - **fit_step**: It is an integer number expressing the interval between two slices of the stack that are used during the optimization. If 1 all the stack is used, for n>1 only the slices having a distance of n on the 0-axis (i.e. the Z-direction). This parameter therefore determine the number of slices used during the optimization.

> **Attention:** When there is the need to restrict some internal mathematical procedure of the plugin to just a subregion of the stack, the `opt_bounding_box` and `fit_step` fields can be found also outside the `optimization_setting` section of the transformation dictionary.
>
> This is the case of the *Registrator* plugin, which has no plugin optimization in the sense discussed *above*. However this plugin still perform a proper mathematical optimization, in order to estimate the necessary registration parameters. This operation can be speed up if just a smaller portion of the stack in the YX-plane is used. For this reason the `opt_bounding_box` field is present in the transformation dictionary of this plugin.

In addition to the options discussed above the transformation dictionary of a plugin, after the `optimization_setting` field, contains all the plugin specific parameters. Below the general structure of a possible optimization plugin is sketched. Concrete examples of transformation plugins are shown in the documentation of plugin, where all the plugin specific parameters are explained.

```
{
'auto_optimize': True,                                  # enable/disable plugin optimization
'optimization_setting':{'opt_param1': ...
                        ...                             # plugin optimization specific␣
↪parameters
                        'opt_paramM': ...
                        'opt_bounding_box':{'use_bounding_box': True,
                                            'y_limits_bbox': [None,None],   # bounding␣
↪box parameters
                                            'x_limits_bbox': [None,None]
                                            },
                        'fit_step':10
                        },
'param1': ...
...                                                     # plugin specific parameters
'paramN': ...
}
```

6.2.1 Bounding box specification convention

For the definition of the bounding box a numpy compatible convention is used. In particular the fields `y_limits_bbox` and `x_limits_bbox` are two lists with just two elements corresponding to to the left and the right element of the numpy sliceing for the correspoing axis. More precisely, given

```
y_limits_bbox: [a,b],
x_limits_bbox: [c,d]
```

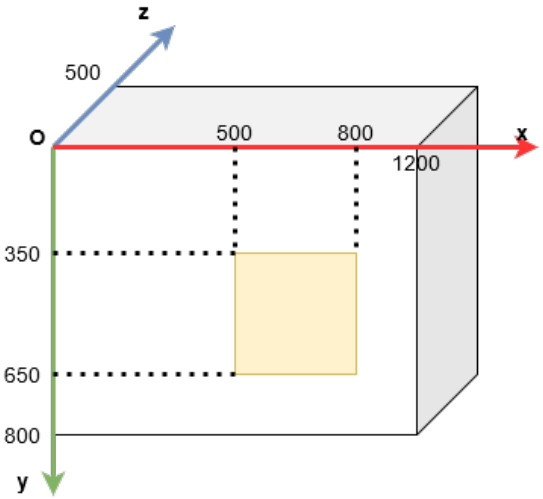
where `a`, `b`, `c`, and `d` are integer number or `None`, this corresponds to the following selection in the (at least) 3-dimensional array `stack` containing the stack

```
stack[:,a:b,c:d]
```

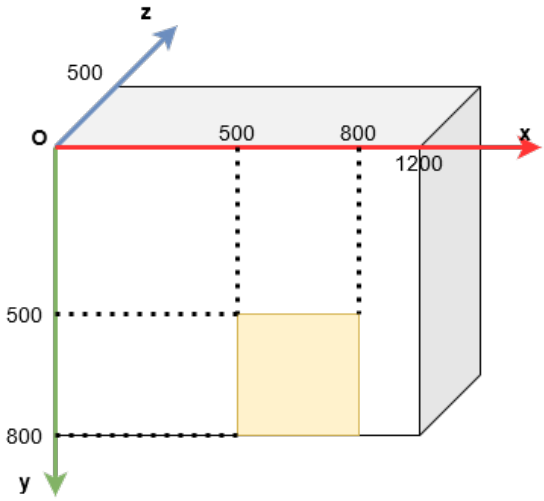
Keep in mind that for a numpy array `arr`, one has

```
arr[None:b] = arr[:b]
arr[a:None] = arr[a:]
arr[None:None] = arr[:] = arr
```

For the user which are not familiar with numpy, the examples below can be helpful. They show a stack of 500x800x1200 voxels and a bounding box on YX-plane of 300x300 in different places. Note that the origin of the reference frame is in the front-top-left corner of the stack.



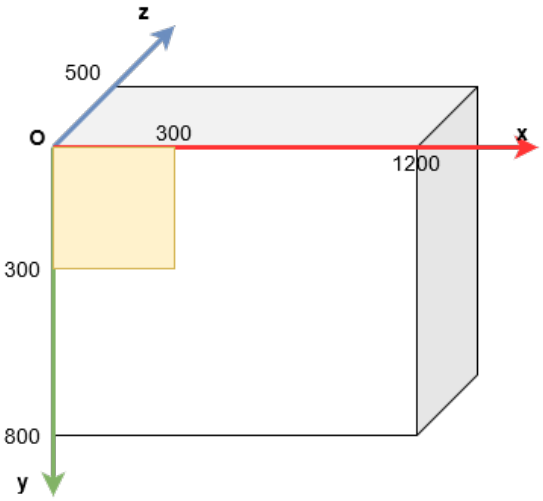

```
y_limits_bbox: [350,650]
x_limits_bbox: [500,800]
```




```
y_limits_bbox: [500,800]
x_limits_bbox: [500,800]
```

equivalent to

```
y_limits_bbox: [500,None]
x_limits_bbox: [500,800]
```




```
y_limits_bbox: [0,300]
x_limits_bbox: [0,300]
```

equivalent to

```
y_limits_bbox: [None,300]
x_limits_bbox: [None,300]
```

Attention: When the bounding box parameters are specified in the pipeline json, keep in mind the simple rules explained *here* about main the differences between the python notation and the json notation.

6.2.2 GUI Plugins setting

The transformation dictionary is not only useful for the python API. Indeed the transformation dictionary of each plugin turns out to be also the model used to construct the plugin GUI via GuiPI (see *GuiPI: automatic GUI generation for plugins*). As such the explanation of the parameters given for each transformation dictionary the plugins, is also the explanation of the possible setting the user may chose using the GUI. There is just on difference in the name of the parameter: in the python API the variables does not have blank space in their name, while in the corresponding GUI parameters every underscore is replaced with a blank space, e.g. `auto_optimize` in the Python API, becomes `auto optimize` in the bmiptools GUI.

6.3 Available plugins

The currently available plugins are:

- *Standardizer*
- *Histogram matcher*
- *Denoiser*
- *Denoiser DNN*
- *Destriper*
- *Flatter*
- *Registrator*
- *Affine*
- *Cropper*
- *Equalizer*

CHAPTER

SEVEN

STANDARDIZER

Standardizer in a nutshell.

1. Plugin for image dynamic standardization;
2. This plugin is multichannel;
3. Python API reference: `bmiptools.transformation.dynamics.standardizer.Standardizer`.

This plugin can be used to rescale and shift the image dynamics. It is a global transformation (the same for all the voxels of the stack) and has no practical effect on the visualization. Image readers which use relative color range would not show any difference with the original image, while the ones which use absolute color range may show some variation. Keep in mind that this variation is apparent, since it depends on how the image is read by the image reader software.

The Python API reference of the plugin is `bmiptools.transformation.dynamics.standardizer.Standardizer`.

7.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'standardization_type': 'mean/std'
 'standardization_mode': 'stack'
}
```

The plugin-specific parameters contained in this dictionary are:

- `standardization_type`: Option selecting standardization methods used by the plugin. The currently implemented methods are:
  - `-1/1`: all the values in the image are suitably rescaled between -1 and 1.
  - `0/1`: all the values of the image are suitably rescaled between 0 and 1.
  - `mean/std`: the image will have zero mean and standard deviation 1.
- `standardization_mode`: Option selecting how the standardization parameters are computed. The available modes are:
  - `slice-by-slice`: the standardization parameters are computed for each slice.
  - `stack`: the standardization parameters are computed considering the whole stack.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Standardizer`

7.2 Use case

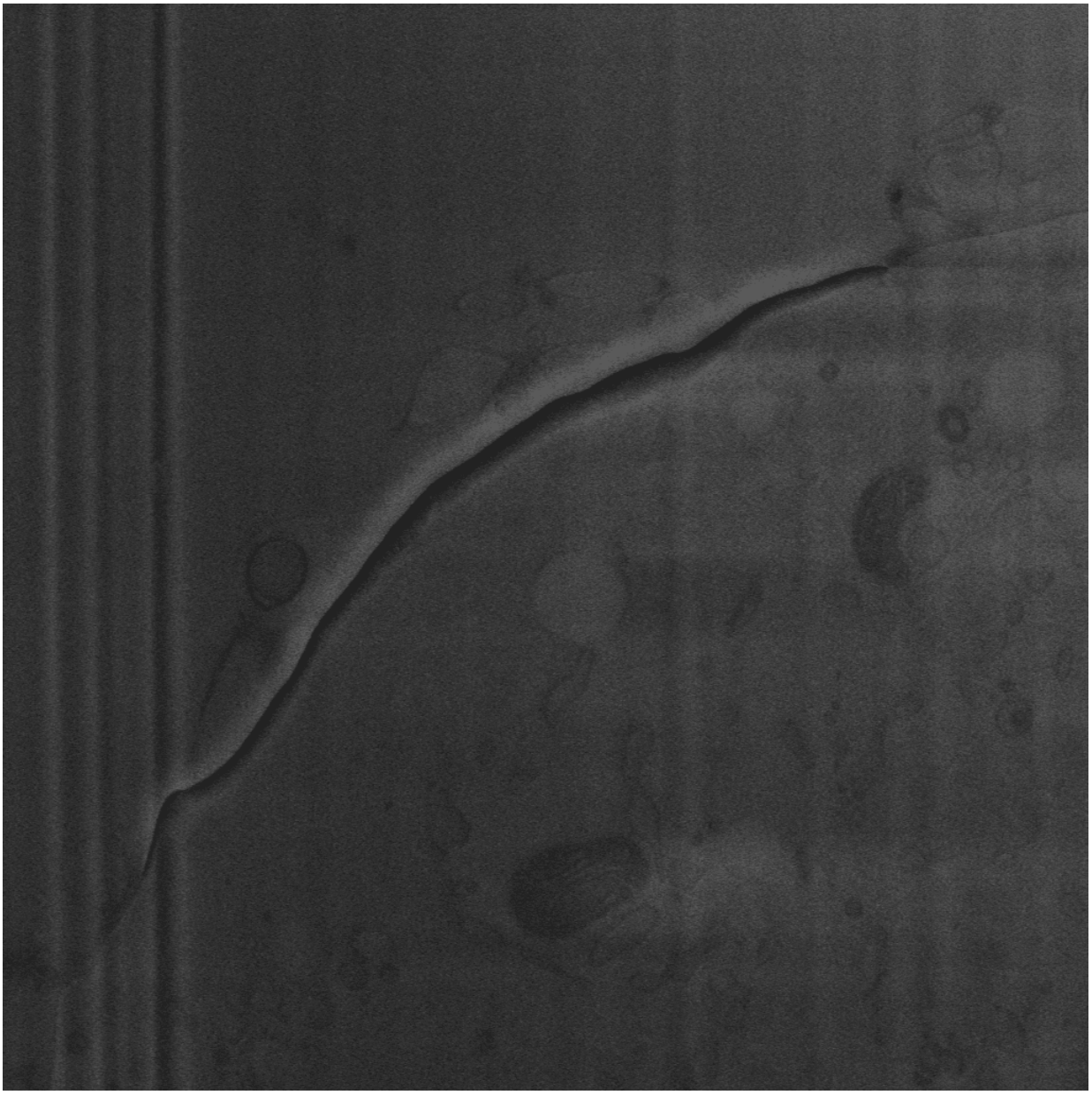
The typical use of this plugin is mainly technical. They are:

1. Increase the dynamics of the image;
2. Rescale the image value through a pipeline, useful when the next plugin need input values with a restricted dynamics (as is the case of man);
3. In case the input image is of type 'int', the `0/1` mode change the image type to 'float' in a way that is typically compatible with the common image readers.

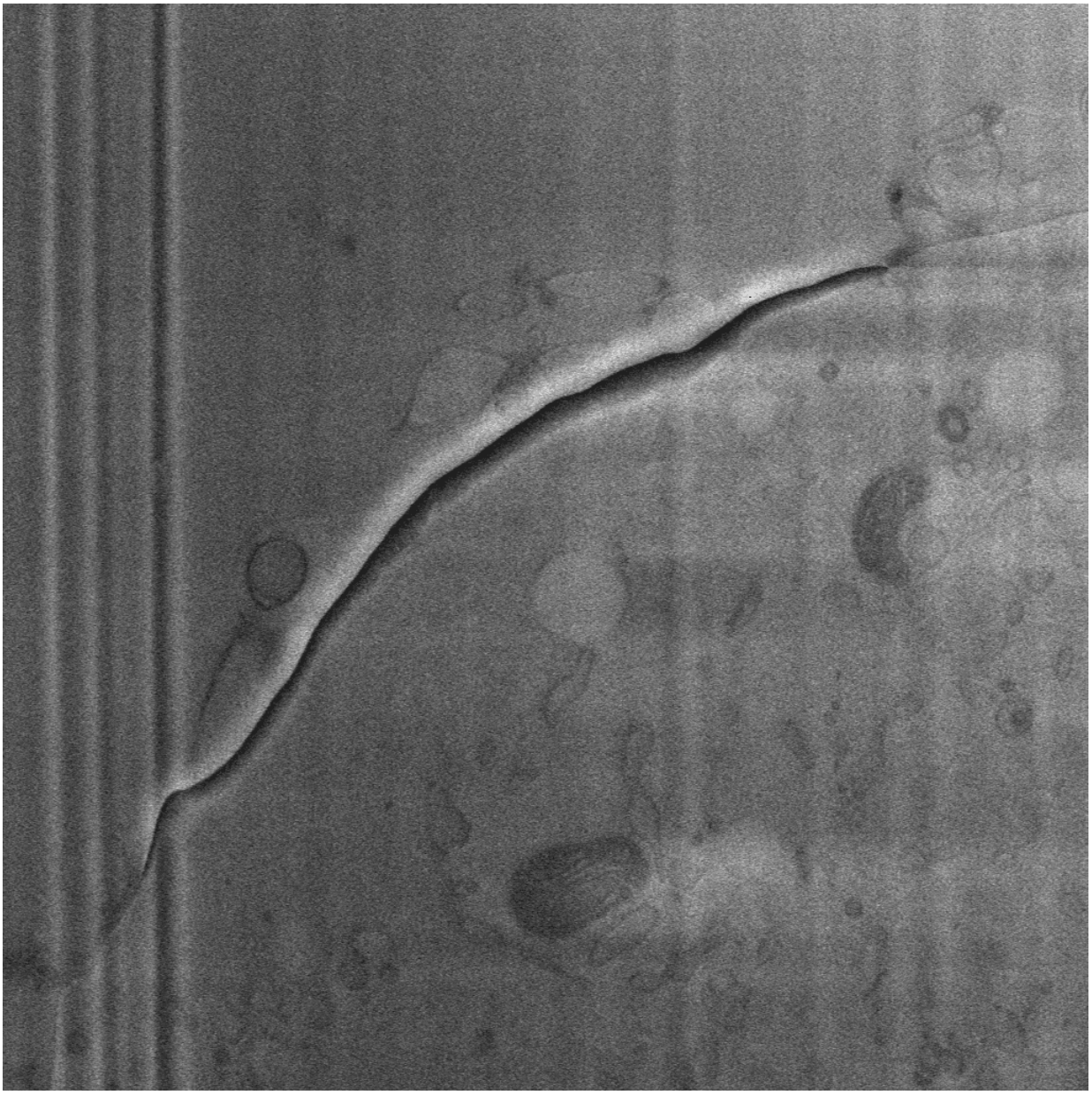
Attention: The effect of this plugin on a stack may depends on the image reader used to visualize the stack. Certain image readers implicitly standardize the images when they are reader: in this case the this plugin would not affect the visualization (despite the value of the pixels are changed in any case).

7.3 Application example

As example consider the slice of a stack of a biological sample obtained via cryo-FIB-SEM, where the brightness slowly increase moving from the left to the top-right of the image.



After the application of the Standardizer plugin with `standardization_type = '0/1'` and `standardization_mode = 'slice-by-slice'`, the result obtained is given below.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

7.4 Implementation details

Let $S(k, j, i)$ be a single-channel $K \times J \times I$ stack, and let

- $M = \max_{k,j,i} S(k, j, i)$ be the maximum of the whole stack,
- $m = \min_{k,j,i} S(k, j, i)$ be the minimum of the whole stack,
- $\mu = \frac{1}{KJI} \sum_{k=0}^{K-1} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} S(k, j, i)$ be the mean value of the stack,
- $\sigma = \sqrt{\frac{1}{KJI} \sum_{k=0}^{K-1} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} (S(k, j, i) - \mu)^2}$ be the standard deviation of the stack.
- $M_k = \max_{j,i} S(k, j, i)$ be the collection of all the maxima of each slice k,
- $m_k = \min_{j,i} S(k, j, i)$ be the collection of all the minima of each slice k,
- $\mu_k = \frac{1}{JI} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} S(k, j, i)$ be the collection of all the mean values of each slice k,
- $\sigma_k = \sqrt{\frac{1}{JI} \sum_{j=0}^{J-1} \sum_{i=0}^{I-1} (S(k, j, i) - \mu_k)^2}$ be the collection of all the standard deviation of each slice k.

Assume to use the plugin with `standardization_mode = 'stack'`. For the `-1/1` standardization type, the input stack $S(k, j, i)$ is transformed as follow

$$S(k, j, i) \to S_{output}(k, j, i) = 2\frac{S(k, j, i) - m}{M - m} - 1.$$

For the `0/1` standardization type, the input stack $S(k, j, i)$ is transformed as follow

$$S(k, j, i) \to S_{output}(k, j, i) = \frac{S(k, j, i) - m}{M - m}.$$

For the `mean/std` standardization mode, the input stack $S(k, j, i)$ is transformed as follow

$$S(k, j, i) \to S_{output}(k, j, i) = \frac{S(k, j, i) - \mu}{\sigma}.$$

When `standardization_mode = 'slice-by-slice'`, the formula above holds true except that rather than use the quantities m, M, μ, and σ computed for the whole stack, the slice dependent quantities m_k, M_k, μ_k, and σ_k are used instead. For multichannel stacks, the transformations above are applied for each channel independently.

CHAPTER

EIGHT

HISTOGRAM MATCHER

Histogram matcher in a nutshell.

1. Plugin for match the brightness of one slice of a Stack with the next one;
2. Do not use this plugin if meaningful global brightness variations along the Z-axis of a stack are expected;
3. This plugin is multichannel;
4. Python API reference: `bmiptools.transformation.dynamics.histogram_matcher.HistogramMatcher`.

This plugin can be used to match the histograms among the various slices of a stack. This makes the brightness level of the slice more uniform, reducing or eliminating the sudden brightness/contrast variation among one slices an the next one.

The Python API reference of the plugin is `bmiptools.transformation.dynamics.histogram_matcher.HistogramMatcher`.

8.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'reference_slice': 0
 }
```

The plugin-specific parameters contained in this dictionary are:

- `reference_slice`: position of the slice used as reference during the matching of the histograms.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `HistogramMatcher`

8.2 Use case

The typical uses of this plugin are:

1. Remove the sudden brightness variation along the Z-axis in a stack.
2. Homogenize the slices in a stack to easier the optimization of the other plugins in a pipeline.

Tip:

- Do not use this plugin on a stack where meaningful brightness variations slice after slice are expected. These variations
- A good idea is to use this plugin in the beginning of a pipeline (first or second plugin), in order to have an homogenized stack as input of the core plugins of a pipeline. In this way, it is more likely that the application of the core plugins (initialized with parameters both set by the user or found by some optimization procedure) produces similar effects for each slice of the stack.

8.3 Application example

As example consider the portion of a FIB-SEM stack of a biological sample, visualized as animated gif (saving mode available in the python API, see `bmiptools.stack.Stack.save_as_gif()`), in order to see how the overall brightness level suddenly change from one slice to the next.

Note the strong reduction of the overall brightness happening at approximately the half of the clip. After histogram matching, the sudden changes in brightness disappear.

.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

8.4 Implementation details

The core operation of this plugin is the matching of the histogram among two images. This is done by using the skimage function `skimage.exposure.match_histogram` (see here for further details). Given two slices of a stack a and b, let $HM(a, b)$ be the function matching the histogram of the image b with the histogram of the image a (used as reference). Then, given the input $K \times J \times I$ stack $S(k, j, i)$ and a reference slice in position k_0, consider k-th slice $S[k](x, y)$. The `HistogramMatcher` plugin perform the following operations:

1. Starting from $k = k_0 + 1$, then

$$S[k](j, i) \rightarrow S_{output}[k](i, j) = HM(S[k-1](j, i), S[k](j, i)).$$

   This operation is repeated *increasing* k by 1 till $k = K - 1$ is reached.

2. Going back to $k = k_0 - 1$

$$S[k](j,i) \rightarrow S_{output}[k](i,j) = HM(S[k+1](j,i), S[k](j,i)).$$

This operation is repeated *decreasing* k by 1 till $k = 0$ is reached.

For multichannel stack, the transformations above are applied for each channel independently.

8.5 Further details

Websites:

- wikipedia
- "Histogram Matching, Earth Engine by Example" - Noel Gorelik
- "Histogram Matching" - Paul Bourke

CHAPTER

NINE

DENOISER

Denoiser in a nutshell.

1. Plugin to denoise with classical methods the slices of a stack;
2. This plugin is multichannel;
3. This plugin can be optimized on a stack;
4. Python API reference: `bmiptools.transformation.restoration.denoiser.Denoiser`.

This plugin can be used to reduce the noise level of the slices using classical denoising techniques. In particular, the following techniques are available:

- *wavelet based denoising*;
- *total variation denoising*, both the Chambolle algorithm and the Split-Bregman based algorithm;
- *bilateral filter*;
- *non-local means*.

The optimization routine which can be used to select the best filter and parameter combination is based on the principle of J-invariance denoising. These denoising algorithms are essentially 2D, therefore they are applied slice-wise to the stack.

The Python API reference of the plugin is `bmiptools.transformation.restoration.denoiser.Denoiser`.

9.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'auto_optimize': True,
'optimization_setting': {'tested_filters_list': ['wavelet','tv_chambolle','nl_means'],
                         'wavelet': {'level_range': [1,9,1],
                                     'wavelet_family': 'db',
                                     'mode_range': ['soft','hard'],
                                     'method_range': ['BayesShrink','VisuShrink']
                                     },
                         'tv_chambolle': {'weights_tvch_range':[1e-5,1,100]
                                          },
                         'tv_bregman': {'weights_tvbr_range':[1e-5,1,100],
                                        'isotropic_range': [False,True]
```

(continues on next page)

(continued from previous page)

```
                                    },
                        'bilateral': {'sigma_color_range': [0.5,1,5],
                                      'sigma_spatial_range': [1,30,2]
                                     },
                        'nl_means': {'patch_size_range': [5,100,5],
                                     'patch_distance_range': [5,100,5],
                                     'h_relative_range': [0.1,1.2,15]
                                    },
                        'opt_bounding_box': {'use_bounding_box': True,
                                             'y_limits_bbox': [-500,None],
                                             'x_limits_bbox': [500,1500]
                                             },
                        'fit_step':10
                        },
'filter_to_use': 'tv_chambolle',
'filter_params': [['weight', 0.2]],
}
```

The optimization-related plugin-specific parameters contained in the `optimization_setting` field of this dictionary are:

- `tested_filter_list`: contains the list of denoiser that are compared among each other during the optimization. The currently available denoiser are
  - `wavelet`, for wavelet based denoising;
  - `tv_chambolle`, for the Chambolle total variational denoising;
  - `tv_bregman`, for the split-Bregman total variational denoising;
  - `bilateral`, for bilateral filter;
  - `nl_means`, for non-local mean denoising.
- `wavelet`: contains a dictionary for the definition of the parameter space used to find the best parameter combination for the wavelet based denoiser (see *below*). The key of this dictionary are:
  - `level_range`, it is a list of positive integer numbers specifying range of the possible decomposition levels used by the filter for wavelet decomposition. This list should look as follow $[l, L, \delta l]$, where l is the smallest level used in the parameter search, L is the largest level used in the parameter search, and δl is the step. This list produces the following decomposition levels for the parameter search: $l, l + \delta l, l + 2\delta l, ..., L - 2\delta l, L - \delta l(, L)$, where the last element is present or not depending if $L - l$ is an integer multiple of δl.
  - `wavelet_family`, name of a discrete wavelet family of the *PyWavelet* library to search only among the wavelets of this family during the optimization (see here for the list of available wavelet families).
  - `mode_range`, it is a list containing the possible thresholding mode used in the wavelet filter to suppress the coefficients containing most of the noise information. The possible values in this list are `soft` and `hard`.
  - `method_range`, it is a list containing the possible methods used in the wavelet filter to define the threshold below which suppress the coefficients containing most of the noise information. The possible values in this list are `BayesShrink` and `VisuShrink`.
- `tv_chambolle`: contains a dictionary for the definition of the parameter space used to find the best parameter combination for the Chambolle total variational denoiser (see *below*). The key of this dictionary are:
  - `weights_tvch_range`, it is a list containing the values defining the parameter space for the 'weight' parameter of the Chambolle total variation denoiser. This list should have 3 numbers, i.e. $[w, W, \delta w]$, generating the range $w, w + \delta w, w + 2\delta w, ..., W - 2\delta w, W - \delta w(, W)$.

- `tv_bregman`: contains a dictionary for the definition of the parameter space used to find the best parameter combination for the split-Bregman total variational denoiser (see *below*). The key of this dictionary are:
  - `weights_tvbr_range`, it is a list containing the values defining the parameter space for the 'weight' parameter of the split-Bregman total variational denoiser. This list should have 3 numbers, i.e. $[w, W, \delta w]$, generating the range $w, w + \delta w, w + 2\delta w, ..., W - 2\delta w, W - \delta w(, W)$.
  - `isotropic_range`: it is a list containing the possible *boolean* value for the 'isotropic' parameter of the split-Bregman total variational denoiser, i.e. the parameter selecting the *kind* of optimization problem solved with the split-Bregman method.
- `bilateral`: contains a dictionary for the definition of the parameter space used to find the best parameter combination for the split-Bregman total variational denoiser (see *below*). The key of this dictionary are:
  - `sigma_color_range` it is a list containing the values defining the parameter space for the 'sigma_color' parameter of the bilateral filter. This list should have 3 numbers, i.e. $[\sigma_c, \Sigma_c, \delta\sigma_c]$, generating the range $\sigma_c, \sigma_c + \delta\sigma_c, \sigma_c + 2\delta\sigma_c, ..., \Sigma_c - 2\delta\sigma_c, \Sigma_c - \delta\sigma_c(, \Sigma_c)$.
  - `sigma_spatial_range` it is a list containing the values defining the parameter space for the the 'sigma_spatial' parameter of the bilateral filter. This list should have 3 numbers, i.e. $[\sigma_s, \Sigma_s, \delta\sigma_s]$, generating the range $\sigma_s, \sigma_s + \delta\sigma_s, \sigma_s + 2\delta\sigma_s, ..., \Sigma_s - 2\delta\sigma_s, \Sigma_s - \delta\sigma_s(, \Sigma_s)$.
- `nl_means`: contains a dictionary for the definition of the parameter space used to find the best parameter combination for the non-local mean denoiser (see *below*). The key of this dictionary are:
  - `patch_size_range`, it is a list containing the values defining the parameter space for the 'patch_size' parameter of the non-local mean denoiser. This list should have 3 numbers, i.e. $[s, S, \delta s]$, generating the range $s, s + \delta s, s + 2\delta s, ..., S - 2\delta s, S - \delta s(, S)$.
  - `patch_distance_range`, it is a list containing the values defining the parameter space for the 'patch_distance' parameter of the non-local mean denoiser. This list should have 3 numbers, i.e. $[d, D, \delta d]$, generating the range $d, d + \delta d, d + 2\delta d, ..., D - 2\delta d, D - \delta d(, D)$.
  - `h_relative_range`, it is a list containing the values defining the parameter space for the 'h_relative' from which the 'h' parameter of the non-local mean denoiser is computed. This list should have 3 numbers, i.e. $[h_r, H_r, \delta h_r]$, generating the range $h_r, h_r + \delta h_r, h_r + 2\delta h_r, ..., H_r - 2\delta h_r, H_r - \delta h_(, H_r)$.

The plugin-specific parameters contained in this dictionary are:

- `filter_to_use`: contain the name of the denoiser that is used if the optimization routine is not used (i.e. when `auto_optimize = False`). The currently available denoiser are
  - `wavelet`, for wavelet based denoising;
  - `tv_chambolle`, for the Chambolle total variational denoising;
  - `tv_bregman`, for the split-Bregman total variational denoising;
  - `bilateral`, for bilateral filter;
  - `nl_means`, for non-local mean denoising.
- `filter_params`: list whose elements are the denoiser parameter. Each denoiser parameter have to be specified with a list of two elements: the parameter name and the parameter value. For example, to initialize the split-Bregman total variational denoiser with `weight = 0.1`, and `isotropic=False`, the following list have to be used

```
[['weight',0.1],
 ['isotropic', False]]
```

  For a list of the parameters for each denoiser see the *Implementation details section*, where each of these denoiser is briefly explained and the reference to their algorithmic implementation is given, which is where the name of all the parameters can be found.

When `auto-optimize = True` the plugin-specific parameters above are ignored, since the one selected by the optimization procedure are used. Finally, the meaning of the remaining parameters can be found in *General information#Transfomation dictionary*.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Denoiser`.

9.2 Use case

The typical use of this plugin are:

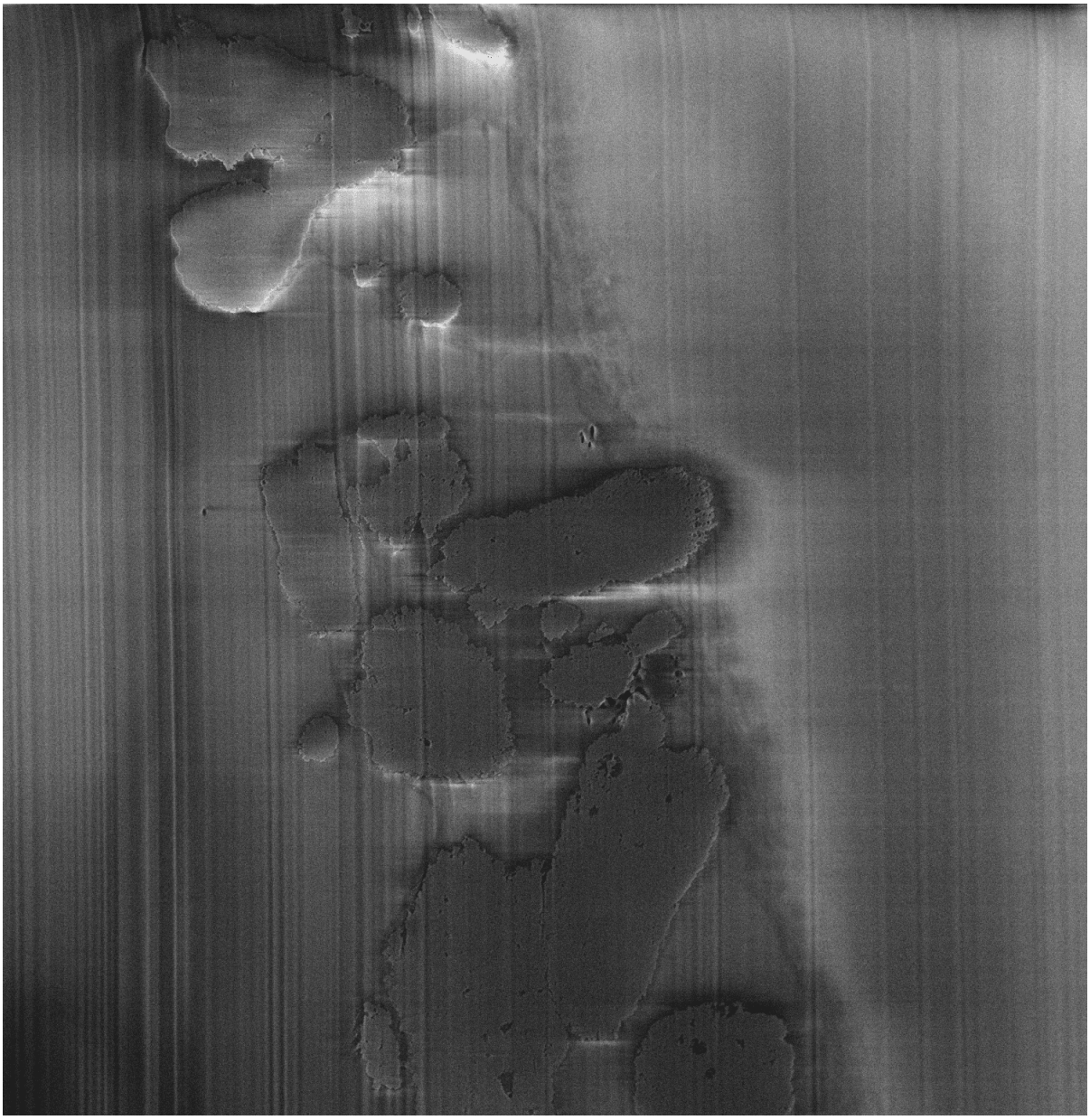
1. Reduce noise level in the input stack.

Tip: From the practical point of view, the following empirical findings

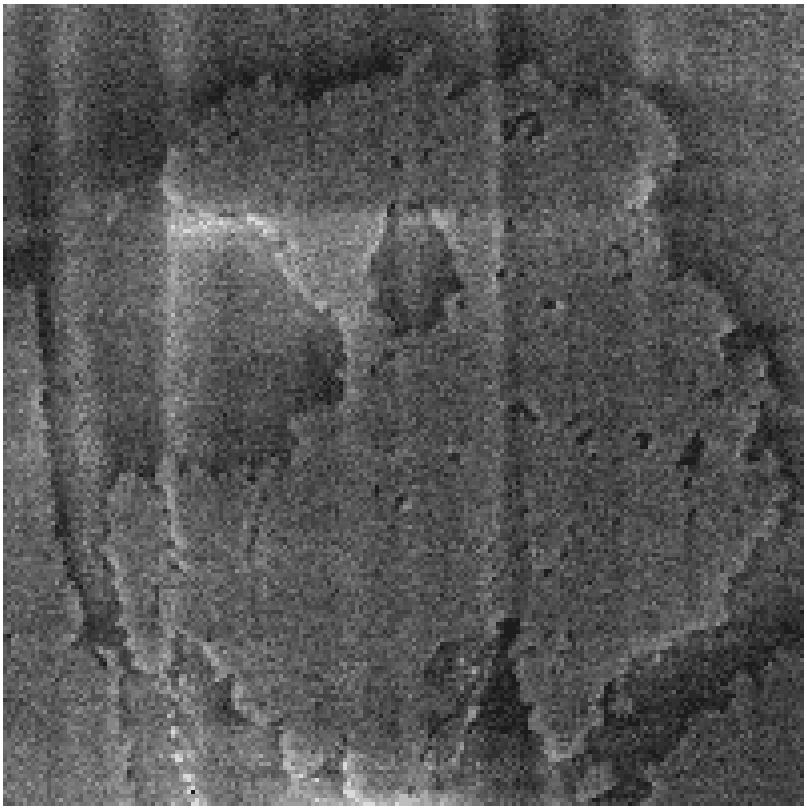
1. `bilateral` should not be added in the list: most of the time is considered as the best filter despite after visual inspection it is not so.
2. The parameter search for `nl_means` can be very time consuming, expecially for big images: consider that when it is added to the list.
3. In general when the optimization is used, the use of a bounding box and a suitable `fit_step` value is highly recommended in order to drastically reduce the optimization time. The noise properties most of the times are the same or very similar in any point of the image. Therefore use the bounding box to select a small region of the stack which is sufficiently representing (in terms of "image variability") the stack should not affect the denoising performance. Similarly, the noise most of the time does not very to much along the Z-axis. Therefore using a suitable value for `fit_step` to reduce the number of slices considered in the optimization process, can reduce the optimization times.

9.3 Application example

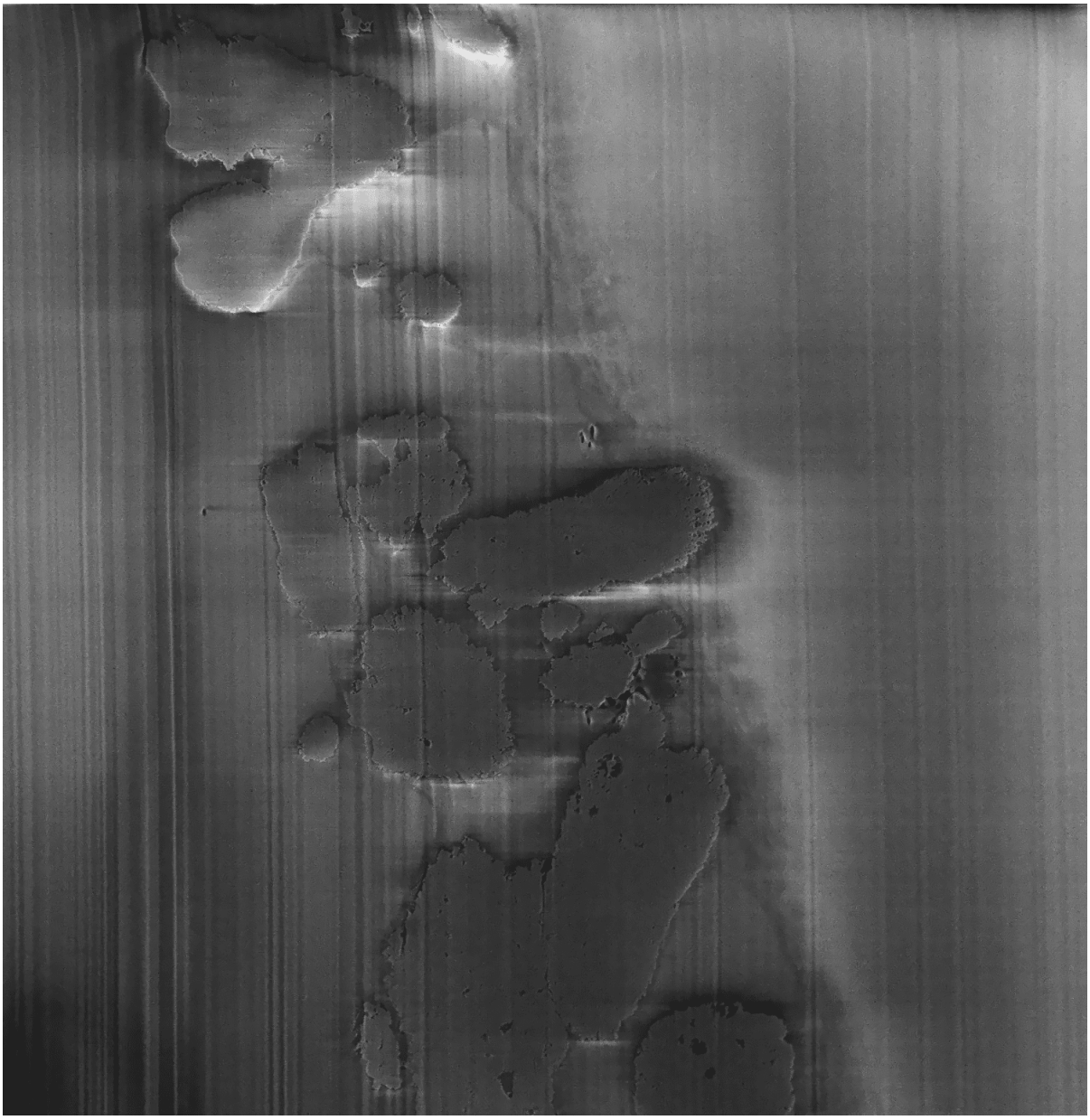
As example consider the slice of a stack of a biological sample obtained via SEM, where the noise is clearly present.



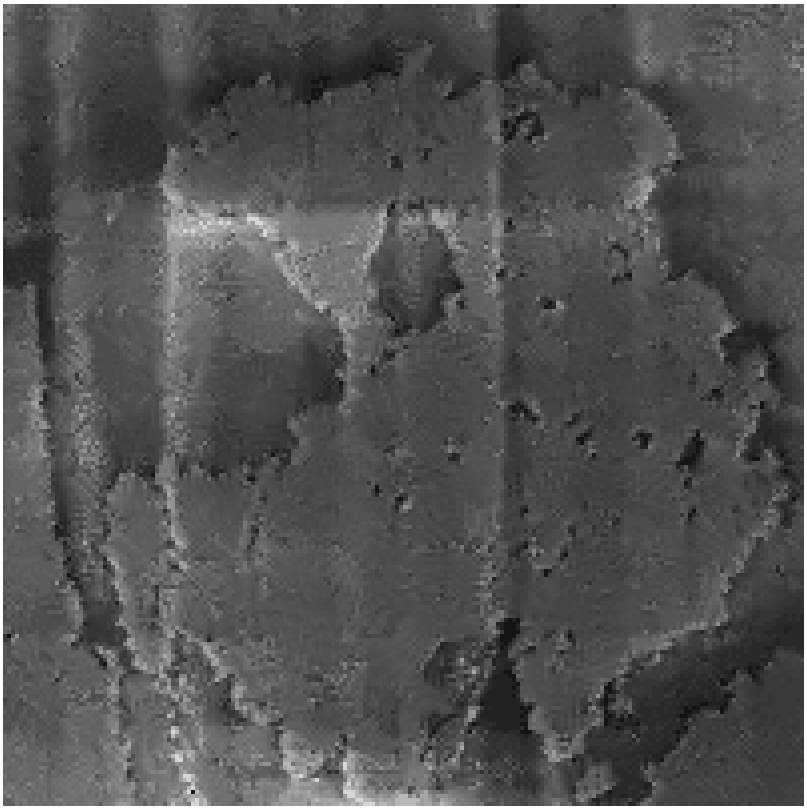
A zoomed part of the center-top/right part of the slice can be found below. One can clearly see some complex structures under the vertical stripes.



Applying the denoiser plugin with default setting, except for the use of the bounding box, which was defined in the central part of the image, the best denoiser with the best parameters has been selected. By applying it, the result one obtains is the following.



Zooming-in in the same place, one can see that the noise level on the image is reduced.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

9.4 Implementation details

The working principle of the different denoisers available in this plugin, are explained in the subsections below. There the meaning of the main parameters are clarified, the reference to the library used is given, where the user can find all parameters of a given filter. Finally the principles behind optimization routine used in the plugin is given.

In case of stack with multiple channels, the Denoiser is applied independently to each channel.

9.4.1 Wavelet denoising

This plugin use the skimage implementation of the wavelet denoiser, where all the parameter that can be used in the `filter_param` field, when the optimization routine is not used, can be found.

The wavelet denoiser of an image is essentially composed by 3 steps [Donoho1994]. First a multilevel wavelet decomposition using a certain *wavelet* w and up a to certain *decomposition level* l. This means to apply the (single-level) 2D wavelet transform with wavelet w iteratively for l times to the input image. Each iterations produces 4 different images with half of the size of the input image (called *subbands*), which are typically labelled with the letters LL, LH, HL, HH. The LL subband is the one used as input of the next iterations. This operation will be denoted with the symbol $WD2d[w, l]$.

The "pixel values" of all the subbands obtained for all the level, are called *wavelet coefficients*. The next step in the wavelet denoising consist in the shrinkage of wavelet coefficients below a certain threshold. Indeed in principle the wavelet coefficients would be higher in those points where the input image at a given scale correlate with the chosen wavelet (at that corresponding scale). The noise, being random, should not correlate particularly well with any wavelet at any level. Therefore, the noise contribution to the wavelet coefficients would be small and more or less constant for all the subbands. Thus it can be suppressed by eliminating the wavelet coefficients below a certain threshold. Given a certain threshold δ, two are the popular *thresholding mode*:

- *hard thresholding* mode, which uses the following thresholding function

$$y_{hard}(x) = \begin{cases} x & \text{if } |x| > \delta \\ 0 & \text{otherwise.} \end{cases}$$

- *soft thresholding* mode, which uses the following thresholding function

$$y_{soft}(x) = \begin{cases} \text{sign}(x)|x - \delta| & \text{if } |x| > \delta \\ 0 & \text{otherwise.} \end{cases}$$

The threshold value δ can be selected according to various criteria. In the current implementation of the wavelet denoiser used here, two are the one available:

- *ViSu shrink*, which employs an universal threshold for all the subbands, equal to

$$\delta = \sigma\sqrt{2\log K}$$

  where K is the total number of pixel of the input image, while σ is the standard deviation of the noise in the image. Typically, σ is estimated from the image from the median of the absolute value of the wavelet coefficients (MAV) of the HH subband of the highest decomposition level, via the formula

$$\sigma = \frac{MAV}{0.6745} \tag{9.1}$$

- *Bayes shrink*, which employs a subband dependent threshold, equal for the level l to

$$\delta_l = \frac{\sigma_l^2}{\sqrt{\max(\sigma_G^2 - \sigma_l^2, 0)}}$$

  where σ_l is the variance of the noise in the image estimated using (9.1) but using the HH subband of the level l and not only the highest, while

$$\sigma_G^2 = \frac{1}{M}\sum_{j,i} c_{j,i}^2$$

  where $c_{j,i}$ are the wavelet coefficients at the level l, and M is the number of those coefficients. With the 'Bayes shrink' each decomposition level is filtered in a different manner, that is why this is an adaptive method for the threshold estimation [Chang2000] [Gupta2015].

The shrinking operation described here will be denoted with the symbol $Sh[\delta, y]$, where δ is the threshold selected according to one of the possible criteria, and y is one of the two possible thesholding function.

The last step is the reconstruction of the filtered image, by using the inverse 2D Wavelet decomposition using the same wavelet w and decomposition level l using in the beginning, operation denoted by $WD2d^{-1}[w, l]$.

Therefore for the wavelet denoise, given a stack $S(k, j, i)$ each slice $S[k](j, i)$ is filtered as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = WD2d^{-1}[w,l](Sh[\delta, y](WD2d[w,l](S[k](j,i)))).$$

9.4.2 TV denoising

This plugin use the skimage implementation bot for the Chambolle and the split-Bregman of the total variational denoiser, where all the parameter that can be used in the `filter_param` field, when the optimization routine is not used, can be found.

Total variational denoising is a techniques based the solution of a suitable optimization problem which is extremely successful in reducing the noise preserving edges in the input image. In particular, in this technique for a given input noisy image I_0, the denoised image I_{output} is the solution of the following problem

$$I_{output} = \text{argmin}_I \mathcal{L}(I; I_0)$$

where the loss function is

$$\mathcal{L}[w](I; I_0) = \frac{1}{2} \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} (I(j,i) - I_0(j,i))^2 + w \sqrt{\sum_{j'=0}^{N-1} \sum_{i'=0}^{M-1} (\nabla_j I(j',i')^2 + \nabla_i I(j',i')^2)}$$

where I is an image, I_0 is the initial image, and w is the *weight* parameter of the loss. In the loss, the gradients of an image have to be understood inn discrete sense. The meaning of this loss is not difficult to understand:

1. the first term is simply the mean square error between the image I and the initial image I_0, which encode the simple requirement that the deionised image is not to different from the initial one;
2. the second term simply require that the variations between one pixel and its next along the X- and Y-axis is small, which is what one should expect for an image without noise, since the variation are slow, while the noise vary a lot from one pixel to the next.

Keeping this in mind, one can understand that the *weight* parameter w determines how much one requirement is important with respect to the other. The *Chambolle* [Chambolle2004] and *split-Bregman in its isotropic version* [Goldstein2009] [Bush2011] are different algorithms which try to solve this problem.

Note: For the *anisotrpic split-Bregman* [Goldstein2009] [Bush2011], the second term of the loss function changes in

$$\sum_{j'=0}^{N-1} \sum_{i'=0}^{M-1} (|\nabla_j I(j',i')| + |\nabla_i I(j',i')|) \,.$$

Therefore, for the total variational denoiser, given a stack $S(k,j,i)$ each slice $S[k](j,i)$ is filtered as follow

$$S[k](j,i) \to S_{output}[k](j,i) = \text{argmin}_S \mathcal{L}[w](S; S[k](j,i)).$$

9.4.3 Bilateral filter

This plugin use the skimage implementation of the bilateral filter, where all the parameter that can be used in the `filter_param` field, when the optimization routine is not used, can be found.

This filter simply perform the convolution of the input image with a filter having *input-dependent* kernel

$$g[\sigma_s, \sigma_c](j,i,j',i',I) = \frac{1}{C_{j,i}} \exp\left(-\frac{(j-j')^2 + (i-i')^2}{2\sigma_s^2} - \frac{(I(j,i) - I(j',i'))^2}{2\sigma_c^2} \right)$$

where

$$C_{j,i} = \sum_{j'} \sum_{i'} \exp\left(-\frac{(j-j')^2 + (i-i')^2}{2\sigma_s^2} - \frac{(I(j,i) - I(j',i'))^2}{2\sigma_c^2} \right)$$

play the role of a pixel-dependent normalization constant. When convolved with the input image this filter perform a gaussian smoothing both in the coordinate space (with spatial standard deviation σ_s) and in the color space (with color standard deviation σ_c) [Paris2009].

Therefore, the denoising with spatial filter consist in the following. Given a stack $S(k, j, i)$ each slice $S[k](j, i)$ is filtered as follow

$$S[k](j,i) \rightarrow S_{output}[k](j',i') = \sum_{j',i'} g[\sigma_s, \sigma_c](j,i,j',i',S[k](j',i'))S[k](j',i')$$

9.4.4 Non-local mean denoising

This plugin use the skimage implementation of the non-local mean denoiser, where all the parameter that can be used in the `filter_param` field, when the optimization routine is not used, can be found.

This denoising technique perform the noise reduction in an image, by using similar regions in the input image for the computation of the denoised one [Buades2011]. For a given image $I(j, i)$, the

$$NL[f,h](I(j,i)) = \frac{1}{C(j,i)} \sum_{j',i'} \exp\left(-\frac{\max(d(j,i;j'i')^2 - 2\sigma^2, 0)}{h^2}\right)$$

where $sigma$ is the standard deviation of the noise in the image, h is the filtering parameters, since it determine how much the noise is suppressed. $d(j, i; j'i')$ is the distance measure used to measure the similarity among two pixels, which is defined to be equal to

$$d(j,i;j'i') = \frac{1}{(2f+1)^2} \sum_{(a,b)\in B((0,0),f)} \left(I(j+a,i+b) - I(j'+a,i'+b)\right)^2$$

with $B((j, i), f)$ denotes the $(2f + 1) \times (2f + 1)$ square patch centered in the pixel (j, i). From this definition one can see that the similarity of two pixel is based not only on its value, but also on the geometrical (and color) information in its surrounding. This implement the idea of using similar regions in an image to compute the denoised value of a given pixel. Therefore, the non-local mean denoising for a given pixel can be understood as a weighted mean of all the pixels in the image, where the weights depends on the similarities between pixels. From the formula above one can see that regions, having distance $d < 2\sigma$, have the maximum weight equal to 1, while as far as the distance is 2 times bigger that the noise standard deviation of the image the weights start to decrease. In bmiptool, the h paramter is not given directly. Instead an $h_{relative}$ parameter is used, from which h is computed with the simple formula below

$$h = \sigma h_{relative}$$

where σ is the estimated standard deviation of the noise present on the input image.

Note: The originally proposed method has been later improved to increase the computational efficieny [Darbon2008]. This improvement was reached with a little but clever modification of the weight function, which makes the computation of the new weights independent on the patch size.

Therefore, for the non-local mean denoiser, given a stack $S(k, j, i)$ each slice $S[k](j, i)$ is filtered as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = NL[f,h](S[k](j,i)).$$

9.4.5 Optimization details

The optimization routine of the denoiser plugin is based on the principe of J-invariance [Batson2019]. In a nutshell, given a certain denoiser f depending on a set of parameters $\alpha_1, \alpha_2, \cdots$ the idea is to find the best parameters by minimizing the following loss

$$\mathcal{L}(\alpha_1, \alpha_2, \cdots) = \frac{1}{N} \sum_{(i,j)} \| f[\alpha_1, \alpha_2, \cdots](I(j,i)) - I(j,i) \|_2^2$$

where I is the noisy image, N is the total number of pixels, and the sum is over the pixels of the image. This loss simple say that the best parameters are the one for which the mean square error between the filtered image and the initial noisy image is smaller. It can be proved, that finding the minimum of this loss (i.e. find the best parameters) is equivalent in finding the minimum of

$$\frac{1}{N} \sum_{(i,j)} \| f[\alpha_1, \alpha_2, \cdots](I(j,i)) - I_0(j,i) \|_2^2$$

where I_0 is the true image without noise, *provided that the filter f is J-invariant*. J-invariant means that the output of the filter in the pixel (j,i) does not depend on a set of pixels J containing (j,i) itself. This means that for a J-invariant filter, the minimization of $\mathcal{L}(\alpha_1, \alpha_2, \cdots)$ allows to find the parameters of the filter such that its output is as closed as possible (in terms of the MSE) to the true image without noise. Given a classical filter, a J-invariant version of it can be obtained by masking: for teh computation of the filter output in the pixel (j,i), the region J around each pixel (j,i) of the input is masked with the mean value of the pixels around this region, leaving the rest of the input image unchanged. By following this procedure, it is possible to obtain the J-invariant version of a given filter f. Empirically, it has been observed that the optimal parameters for the J-invariant version of a filter are very close to the optimal parameter of the filter, most of the times.

9.5 Further reading

Articles:

CHAPTER

TEN

DENOISER DNN

Denoiser DNN in a nutshell.

1. Plugin to crop region of a stack;
2. This plugin is **not** multichannel;
3. This plugin can be optimized on a stack;
4. Python API reference: `bmiptools.transformation.restoration.denoiser.DenoiserDNN`.

This plugin can be used to reduce the noise level on the slices of a stack by using Deep Neural Network based techniques. In particular, in this plugin 2d and 3d Noise2Void denoiser are available.

The Python API reference of the plugin is `bmiptools.transformation.restoration.denoiser.DenoiserDNN`.

10.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'auto_optimize': True,
'optimization_setting': {'tested_filters_list': ['n2v_2d'],
                         'n2v_2d': {'unet_kern_size_2d_list': [3,5,7],
                                    'train_batch_size_2d_list': [128],
                                    'n2v_patch_shape_2d_list': [64],
                                    'train_epochs_2d_list': [30],
                                    'train_loss_2d_list': ['mse','mae'],
                                    'n2v_manipulator_2d_list': ['uniform_withCP','normal_
↪withoutCP',
                                                                'normal_additive',
↪'normal_fitted'],
                                    'n2v_neighborhood_radius_2d_list':[5,10,15,20]
                                    },
                         'n2v_3d': {'unet_kern_size_3d_list': [3,5,7],
                                    'train_batch_size_3d_list': [128],
                                    'n2v_patch_shape_3d_list': [(32,64,64)],
                                    'train_epochs_3d_list': [30],
                                    'train_loss_3d_list': ['mse','mae'],
                                    'n2v_manipulator_3d_list': ['uniform_withCP','normal_
↪withoutCP',
                                                                'normal_additive',
```

(continues on next page)

(continued from previous page)

```
↪'normal_fitted'],
                                    'n2v_neighborhood_radius_3d_list': [5,10,15,20]
                                    },
                          'opt_bounding_box': {'use_bounding_box': True,
                                               'y_limits_bbox': [-500,None],
                                               'x_limits_bbox': [500,1500]
                                               },
                          'fit_step': 10
                          },
'filter_to_use': 'n2v_2d',
'filter_params': None,
'trained_n2v_setting': {'use_trained_n2v_model': False,
                        'path_to_trained_n2v_model': '',
                        'save_trained_n2v_model': False,
                        'saving_path': ''}
}
```

The optimization-related plugin-specific parameters contained in the `optimization_setting` field of this dictionary are:

- `tested_filter_list`: contains the list of denoiser that are compared among each other during the optimization. The currently available denoiser are:
  - `'n2v_2d'`, for 2d Noise2Void denoiser;
  - `'n2v_3d'`, for 3d Noise2Void denoiser.
- `n2v_2d`: contains a dictionary which is used to define the parameter space used for the optimization of the 2d Noise2Void denoiser. It contains the following keys:
  - `unet_kern_size_2d_list`, contains a list of all the possible kernel sizes of the convolution layers used in the 2d Noise2Void which are tested during the hyperparameters optimization routine. The kernel size can be equal only to 3,5 or 7. Putting other numbers in this list would give rise to errors.
  - `train_batch_size_2d_list`, contains a list of all the possible batch sizes used for the training of the 2d Noise2Void model which are tested during the hyperparameters optimization routine.
  - `n2v_patch_shape_2d_list`, contains a list of the shapes of all the possible patches used in the Noise2Void which are tested during the hyperparameters optimization routine. The shape can be specified as a single integer number, meaning that a square patch of that size is used, or a usual tuple according the usual numpy convention.
  - `train_epochs_2d_list`, contains a list of all the possible epoch parameter used for the training of the 2d Noise2Void model which are tested during the hyperparameters optimization routine.
  - `train_loss_2d_list`, contains a list of all the possible loss function for the training of the 2d Noise2Void model which are tested during the hyperparameters optimization routine. This parameter can be:
    - `'mse'`;
    - `'mae'`.
  - `n2v_manipulator_2d_list`, contains a list of all the possible 'n2v_manipulator' parameter used for the training of the 2d Noise2Void model which are tested during the hyperparameters optimization routine. The 'n2v_manipulator' is the criteria used to replace the value of the masked pixels during the Noise2Void training. This parameter can be:
    - `'uniform_withCP'`, to replace the masked pixel with a randomly selected pixel of the patch *with* the pixel to mask;

        * `'normal_withoutCP'`, to replace the masked pixel with a randomly selected pixel of the patch *without* the pixel to mask;
        * `'normal_additive'`, to replace the masked pixel with a pixel having the value of the pixel itself plus some guassian noise with 0 mean standard deviation equal to the parameters specified in the `n2v_neighborhood_radius_2d_list` field ;
        * `'normal_fitted'`, to replace the masked pixel with a pixel having the value of the pixel itself plus some guassian noise with 0 mean and standard deviation estimated from the patches;
        * `'idenitity'`, the pixel is not replaced.
    - `n2v_neighborhood_radius_2d_list`, contains a list of all the possible radii used to define the neighborhood of a pixel used in the training of the Noise2Void models which are tested during the hyperparameters optimization routine.
- `n2v_3d`: contains a dictionary which is used to define the parameter space used for the optimization of the 2d Noise2Void denoiser. It contains the same keys of the previous dictionary, except that '3d' have to be used in the name rather than '2d'.

The plugin-specific parameters contained in this dictionary are:

- `filter_to_use`: it contains the name of the filter chosen. This field is ignore when the auto-optimization is done. It can be:
    - `'n2v_2d'`, for 2d Noise2Void denoiser;
    - `'n2v_3d'`, for 3d Noise2Void denoiser.
- `filter_params`: list whose elements are the denoiser parameter. Each denoiser parameter have to be specified with a list of two elements: the parameter name and the parameter value. This field is ignored when the plugin optimization is done, and in that case the parameter of the best filters found during the optimization routine are used. For manual specification of the parameters of the filter available in this plugin, see https://github.com/juglab/n2v (in particular in the 'n2v/models/n2v_config.py' file). It has to be specified as below

```
[[name_parameter_1, value_parameter_1], [name_parameter_2, value_parameter_2], ...].
```

- `trained_n2v_setting`: it is a dictionary containing the setting relative to the loading/saving of trained n2v models. This dictionary has the following fields:
    - `use_trained_n2v_model`, a boolean such that if True, a trained n2v model is loaded from the path contained in the field `'path_to_trained_n2v_model'`.
    - `path_to_trained_n2v_model`, which contain the path to a trained n2v model. This field is ignored if the previous field is `False`.
    - `save_trained_n2v_model`, a boolean such that if True after training, the best n2v model is saved a the path contained in the field `'saving_path'`.
    - `saving_path`, which containt the path where the best n2v model is saved. This field is ignored if the previous field is `False`.

When `auto-optimize = True` the plugin-specific parameters above are ignored, since the one selected by the optimization procedure are used. Finally, the meaning of the remaining parameters can be found in *General information#Transfomation dictionary*.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `DenoiserDNN`.

10.2 Use case

The typical use of this plugin are:

1. Reduce noise level in the input stack.

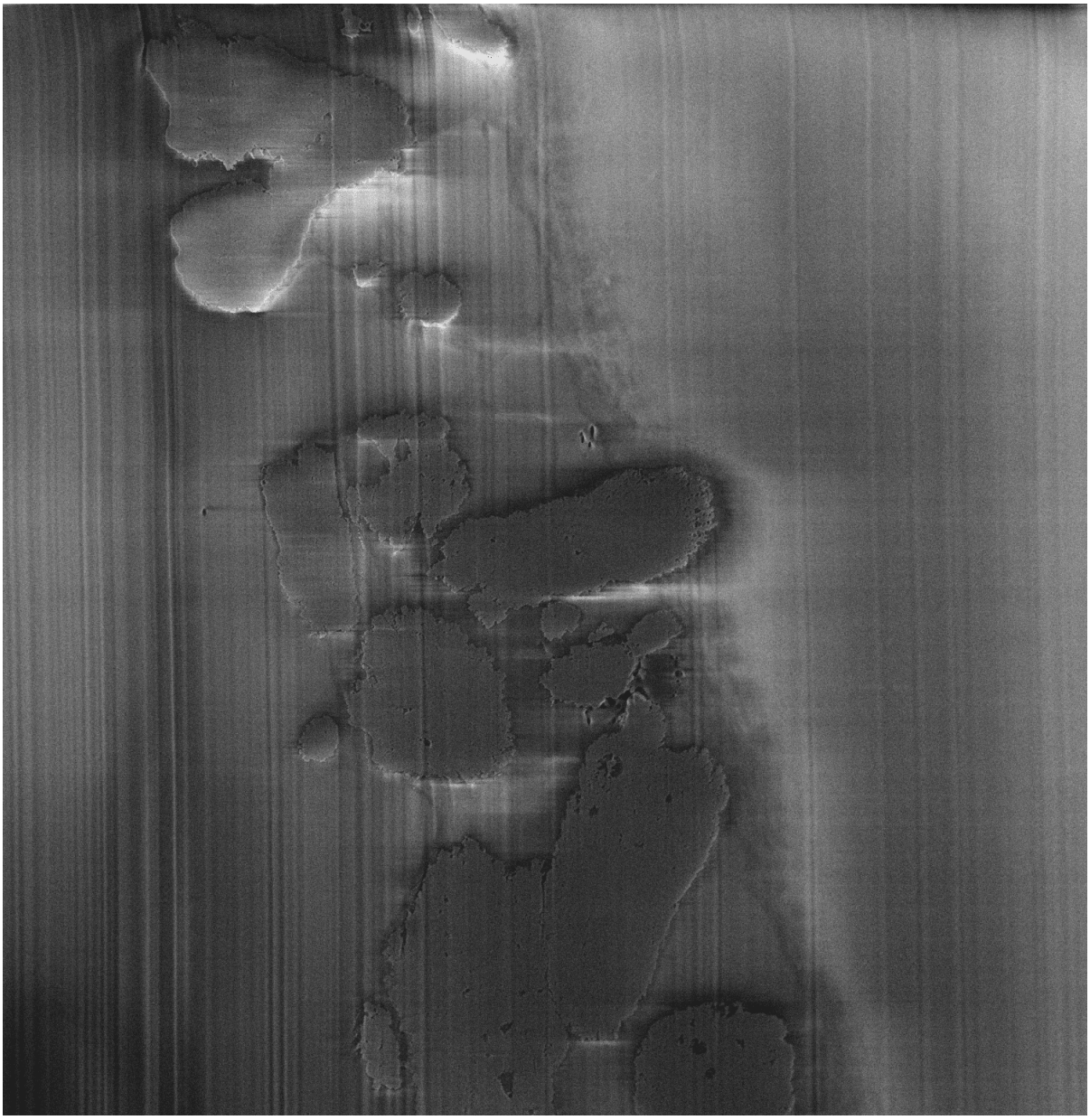
Tip: From the practical point of view, the following empirical findings

1. It has been observed that use the optimization routine for the hyperparameter search does not perform well when 2d and 3d denoiser are compared among each other. Therefore, the option `tested_filter_list = ['n2v_2d', 'n2v_3d']` is not advised. Hyperparamter search based on the optimization routine of this plugin, has shown to perform well when restricted to 2d models only or 3d models only, i.e. with the option `tested_filter_list = ['n2v_2d']` or `tested_filter_list = ['n2v_3d']`.

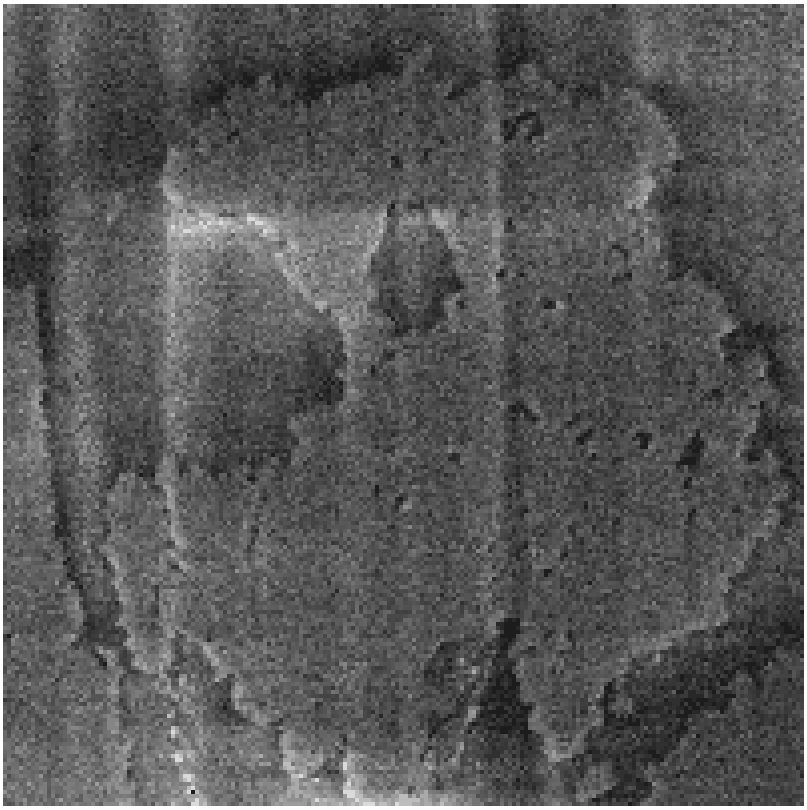
Keep also in mind that, for the application of 3d Noise2Void model, the stack has to be already aligned, for example with the *Registrator plugin*.

10.3 Application example

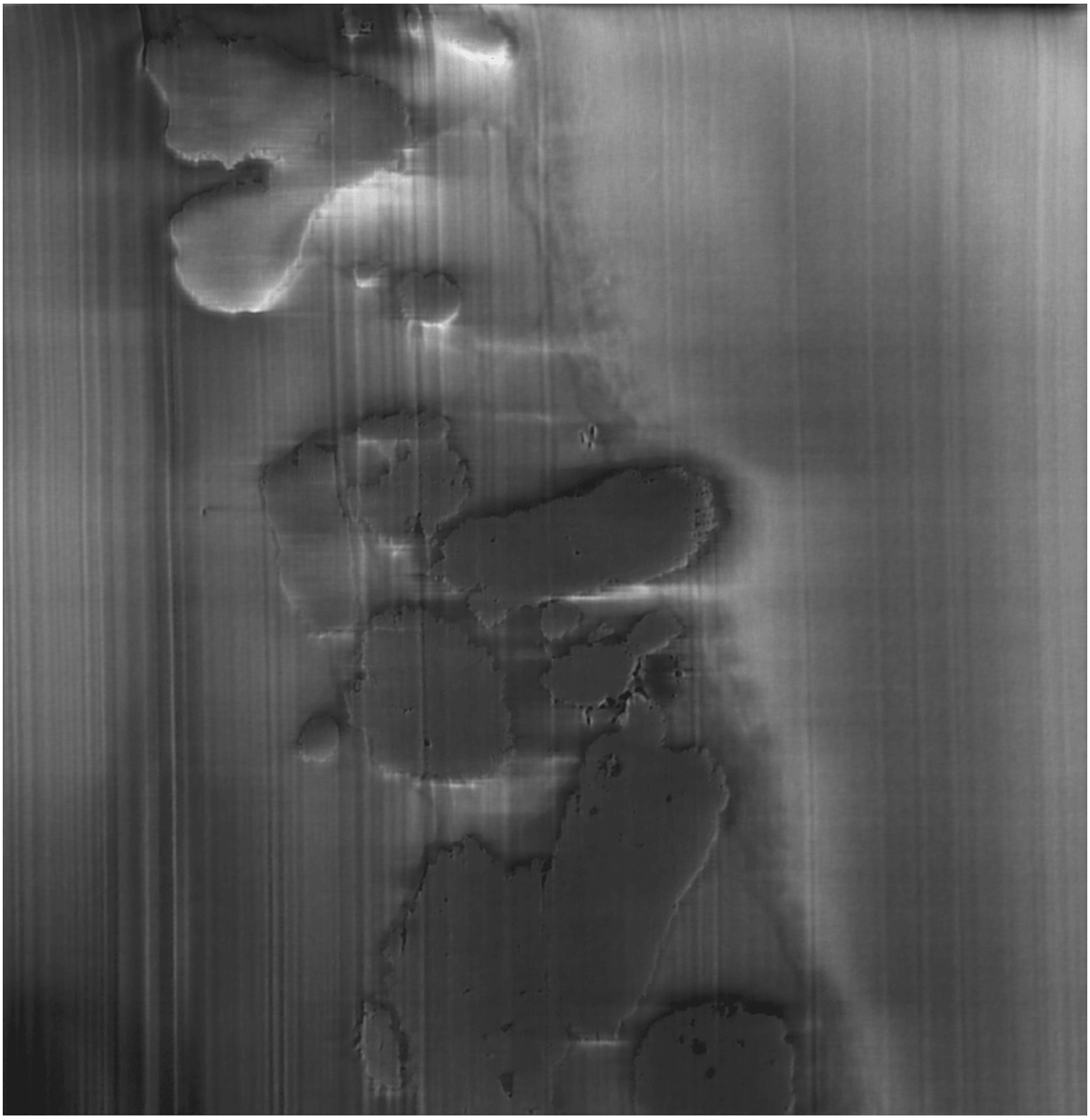
As example consider the slice of a stack of a biological sample obtained via SEM, where the noise is clearly present.



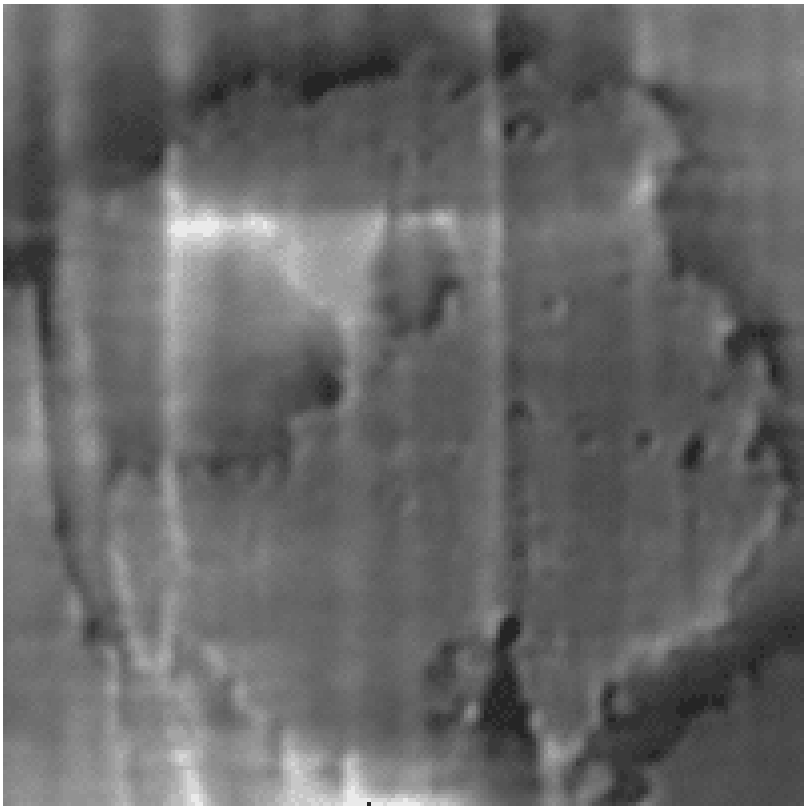
A zoomed part of the center-top/right part of the slice can be found below. One can clearly see some complex structures under the vertical stripes.



Applying the denoiserDNN plugin with default setting (i.e. with using a 2d noise2void model), except for the use of the bounding box, which was defined in the central part of the image, the best hyperparameters for the noise2void model has been selected. By applying the trained model with these hyperparameters, the result one obtains is the following.



Zooming-in in the same place, one can see that the noise level on the image is reduced.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

10.4 Implementation details

This plugin use the original implementation of Noise2Void which can be found here. All the parameters can be found in the 'n2v/models/n2v_config.py' file ot that repository, where a brief explanation of their meaning is given. For some of them, the original work [Krull2019] may help in clarifying the meaning.

The Noise2Void idea can be easily understood by looking the previous work about denoising with deep neural network. More precisely the Noise2Noise work [Lehtinen2018] can be very helpful. In Noise2Noise the idea is the following. Consider two noisy images reproducing exactly the same object (i.e. the image content is the same but the noise, being random, is different). Given a dataset of couples of this kind, it is possible to train a deep neural network in the following manner. For any couple in the dataset, one of the two images is given as network input and the other as target. What has been observed is that, since the network cannot learn random input-output relation, the only thing that the network can learn to reproduce is the image without noise, which is the same for both the images of the couple. Noise2Void goes one step further showing that by masking the value of a pixel in the input of the network, and giving this pixel value as target, the network can learn to denoise the image. Again, since the network cannot learn to reproduce the random component in the pixel value, the best it can do is to estimate the deterministic component the masked pixel would have based on the information available in the surrounding pixels. This procedure, if done exactly as described here is very inefficient, and the Noise2Void authors derive an approximated procedure, which speed up the network training a lot.

Summarizing given a stack $S(k, j, i)$ and call $N2V_{2d}[\alpha]$ and $N2V_{3d}[\alpha]$ represent the trianed 2d and 3d Noise2Void

model, where α represents the set of the possible hyperparameters of the network. In the 2d case, given a slice $S[k](j,i)$ the output stack is composed as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = N2V_{2d}[\alpha](S[k](j,i)).$$

In the 3d case, the network is applied to the whole stack directly, therefore

$$S(k,j,i) \rightarrow S_{output}(k,j,i) = N2V_{3d}[\alpha](S(k,j,i)).$$

10.4.1 Optimization details

The optimization routine of this plugin is done in order to find the best combination among (a reasonable subset of) the possible hyperparameters of network, and is based on the J-invariance principle [Batson2019]. A brief discussion of the J-invariance and why it can be used for this kind of optimization can be found *here*. Despite the Noise2Void training scheme does not guarantee the trained model to be J-invariant (see section 2 in [Batson2019]), it has been empirically observed that the model with hyperparameters selected using the J-invariance criteria lead to very good results.

10.5 Further reading

Articles:

CHAPTER

ELEVEN

DESTRIPER

Destriper in a nutshell.

1. Plugin to eliminate the curtaining artifacts (typical of Cryo FIB-SEM images) in a stack;
2. This plugin is multichannel;
3. This plugin can be optimized on the stack;
4. Python API reference: `bmiptools.transformation.restoration.destriper.Destriper`.

This plugin can be used to eliminate the curtaining artifacts, typical of Cryo FIB-SEM images. It implements the Wavelet-FT filter described in [Beat2009] with an optimization procedure for the automatic selection of the filter parameter. The Wavelet-FT filter is applied slice-wave to the input stack.

The Python API reference of the plugin is `bmiptools.transformation.restoration.destriper.Destriper`.

11.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'auto_optimize': True,
'optimization_setting': {'wavelet': {'use_wavelet': 'all',
                                     'wavelet_family': 'db'
                                    },
                         'sigma': {'sigma_min':0.01,
                                   'sigma_max': 50,
                                   'sigma_step':1
                                  },
                         'decomposition_level': {'set_decomposition_level_to_max_
↪compatible': True,
                                                 'decomposition_level_min': 2,
                                                 'decomposition_level_max': 9,
                                                 'increase_decomposition_level_during_
↪inference': False
                                                 },
                         'opt_bounding_box':{'use_bounding_box': True,
                                             'y_limits_bbox': [-500,None],
                                             'x_limits_bbox': [500,1500]
                                            },
                         'fit_step':10
```

(continues on next page)

(continued from previous page)

```
                        },
'wavelet_name':'db1',
'decomposition_level':4,
'sigma': 4,
'match_in_out_contrast': True
}
```

The optimization-related plugin-specific parameters contained in the `optimization_setting` field of this dictionary are:

- `wavelet`: contains dictionary with the parameters related to the possible wavelet type which can be used by the Wavelet-FT filter during the parameter search. The key of this dictionary are:
  - `use_wavelet`: it specify the kind of wavelet(s) and can take the following values:
    - `'all'`, to check all the discrete wavelets available in the *PyWavelet* library during the optimization.
    - `'family'`, to check just a single wavelet family during the optimization. When this option is used the wavelet family have to be specified in the field `wavelet_family`.
    - A name of a discrete wavelet of the *PyWavelet* library which is kept fixed during the optimization (see here for the list of available wavelet).
  - `wavelet_family`: name of a discrete wavelet family of the *PyWavelet* library to search only among the wavelets of this family during the optimization. This field is ignored if `use_wavelet` is not set equal to `family`.
- `sigma`: contains a dictionary for the setting defining the parameter space for the standard deviation of the gaussian filter used on vertical components of the wavelet decomposition of the image to be filtered. This dictionary has the following keys:
  - `sigma_min`: minimum value of standard deviation of the gaussian filter used by the Wavelet-FT filter used during the parameter search.
  - `sigma_max`: maximal value of standard deviation of the gaussian filter used by the Wavelet-FT filter used during the parameter search.
  - `sigma_step`: step value used to define the possible values of standard deviation of the gaussian filter used by the Wavelet-FT filter used during the parameter search.
- `decomposition_level`: contains a dictionary for the setting related to the research of the decomposition level used in the wavelet decomposition of the image. It has to be specified as follow:
  - `set_decomposition_level_to_max_compatible`: if set True only the maximal possible level of the wavelet decomposition, which would not not introduce boundary artifacts in the reconstruction of the unfiltered image, is used during the parameter search (see *here*).
  - `decomposition_level_min`: it is the minimum decomposition level of the 2D wavelet transform used during the parameter search. If set equal to 'max' the maximal possible level of the wavelet decomposition, which would not not introduce boundary artifacts in the reconstruction of the unfiltered image, is used.
  - `decomposition_level_max`: it is the maximal decomposition level of the 2D wavelet transform used during the parameter search. If set equal to `'max'` the maximal possible level of the wavelet decomposition, which would not introduce boundary artifacts in the reconstruction of the unfiltered image, is used.
  - `increase_decomposition_level_during_inference`: if set True the decomposition level during the inference is increased by one. This typically further reduces the stripe artifacts in case the optimization does not find the visually best combination of parameters.

The plugin-specific parameters contained in this dictionary are:

- `wavelet_decomposition`: contains a dictionary for the setting of the wavelet transform part of the Wavelet-FT filter. The dictionary has to be specified as below:
  - `wavelet_name`: is the wavelet name used by the Wavelet-FT filter
  - `decomposition_level`: when an integer number is given, this number is the maximal decomposition level used in the wavelet transform. If `'max'` is given, the highest level which can be used in the wavelet decomposition without introduce border artifacts in the reconstruction of the unfiltered image is used.
- `fourier_space_filter`: contain a dictionary for the setting related to the Fourier transform part of the filter used in this plugin. The only possible parameter is:
  - `sigma`: is the standard deviation of the gaussian filter used in the Fourier space to remove vertical artifacts.
- `match_in_out_contrast`: when True, the histogram of each slice of the stack after the Wavelet-FT filter is matched with the histogram of the corresponding input slice, increasing the contrast in the output.

When `auto-optimize = True` the plugin-specific parameters above are ignored, since the one selected by the optimization procedure are used. Finally, the meaning of the remaining parameters can be found in *General information#Transfomation dictionary*.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Destriper`.

11.2 Use case

The typical use of this plugin are:

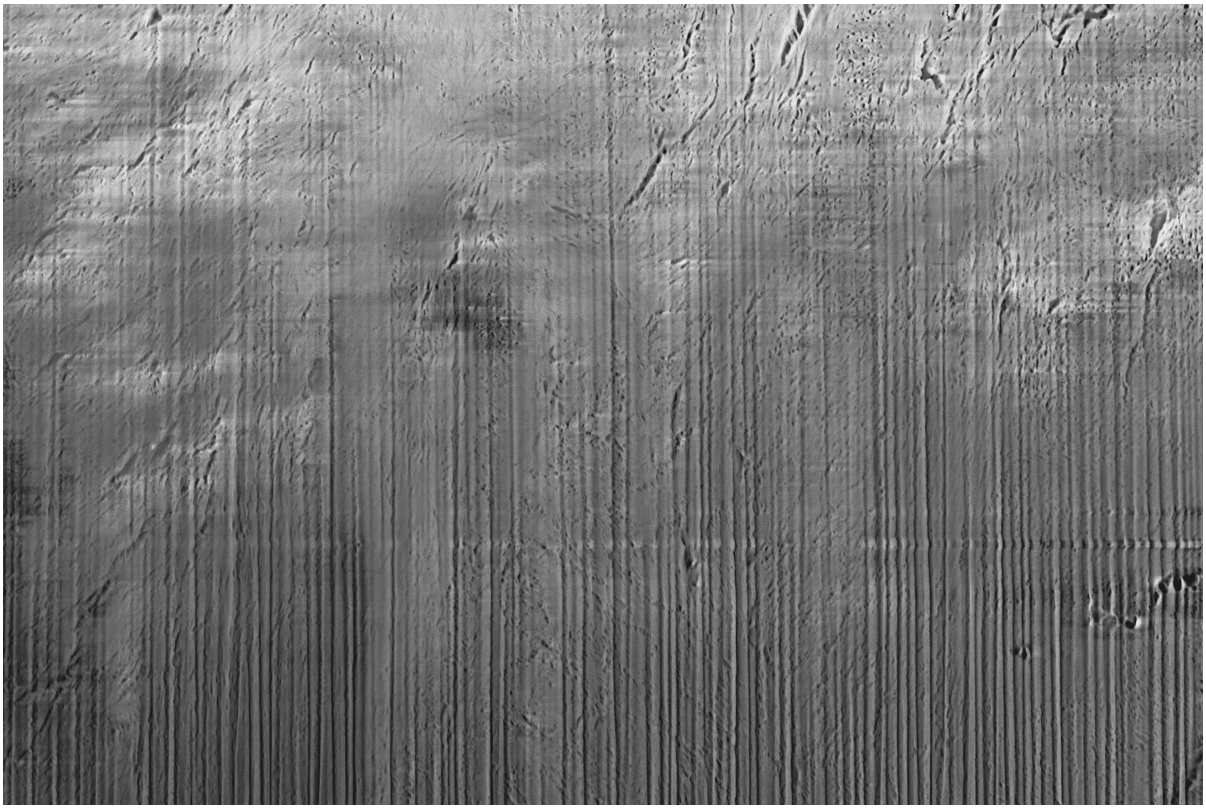
1. Reduce the curtaining artifact present on the input stack.

Tip: The following things turn out to be useful, from a practical point of view.

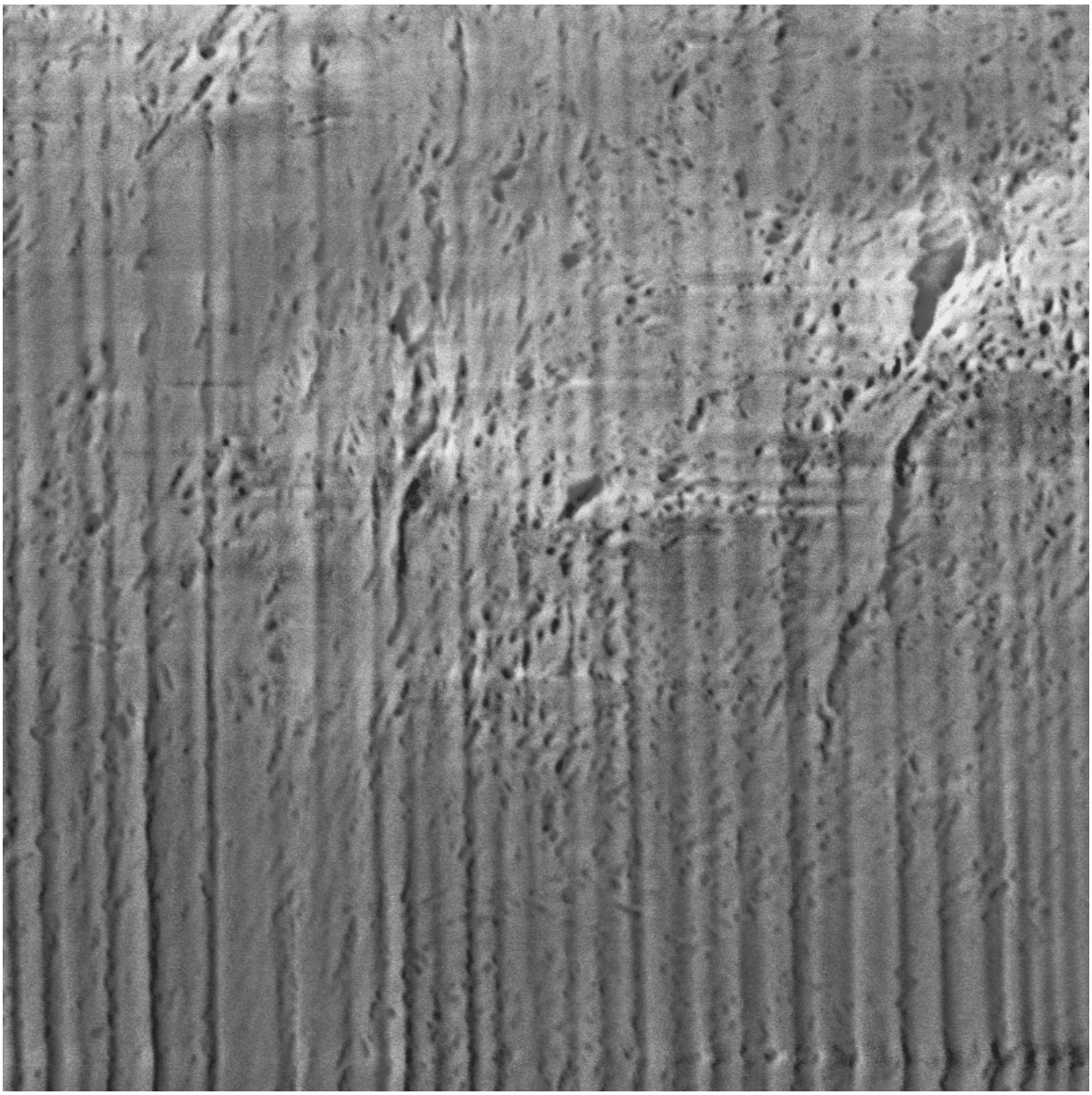
1. If the bounding box is used during the optimization, it should contain the region of the stack where the curtaining artifacts are stronger. Optimizing the plugin in a different region would lead to sub-optimal filter parameter for the input stack.
2. Most of the time, setting the decomposition level to the maximum value allowing to the wavelet decomposition to reconstruct the image without introducing artifacts related the image boundaries, gives good result. This number depends on the image size and can be computed in advance. During the optimization by setting `set_decomposition_level_to_max_compatible = True` the decomposition level is kept constant to maximum value. If in `wavelet_decomposition` one set `decomposition_level = 'max'`, the decomposition level used for the filter application is automatically set to the maximum value, without the need to the user to specify it.
3. Empirically, it has been observe that:
   - if stripes are of the same intensity along the entire image, not using the bounding box (i.e set `use_bounding_box = False`) and increasing the decomposition level of the Wavelet transform during the filter application (i.e. set `increase_decomposition_level_during_inference = True`) gives good result.
   - if stripes have a stronger intensity in some part of the image, using the bounding box "centered" in this region (i.e set `use_bounding_box = True` and *set the correct coordinates for the bounding box*) *without* increasing decomposition level during the filter application (i.e. set `increase_decomposition_level_during_inference = True`) gives good result.
4. The `match_in_out_contrast` option typically increase the contrast in the output image. However, it may also amplify some artifact that can be introduced by the Wavelet-FT filter.

11.3 Application example

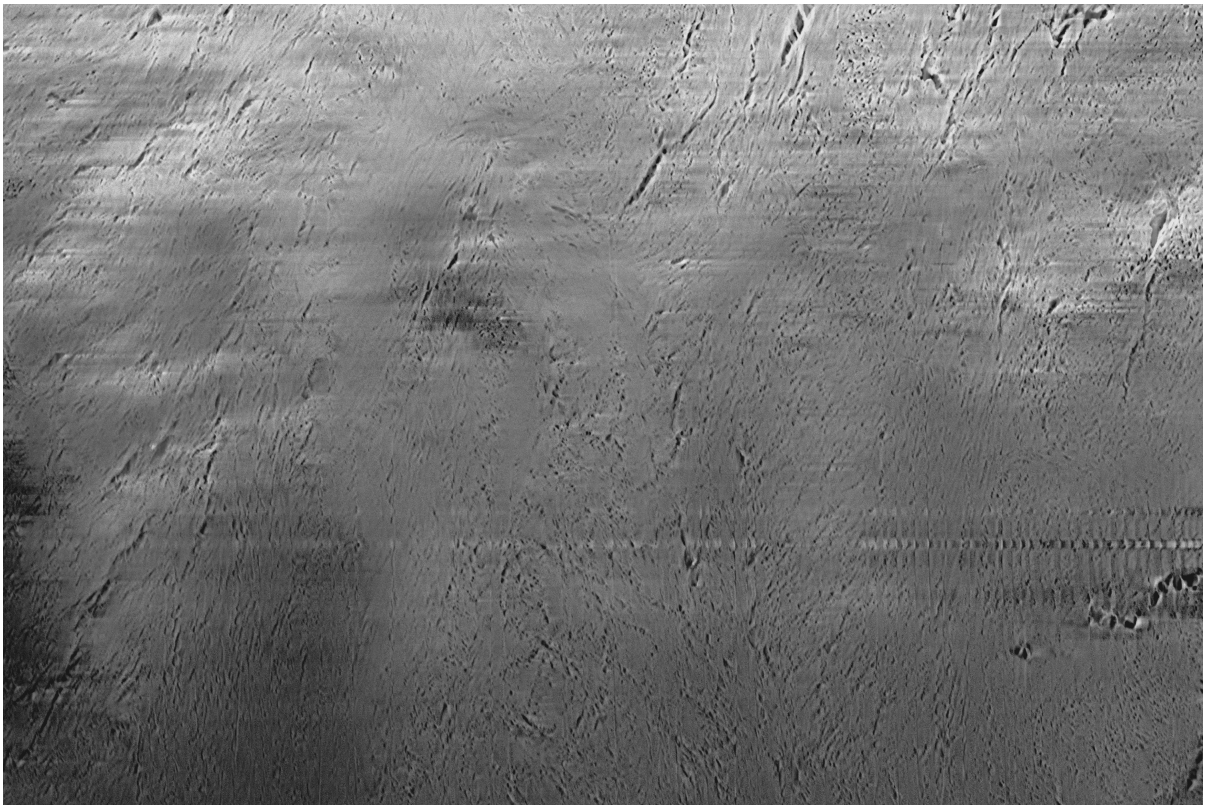
As example consider the slice of a stack of a biological sample obtained via FIB-SEM, where the striping artifact is clearly present.



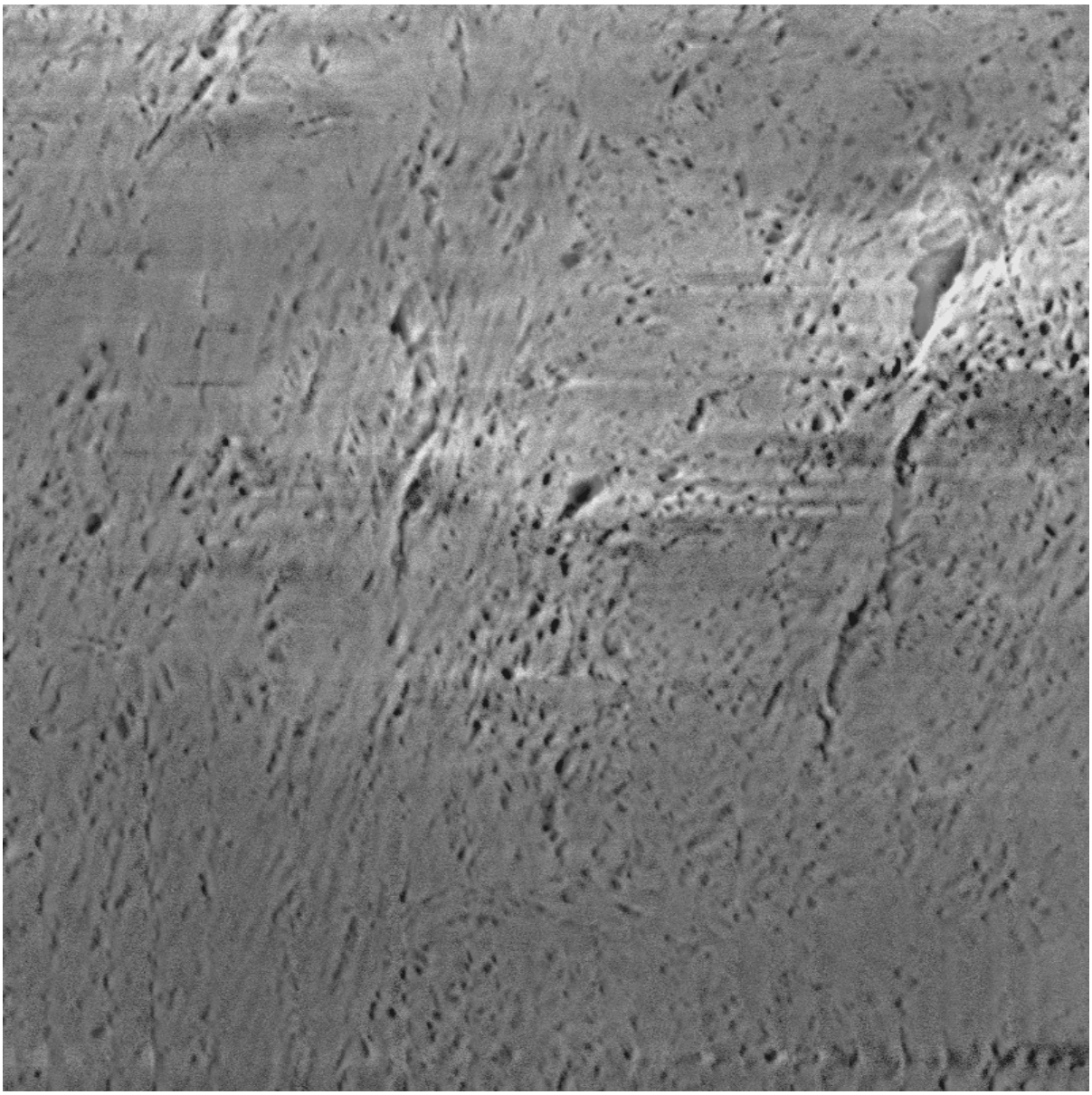
A zoomed part of the center-top/right part of the slice can be found below. One can clearly see some complex structures under the vertical stripes.



Applying the destriper plugin with default setting, except for the use of the bounding box, which was defined in the center-bottom part of the image and increasing the decomposition level during the inference (i.e. setting `increase_decomposition_level_during_inference = True`, see *tip number 3* of the previous section)



Zooming-in in the same place, one can see that the structure now are well visible and the stripes are almost absent.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

11.4 Implementation details

The core operation of this plugin is the application of the Wavelet-Fourier filter described in [Beat2009]. Given a $K \times J \times I$ stack $S(k, j, i)$, the Wavelet-Fourier filter is applied to each slice $S[k](j, i)$. The filter consists essentially of 3 steps:

1. 2D wavelet decomposition using a certain wavelet w up a to certain decomposition level l of the input image. This operation consists essentially in the iterated application of a (single-level) 2D wavelet transform with wavelet w for l times. Given an input slice $S[k](j, i)$, the application of a (single-level) 2D discrete wavelet transform $WT[w]$ produces 4 different output images, having half of the size of the input, (also called *subbands*) namely

$$\{S_{LL}[k](j', i'), S_{LH}[k](j', i'), S_{HL}[k](j', i'), S_{HH}[k](j', i')\} = WT[w](S[k](j, i)),$$

   which represents the wavelet decomposition at the level 1. S_{LL} contains an approximation of the input image, S_{LH} contains mainly the horizontal features of the input image, S_{HL} contains mainly the vertical features of the input image, while S_{HH} contains essentially the diagonal features of the input image. To go on to the next level, the 2D wavelet transform is applied recursively to the approximation image S_{LL} for l times. The full transformation will be denoted with $WD2d[w, l]$.

2. Filtering in the fourier space the vertical component at all levels. This filtering consist in the application of a 2D discrete Fourier transform, DFT, then multiply the result by a suitable filter function $g(\nu_j, \nu_i)$, and finally applying the 2D inverse discrete fourier transform. The filter function $g(\nu_j, \nu_i)$ is chosen such that the vertical components due to stripes artifact are suppressed, and turns out to be equal to

$$g(\nu_j, \nu_i) = g_\sigma(\nu_j, \nu_i) = 1 - e^{-\frac{\nu_j^2}{2\sigma^2}},$$

   which depends only on the parameter σ. Let $S_{HL,u}[k](j, i)$ be the vertical component produced by the wavelet transform at the level $u \in [1, ..., l]$.

$$S^{filt}_{HL,u}[k](j, i) = DFT^{-1}[g_\sigma \cdot DFT[S_{HL,u}[k]]](j, i).$$

   This is done for all decomposition levels $u \in [1, ..., l]$. The whole filtering operation will be denoted by $FF[\sigma]$.

3. 2D wavelet reconstruction using the wavelet w and decomposition level l used in initial step. It works exactly as the 2d wavelet reconstruction, but rather than using the 2D wavelet transform, it uses its inverse. For each level u, the input of the inverse wavelet transform is composed by the 4 outputs described in step 1, except that $S^{filt}_{HL,u}$ is now used instead of $S_{HL,u}$. This operation will be denoted with $WD2d^{-1}[w, l]$.

Summarizing, the Wavelet-Fourier filter consists in the following operation

$$S[k](j, i) \rightarrow S_{output}[k](j, i) = WD2d^{-1}[w, l](FF[\sigma](WD2d[w, l](S[k](j, i)))).$$

This operation is applied on all the slices of the stack. It can be seen that this plugin depends on 3 parameters:

- the wavelet used w,
- the decomposition level l,
- the parameter σ of the filter in the Fourier space,

which corresponds to the parameters of the transformation dictionary contained in the fields `wavelet_decomposition` and in `fourier_space_filter`. In case of stack with multiple channels, the destriper is applied independently to each channel.

11.4.1 Optimization method

The possible combinations of the 3 parameters for the wavelet-Fourier filter are a lot. Most of the time a simple criteria can be used for the selection of the level (see *here*), but even in this case the parameters combinations are too many. Find the right combination of parameters can be therefore a time consuming operation. An optimization procedure has been developed to automatize this step, rendering the plugin almost parameter-less. What the user have to do is just to define the parameters space boundaries. This is done in the in `optimization_setting` section of the transformation dictionary. The default parameter space boundaries specified in the `empty_transformation_dictionary` of the plugin, seem to be good enough for majority of the typical application cases of this plugin.

The optimization is done by finding the parameters combinations which minimize a suitable loss function. More precisely, let $f_{w,l,\sigma}$ be the wavelet-Fourier filter described in the previous section. Given an input image $I(j,i)$ of size $J \times I$ with curtaining artifact, one can trivially write that

$$I(j,i) = O(j,i) + D(j,i),$$

where $O(j,i)$ is the *output image*, i.e. $O(j,i) = f_{w,l,\sigma}[I](j,i)$, while $D(j,i)$ is called *stripe image*, which can be simply defined as $D(i,j) := O(i,j) - I(i,j)$. For the perfect filter, the stripe image would contain only the stripes, while the output image would contain the image without any curtaining artifact. Consider the ideal case, where the curtaining artifacts consist in perfectly vertical stripes of constant intensity superimposed to the true image (in the real case the stripes are slightly swinging and the intensity may vary). Assume also that the stripe intensity is high if compared to the typical intensity of the true image. By using the perfect filter, the gradient along the vertical direction of the destriped output image would match exactly gradient of the true image, while the gradient of the stripe image along the same direction would be zero everywhere. For the horizontal direction the situation is different: the gradient along the horizontal direction of the stripe image would be close but not equal to the gradient of the input image, since most of the variations along horizontal direction are due to the stripes. Keeping that in mind, one can define the following loss function

$$\mathcal{L}[w,l,\sigma](I) = P + Q + R,$$

with

- $P = \frac{1}{JI}\sum_{j,i} |\nabla_y O(j,i) - \nabla_y I(j,i)|$, is the term favoring the match between the gradient along the vertical direction between the output image and the input image;
- $Q = \frac{1}{JI}\sum_{j,i} |\nabla_x D(j,i) - \nabla_x O(j,i)|$, is the term favoring the match between the gradient along the horizontal direction of the stripe image and the output image;
- $R = \frac{1}{JI}\sum_{j,i} |\nabla_y D(j,i)|$, is the term favoring the vanishing of the gradient of the stripe image along the vertical direction.

These three conditions are *weighted in equal manner* in the loss. ∇_y and ∇_x are the gradient operators along the corresponding directions. In the current implementation, the gradient is approximated using the central difference scheme, typically used to discretize derivatives. By using simple math, it easy to see that

$$\mathcal{L}[w,l,\sigma](I) = 2R + Q.$$

Note: The loss function $\mathcal{L}[w,l,\sigma]$ is asymmetric in how the X- and Y- direction are treated: apparently what happens in the Y-direction seems to have the double of the importance of what happens in the X-direction. The reason for this asymmetry lies in the fact that there is no term keeping into account that the variation of the destriped output image along the X-direction are small, compared to the variation of the stripe image in the same direction. By the way, a term encoding this requirements can be added to the loss. It has a similar structure to the one of the R term , and in particular it is

$$W = \frac{1}{JI}\sum_{j,i} |\nabla_x O(j,i)|.$$

It is however important to note, that the derivative of the destriped output image along the horizontal directions, even in the ideal case cannot vanish: the true image still vary along the horizontal direction in general. This means that if R is present in the loss to force $\nabla_y D$ to vanish, the term W need to have less importance in the loss with respect to R. This can be achieved by multiplying W for a suitable weight λ, i.e.

$$W = \lambda \cdot \frac{1}{JI} \sum_{j,i} |\nabla_x O(j,i)|,$$

with $\lambda << 1$. The condition on λ, encode exactly the fact that W have to be less important than R. However if $\lambda << 1$ this term can be neglected without altering too much the position of the minimum of the loss. That is the reason why this term is absent in the definition of the loss.

At this point, given a stack $S(k,j,i)$ of size $K \times J \times I$, the optimization problem can be formulated as follow:

$$(w_{best}, l_{best}, \sigma_{best}) = \text{argmin}_{w,l,\sigma} \left(\frac{1}{N} \sum_{n=0}^{N-1} \mathcal{L}[w,l,\sigma](S[k_n]) \right)$$

where the loss function is the average over some subset of slices $S[k_0], ..., S[k_{N-1}]$ (with $N \leq K$) of the stack $S(k,j,i)$. This subset of slices is defined by means of the parameter `fit_step` in the `optimization_setting` field of the transformation dictionary.

Note: When a bounding box is used, the loss $\mathcal{L}[w,l,\sigma]$ is not computed using the whole slice $S[k_n](j,i)$ but using the part of the slice selected with the bounding box.

From the algorithmic point of view, the current implementation of the optimization routine is rather trivial. The parameter space is defined according to the parameter specified in the `optimization_setting` field of the transformation dictionary, and the best parameter combination is found with a simple grid search.

11.5 Further details

Tutorials:

- *Case of study: destriper optimization*.

Articles:

CHAPTER

TWELVE

FLATTER

Cropper in a nutshell.

1. Plugin to remove slowly-varying brightness variation on the slices of the input stack;
2. This plugin is multichannel;
3. This plugin can be optimized on the stack;
4. Python API reference: `bmiptools.transformation.restoration.flatter.Flatter`.

This plugin can be used remove in the slices of a stack the slowly-varying brightness variation.

The Python API reference of the plugin is `bmiptools.transformation.restoration.flatter.Flatter`.

12.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'auto_optimize': True,
'optimization_setting':{'sigma_deriv_smoother': 5,
                        'sigma_min': 5,
                        'sigma_max': 'auto',
                        'sigma_step': 5,
                        'entropy_setting':{'image_range': (0,1),
                                           'derivative_range': (0,0.039),
                                           'n_bins': 1024},
                        'fit_step': 10,
                        'regularization_strength': 1,
                        'use_early_stopping': True,
                        'patience': 5},
'sigma_low_pass': 80}
```

The optimization-related plugin-specific parameters contained in the `optimization_setting` field of this dictionary are:

- `sigma_deriv_smoother`: it is the standard deviation used to compute the image derivative. Its default value is `5` is usually a good choice.
- `sigma_min`: is the smallest value of the parameter `sigma_low_pass` used during the optimization. This value is also used for the definition of the loss parameter (see *below*).
- `sigma_max`: is the maximum value of the parameter `sigma_low_pass` used during the optimization. It can be:

  - the actual maximum value of `sigma_low_pass`;
  - `'auto'`, to automatically infer from the image shape the maximum value of `sigma_low_pass`, imposing the filter size to be not to big with respect to the image.
- `sigma_step`: is the step size used for the variation of `sigma_low_pass` during the optimization routine.
- `entropy_setting`: its a dictionary containing the setting to compute the entropy od an image and of its derivative using histograms having the same binning, so that they are compatible. The keys of this dictionary are:
  - `image_range`, which is a tuple containing the minimum and maximum value of an image (i.e. the dynamic range of the image) which are used to define the histogram range for the image;
  - `derivative_range`, which is a tuple containing the minimum and maximum value of the modulus of the derivative of the image (i.e. the dynamic range of the modulus of the image derivative) which are used to define the histogram range for the image derivative;
  - `n_bins`, which is the number of bins used to compute the histograms needed in the optimization.
- `regularization_strength`: is the strength of the regularization used during the plugin optimization.
- `use_early_stopping`: if `True` an early stopping policy for the line-search optimization is used, with patience specified in the field below.
- `patience`: patience of the early stopping policy, i.e. the number of epochs with no improvement in the loss before to stop the optimization routine.

The plugin-specific parameters contained in this dictionary are:

- `sigma_low_pass`: it is the standard deviation parameter of the low-pass filter in the flatter transformation. For stack with multiple channels, a list with the low-pass filter parameters for each channel can be given.

When `auto-optimize = True` the plugin-specific parameters above are ignored, since the one selected by the optimization procedure are used. Finally, the meaning of the remaining parameters can be found in *General information#Transfomation dictionary*.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Flatter`.

12.2 Use case

The typical use of this plugin are:

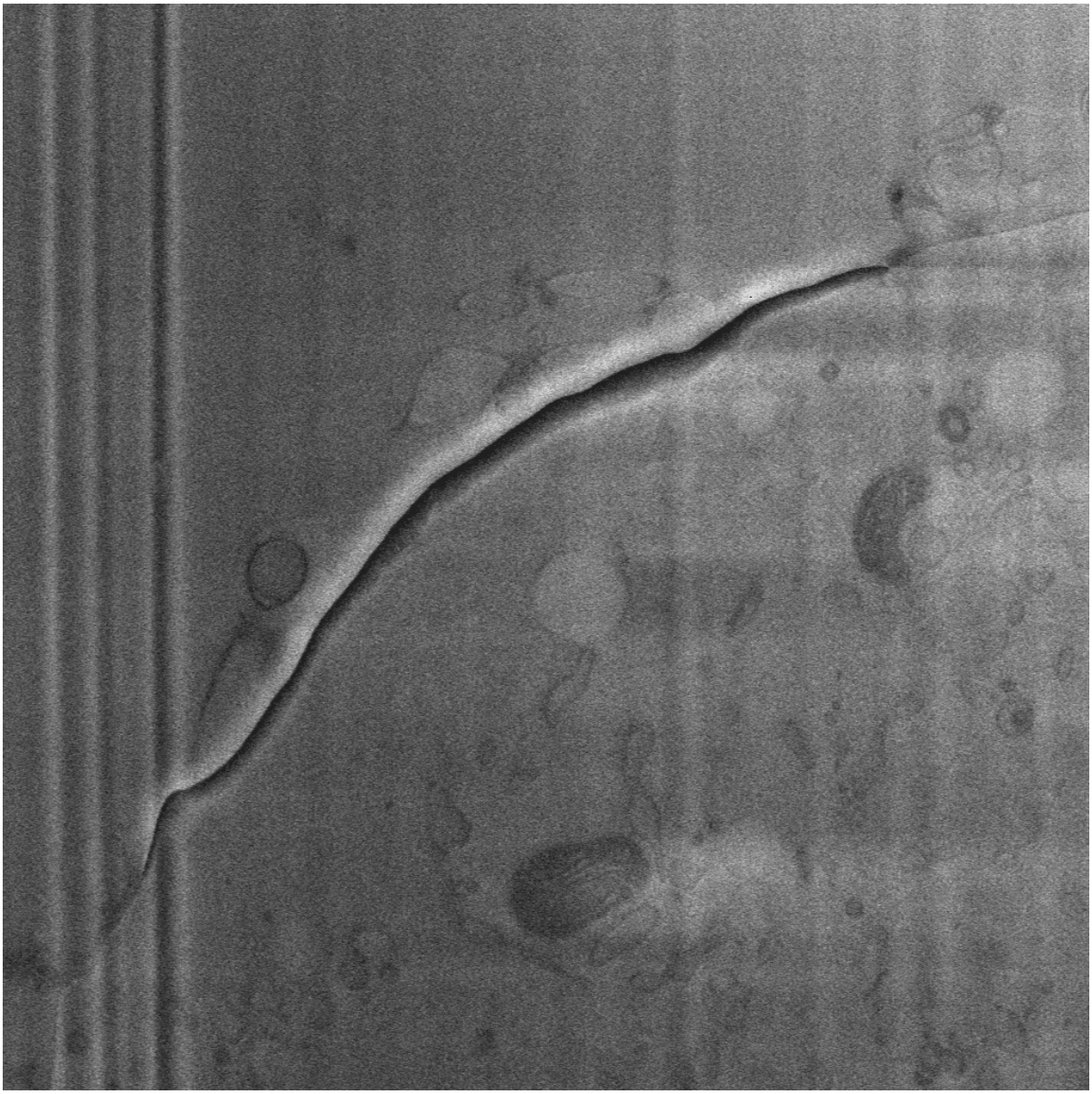
1. remove the slowly-varying brightness in the input stack;
2. flatten the image in order to make it easier the finding of an universal threshold to segment objects;
3. when charging is particularly strong and diffuse on the input slices, part of the charging can be removed/reduced with this plugin.

Tip: The following things turn out to be useful, from a practical point of view.

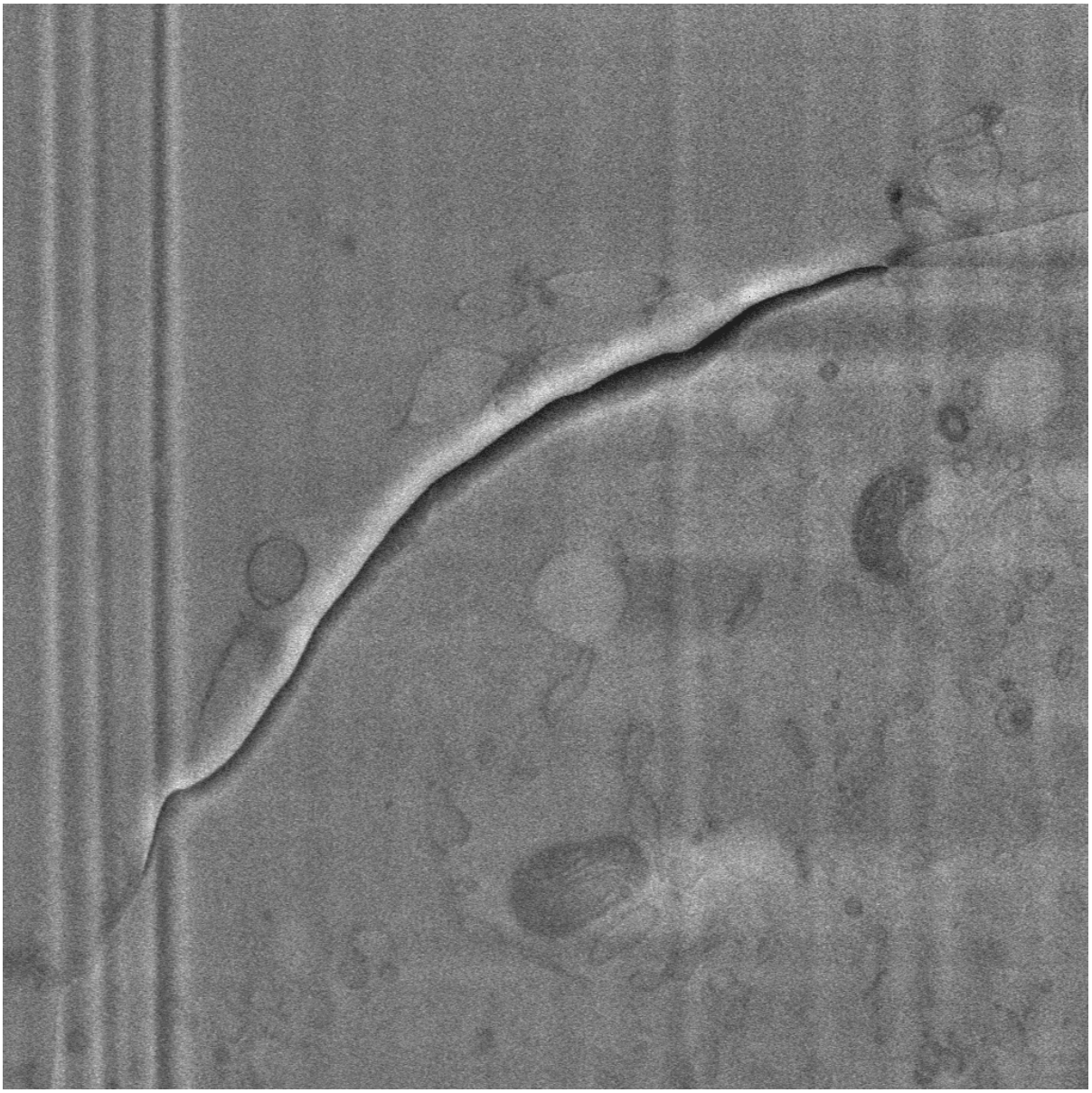
1. When slow brightness variation are expected for some reason, this plugin may simply remove part of the information content. Therefore its use is not suggested in this case.
2. It is warmly suggested to avoid to change the `entropy_setting`, unless there is a good reason for doing that.

12.3 Application example

As example consider the slice of a stack of a biological sample obtained via cryo-FIB-SEM, where the brightness slowly increase moving from the left to the top-right of the image. In order to make this artifact clearly visible in the slice showed below, a preliminary standardization step (using the *Standardizer* plugin with `standardization_type = '0/1'`) has been applied.



After the application of the Flatter plugin with the default parameters (i.e. the one present in the `empty_transformation_dictionary` of the plugin), the result obtained is given below.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

12.4 Implementation details

The plugin simply subtract from the input image a low-pass filtered version of the same image. Low-pass filtering is obtained with the application of a simple gaussian filter with standard deviation σ_{LP}. More precisely, given an image $I(j,i)$ and let $G[\sigma_{LP}]$ be a gaussian kernel with standard deviation σ_{LP}, the flattened version of the input image, $I_{flat}(j,i)$ is obtained by simply using the equation below

$$I_{flat}(j,i) = I(j,i) - (G[\sigma_{LP}] * I)(j,i)$$

where $*$ denotes the 2d convolution. The operation above will be denotes with the symbol $FL[\sigma_{LP}]$. The cut-off (spatial) frequency of the gaussian filter, is clearly determined by the kernel standard deviation. In practical terms, the details in the input image with are significantly smaller than σ_{LP} are are de fact eliminated in the gaussian filter output, leaving unchanged only those structures of the image which are significantly larger than σ_{LP}. These remaining structure are then subtracted from the original image. Therefore the σ_{LP} parameter in this plugin defines the scale below which the brightness variation are considered slowly-varying.

Summarizing, given a $K \times J \times I$ stack $S(k,j,i)$, the flatter plugin acts on each slice $S[k](j,i)$ as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = FL[\sigma_{LP}](S[k])(j,i).$$

In case of stack with multiple channels, the Flatter is applied independently to each channel.

12.4.1 Optimization details

The optimization routine try to find a reasonable value for σ_{LF}. It is reasonable both in the sense of preserving the image content, and in the sense of computational resources needed. The optimization itself is multichannel: when more than one channel is present in the input stack, the best parameters is found for each channel independently.

For the optimization of this plugin, the shannon entropies associated to the histogram of image and the histogram of the (modulus of the) gradient of it turns out to be useful. Given an image I, let $h_I(b)_{b=0}^{N-1}$ be its N bin histogram, obtained by dividing the image range in N intervals and counting how many times a pixel has its value falling in a given bin. The shannon entropy associated to this image can be defined as follow

$$H(I) = -\sum_{b=0}^{N-1} \tilde{h}_I(b) \log \tilde{h}_I(b)$$

where $\tilde{h}_I(b) = h_I(b) / \sum_{b'} h_I(b')$ is the normalized histogram. Given an image $I_0(j,i)$ a slowly varying would contribute to the image histogram as a more or less constant contribution to all the non-zero bin of the histogram. Therefore the removal of this contribution would lead to a decreasing in the image entropy. On the other hand if value of the image entropy decrease too much, this may indicate that less and less details are present in image. To counterbalance that, two things can be taken into account:

- given the modulus of the gradient of the image, $dI(j,i) = \sqrt{[\nabla_y I(j,i)]^2 + [\nabla_x I(j,i)]^2}$ (here ∇_l for $l = x, y$ corresponds to the a discrete differentiation followed by a suitable smoothing, as usually computed the derivative for an image), the entropy of dI is expected to grow when more details (edges in particular) are present. Therefore its inverse is expected to be very small if the flatter preserve as many details as possible when applied to the input image.
- the contribution due to a slowly-varying component is not only more or less constant for all the non null bins, but is also small. Therefore, it is not expected that entropy of the flatten image is too differnt with respect to the input image.

The three terms of the loss below contain the 3 conditions explained above. Given a image $I(j,i)$, obtained by applying the flatter to the corresponding input image $I_0(j,i)$, one can define the following:

$$\mathcal{L}_0(I) = \frac{1}{3}\left[\alpha H(I) + \frac{\beta}{H(dI)} + \gamma|H(dI) - H(dI_0)|\right],$$

where α, β and γ are three parameters which are used to make the ranges of the three terms in the loss comparable. The combination below gives good results

$$\alpha = \frac{1}{H(I_{min})}\,,\, \beta = H(dI_{min})\,,\, \gamma = \frac{1}{|H(dI_{min}) - H(dI_0)|},$$

where $I_{min} = FL[\sigma_{LP}^{min}](I)$ and $dI_{min} = FL[\sigma_{LP}^{min}](dI)$. Since the typical value of σ_{LP} is usually high, the convolution operation turns out to be particularly demanding from the computational point of view. From an empirical point of view, it has been observed that little or no visible is obtained by increasing too much the value of σ_{LP}. For this reason a regularization term is added to the previous loss

$$\mathcal{L}_{reg}(\sigma_{LP}) = \frac{(\sigma_{LP} - \sigma_{LP}^{min})^n}{\sigma_{LP}^{max}},$$

where σ_{LP}^{min} and σ_{LP}^{max} are the minimum maximum allowed values for σ_{LP}, while $n \geq 2$. To reduce the influence of this term for the low value of σ_{LP}, an high value of n is be used. The loss function use in the optimization routine is the following

$$\mathcal{L}(I, \sigma_{LP}) = \mathcal{L}_0(I) + \lambda\mathcal{L}_{reg}(\sigma_{LP}),$$

where λ is a parameter regulating the strength of the regularization. Given a $K \times J \times I$ stack $S(k, j, i)$ take a collection of its slices $\{S[k](j, i)\}_{k \in V_K}$ where $V_K \subset \{0, 1, \cdots, K-1\}$ (i.e. the subset of slices selected for the optimization), then the flatter optimization routine solve the following problem

$$\sigma_{LP}^{best} = \text{argmin}_{\,\sigma_{LP}} \sum_{k \in V_K} \mathcal{L}(FL[\sigma_{LP}](S[k]), \sigma_{LP}).$$

12.5 Further details

Websites:

- Shannon entropy on wikipedia

CHAPTER

THIRTEEN

DECHARGER

Dechargere in a nutshell.

1. Plugin to reduce the charging artefacts in a stack;
2. This plugin is multichannel;
3. When `local_GF2RBGF` is used as decharging method, this plugin can be optimized on the stack.
4. Python API reference: `bmiptools.transformation.restoration.decharger.Decharger`.

This plugin can be used to reduce the charging artifact, typical of Cryo FIB-SEM images. Two methods are available to reduce the charging artifact:

- *local GF2RBGF*: where the decharger first try to estimate the regions in each slice where the charging is present, by using a down-hill filter, and then correct the distortion locally. The correction is performed by subtracting an estimated increase of the brightness due to the charging in all the regions. This kind of correction method, try to correct the less is possible, but it may be slow.
- *global GF2RBGF*: which is like the local GF2RBGF but the correction algorithm is applied to the whole image skipping the step in which the regions in which charging is present are estimated. It is typically faster than the local one, but it changes more the whole image.

The charging can be both the "normal" one, where the brightness is increased locally, or the "inverse" one, where the brightness is instead decreased.

The Python API reference of the plugin is `bmiptools.transformation.restoration.decharger.Decharger`.

13.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'auto_optimize': True,
'optimization_setting': {'dilation_iterations': 10,
                         'N_regions_for_opt': 20,
                         'gf1_sigma_list': [40,80,120],
                         'color_shift_list': [0.05,0.1,0.2],
                         'gf2_sigma_list': [40,80,120],
                         'RB_radius_list': [2,10,50],
                         'gf3_sigma_list': [4,25,50],
                         'opt_bounding_box': {'use_bounding_box': False,
                                              'y_limits_bbox': [0,500],
```

(continues on next page)

(continued from previous page)

```
                                                'x_limits_bbox': [0,500] },
                          'fit_step': 10},
'decharger_type': 'local_GF2RBGF',
'GF2RBGF_setting': {'gf1_sigma': 80,
                    'gf2_sigma': 80,
                    'RB_radius': 2,
                    'gf3_sigma': 4,
                    'local_setting':{'A_threshold': 50,
                                     'color_shift': 0.1,
                                     'n_px_border': 10}},
'inverse': False}
```

The optimization-related plugin-specific parameters contained in the `optimization_setting` field of this dictionary are:

- `dilation_iteration`: is the number of dilation done to correction mask in order define the region in which charging is not present but the histogram is still comparable with the one obtained from the region in which charging is present.
- `N_regions_for_opt`: is the maximum number of regions considered in a correction mask for the loss computation.
- `gf1_sigma_list`: is the list containing the of possible values of the 'gf1_sigma' parameter tested during the optimization.
- `color_shift_list`: is the list of possible values of the 'color_shift' parameter tested during the optimization.
- `gf2_sigma_list`: is the list of possible values of the 'gf2_sigma' parameter tested during the optimization.
- `RB_radius_list`: is the list of possible values of the 'RB_radius' parameter tested during the optimization.
- `gf3_sigma_list`: is the list of possible values of the 'gf3_sigma' parameter tested during the optimization.

The plugin-specific parameters contained in this dictionary are:

- `decharger_type`: contain name of the decharging method applied to the images. The available decharger are:
  - `local_GF2RBGF`,
  - `global_GF2RBGF`.
- `GF2RBGF_setting`: is a dictionary containing the setting of the GF2RBGF method. This field is ignored when `decharger_type = 'local_GF2RBGF'` and `auto_optimize = True`. It contains the keys below:
  - `gf1_sigma`, which is the standard deviation of the first gaussian filter which flatten the slice.
  - `gf2_sigma`: which is the standard deviation of the second gaussian filter, which should be used if the image is not sufficiently flat.
  - `RB_radius`, which is the radius parameter of the rolling ball algorithm used to estimate the charging related increase in brightness.
  - `gf3_sigma` which is the standard deviation of the gaussian filter to smooth the estimated charging related increase in brightness, since after the rolling ball algorithm the estimated background is typically to 'regular'.
  - `local_setting`, containing a dictionary with the decharger setting used by the `'local_GF2RBGF'` method. It is ignored if the global decharger type is specified in the `decharger_type` field. This dictionary has the following fields:

    * `A_threshold`, which is a threshold on the area (expressed in pixel), used to disregard all the estimated charged regions that are too small. All the regions having area in pixel below this threshold are not corrected.
    * `color_shift`, which is number between 0 and 1 indicating the shift in the grey-level values used by the down-hill filter to identify the charged regions. Typically this value is small.
    * `n_px_border`, which is the number of pixels used to smoothly pass from the regions corrected, to the regions that are not. This is done to avoid a too drastic difference between the corrected and not-corrected regions.

- `inverse`: when `True` the decharger is applied to corrected the 'inverse-charging artifact', namely when charged regions are shifted towards low-brightness, rather than high-brightness (as it happens for normal charging). The default value is `False`.

When `auto-optimize = True` the plugin-specific parameters above are ignored, since the one selected by the optimization procedure are used. Finally, the meaning of the remaining parameters can be found in *General information#Transfomation dictionary*.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Decharger`.

13.2 Use case

The typical use of this plugin are:

1. Reduce the amount of charging artifacts in the input stack.

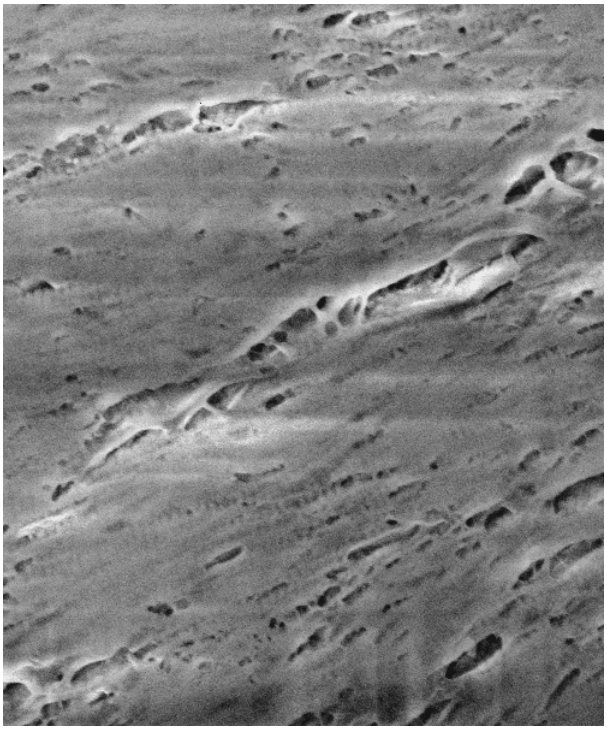
Tip: The following things turn out to be useful, from a practical point of view.

1. Even when the `local_GF2RBGF` decharger is used, the `A_threshold` parameter does not have an optimization procedure. By the way, by keeping it constant to a reasonable value (`A_threshold = 50`, for instance) gives good results for different input stacks.
2. The decharger plugin should be applied to images having sufficiently smooth background. The background should not necessarily need to be flat, but a smooth variation of it is assumed in the plugin. To have a rough idea of what can be considered smooth variation, the following criteria can be applied. Consider the typical dimension of the chargerd regions in an image, if another artifacts have comparable dimension, the background cannot be considered smooth for the application of the decharger. As such, in images with striping artifacts, it is recommended to remove this artifact (e.g. by applying the *Destriper* plugin) *before* the application of this plugin. On the other hand, the image noise should not disturb the application of a plugin, since it typically involves at most few pixels (if noise is pixelwise correlated) and decharger is insensitive to such a small variations. Therefore there is no need to apply a denoiser before the application of this plugin (but of course it can be done if needed).

13.3 Application example

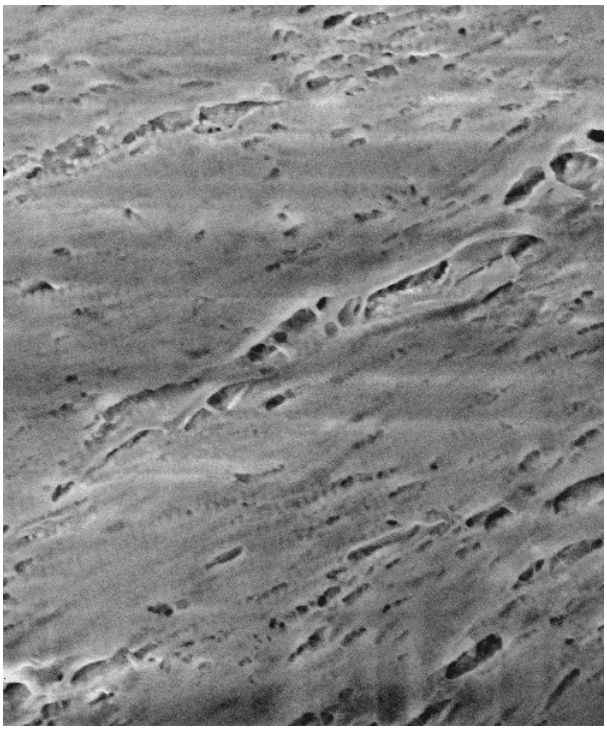
As example consider the slice of a stack of a biological sample obtained via FIB-SEM, where the charging artifact is present. According to the *tip 2*, the *Destriper* plugin was applied before to apply the decharger. Below the starting point for the application of the decharger.

A zoomed part of the center-bottom/center-left part of the slice can be found below. One can clearly see some complex structures surrounded by a brightness halo, which is due to the charging artifact.



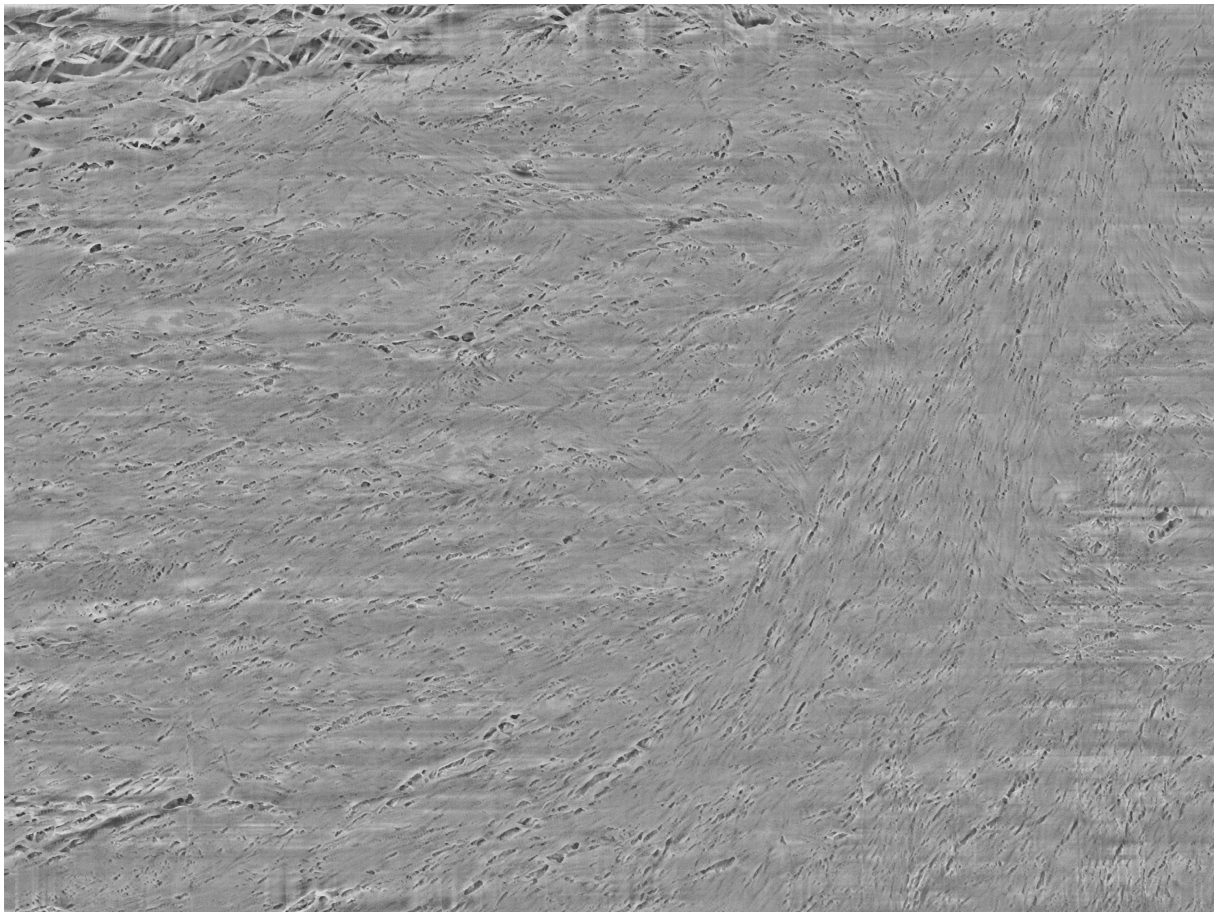
Applying the decharger plugin with default setting (which uses the `local_GF2RBGF` algorithm, which can be optimized), except for the use of the bounding box, the result obtained is showed below.

Zooming-in in the same place, one can see that the structure now are well visible and charging is reduced.

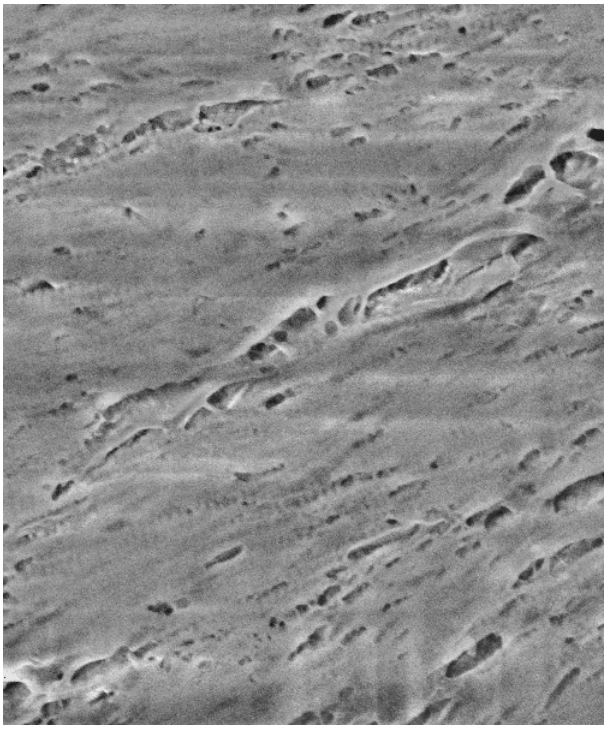


.

Attention: As one can see there is till some brighter regions in the final image. These local brightness variations probably have the same physical origin of the artifact here called charging. By the way, to reduce them, Decharger alone is not sufficient: one need to use also the *Flatter* plugin. Applying the Flatten with its standard setting on the images above (i.e. after the Decharger) one obtain the following.



The zoomed part now looks as below.



Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

13.4 Implementation details

This plugin has two decharging methods implemented: the *global GF2RLGF* and the *local GF2RLGF*. They work according to the same principle, but in the local one correct the correction is done only locally, in order reduce as much as possible the modification in the input image.

13.4.1 Global GF2RBGF

In the *global GF2RBGF*, the correction algorithm is applied to the whole input image I directly. GF2RLGF stands for "Gaussian Filter 2 times - Rolling Ball - Gaussian Filter" which are the basic operations applied in order to estimate the charging contribution to the input image, $I_{charged}$. Once that $I_{charged}$ has been estimate, it is simply subtracted to the input image, obtaining the decharged image $I_{decharged}$. More precisely, let $G[\sigma]$ be a gaussian kernel with standard deviation σ, and let $*$ denote the usual 2d convolution operator. Let $RB[r]$ denote the *rolling ball* algorithm [Sternberg1983] with radius parameter r, an algorithm used to estimate the background in an image. Then the GF2RLGF charging correction procedure can be summarized as follow:

$$\begin{cases} I_{HF_1}(j,i) & = I(j,i) - (G[\sigma_{GF_1}] * I)(j,i) \\ I_{HF_2}(j,i) & = I_{HF_1}(j,i) - (G[\sigma_{GF_2}] * I_{HF_1})(j,i) \\ I_{charged}(j,i) & = G[\sigma_{GF_3}] * RB[r](I_{HF_2})(j,i) \\ I_{decharged}(j,i) & = I(j,i) - I_{charged}(j,i). \end{cases}$$

The procedure is simple. In the first step the slowly varying part of the image (low spatial frequencies) are subtracted to the input image, following the same strategy adopted for the *Flatter* plugin. This procedure is iterated two times. At the end of this process the image I_{HF_2} should contains only the high frequency details, i.e. the small structure in the image *plus* most of the charging artifact. However, the charging artifact is still expected to vary more slowly with respect to the small structure of the image: as such it can be considered as a background. This background can be estimated by applying the rolling ball algorithm with a suitable radius. The output of the rolling ball algorithm may have to much high frequency details in it, therefore a further gaussian smoothing with small standard deviation is applied. The result obtained is $I_{charged}$, which is then subtracted to the input image. The whole procedure described here will be denoted with he symbol $gDECH[\sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}]$.

Summarizing, given a stack $S(j,k,i)$ the global GF2RBGF decharger is applied to each slice $S[k](j,i)$ as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = gDECH[\sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}](S[k](j,i))$$

Note: Note that with this procedure it is assumed that the charging is present in any pixel of the image: that is why the correction algorithm is said 'global'.

13.4.2 Local GF2RBGF

In the *local GF2RBGF* the correction algorithm above is not applied to the whole image: the charging contribution is subtracted only locally, where charging is present.

In order to do that a mask, indicating where the charging is, is estimated from the input image. Proceeding similarly to [Spehner2020], the estimation is done by using the downhill filter [Robinson2004] followed by a thresholding operation. To proceed with the morphological reconstruction done with the downhill filter, one needs to define a seed image: this is obtained from the original image whose color are shifted down of a value c_{shift}, and then setting every pixel value equal to the minimum pixel value of the image *except* at the image borders, which are left unchanged. The threshold operation is done after the downhill filter to select the parts of the reconstructed image with the highest brightness. Regions found in this way are filtered according to their area: only the regions having area above a threshold A_{th} are considered as region with charging. The operations briefly described here will be all denoted with the symbol $DHF[c_{shift}, A_{th}]$. To better estimate the charged image it is better, to apply a low pass filter to remove the slowly varying part of the image from the charging estimation. Summarizing, let $M(j,i)$ be the mask containing the estimated charged region, then

$$\begin{cases} I_{HF}(j,i) & = I(j,i) - (G[\sigma_{GF_1}] * I)(j,i) \\ M(j,i) & = DHF[c_{shift}, A_{th}](I_{HF})(j,i). \end{cases}$$

The mask $M(j,i)$ obtained clearly depends on the input image I and will we useful also for the optimization. Once that the mask is obtained, the estimation of the charged image proceed exactly as for the global GF2RBGF correction, except for a difference in the last step, i.e. when the estimated $I_{charded}$ is subtracted from the input image. In the local GF2RBGF correction the following formula is used

$$I_{decharged}(j,i) = [1 - M(j,i) - \partial M(j,i)] \cdot I(j,i) - [M(j,i) + \partial M(j,i)] \cdot I_{charged}(j,i),$$

where ∂M is obtained from the mask M and is zero everywhere expect in external border region of M, where the values are increased linearly form 0 to 1 as one moves from the most external regions outside M to M itself. ∂M is used to smoothly pass form the corrected to the uncorrected parts of the input image. The whole procedure described here will be denoted with he symbol $lDECH[c_{shift}, A_{th}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}]$. This kind of correction method, try to correct charging by removing the less is possible, but it may be slow.

Summarizing, given a stack $S(j,k,i)$ the local GF2RBGF decharger is applied to each slice $S[k](j,i)$ as follow

$$S[k](j,i) \rightarrow S_{output}[k](j,i) = lDECH[c_{shift}, A_{th}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}](S[k])(j,i).$$

In both versions, in case of stack with multiple channels, the Decharger is applied independently to each channel.

Optimization details

The optimization is possible only for the local method, since only in this case one can get an estimation of the typical value one would have in the images without charging. Given the mask $M(j,i)$, one can decompose it into connected regions. Let Q be one of these connected region, then one defines the region Q^s by dilating s times the mask Q, and then subtract (in set theoretic sense) Q itself to the dilation result, i.e.

$$Q^s = \text{ dilate } (Q,s) - Q$$

This region is interesting because, if it has no superposition with any non-null part of M, this can be considered as a region without charging close to Q itself, therefore is can be a good guess on how the charged region Q would look like if no charged is present. Therefore one can compare the two regions in order to find the best parameter combination for the decharger as the ones which make the two regions closest as possible. The 'closeness' of the two regions need to be defined in a proper manner. Charging is expected to shift up (or down in the case of inverse charging) the brightness of the image. This should be visible at the level of the "local" image histograms, i.e. the histograms made with the gray levels of the pixels belonging to the two regions Q and Q^s.

Let hist $[I,Q]$ be the *normalized* histogram constructed using the value of the pixels of the image I in the region Q only. It is expected that charging simply shift the (normalized) histogram (up or down depends on the kind of charging). Therefore, the decharger should reduce/eliminate this shift making the normalized histograms in Q and Q^s closer. Using the total variation distance between discrete probability distributions to measure the closeness of the two normalized histograms, the loss below can be defined

$$\mathcal{L}[\alpha](I,Q,Q^s) = \sum_b |\text{hist } [lDECH[\alpha](I),Q]\,(b) - \text{hist } [lDECH[\alpha](I),Q^s]\,(b)|,$$

where $\alpha = (c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3})$ are the optimizable parameters of the decharger and the sum is over the normalized histograms bins.

Note: Note that the parameter A_{th} is not present in the loss and no optimization procedure is available for it.

Given a $K \times J \times I$ stack $S(k,j,i)$ take a collection of its slices $\{S[k](j,i)\}_{k\in V_K}$ where $V_K \subset \{0,1,\cdots,K-1\}$ (i.e. the subset of slices selected for the optimization), then the decharger optimization routine solve the following problem

$$c_{shift}^{best}, \sigma_{GF_1}^{best}, \sigma_{GF_2}^{best}, r^{best}, \sigma_{GF_3}^{best} = \text{argmin}_\alpha \left(\sum_{k\in V_K} \sum_{Q_k} \mathcal{L}[\alpha](S[k], Q_k) \right)$$

The solution is obtained via a simple grid search over a given parameters space. To speed up this operation, the sum over Q_k is not done over all the connected components of the mask M_k, but over a subset of the first N connected component having the biggest area.

13.5 Further details

Tutorials:

- skimage tutorial on rolling ball algorithm.
- skimage tutorial on downhill filter algorithm.
- *Case of study: decharger optimization*.

Articles:

CHAPTER

FOURTEEN

REGISTRATOR

Registrator in a nutshell.

1. Plugin to align among each other the slices of a stack;
2. This plugin is **not** multichannel;
3. Python API reference: `bmiptools.transformation.alignment.registrator.Registrator`.

This plugin can be used to align among each other the slices of the input stack, in order to get a proper 3d reconstruction. The registration procedure of this plugin consist in at most two steps:

1. *rigid registration*, where a global affine transformations is applied to each slice to match it geometrically with the next. The rigid registration can happen in two ways: either via ECC based matching algorithm [Evangelidis2008] (slower but it is more precise in principle) or via phase correlation based matching algorithm [Reddy1996] (faster but it may be less precise).
2. *non-rigid registration*, which is used to eventually refine the result of the previous step, where a pixelwise transformation is applied to each pixel of a slice to match it with the corresponding pixel in the next slice (keeping into account the pixel surrounding) based on optical-flow-based matching [LeBesnerais2005]. This step is optional.

The Python API reference of the plugin is `bmiptools.transformation.alignment.registrator.Registrator`.

14.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'load_existing_registration': False,
'loading_path': ' ',
'registration_algorithm':'ECC',
'padding_val': 0,
'destandardize': True,
'template_lh_boundary_factor': 1,
'template_rb_boundary_factor': 1,
'ECC_registration_setting': {'n_iterations': 5000,
                             'termination_eps': 1e-10,
                             'motion_model':'Translation',
                             'ecc_threshold': 0.7},
'phase_correlation_registration_setting': {'motion_model': 'Translation',
                                           'phase_corr_threshold': 0.4,
```

(continues on next page)

(continued from previous page)

```
                                                },
'opt_bounding_box': {'use_bounding_box': True,
                     'y_limits_bbox': [-500,None],
                     'x_limits_bbox': [500,1500]},
'refine_with_optical_flow': False,
'OF_setting':{'optical_flow_attachment': 5,
              'save_mod_OF': False,
              'mod_OF_saving_path': ''},
'save_fitted_registration': False,
'saving_path': ' '
}
```

The plugin-specific parameters contained in this dictionary are:

- `load_existing_registration`: if `True` an existing registration produced by this plugin is loaded and all the other fields below except `loading_path` are ignored.
- `loading_path`: contains the path to the files containing the existing registration parameters.
- `registration_algorithm`: field where one has to specify the algorithm used to perform the *rigid* registration. The possible options are:
  - `'ECC'`
  - `'Phase_correlation'`
- `padding_val`: it is the value used for padding the images in order to reach a certain shape during the application of the registration.
- `destandardize`: when True at the end of the registration ,the image is destandardized (image standardization take place before the optimization of the registration algorithm and is done according to the 0/1 mode of the *Standardizer* plugin).
- `ECC_registration_setting`: contains the setting for the ECC registration algorithm. It is a dictionary and has to be specified as below:
  - `n_iterations`, is number of iterations used for the maximization of the ECC loss.
  - `termination_eps`, is epsilon value used to determine the convergence of the ECC maximization algorithm: if the difference between the ECC values after two iterations is less then this value, then the maximization stops.
  - `motion_model`, is the kind of motion model used for the estimation of the parameters used for the registration, and can be equal to
    - `'Translation'`;
    - `'Euclidean'` (i.e. rotation + translation);
    - `'Affine'`.
  - `template_lh_boundary_factor`, it is the left/high boundary factor for the template window definition (recommended value: 1)
  - `template_rb_boundary_factor`, it is right/bottom boundary factor for the template window definition (recommended value: 1)
  - `ecc_threshold`, is the threshold on the ECC value at the end of the maximization procedure below which a two steps estimation procedure for the estimation of the transformation parameters is run.
- `phase_correlation_registration_setting`: contains the setting for the registration algorithm based on the phase correlation techniques. It is a dictionary having the keys below:

  - `motion_model`, *do not change. currently only* `'Translation'` *is possible*.
  - `phase_corr_threshold`, is threshold on the normalized correlation below which the two steps registration optimization is executed.
- `refine_with_optical_flow`: if `True` a final refinement with optical flow registration is applied at after that the rigid registration has been applied on the stack.
- `OF_setting`: is a dictionary containing the setting of the optical flow registration. This dictionary has to be specified as follow:
  - `optical_flow_attachment`, it is the attachment parameter of the optical flow registration algorithm.
  - `save_mod_OF`, if `True` the modulus of the optical flow field is saved.
  - `mod_OF_saving_path`, is the path where the modulus of the optical flow registration field is saved. If the above field is False this field is ignored.
- `save_fitted_registration`: if `True` the parameters estimated for the rigid-registration are saved.
- `saving_path`: is the path where the registration parameters are saved.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Registrator`.

14.2 Use case

The typical use of this plugin are:

1. Align the slices of a stack in order to get a proper 3d reconstruction.

Tip: The following things turn out to be useful, from a practical point of view.

1. Using the bounding box the the time necessary for the estimation of the rigid registration can be reduced. By the way the time for the refined optical flow registration (if used to refine the result), does not change.
2. The parameter describing the *rigid registration* can be saved and loaded in a later time, allowing the registration of a stack by using the parameter estimated using a different stack. This may be particularly useful in case one has to register two stack which are produced at the time from the same sample but with different imaging techniques.
3. It can be reasonable to fit this plugin in the beginning of the image processing pipeline but *apply it only at the end*. This because this plugin change the dimension of the images, making the stack bigger. This means that more RAM memory is required and the overall computation is increased. Therefore, *if there is no need to use the 3-dimensional information for the application of a plugin*, is a good strategy. Note that practically all the image processing plugin in bmiptool act on the slices directly, and works without the need of information coming from (the local structures of) the other slices.
4. The use of the `'Translation'` motion model advised for the rigid-registration step.
5. To achieve a non-rigid registration, the a rigid registration is always done before. This is done because the non-rigid registration is not good in estimating big translations, which can be vary easily estimated with rigid methods. Since the non-rigid method is particularly slow and is used to refine the registration, it is recommended to use `Phase_correlation` which is fast and enough to get a good result.
6. In the `ECC_registration_setting`, if there are no particular reasons, `template_lh_boundary_factor` and `template_rb_boundary_factor` should not be changed.

14.3 Application example

As example consider the portion of a FIB-SEM stack of a biological sample, visualized as animated gif (saving mode available in the python API, see `bmiptools.stack.Stack.save_as_gif()`), in order to get the feeling of the 3-dimensional structure of the sample. Before any registration the stack look like below.

A registration based on `Phase_correlation` only and using practically all the default parameters of the plugin (i.e. the ones in the `empty_transformation_dictionary`, but disabling the use of the bounding box only, due to the small dimension of the image) would give the result below.

When after the `Phase_correlation` step, also the refinement with optical flow is applied (i.e. setting `refine_with_optical_flow = True` in the transformation dictionary of the plugin) would give the result below.

Note that after the optical flow refinement, the central structure in the top-right corner of the image changes more smoothly from one slice to the other. The other points of the slices are practically unchanged, showing the local nature of the refinement step of the registrator plugin.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

14.4 Implementation details

A registration algorithm in general takes as input two things: an input image and a reference image. The reference image is the reference on which the input image is aligned. The registration algorithms in this plugin consist essentially in two steps:

1. In the first step a map between the pixel positions of the input image and the pixel position is established. This map can be the same for all pixels (in case of rigid registration) or be pixel dependent (in the non-rigid case). These maps are parametrized in a suitable manner, and the best parameters for this map are found by various optimization techniques.
2. In the second step the input image is evaluated on the new pixel position obtained from the map derived in before. This evaluation is done by computing the pixel value in the new position via interpolation.

In what follows $\phi[\alpha]$ will denote the registration procedure described by the two steps above, where α are the registration parameters. For the rigid registration, implemented in this plugin it may take the following form

- for translation, $\phi[\alpha](I(x)) = I(x + t)$, where t is a suitable translation vector.
- for a generic euclidean transformation, $\phi[\alpha](I(x)) = I(Rx + t)$, where t and R is a suitable translation vector and rotation matrix respectively.
- for a generic affine transformation, $\phi[\alpha](I(x)) = I(Ax + t)$, where t and A is a suitable translation vector and (invertible) matrix respectively.

For non-rigid transformations one can formally write $\phi[\alpha](I(x)) = I(f_\alpha(x))$, where f_α is a suitable parameterizable function which may depends also on other image pixels and not only on x.

Given a $K \times J \times I$ stack $S(k, j, i)$ for each slice $S[k](j, i)$ the output of the registration algorithm is given as follow

$$S[k](j, i) \rightarrow S'_{output}[k](j', i') = \phi[\alpha](S[k](j, i))$$

where $S'_{output}[k](j', i')$ is the k-th slice of the $K \times J' \times I'$ output stack $S'_{output}(k, j', i')$. The input and output stack have different size: indeed this plugin change the image dimension in order to accommodate the image "movements" due to the registration.

14.4.1 ECC

The ECC (Enhanced Correlation Coefficient) registration algorithm [Evangelidis2008] is a gradient-based registration technique. Given an input image $I(x)$ and a reference image $I_{ref}(x)$, the parameters of $\phi[\alpha]$ are found by minimizing the following differentiable loss

$$\mathcal{L}_{ECC}(\alpha) = \sum_x \| \frac{I_{ref}(x)}{\|I_{ref}(x)\|} - \frac{\phi[\alpha](I(x))}{\|\phi[\alpha](I(x))\|} \|^2$$

where $\| \cdot \|$ is the euclidean norm. The implementation used in this plugin is based on the one available in openCV.

Attention: If the optimization using this loss fails for a pair of slices in a stack (i.e. if the value of the loss function remains above a certain threshold ECC_{th}), a two step estimation is done: a first translation is estimated using the Phase correlation routine on a smaller part of the image, and later a refined estimation is done using again the ECC on the whole image. If the second estimation is still unsuccessful, only the result of the first step is used.

14.4.2 Phase correlation

The phase correlation registration algorithm [Reddy1996] simply compute the normalized cross-correlation between the input image and the reference one. Since the cross-correlation can be computed with a simple multiplication in Fourier space, by using Fourier shift theorem, if the input and reference image are linked by a rigid translation,the cross-correlation would be just a phase factor. When transformed back to the real space, the cross-correlation corresponds to a delta function centered in a certain point p. The vector linking p with the center of the image correspond to the translation linking the two images.

In this plugin only translation vectors can be estimated with this method, despite in principle it is possible to estimate also rotation angles by means of a change of coordinates. The implementation of this algorithm used in this plugin is the one of openCV

Attention: If the optimization using this loss fails for a pair of slices in a stack (i.e. if the value of normalized cross-correlation remains above a certain threshold PC_{th}), a two step estimation is done: a first translation is estimated using the Phase correlation routine on a smaller part of the image, and later a refined estimation is done using again the Phase correlation routine on the whole image. If the second estimation is still unsuccessful, only the result of the first step is used.

14.4.3 Optical flow

The Lucas-Kanade optical flow registration algorithm [LeBesnerais2005] is another gradient based optimization techniques allowing for non-rigid registration. It is based on the minimization of the following loss function

$$\mathcal{L}_{OF}(\alpha) = \sum_{(i,j)} \sum_{(j',i') \in G(j,i)} \left[I_{ref}(j',i') - \tilde{I}(j' + f_j[\alpha](j',i'), i' + f_i[\alpha](j',i')) \right]^2$$

where $\tilde{I}$ is the interpolated version of the input image I, so that it can be evaluated on a generic point $(j + f_j[\alpha](j,i), i + f_i[\alpha](j,i))$, which does not lie necessarily on the pixel grid. The non rigid character of this algorithm lies in the second sum, since $G(j,i)$ is a patch centred in (j,i). The goal of the optimization problem is to find the parameters α defining the *optical flow vector field*, which tells how the position of the pixels in the input image and the one in the reference image are mapped. In particular, this quantity can be useful

$$\begin{pmatrix} j_{input} \\ i_{input} \end{pmatrix} = \begin{pmatrix} j_{ref} + f_\alpha(j_{ref}, i_{ref}) \\ i_{ref} + f_\alpha(j_{ref}, i_{ref}) \end{pmatrix}.$$

The two components of the optical flow field, $f_\alpha(j_{ref}, i_{ref})$ and $f_\alpha(j_{ref}, i_{ref})$, can be thought as two 2d images where the "movements" between the input and the reference image can be visualized. The magnitude of these "movements" can be summarized in the image containing the magnitude of the optical flow field.

This plugin uses the skimage implementation of this algorithm.

14.5 Further details

Websites:

- Affine transformation on wikipedia.
- Phase correlation on wikipedia
- Optical flow on wikipedia

Articles:

CHAPTER

FIFTEEN

AFFINE

Affine in a nutshell.

1. Plugin to apply affine 3D geometric transformation to a stack;
2. This plugin is **not** multichannel: the current implementation works only for single channel stack;
3. This plugin uses the XYZ convention for expressing the coordinates in most of its field;
4. Python API reference: `bmiptools.transformation.geometric.affine.Affine`.

This plugin can be used to apply a generic geometric affine 3D transformation (e.g a rotation,...) on a stack.

The Python API reference of the plugin is `bmiptools.transformation.geometric.affine.Affine`.

15.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'apply': 'translation',
'reference_frame_origin': 'center-yx',
'translation': {'translation_vector': [0,0,0]
                },
'rotation': {'rotation_angle': 0,
             'rotation_axis': [0,0,1]
             },
'scaling': {'scaling_factors': 1
            },
'shear': {'shear_factors': [0,0,0]
          },
'custom': {'affine_transformation_matrix': None
           }
}
```

The plugin-specific parameters contained in this dictionary are:

- `apply`: it is the field where name of the transformation to apply is specified. Currently possible options are:
  - `'translation'`, for translation in 3D space;
  - `'rotation'`, for rotation in 3D space (specified using the angle-axis notation);
  - `'scaling'`, for scaling transformation in 3D space;

  - `'shear'`, for shear transformation in 3D space;
  - `'custom'`, for custom affine transformation, which as to be specified as 4x4 matrix representing the transformation in a projective 3D space.

- `reference_frame_origin`: in this field the reference frame origin for the application of an affine geometric transformation is specified. In this field one have to specify the origin position *using the XYZ convention with respect in the front-top-left corner of the stack* (i.e. the [0,0,0] position in the numpy array containing the stack) using a tuple/list/array, if one wants to specify a precise point. On the other hand, one can also use the options below.
  - `'center'`, to chose the origin of the reference frame is placed in the exact center of the stack;
  - `'center-zy'`, to chose the origin of the reference frame is placed in the exact center of the ZY-plane and with x=0;
  - `'center-zx'`, to chose the origin of the reference frame is placed in the exact center of the ZX-plane and with y=0;
  - `'center-yx'`, to chose the origin of the reference frame is placed in the exact center of the YX-plane and with z=0;

  Attention: The position of the reference frame origin can be any arbitrary point in the stack. The default origin is the one of the reference frame used for the definition of the bounding box. For some transformation (e.g translation) the position of the origin does not affect the result, while for other transformations the origin position may have non trivial effects. For example, the rotation is implemented such that the stack is rotated around the specified axis passing trough the origin: change the origin means to change the point around which the stack is rotated.

- `translation`: contains a dictionary with the parameters for the translation transformation. These parameters are read only when `'apply': 'translation'` is used. The dictionary with the parameters contains the following field:
  - `translation_vector`: is a tuple/list/numpy array containing the translation vector in a 3D space written using the XYZ convention.
- `rotation`: contains a dictionary with the parameters for the translation transformation. These parameters are read only when `'apply': 'rotation'` is used. The rotation is expressed using the axis-angle convention. The dictionary with the parameters contains the following field:
  - `rotation_angle`, where one express the rotation angle in grad;
  - `rotation_axis`, where one express the vector (as tuple/list/numpy array) representing the direction of the rotation axis

  The rotation axis is automatically assumed to pass for the point specified in the `reference_frame_origin` field.
- `scaling`: contains a dictionary with the parameters for the scaling/dilating transformation. These parameters are read only when `'apply': 'scaling'` is used. The dictionary with the parameters contains the following field:
  - `scaling_factor`, is the field where one specify scaling factor of the transformation. If a single number is given, it is assumed the same scaling transformation for each axis. On the other hand, if tuple/list/numpy array with 3 entries is given, it is assumed a different scaling for each axis, using the XYZ-convention.
- `shear`: contains a dictionary with the parameters for the shear transformation. These parameters are read only when `'apply': 'shear'` is used. The dictionary with the parameters contains the following field:

  - `shear_factors`, where the three parameters of a 3D shear transformation are specified in a tuple/list/numpy array using the XYZ-convention.

- `custom`: contains a dictionary with the parameters for a generic affine transformation. These parameters are read only when `'apply': 'custom'` is used. The dictionary with the parameters contains the following field:
  - `affine_transformation_matrix`, which is numpy array containing a 4x4 matrix representing a generic affine transformation. The affine transformation have to be expressed using the *augmented matrix* representation for affine transformations (see here).

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Affine`.

15.2 Use case

The typical use of this plugin are:

1. Apply affine geometric transformations (e.g translations, rotations,...) to the input stack.

Tip: Since the result of an affine transformation is computed from the interpolation of voxels value of a stack, the quality of the interpolated function determine the quality of the final result. A 3d stack which is not aligned along one of their axis, would not produce the best interpolation function. Therefore, it is suggested to use the this plugin only after the stack alignment (for example via the *Registrator* plugin).

15.3 Application example

As example consider the slice of a stack of a biological sample obtained via cryo-FIB-SEM. The stack considered here has been aligned, and below the 50 slices of it are showed in a single gif.

After the application of the Affine plugin to rotate the stack of 3° around the z-axis, and 5° around the x-axis, the result obtained is given below.

The weird behavior at the boundaries is unavoidable: it is due to the fact the the interpolation function in those points is constructed interpolating between the image values at the border of some slice, and the empty part of the image in the next slice, due to the translations necessary for the stack alignment.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

15.4 Implementation details

This plugin rely on the basic transformations implemented in `scipy.ndimage` (see here. Useful technical notes about this kind of transformations can be found *here*.

15.5 Further details

Websites:

- Affine transformation on wikipedia.

Technical notes:

- "NumPy/sciPy recipes for image processing: affine image warping" - Christian Bauckhage.

CHAPTER

SIXTEEN

CROPPER

Cropper in a nutshell.

1. Plugin to crop region of a stack;
2. This plugin is multichannel;
3. Python API reference: `bmiptools.transformation.geometric.cropper.Cropper`.

This plugin can be used to crop a specific region of a stack. When used inside a pipeline, it is better to apply this plugin as soon as possible, in order to reduce the computation resource needed for the application of the other plugins to a stack.

The Python API reference of the plugin is `bmiptools.transformation.geometric.cropper.Cropper`.

16.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'z_range': [None,None],
'y_range': [None,None],
'x_range': [None,None]
}
```

The plugin-specific parameters contained in this dictionary are:

- `z_range`: list specifying the two extrema for the cropping along the Z-direction,
- `y_range`: list specifying the two extrema for the cropping along the Y-direction,
- `x_range`: list specifying the two extrema for the cropping along the X-direction,

The ranges above have to be expressed according to the convention explained *here* (also for the Z-axis). Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Cropper`.

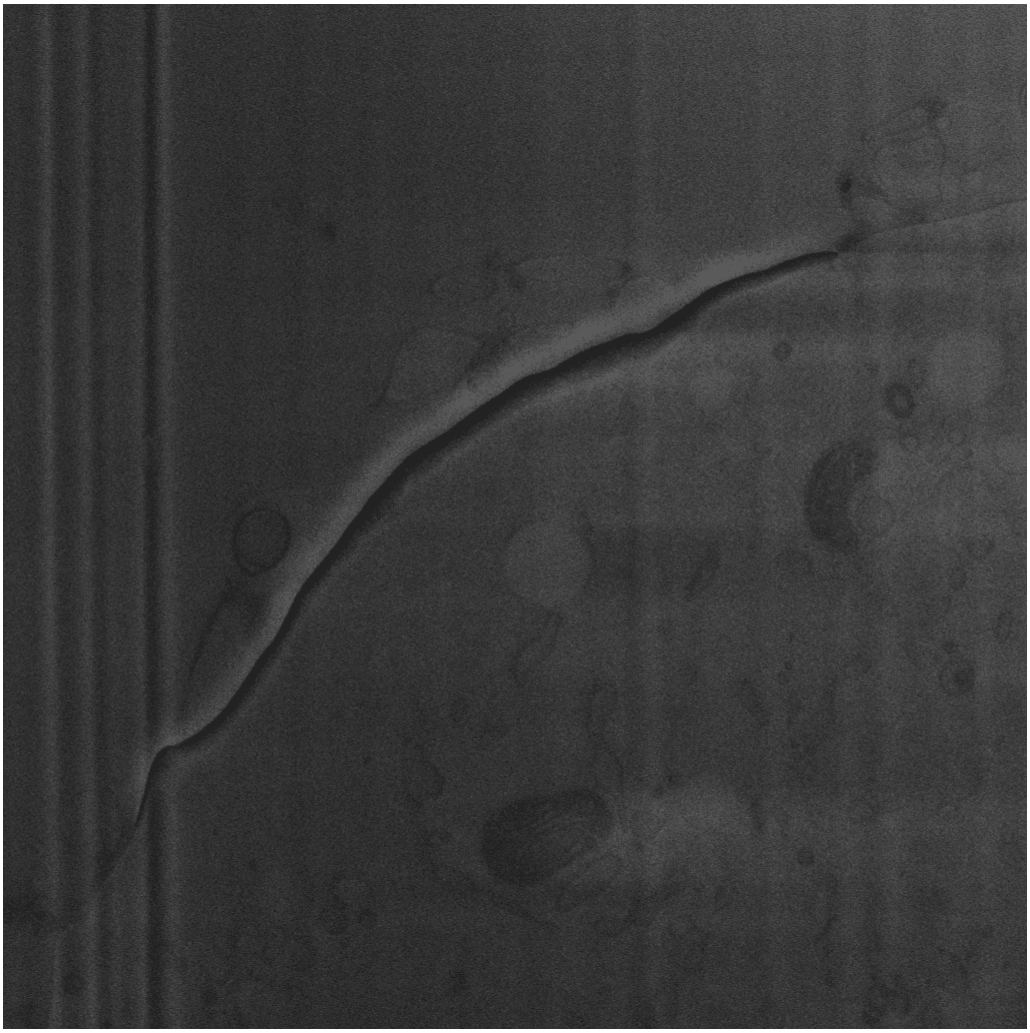
16.2 Use case

The typical use of this plugin are:

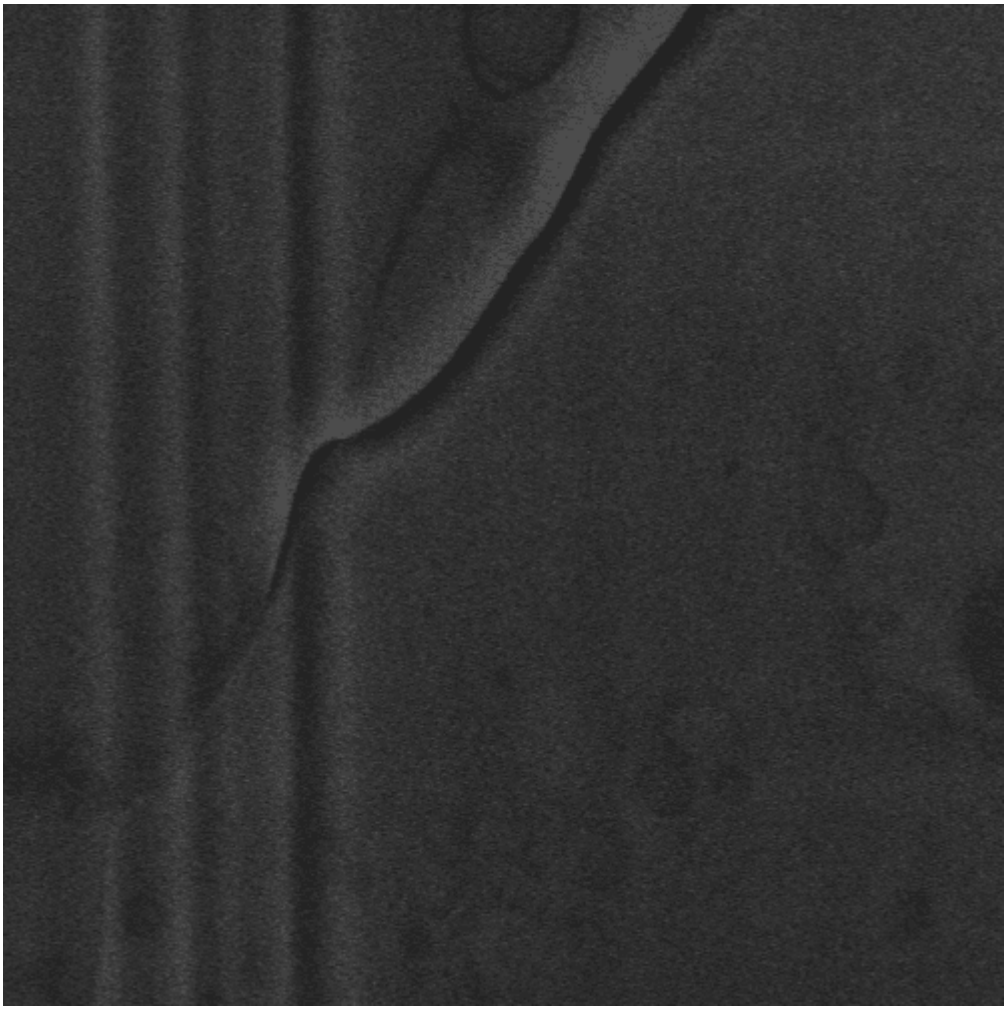
1. Crop part of the input stack.

16.3 Application example

As example consider the slice of a stack of a biological sample obtained via cryo-FIB-SEM.



After cropping a square 500×500 pixels in bottom-left part of the slice, the result obtained is given below.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

16.4 Implementation details

In case of stack with multiple channels, the Cropper is applied independently to each channel.

CHAPTER

SEVENTEEN

EQUALIZER

Equalizer in a nutshell.

1. Plugin apply the CLAHE equalization algorithm to each slice of a stack;
2. This plugin is multichannel;
3. Python API reference: `bmiptools.transformation.dynamics.equalizer.Equalizer`.

This plugin can be used to enhance the contrast in stack using CLAHE algorithm. This plugin uses the skimage implementation of CLAHE, which is applied slice-by-slice to the whole stack.

The Python API reference of the plugin is `bmiptools.transformation.dynamics.equalizer.Equalizer`.

17.1 Transformation dictionary

The transformation dictionary for this plugin look like this.

```
{'kernel_size': None,
'clip_limit': 0.01,
'nbins': 256
}
```

The plugin-specific parameters contained in this dictionary are:

- `kernel_size`: shape of the contextual region around a pixel from which the histogram is constructed. When `None` is given, this parameter is set to the `skimage.exposure.equalize_adapthist()` function.
- `clip_limit`: number between 0 and 1 used as clipping limit.
- `nbin`: number of bins used to construct the histogram for the equalization.

Further details useful the the usage of this plugin with the Python API can be found in the `__init__` method of the class `Equalizer`.

17.2 Use case

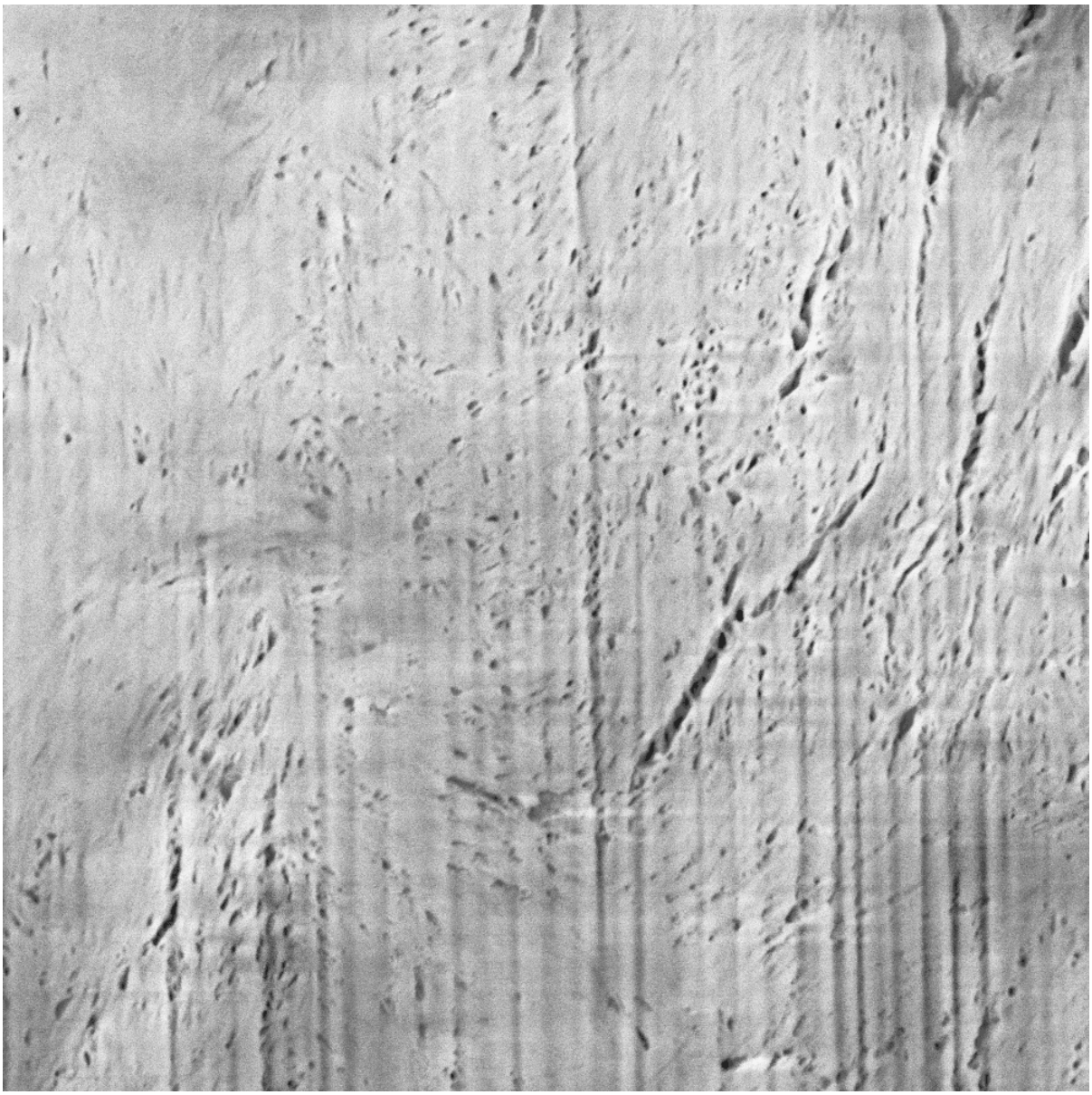
The typical use of this plugin are:

1. Apply then CLAHE equalization algorithm to each slice of the stack, to increase the contrast.

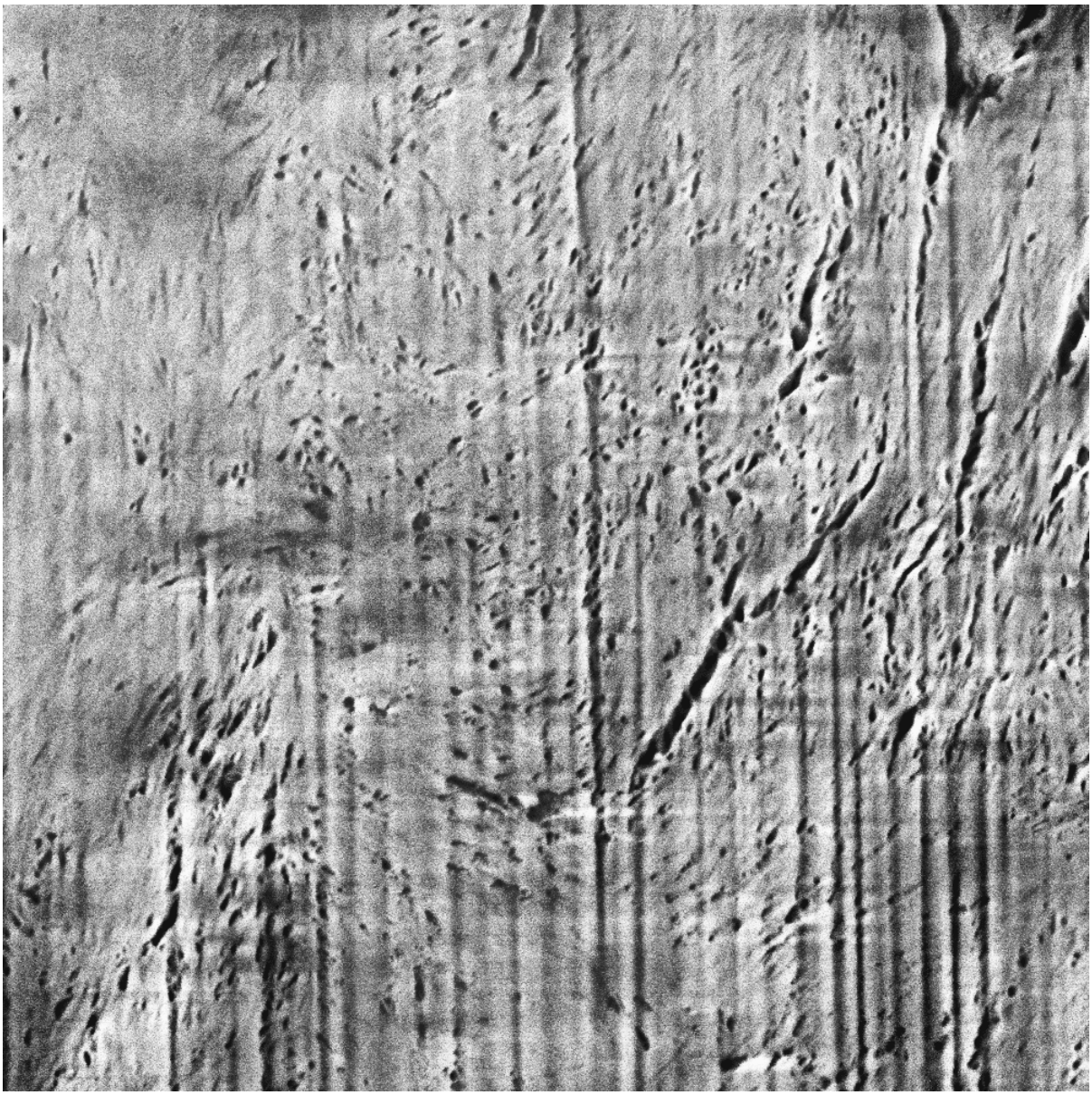
Tip: Equalization may lead to an increase of the noise level present in a image. As such it suggested to apply this plugin after a denoising step (if any), and, more generally, after the removal of all the artifacts in the image if this is possible.

17.3 Application example

As example consider a portion of slice of a stack of a biological sample obtained via FIB-SEM, with low contrast.



Applying the Equalizer plugin with its default parameters (i.e. the one present in the `empty_transformation_dictionary` of the plugin), lead to the high contrast image below.



.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

17.4 Implementation details

This plugin applies the CLAHE algorithm to each slice to the input stack. This plugin is essentially a wrapper around the `skimage.exposure.equalize_adapthist` implementation of CLAHE. The reference for this implementation can be found in the corresponding page of the scikit-image documentaion. In case of stack with multiple channels, the CLAHE equalization algorithm is applied independently to each channel. Note that this behavior is different from what is typically done for standard RGB/RGBA colored images.

17.5 Further details

Websites:

- wikipedia page

Tutorials:

- scikit-image technical notes: local histogram equalization

CHAPTER

EIGHTEEN

BASIC VISUALIZATION TOOLS IN BMIPTOOLS

bmiptools has also *basic* visualization tools based on matplotlib, which can be used for a rapid inspection of the slices of a stack. In particular, there are:

- the `Basic2D` class, for some basic 2d visualization tools;
- the `Basic3D` class, for some basic 3d visualization tools.

About that, the most useful visualisation tools are the 2d ones, whose usage is showed below.

18.1 Basic slice visualization

The code below shows how to use some of them to visualize the slice of the stack `stack`.

```
from bmiptools.visualization.graphic_tools.basic_graphic_tools import Basic2D as b2d

# plot the slice 0
b2d.show_image(stack[0])
```

18.2 Compare two slices

Another useful operation can be the comparison of two slices of a stack. This can be done easily with the code below.

```
# compare slice 0 and slice 1
b2d.compare_images(stack[0],stack[1])
```

18.3 Visualizing image grey-levels as surface

For segmentation purposes, it can be useful to visualize the gray level of a slice as a 3d surface. In bmiptools this can be done as follow.

```
# plot the 2d image of slice 0 as 3d surface
b2d.plot_image_as_surface(stack[0])
```

18.4 Plot masks on slices

Given a mask, it is also possible to visualize it superimposed over the slice of a stack.

```
# plot a mask on the slice 0
mask = ....                                    # numpy array containig the mask
b2d.show_threshold_on_image(stack[0],mark)
```

CHAPTER

NINETEEN

CHANGE COORDINATE SYSTEM

An useful visualization tool in bmiptools is `ChangeCoordinateSystem`, which allows to change the coordinate system on which the stack is visualized. Its structure is similar to a plugin, but it has no transformation dictionary. After the initialization, this tool takes a given stack $S(k, j, i)$ evaluated on a 3d grid with coordinates (k, j, i) and return another stack $S'(r, s, t)$ evaluated on a different 3d grid with coordinates (r, s, t), related to the previous coordinate by a change of reference frame.

Attention: This tool can be used via Python API only.

19.1 Preliminary facts

Given a stack S the value assumed in the voxel (k, j, i) has to be understood as the value of some physical quantity describing the sample in the point $(k\Delta Z, j\Delta Y, i\Delta X)$ of the 3d space, where $(\Delta Z, \Delta Y, \Delta X)$ are the voxel size. All the points (k, j, i) associated to the various voxels of the stack form a 3d grid, which is typically cartesian, i.e. expressed in a cartesian reference frame. For visualization purpose one may want to see the stack using a different coordinate system.

Let (z, y, x) and :(s,t,u): be two coordinate system related by the invertible function f such that

$$(s, t, u) = f(z, y, x)$$
$$(z, y, x) = f^{-1}(s, t, u).$$

Given a $K \times J \times I$ stack $S(k, j, i)$ evaluated in the (z, y, x) coordinates with voxel size given by $(\Delta z, \Delta y, \Delta x)$, it can be evaluated on the (s, t, u) coordinate system by following this procedure.

1. Define the new grid in the (s, t, u) coordinate system. Practically this means to specify two of the following 3 quantities: number of points for each coordinate, coordinate range, and step size for each coordinate. By specifying two of them, the third follow. Indeed:

   - specifying for the coordinate s the number of points N_s and its range $[S_{min}, S_{max}]$, then the step step is given by $\Delta s = (S_{max} - S_{min})/N_s$. Similar considerations hold for the other two coordinates.
   - specifying for the coordinate s the number of points N_s and its step size Δs, then the coordinate range is $[0, N_s\Delta s]$ which can be translated by specifying an offset value. Similar considerations hold for the other two coordinates.
   - specifying for the coordinate s its range $[S_{min}, S_{max}]$ and its step size Δs, then the number of points is given by $N_s = (S_{max} - S_{min})/\Delta s$. Similar considerations hold for the other two coordinates.

   The new grid will contain $N_s \times N_t \times N_u$ points (l, m, n) corresponding to the point of the 3d space having (s, t, u)-coordinates $(l\Delta s, m\Delta t, u\Delta u)$.

2. Interpolate the stack S on the grid defined in the (z, y, x) coordinate system, obtaining the function $\tilde{S}(k, j, i)$.

3. Evaluate the interpolated stack on the new grid, by using the inverse function f^{-1}. More precisely, given a point in the new grid (l, m, n) one computes the corresponding point in the 3d space $(s_l, t_m, u_n) = (l\Delta s, m\Delta t, n\Delta u)$. Then one compute the corresponding point in the grid expressed in the old coordinate system (z, y, x), this can be done first by computing the point

$$(z_{(l,m,n)}, y_{(l,m,n)}, x_{(l,m,n)}) = f^{-1}(l\Delta s, m\Delta t, n\Delta u),$$

and later diving each coordinate by the corresponding step size, obtaining

$$(k_{(l,m,n)}, j_{(l,m,n)}, i_{(l,m,n)}) = \left(\frac{z_{(l,m,n)}}{\Delta z}, \frac{y_{(l,m,n)}}{\Delta y}, \frac{x_{(l,m,n)}}{\Delta x}\right).$$

At this point the stack in the expressed in the new coordinate system S', is defined as

$$S'(l, m, n) = \tilde{S}(k_{(l,m,n)}, j_{(l,m,n)}, i_{(l,m,n)}).$$

Spherical and cylindrical may be particularly useful and will be briefly described below.

19.1.1 Spherical coordinates

The function f mapping the cartesian coordinates (z, y, x) into spherical coordinates (r, θ, ϕ) is defined as follow

$$\begin{pmatrix} r \\ \theta \\ \phi \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2 + z^2} \\ \arccos\left(\frac{z}{r}\right) \\ \text{arctan2}\,(y, x) \end{pmatrix}$$

where arctan2 is the "2-argument arctangent", and the inverse transformation is

$$\begin{pmatrix} z \\ y \\ x \end{pmatrix} = \begin{pmatrix} r\cos(\theta) \\ r\sin(\theta)\sin(\phi) \\ r\sin(\theta)\cos(\phi) \end{pmatrix}$$

In bmiptools f and f^{-1} corresponds to the function `cartesian_to_spherical` and `spherical_to_cartesian`.

19.1.2 Cylindrical coordinates

The function f mapping the cartesian coordinates (z, y, x) into spherical coordinates (r, θ, z) is defined as follow

$$\begin{pmatrix} r \\ \theta \\ z \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \text{arctan2}\,(y, x) \\ z \end{pmatrix}$$

where arctan2 is the "2-argument arctangent", and the inverse transformation is

$$\begin{pmatrix} z \\ y \\ x \end{pmatrix} = \begin{pmatrix} z \\ r\sin(\theta) \\ r\cos(\theta) \end{pmatrix}$$

In bmiptools f and f^{-1} corresponds to the function `cartesian_to_cylindrical` and `cylindrical_to_cartesian`.

19.2 Tool usage

To use this tool the first thing to is to initialize it. During the initialization the following information need to be declared:

- `reference_frame_origin`: it is the position of the origin in the new reference frame, and have to be specified as a list, or tuple, or a numpy array.
- `xyz_to_XYZ_inv_map`: this is a python function with 3 input representing the inverse mapping between the reference frame 'xyz' and the new reference frame 'XYZ' (i.e. is the function f^{-1} *above*).
- `xyz_to_XYZ_specs`: this is a dictionary used to define the new reference frame, which need to have the following structure:
  - `new_shape`, a tuple specifying the shape of the stack in the new reference frame, i.e. the number of points for each new coordinate of the new grid on which the stack will be define after the coordinate change.
  - `X_bounds`, a list with two arguments: the minimum and maximum value of the new X coordinate.
  - `Y_bounds`, a list with two arguments: the minimum and maximum value of the new Y coordinate.
  - `Z_bounds`, a list with two arguments: the minimum and maximum value of the new Z coordinate.
  - `XYZ_ordering`, is used to specify the ordering of the new coordinates. If None the order is the 'ZYX' otherwise on can specify a list with the new ordering, e.g. for 'XYZ' use [2,1,0], for 'YXZ' use [1,2,0], while for 'XZY' use [2,0,1].
- `use_xyz_ordering`: if `True`, the default ordering of the axis in the stack (which is 'zyx') is converted to the cartesian ordering (i.e. 'xyz').

Therefore to initialize this tool the code below can be used, which assume the initial coordinate system as cartesian, the new coordinate system as spherical.

```
import numpy as np
from bmiptools.visualization.geometric.change_coordinate_system import␣
↪ChangeCoordinateSystem
from bmiptools.core.math_utils import spherical_to_cartesian


ccs = ChangeCoordinateSystem(reference_frame_origin = [100,100,100],
                             xyz_to_XYZ_inv_map = spherical_to_cartesian,
                             xyz_to_XYZ_specs = {'new_shape': (100,180,360),
                                                 'X_bounds': [0,100],
                                                 'Y_bounds': [0,np.pi],
                                                 'Z_bounds': [0,2*np.pi]},
                             use_xyz_ordering = True)
```

Once the initialization is done, the application to this tool to a stack `stack` can be done as for the plugin with the method `.transform`.

```
from bmiptools.visualization.graphic_tools.basic_graphic_tools import Basic2d as b2d

# load/ create a stack.
# stack = ...

# See slice 0 before the change of coordinate system.
# b2d.show_image(stack[0])

ccs.transform(stack)
```

(continues on next page)

(continued from previous page)

```
# See slice 0 after the change of coordinate system.
b2d.show_image(stack[0])
```

Attention: Also in this case the `inplace` option is present: when `inplace = True` (default value) the stack content is overwritten with the new stack, while using `inplace = False` the new stack is returned as numpy array.

```
from bmiptools.visualization.graphic_tools.basic_graphic_tools import Basic2d as b2d

# load/ create a stack.
# stack = ...

transformed_stack = ccs.transform(stack,inplace=False)

# Compare slice 0 of the stack and the same slice of the transformed_stack numpy array
b2d.compare_images(stack[0],transformed_stack[0])
```

19.3 Further reading

Technical notes:

- "NumPy/sciPy recipes for image processing: general image warping" - Christian Bauckhage.

CHAPTER

TWENTY

GENERAL CONVENTIONS

For developer it can be useful to have a rough idea about the bmiptools library organization and its basic building blocks, which is briefly described here.

The source code of bmiptools is logically organized as follow:

1. in the *core* folder, all the low level utility functions and classes are collected. The utility functions are roughly divided by scope: there are the gpu related low level functions in `gpu_utils.py`, the image processing related low level functions in `ip_utils.py`, the math related functions in `math_utils.py`, and the other general purpose utility functions are collected in `utils.py`. In `base.py` the basic class used to propagate all the global setting through the bmiptool library is present. *Below* this class is decribed.
2. in the *gui* folder, all the gui related functions and classes are collected. The basic one are in `gui_basic.py`, while classes producing the actual bmiptools gui are collected in `bmiptools_gui.py`. For more about that, see *GuiPI: automatic GUI generation for plugins*.
3. in the *setting* folder, all the functions an methods used to set global feature to the bmiptools library are collected. In the folder `file` can be found the `global_setting.txt` where all the global setting of the library are stored. Always in `file` the list of metadata tags used by the `ExperimentalMetadataInspector` saved in a txt file can be also found.
4. in the *transformation* folder, all the plugin are collected. The plugins are organized according to their scope in different folder (`alignment`, `dynamics`, `geometric`, and `restoration`), while functions and classes which may be useful for all kind of plugins are collected in the folder `basic`. In `base.py` the basic structure of a plugin is contained (see *General plugin structure*, for a code oriented description, and *General information about plugins*, for plugin usage oriented description).
5. in the *visualization* folder, all the part of the library related to visualization tools are collected.

As general rule, bmiptools has been developed by following the Object-Oriented Programming (OOP) paradigm: each tool of the library which can be used by the user, is a Python class. The propagation of global setting to the various components of the library happens via class inheritance.

20.1 The `CoreBasic` and the global settings

In bmiptools the global setting of the library are stored in `global_setting.txt` which can be found in `.\setting\file`. This file is read during the initialization of any object of the library, so that the library behaves according to these setting. Every time a bmiptool onject is imported, the `CoreBasic` class is run, since it is inherited by all the bmiptools objects. This class is reported fully below (to show all its hidden methods) to show basic operations that are executed.

```
class CoreBasic:
    """
    Basic class inherited by all the classes of bmiptools.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self):


        self._global_setting_dict = ut.read_global_setting(bmiptools.__global_setting_
↪path__)
        self._basic_setup()

    def _basic_setup(self):

        # verbosity level
        self.verbosity = self._global_setting_dict['verbosity']

        # multiprocessing
        use_multiprocessing = {0:False,
                               1:True}
        self._use_multiprocessing = use_multiprocessing[self._global_setting_dict['use_
↪multiprocessing']]
        multiprocessing_type = {0:'parallelize_pipeline',
                                1:'parallelize_plugin'}
        self._use_multiprocessing_type = multiprocessing_type[self._global_setting_dict[
↪'multiprocessing_type']]
        self._cpu_buffer = self._global_setting_dict['cpu_buffer']
        self.configure_multiprocessing()

        # gpu optimization
        self._use_gpu = self._global_setting_dict['use_gpu']

    # verbosity controlled i/o methods
    def write(self,x,**kwargs):
        """
        Print the input based to the chosen verbosity level.

        :param x: input to print.
        """
        if self.verbosity == 1:

            print(x,**kwargs)

    def progress_bar(self,i, i_max, bar_length, text_after='', text_before=''):
        """
        Simple and light verbosity controlled progress bar.

        :param i: (int) current index
        :param i_max: (int) max index
        :param bar_length: (int) max length of the progress bar
        :param text_after: (str) text after the progress bar
        :param text_before: (str) text before the progress bar
        """
        if self.verbosity == 1:
```

(continues on next page)

(continued from previous page)

```
            bar = ''
            if text_before != '':

                bar = text_before + ' | '

            bar = bar + '[' + '#' * int(i / i_max * bar_length) + ' ' * (bar_length -␣
↪int(i / i_max * bar_length)) + ']'
            if text_after != '':

                bar = bar + ' | ' + text_after

            print(bar, end='\r')

    def vtqdm(self,x):
        """
        Verbosity controlled tqdm counter for for cycle.

        :param x: iterator
        :return: tqdm(iterator)
        """
        if self.verbosity == 1:

            return tqdm(x)

        else:

            return x

    # multiprocessing methods
    def configure_multiprocessing(self):

        self._n_available_cpu = joblib.cpu_count() - self._cpu_buffer
        if self._n_available_cpu <= 1:

            self._use_multiprocessing = False
            warnings.warn('No multiprocessing possible due to an insufficient number of␣
↪CPUs. Consider to change the'
                          '\'cpu_buffer\' global variable of the library. Execution␣
↪continues in normal mode.')
```

This class is inherited by all the classes of bmiptools and should be inherited by the new ones. Inheritance is what ensure the propagation of the setting present in `global_setting.txt` to any bmiptools object. The methods responsible for that are the

- `__init__`, which read the global setting and call the `_basic_setup()` to interpret it;
- `_basic_setup`, which "translate" the global settings in more user friendly flags;
- `configure_multiprocessing`, which perform basic configuration for the multiprocessing, which is done using the joblib library.

There are also basic functions whose behavior depends on the global setting of bmiptools. This is that case for the methods below.

write(*self*, *x*)

Verbosity controlled print.

progress_bar(*self*, *i*, *i_max*, *bar_length*, *text_after=''*, *text_before=''*)

Verbosity controlled progress bar for standard for cycles (it does not work for the one executed in parallel). The user needs to specify the current iteration index `i`, the max number of iteration `i_max`, and the `bar_length` parameter, which is the leght in character of the progress bar printed. Text can be added before or after the progress bar via the `text_before` and `text_after`, respectively.

vtqdm(*self*, *x*)

Verbosity controlled tqdm. It can be used to have a verbosity controlled progress bar also fo parallelized for cycles.

These methods should be used in any new plugin to print out messages and/or progress bar, so that the verbosity setting in the `global_setting.txt` effectively control the verbosity level of bmiptools.

20.2 General recommendations

The following general recommendations may help in keeping the code consistent and was (more or less) followed during the development of the library.

1. The name of the variables, functions or classes should reflect their scope to help the user to inspect the code. It is recommend to avoid critical names: a long name made of few words, but with a clear meaning, is always better than a short name, which is fast to write for the developer but hard to follow for the rest of the word.
2. Variables names are written with small letters. If more than one word is present in the variable name, it is recommended to use underscores for blank spaces.

   e.g. breaking bad episode -> `breaking_bad_episode`
3. Functions names are written with small letters. If more than one word is present in the function name, it is recommended to use underscores for blank spaces.

   e.g. play episode -> `play_episode()`
4. Classes names begin with capital letter. If more than one word is present in its name, every word start with capital letters and blank spaces are removed.

   e.g. House of cards -> `HouseOfCards`
5. Each function and class should be properly commented. Commenting in a .rst compatible format would speed up the creation of new documentation for the new developed plugins.

CHAPTER

TWENTYONE

GENERAL PLUGIN STRUCTURE

The general structure of a plugin is explained here.The developer creating a new plugin according to the rules explained here, get the following functionalities for free:

1. The plugin can be used in a pipeline in automatic way, i.e. with automatized fitting and application on the input stack of the pipeline.
2. The all parameters of the plugin are tracked during the pipeline application and save when the pipeline is saved. More generally, they can be all stored in a single dictionary with their current value at any time by calling a single command.
3. The GUI for the plugin is created automatically and is integrated with the general bmiptools GUI for free.

Therefore, by using the structure and conventions described here, a new plugin can be fully integrated with bmiptool.

21.1 Basic transformation structure

The prototype plugin is declared in the class `bmiptools.transformation.base.TransformationBasic`, which is copied entirely below. **This class need to be always inherited by any new plugin**.

```
class TransformationBasic(CoreBasic):

    empty_transformation_dictionary = {}
    _guipi_dictionary = {}
    # _undillable_attribute_path = ''
    #
    # P.A. : A global attribute of the class, named '_undillable_attribute_path' need to␣
↪be created
    #        specifying the name of the class attribute used to specify the loading link.
↪ See
    #        DenoiseDNN for an example.
    #
    def __init__(self,*args,**kwargs):
        """
        Initialize here all the parameters of the transformation and execute all the␣
↪setup operations.
        """
        super(TransformationBasic,self).__init__()
        self.fit_enable = True
        pass

    def _setup(self,*args,**kwargs):
```

(continues on next page)

(continued from previous page)

```
        """
        Execute all the setup operations of the transformation. All the operations which␣
↪have to be executed before
        to apply the transformation and does not depend on the Stack object on which␣
↪they are applied, should be placed
        here.
        """
        return None

    def fit(self,x,*args,**kwargs):
        """
        Fit the transformation to the stack on which is applied.

        :param x: (bmiptools.stack.Stack) stack object on which the transformation is␣
↪applied.
        """
        return None

    def transform(self,x,inplace=True,*args,**kwargs):
        """
        Apply the initialized transformation.

        :param x: (bmiptools.stack.Stack) stack object on which the transformation is␣
↪applied.
        :param inplace: (bool) if True the result of the transformation substitute the␣
↪content of the input Stack. When
                        False, the transformation result is returned in the for of numpy␣
↪array and the content of the
                        input Stack is left unchanged.
        """
        return None

    def inverse_transform(self,x,inplace=True,*args,**kwargs):
        """
        Apply the inverse transformation (if possible) on the stack

        :param x: (bmiptools.stack.Stack) stack object on which the inverse␣
↪transformation is applied.
        :param inplace: (bool) if True the result of the transformation substitute the␣
↪content of the input Stack. When
                        False, the transformation result is returned in the for of numpy␣
↪array and the content of the
                        input Stack is left unchanged.
        """
        return None

    def save(self,*args,**kwargs):
        """
        Save the plugin state (or all the necessary information to recover a functional␣
↪plugin state). This method need
        to be implemented ONLY if the plugin contain some "non-pickable"/"non-dillable"␣
↪object. In this case the default
```

(continues on next page)

(continued from previous page)

```
        saving methods of the saving class will not be able to save the plugin state. A␣
↪simple way to check the
        "pickability/dillability" of an object, the code below can be used to check if␣
↪the object f is dillable:

        >>> import dill
        >>> dill.pickles(f)

        It is recommended to make this test with a plugin that has already been␣
↪initialized, (eventually) fitted and
        applied to some stack, so that all the attributes of the plugin has been␣
↪initialized.

        P.A. : In case of undillable plugin, a global attribute of the plugin class,␣
↪named 'undillable_path_attributes'
        need to be created specifying the name of the class attribute used to specify␣
↪the loading link of the undillable
        objects.
        See DenoiseDNN for an example.

        """
        return None

    def get_transformation_dictionary(self,*args,**kwargs):
        """
        Return the transformation dictionary of the plugin filled with the current␣
↪values of the variables of the
        plugin class at the time at which this method is called. The transformation␣
↪dictionary of the plugin has the
        same organization of the 'empty_transformation_dictionary', a (global) attribute␣
↪of the plugin class.
        """
        if hasattr(self.__class__, 'empty_transformation_dictionary'):

            if not self.__class__.empty_transformation_dictionary is None:

                transformation_dictionary = copy(self.__class__.empty_transformation_
↪dictionary)
                key_branches_list = ut.get_branch_of_key_tree(transformation_dictionary)
                for element in key_branches_list:

                    if element[-1] in self.__dict__.keys():

                        ut.set_by_path(transformation_dictionary, element, eval('self.{}
↪'.format(element[-1])))

                return transformation_dictionary

            return None

        else:
```

(continues on next page)

(continued from previous page)

```
            return None
```

As said above, this template class have to be always inherited by the plugin class. In this way all the flags specified in the global setting can be used also in the new plugin. The ones useful for the creation of a plugin may be:

- `use_multiprocessing`, when `True` multiprocessing can be used.
- `plugin_parallelization`, when equal to 1 the parallelization is done at plugin level (currently this flag is not used, it is always equal to 1).
- `use_gpu`, when `True` the gpu can be used (currently this flag is not used, it is always equal to 0).
- `cpu_buffer`, number of cpu not used during the parallelization.

21.1.1 Default global attributes

Every plugin has always two dictionaries by default.

- `empty_transformation_dictionary`: every plugin need to have its own empty transformation dictionary, where all the parameters are initialized with some default value. The structure of this dictionary has to be the same of the `transformation_dictionary`, which is used to initialize the plugin. This for two reasons:
  1. the user can check how the transformation dictionary of the plugin have to look like, *before* the plugin initialization;
  2. the empty transformation dictionary is used as template to register the state of then plugin when the `.get_transformation_dictionary` method is called.
- `_guipi_dictionary`: this dictionary contains the information need to automatically create the GUI out of the python class. In particular, here is where one has to specify which kind of widget is used to create the graphical interface for the input of a given parameter. It has to have exactly the same structure of the transformation dictionary (i.e. the same structure of the empty transformation dictionary), but the value associated to the key corresponding to a given parameter (see below for the convention about these keys) have now a *GuiPI* object specifying the nature of the parameter, from the GUI point of view.

Attention: For plugins which are not dill-compatible, i.e. they cannot be pickled as Python objects using dill the user need to specify a custom saving method (see below) capable to store the status of the Python object in a way that can be loaded in a second moment. In this case an additional global attribute need to be present

- `_undillable_attribute_path`, which contains the name of the key in the transformation dictionary containing the path to the file to be used to load the previous plugin status.

The dill compatibility of a Python object `f` can be checked with the simple code below

```
import dill
dill.pickles(f)
```

21.1.2 Default methods

__init__(*self*, **args*, ***kwargs*)

It is the standard initializer of a class, and it has typically the two input parameters below:

- `transformation_dictionary`: it is a dictionary containing all the input parameters needed to initialize the plugin. It can be a nested dictionary, i.e. a key can have another dictionary as value. Each key of this dictionary is the name of a parameter, except if the value of that key is another dictionary: in this case the keys of the deepest dictionary are the names of the parameters. For more about the transformation dictionary from the plugin usage point of view, see *here*.
- `force_serial`: it is used as flag to force the serial execution of the internal operation of a plugin. It is an optional argument and is not tracked by a `Pipeline` object.

Attention: Since the `TransformationBasic` need to be inherited by any plugin, recall that one always needs to add the `super` function in the `__init__()` of the new plugin. The example below, show that

```
from bmiptools.transformation.base import TransformationBasic


class MyPlugin(TransformationBasic):

      empty_transformation_dictionary = {}
      _guipi_dictionary = {}
      def __init__(self,transformation_dictionary):

          super().__init__(MyPlugin)    # <- this line is always needed.
          ...
```

_setup(*self*, **args*, ***kwargs*)

This method execute all the setup operations of the plugin. The setup operations are all those preliminary operations which can be done by the plugin without the need to have access to some input stack.

fit(*self*, *x*, **args*, ***kwargs*)

This method execute fit the plugin on a given input stack `x`. This fitting operations have to be understood either as the actual optimization routine, or as all those operations which need a stack in order to be done (e.g. get the stack shape, get the slice dimension, ecc…).

transform(*self*, *x*, *inplace=True*, **args*, ***kwargs*)

This method apply the initialized transformation of the plugin on the input stack `x`. The `inplace` flag is used to decide if the result of the transformation is returned as numpy array (`inplace = False`), or the input stack is overwritten with the result (`inplace = True`).

inverse_transform(*self*, *x*, *inplace=True*, **args*, ***kwargs*)

This method apply the inverse of the initialized transformation of the plugin on the input stack `x`. The `inplace` flag is used to decide if the result of the transformation is returned as numpy array (`inplace = False`), or the input stack is overwritten with the result (`inplace = True`).

save(*self*, **args*, ***kwargs*)

This method need to be implemented **only** if the plugin contain some dill-incompatible object, and is used to save the plugin state, or all the necessary information to recover a functional plugin state able to replicate the plugin output.

get_transformation_dictionary(*self*, **args*, ***kwargs*)

This method is used to get the transformation dictionary with the values all the parameter has when it is called.

These methods are the default one, which should be present to integrate a plugin in bmiptools. Other methods can be clearly added if needed, but they will not be called by the other tools in bmiptools. Typically they are used for the internal operations in a plugin.

21.2 Conventions for the plugin construction

The following conventions are also used:

1. **All the parameters of a given plugin need to be specified in the transformation dictionary of the plugin**, which is the main input of the `__init__()` method of the plugin. This transformation dictionary can be a nested dictionary and only the keys at the deepest level corresponds to the parameter name. In the initialization of the plugin, **each parameter need to be assigned to local attribute of the plugin class having exactly the same name of the keys of the (deepest level of the nested) dictionary**. For example, given a plugin with transformation dictionary equal to

```
{'a': 1,
 'b':{'nested_w': 4,
      'nested_v':{'nested2_x': [3,2,1],
                  },
      },
 'c': 'val',
}
```

   has the following parameters: `a`, `nested_w`, `nested2_x`, and `c`. Therefore in the plugin initialization one need to have the following local attributes.

```
...
def __init__(self,transformation_dictionary):

    ...
    self.a = transformation_dictionary['a']
    self.nested_w = transformation_dictionary['b']['nested_w']
    self.nested2_x = transformation_dictionary['b']['nested_v']['nested2_x']
    self.c = transformation_dictionary['c']
    ...

...
```

2. When a new plugin is created, the methods of `TransformationBasic` are overwritten with the one of the new plugin. All except `get_transformation_dictionary` , since it is able to return the update transformation dictionary filled with the current value of the parameters, provided that the point 1. is fulfilled.

3. **The input of a plugin is always a stack**. Therefore in the `fit`, `transform` and `inverse_transform` (when available) one has to use the methods and attributes of the stack class, like the `.data` attribute and the `from_array` method. As example, below is is shown two common step in the `transform` method of practically all the plugins

```
...
def transform(self,x,inplace=False):
```

(continues on next page)

(continued from previous page)

```
    x_to_transform = x.data          # get the stack content and store in a numpy␣
↪array
    ...
    x_transformed = ...              # result of the transformation stored in a␣
↪numpy array
    if not inplace:

        return x_transformed         # return a numpy array

    else:

        x.from_array(x_transformed)  # overwrite the stack content

...
```

4. The new implemented methods of the plugin class should be divided in two groups: *visible* and *hidden*. Visible methods are the ones which can be access to the user normally, while the hidden ones contain the parts of the plugin which are necessary for its correct working, but that does not perform any operation useful to the user in a normal usage scenario.

CHAPTER

TWENTYTWO

GUIPI: AUTOMATIC GUI GENERATION FOR PLUGINS

bmiptools has a system for the automatic GUI creation, which uses magicgui as backend. In bmiptools, each parameter of a plugin corresponds to a widget, which represent the part of the graphical interface responsible for the setting of the parameter values. For each plugins widgets are packed one on top of the other in order to create the plugin gui. At the end a button is always present. Pressing this but the plugin is initialized with the parameters specified in the gui in that moment.

In addition to the native magicgui widgets, in bmiptools custom widgets for specific data types and format has been developed (see `bmiptools.gui.gui_basic`). The association between a given data type and a given widget is regulated by GuiPI.

22.1 GUI Parameters Information, i.e. GuiPI

GuiPI, i.e. Gui Parameter Information, is an object which store information about the parameters relevant for the automatic gui construction. A `GuiPI` object have the following initialization

__init__(*self*, *p_type=None*, *min=None*, *max=None*, *options=None*, *description=None*, *name=None*, *filemode=None*, *visible=True*)

The inputs required for initialize a GuiPI object are:

- `p_type`, which stands for *parameter type*, which determine the selection of the corresponding widget. In addition to the standard python data-type **int**, **bool**, **float**, **str**, and **list**, there are also other useful GuiPI parameters type. They are:
  - `'path'`, to specify a path to a file or a folder;
  - `'options'`, to specify list of objects among which the user can choose;
  - `'range int'`, to specify an integer range object, i.e. integer numbers going from A to B *with a step of C*, where A,B,C integer;
  - `'range float'`, to specify a floating point range object, i.e. floating point numbers going from A to B *with a step of C*, where A,B,C float;
  - `'span int'`, to specify an integer linear span object, i.e. integer numbers going from A to B *in C steps*, with A,B,C integer;
  - `'span float'`, to specify a floating point linear span object, i.e. floating point numbers going from A to B *in C steps*, with A,B,C float;
  - `'math'`, to specify a mathematical object, e.g. vector,matrix,ecc…, and the slicing notation described in the *Bounding box specification convention* subsection;
  - `'table'`, to specify a tabular data. This has to be done as a list-of-list having structure

```
[[key1, value1],[key2,value2],...].
```

- `min`, minimum value the parameter can take. This field is optional and ignored for non-numeric fields;
- `max`, maximum value the parameter can take. This field is optional and ignored for non-numeric fields;
- `options`, when `p_type = 'options'` in this field, the list with the available options for a given parameter is specified. For other values of `p_type` this field is ignored.
- `description`, is an optional brief description of which should appear when the mouse stops on the widget.
- `name`, is the name of the parameter displayed in the widget.
- `filemode`, when `p_type = 'path'` this is the kind of widget for the specification of a path. It can be:
  - `'r'`, to specify one existing file;
  - `'rm'`, to specify one or more existing files;
  - `'w'`, to specify one file name that does not have to exist;
  - `'d'`, to specify one existing directory.

  For other values of `p_type` this field is ignored.
- `visible`, when `False` the widget is not visualized in the final plugin GUI.

To get the corresponding widget one have to call the method

widget(*self*, *name=None*)

This method returns the widget object. The name of the widget can be set by specifying the `name` field (if needed, this is optional). In addition to the widget itself, two functions are returned:

- the setter function, i.e. the function to set the values initially displayed in the widget;
- the reader function, i.e. the function to read the value currently set in the widget, and possibly organize them in a suitable format.

Note: GuiPI objects are used in the `_guipi_dictionary` attribute of the various plugins (see also here).

22.2 Guize

In bmiptools the automatic gui creation of from a class is possible. This is done by means of the guize methods. Two are the available guize methods:

1. `GuizeObject`, which creates a GUI out of a Python class. The parameters displayed in the gui are the input parameter of the `__init__` method.
2. `GuizeObjectFromDict`, which creates a GUI out of a Python class, when the class input parameters are all contained in a dictionary (as in the case of bmiptools plugins). The parameters displayed in the gui are the one contained in the `empty_transformation_dictionary`.

In both cases, the widget can be assigned in two ways:

- the widgets are determined from parameters types using the magicgui-data type conversion rules. In case of bmiptools this does not always give a good result: for this reason GuiPI objects has been developed.
- the widgets are determined from the `_guipi_dictionary` attribute of the input classes.

In both cases, first the method look for a `_guipi_dictionary` and only if it is not found, creates the gui using the standard magicgui-data type conversion rules.

As example of application of these two methods, one can consider the `Stack` class and a plugin, say the the `Destriper`. The `Stack` class has a `_guipi_dictionary`, but it is not initialized via a dictionary. Therefore ones have to use `GuizeObject`, as the example below shows.

```
from bmiptools.stack import Stack
from bmiptools.gui.gui_basic import GuizeObject

stack_gui = GuizeObject(Stack)
stack_gui()                                    # run the gui
```

Note that `GuizeObject` returns a gui object. To actually run the gui, one has to call this object, which is the last line of the code snapshot above. The `Destriper`, being a plugin, needs `GuizeObjectFromDict` in order to me the gui. The code below shows how this is done.

```
from bmiptools.transformation.restorer.destriper import Destriper
from bmiptools.gui.gui_basic import GuizeObject

destriper_gui = GuizeObjectFromDict(Destriper)
destriper_gui()                                    # run the gui
```

Attention: The gui obtained both from `GuizeObject` and from `GuizeObjectFromDict` have always a button 'ok' at the end. No action is performed when one clicks it: one has to connect this button to a suitable function in order to make some action according to the magicgui rules (see here). This button always correspond to the gui object attribute `pbutton`. If the plugin transformation dictionary is nested, for each level of the dictionary a separator contained a series of '###' and the key corresponding to the level is added in the final gui. The whole plugin gui organized as a magicgui container can be found in the attribute `gui` of the gui object produced by the guize methods.

CHAPTER

TWENTYTHREE

PLUGIN INSTALLATION

bmiptools come with a series of plugins. However any user can develop its own plugin, and by following the guidelines described in the section *General plugin structure*, compatibility with the bmiptools ecosystem is ensured. Once that a new plugin has been developed, to integrate it in bmiptools one need to install it. In this way:

- the new plugin will be available in the GUI;
- the new plugin will be available for the creation of pipeline object;
- the new plugin can be imported in a simplified way.

23.1 Add and use a new plugin

The development of a new plugin means that the user created some file `my_plugin.py` where the plugin is contained in a suitable class `MyPlugin`. Assuming that the file `my_plugin.py` can be found at the path `../[...]/my_plugin.py`, to install a plugin it is enough run the two lines of code below.

```
from bmiptools.setting.configure import install_plugin


install_plugin(r'../[...]/my_plugin.py','MyPlugin')
```

The function `install_plugin` takes two arguments: the path to the python script containing plugin class, and the name of the plugin class (which is also considered the name of the plugin). The installation of a plugin is clearly meant to be done just one time for each new plugin: simply add this file to the list of files that are imported when bmiptools is used, making the new plugin always available.

> **Attention:** Since once installed, bmiptools will always assumes to find the plugin file at the path specified at the moment of plugin installation. If this file is moved or eliminated a warning message will be rised and the corresponding plugin cannot be anymore used.

Once that the plugin has been installed, it can be imported in a simplified manner. Indeed the plugin class will be stored both in the `PLUGINS` and in the `LOCAL_PLUGINS` dictionaries. Both of them can be easily imported from `bmiptools.setting.installed_plugins`.

```
from bmiptools.setting.installed_plugins import LOCAL_PLUGINS     # import all local␣
↪plugins

print(LOCAL_PLUGINS)
```

23.2 Remove a plugin

Any *local* plugin can be uninstalled. This can be done with the method `uninstall_plugin` simpy specifying the name of the plugin to uninstall.

```
from bmiptools.setting.configure import uninstall_plugin

uninstall_plugin('MyPlugin')
```

With the code above the uninstalled plugin will not be loaded anymore and therefore removed from the `PLUGINS` and `LOCAL_PLUGINS` dictionary (therefore it cannot be used in a pipeline and in the GUI).

CHAPTER

TWENTYFOUR

CONTRIBUTE OR ISSUE REPORT

The source code of bmiptools and this documentation, and may examples code can be found on the MPIKG GitLab at this address:

- https://gitlab.mpikg.mpg.de/curcuraci/bmiptools .

For questions about bmiptools not addressed in this documentation, one can . An answer will be delivered maybe time permitting as soon as possilble.

24.1 Issue request

To signal bugs, unexpected behavior of bmiptool, or simply ask for a feature request open an issue request on GitLab repository of the bmiptools library. As general rule, for bugs or unexpected behavior it is important to attach a code snapshot able to reproduce the bug, or a detailed description of all the operations executed with the GUI (possibly with a minimal amount of input data), so that one can reproduce the issue. If these conditions are not met, it is difficult to have a positive end for the issue request.

24.2 Integrate custom plugins

To integrate custom plugins or new functionalities in bmiptools, create a new branch on the MPIKG GitLab and update the custom features there. For new plugins, it is always a good idea to install them locally and perform all the tests locally on the developer machine. As series of unit test are available here to test further compatibility. Once a final version of the custom plugin is available, ask for a merge of the branch to the repository administrator.

24.3 To do list

The following list contains possible direction of improvements:

- Implement better optimization strategies. Optimization for all the plugins is done by using simple grid search, and this is sufficient ot get nice results in a reasonable amount of time. By the way more clever optimization methods may reduce the optimization time. A good and not too complicated idea can be to use bayesian optimization, since it is able to deal with continuous, discrete and categorical parameter type during the optimization.
- Multichannel implementation for all the plugins: Registrator, and DenoiseDNN are still single channel. DenoiseDNN can be particularly different with respect to the usual bmiptool procedure to implement a multichannel plugin.
- Implement cycle spin for wavelet denoising.
- Add BM3D denoiser algorithm.

- It should be checked if Noise2Same, rather than Noise2Self, gives rise to better self supervised parameter selection for a denoiser (see here)

CHAPTER

TWENTYFIVE

API REFERENCES

The full bmitools repository is available at

- https://gitlab.mpikg.mpg.de/curcuraci/bmiptools

Here below, most of the bmiptools objects are listed.

bmiptools.stack	I/O methods of bmiptools.
bmiptools.pipeline	A pipeline object manage the creation/initialization/application on a stack of sequence of transformation tools (i.e.
bmiptools.setting.installed_plugins	List of the available installed plugins organized in two dictionaries:
bmiptools.setting.configure	Collect functions for the bmiptools configuration.
bmiptools.core.base	Basic class containing general function which can be useful in any object of the library and possibly depending on the global setting of the library..
bmiptools.core.gpu_utils	Work in progress...
bmiptools.core.ip_utils	Utility functions related to image processing.
bmiptools.core.math_utils	Mathematics related utility functions.
bmiptools.core.utils	Generic utility functions
bmiptools.gui.bmiptools_gui	Main objectus used to render the bmiptools gui
bmiptools.gui.gui_basic	Collection the basic elements which can be useful for the gui creation in bmiptools.
bmiptools.transformation.base	Base class for all the transfomation plugins.
bmiptools.transformation.alignment.registrator	Plugin used to apply registration algorithms to a given stack object.
bmiptools.transformation.basic.filters	Basic filters used in bmiptools.
bmiptools.transformation.dynamics.standardizer	Plugin performing the standardization of a stack.
bmiptools.transformation.dynamics.histogram_matcher	Plugin performing the histogram matching of a stack.
bmiptools.transformation.dynamics.equalizer	Plugin performing the equalization of the slices of a stack.
bmiptools.transformation.geometric.cropper	Plugin performing the cropping of a stack.
bmiptools.transformation.geometric.geometric_tools	Collection of a series of tools useful for geometric transformations.
bmiptools.transformation.geometric.affine	Plugin used to apply affine transformations on a stack object.
bmiptools.transformation.restoration._restoration_shared	Collection of core functions/ variables shared among the restoration plugins.
bmiptools.transformation.restoration.decharger	Plugin to remove/reduce the charging artifact typical of FIB-SEM images.
bmiptools.transformation.restoration.denoiser	Plugins performing denoising on a stack.
bmiptools.transformation.restoration.destriper	Plugin for the removal of the stripe artifacts, typical of FIB-SEM images.
bmiptools.transformation.restoration.flatter	Plugin applying the flatter transformation on a stack.
bmiptools.visualization.geometric.change_coordinate_system	Plugin used to visualize a stack in a different reference frame.
bmiptools.visualization.graphic_tools.basic_graphic_tools	Basic visualization tools for a stack object or slices of it.

CHAPTER

TWENTYSIX

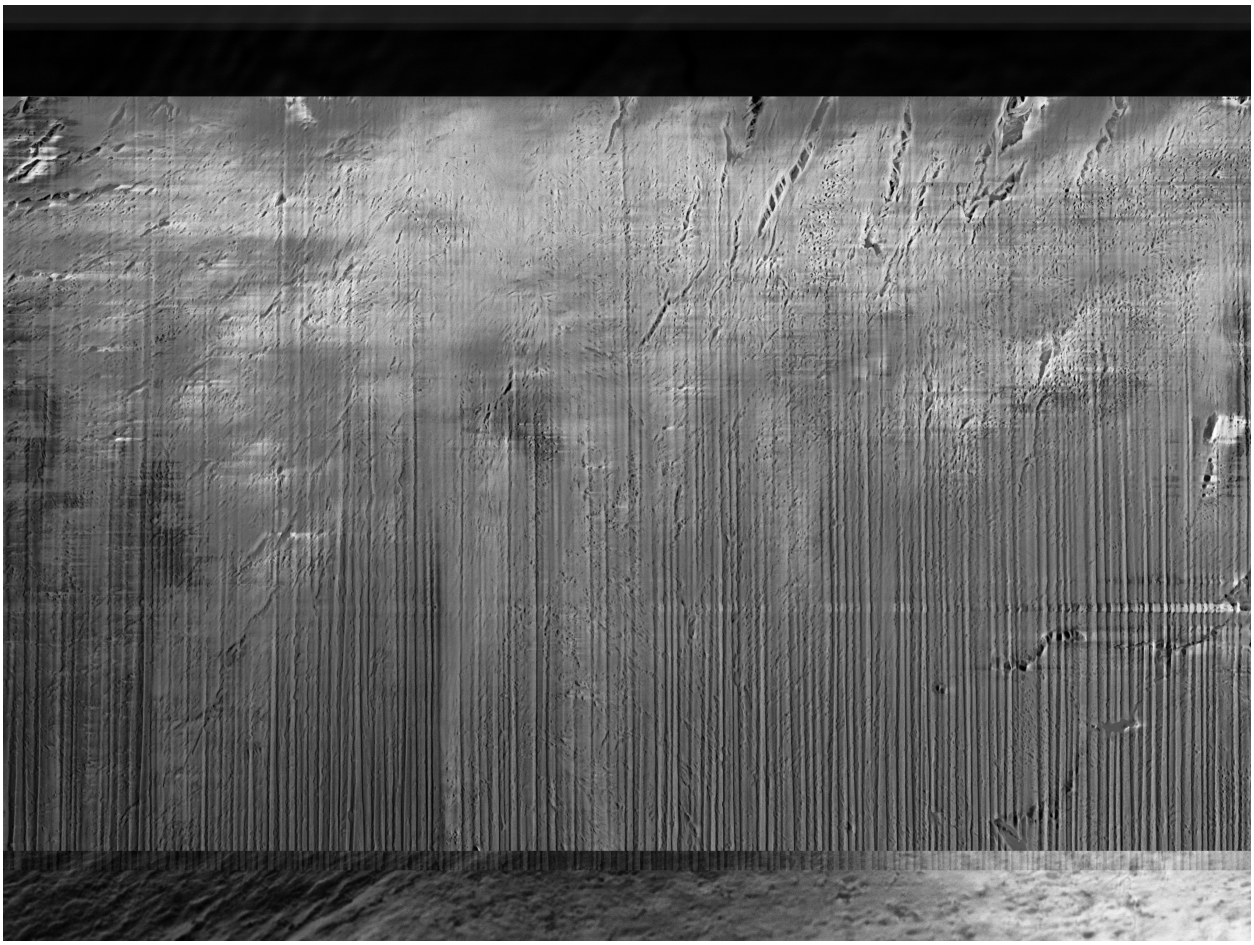
BASIC OPERATIONS WITH STACKS AND PIPELINES

In this tutorial it is showed how to construct a pipeline for a given input stack, by arguing why a certain plugin is used and its position in the pipeline when this is possible.

26.1 The input stack and its problems

The input stack used for this tutorial is showed below.

The first thing to do, is to understand which kind of artifacts are present on these images, and define a pipeline accordingly. Consider the slice below.



Since the interesting part of the sample is in the center, the two horizontal bands on the top and on the bottom of the images should be eliminated. This can be done with the *Cropper* plugin. This should helps for two reasons:

1. First of all, eliminating the uninteresting parts of the stack reduce the overall ammount of computations needed for the artifacts correction, i.e. the time and RAM memory taken to perform the corrections are reduced.
2. Second, the kind of discontinuities between the central part of the sample and the two horizontal bands, may introduce additional artifacts in the border regions, when certain plugins are applied (e.g. the *Destriper* plugin).

Therefore the first plugin in the pipeline in this case should be the *Cropper*.

Many algorithm works better if the pixels of the input images takes value in [0,1]. Moreover may of the default parameters of the bmiptools plugins has been derived with images taking value son the [0,1] range. Therefore at this point it is a good idea to apply the *Standardizer* plugin, to rescale each slice of the stack in the [0,1] range.

After that it is logical to apply all the plugins which may "homogenize the stack along the z-axis" which does not need an optimization procedure. The only plugin available for this scope at the moment is the *HistogramMatcher*. The logic behind this "homogenization along the z-axis" is that the estimation of the parameters of the optimizable plugins should be more robust in this way. By the way keep in mind, that this step can be applied only if no meaningful variation along the z-axis of the stack is expected (which is assumed for this example). Moreover from the animation above, one can clearly see some sudden brightness variation which is typically not expected.

Summarizing according to the motivations above, the next two plugins of the pipeline should be the *Standardizer* followed by the *HistogramMatcher*.

At this point it is reasonable to start to correct the main artifacts of the stack: curtaining and charging. By the way it can be reasonable to estimate now the parameters of the *Registrator* plugin. Indeed, being the vertical stripes real, they

can be very helpful in the estimation of the parameters for the registration. Because of that, at this point of the pipeline the *Registrator* is *fitted*.

After that one can really start to remove the curtaining artifact by means of the *Destriper* plugin. According to the tips which can be found in the plugin documentation page, being the stripes stronger in the bottom part of the image, one should use a bounding box centered on this part of the slice.

After destriping, one can proceed with the reduction of the charging artifact. Therefore at this point the *Decharger* plugin can be applied. After decharging one may flatten the image using the *Flatter* plugin, however here it is assumed that the variation of brightness in the yx-plane still has meaning: as such this plugin is not applied.

The last artifact to remove is the noise. This can be done before applying the registration algorithm provided that only 2d denoising algorithm are applied (otherwise the denoising step has to be applied *after* the application of the registration algorithm). It is a good idea to denoise the stack towards the end of the pipeline since the other steps may "introduce some further noise" or amplify the existing one (a denoiser reduce the noise level but does not eliminate it in general). At this point of the pipeline it is not clear how the noise distribution should look like, therefore it is better to use denoising algorithm which do not require assumptions about the noise structure. The *DenoiserDNN* plugin has this feature, and therefore will be used in this pipeline.

The last operation to do is the application of the *Registrator* plugin, with the parameters estimated before the removal of the curtaining artifact. Possibly the refinement with the optical flow registration can be done. However before to do that, a technical step may be necessary. It is not indeed guaranteed that the range of the images remains between 0 and 1, after the application of the 3 previous plugins. The registration step need to "expand" the input images, in order to accommodate the possible movements. This "expansion" is done by filling the new pixels with some value (0 is the default value for that). By the way if the images may assume negative values, it can be that 0 (assuming to use the default value) is in between the image range, i.e. would correspond to regions of the sample which are not empty. This would give rise to a final result which is not very natural. For that reason it is a good idea to apply again the *Standardizer* (with `standardizer_mode = 0/1`) *before* to apply the *Registrator* plugin. In this way, one is certain that the lowest possible value in the image corresponds to the value used to "expand" the image during the registration.

To conclude, according to all the considerations above, the pipeline that can be deduced from the observation of a slice of the stack and from some additional consideration on the nature of the sample, is the following

```
[Cropper, Standardizer, HistogramMatcher, fit_Registrator, Destriper, Decharger,␣
↪DenoiseDNN,
 Standardizer, Registrator].
```

26.2 Post-processing using bmiptools pipelines

The code below show how to load the stack and apply the pipeline described in the previous section, saving the result obtained at the end. For the details about the various function used and their meaning, the section *Basic API usage* contains all the necessary informations.

```
### Imports

import numpy as np

from bmiptools.stack import Stack
from bmiptools.pipeline import Pipeline
from bmiptools.visualization.graphic_tools.basic_graphic_tools import Basic2D as b2d


### Inputs

```

(continues on next page)

(continued from previous page)

```
# stack related
path_to_sample_stack = ...  # path to the stack to correct
final_stack_path = ...      # saving path of the final stack
final_stack_name = ...      # saving name of the final stack

# pipeline related
pipeline_folder_path = ...  # path to the pipeline working folder
pipeline_name = None        # name of the pipeline (optional)
pipeline_op_list = ['Cropper','Standardizer','HistogramMatcher',
                    'fit_Registrator','Destriper','Decharger',
                    'DenoiseDNN','Standardizer','Registrator']


### Correct stack using a pippeline

# load a sample stack
sample = Stack(path=path_to_sample_stack,load_metadata=True,from_folder=True)

# create a pipeline
pip = Pipeline(operation_list = pipeline_op_list,
               pipeline_folder_path = pipeline_folder_path,
               pipeline_name = pipeline_name)

# initialize the pipeline AFTER specification of the parameters in json of the pipeline
pip.initialize()

# apply the pipeline
pip.apply(sample)

# save the pipeline
pip.save()

# save the final stack
sample.save(saving_path = final_stack_path,
            saving_name = final_stack_name,
            standardized_saving = True,
            data_type = np.uint8,
            mode = 'slice_by_slice')
```

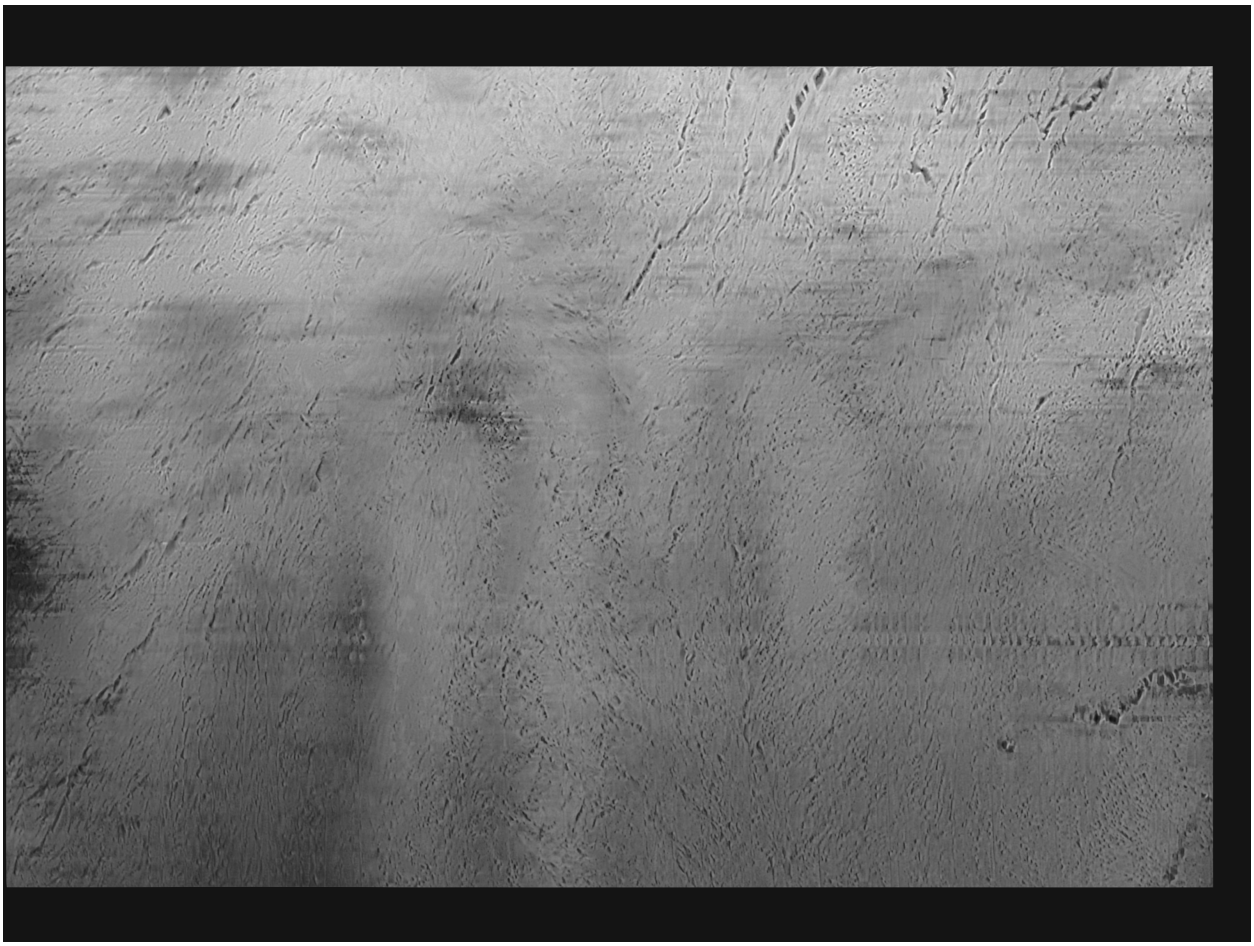
Alternatively one can use the bmiptools GUI. The section *GUI usage* and the videos therein should be sufficient to clarify how this can be done.

The animation below shows how a slice of the input stack changes throughout the pipeline (see *here* to understand how previews can be obtained in the python API, or *here* in the bmiptools GUI).

26.3 Final result

The result of the pipeline described above using the code described in the previous section can be seen in the animation below.

Considering the same slice considered above, the result obtained is the following.



Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

CHAPTER

TWENTYSEVEN

CREATE AND INSTALL A CUSTOM PLUGIN

In this tutorial, it will be showed how a custom plugin can be created and installed in bmiptools. The plugin created will perform the gamma correction of images, and a simple optimization procedure to detect low contrast images and good parameters to enhance it is also sketched. It is not guaranteed that the plugin here perform in a meaningful manner on images, and have to be understood just as an example to illustrate the plugin creation and installation in bmiptools.

27.1 An example of custom plugin

The custom plugin created perform the gamma correction of the images. In particular, given a stack $S(k, j, i)$ it perform the following transformation

$$S(k, j, i) \rightarrow S_{output}(k, j, i) = S(k, j, i)^{\gamma}$$

with $\gamma > 0$. This plugins has also a simple optimization routine. By checking on how the average value of the image compare with respect to the value corresponding to half of the image dynamical range associate to the image type one can decide if the image dynamics need to be stretched (i.e. $\gamma > 1$) or compressed (i.e. $0 < \gamma < 1$). Then the initial value of $\gamma = 1$ is increase or decreased till the output image is not anymore a low contrast image (low contrast images are detected via the `is_low_contrast` function of skimage.exposure). Without optimization, this plugin simply reduce to a wrapper round the `adjust_gamma` function of skimage.exposure

```
import numpy as np

from skimage.exposure import is_low_contrast,adjust_gamma

from bmiptools.transformation.base import TransformationBasic
from bmiptools.gui.gui_basic import GuiPI


class MyPlugin(TransformationBasic):

    empty_transformation_dictionary = {'auto_optimize': True,
                                       'optimization_setting': {'gamma_step': 0.1,
                                                                'gamma_min': 0.1,
                                                                'gamma_max': 3
                                                                },
                                       'gamma': 1}
    _guipi_dictionary = {'auto_optimize': GuiPI(bool),
                         'optimization_setting': {'gamma_step': GuiPI(float,min=0.01,
↪max=1),
                                                  'gamma_min': GuiPI(float,min=0.01,
```

(continues on next page)

(continued from previous page)

```
↪max=1),
                                                    'gamma_max': GuiPI(float,min=1,max=5),
                                                    },
                            'gamma': GuiPI(float,min=1,max=5)}
    def __init__(self,transformation_dictionary):

        super().__init__(TransformationBasic)
        self.fit_enable = True

        self.auto_optimize = transformation_dictionary['auto_optimize']
        if self.auto_optimize:

            self.gamma_step = transformation_dictionary['optimization_setting']['gamma_
↪step']
            self.gamma_min = transformation_dictionary['optimization_setting']['gamma_min
↪']
            self.gamma_max = transformation_dictionary['optimization_setting']['gamma_max
↪']

        self.gamma = transformation_dictionary['gamma']
        self._setup()

    def _setup(self):

        if self.auto_optimize:

            self.gamma_range_above = np.arange(1,self.gamma_max,self.gamma_step)
            self.gamma_range_below = np.arange(1,self.gamma_min,-self.gamma_step)

    def fit(self,x):

        slice_tested = x[0,:,:]
        slice_tested = (slice_tested-slice_tested.min())/(slice_tested.max()-slice_
↪tested.min())
        if np.mean(slice_tested) >= 0.5:

            par_range = self.gamma_range_above

        else:

            par_range = self.gamma_range_below

        parameter_found = False
        for gamma_tested in par_range:

            res = adjust_gamma(slice_tested,gamma=gamma_tested,gain=1)
            if not is_low_contrast(res,fraction_threshold=0.5,lower_percentile=10,upper_
↪percentile=90):

                self.gamma = gamma_tested
                self.fit_enable = False
                parameter_found = True
```

(continues on next page)

(continued from previous page)

```
                break

        if parameter_found == False

            self.gamma = 1
            self.fit_enable = False
            self.write('No gamma value found: gamma = 1 used instead.')

    def transform(self,x,inplace=True):

        if self.fit_enable:

            self.fit(x)

        result = []
        for i in range(len(x)):

            result.append(adjust_gamma(x[i,:,:],gamma=self.gamma,gain=1))

        if not inplace:

            return np.array(result)

        x.from_array(np.array(result))
```

This custom plugin has all the necessary features in order to be fully compatible with bmiptools. They are summarized below

1. The plugin is equipped with the `empty_transformation_dictionary` containing all the plugin setting and it is organized in the standard way (see *here* and *here*).
2. The `_guipi_dictionary` contains all the variable specified in the empty transformation dictionary with the corresponding GuiPI (see *here*).
3. The attribute names are the ones used in the transformation dictionary.
4. The input independent preliminary operations are contained in the `_setup` method, while the optimization routine is written in the `fit` methods. The `fit enable` attribute is correctly created in the `__init__` methods, and disabled when the fitting procedure terminate.
5. The `transform` method is correctly implemented.
6. `TransformationBasic` correctly inherited by the plugin (see *here* for more information).
7. Messages are printed using the verbosity controlled function `write`, so that the bmiptools global setting can silent them.

27.2 Plugin installation

What is left to do is to install the custom plugin. According to the *Plugin installation* section, it is sufficient to create a new python script containing the two lines below

```
from bmiptools.setting.configure import install_plugin


install_plugin(r'../[...]/my_plugin.py','MyPlugin')
```

where here is assumed that script containing the custom plugin `MyPlugin` has (absolute) path `../[...]/my_plugin.py`. It is sufficient to run the created script with the python interpreter (clearly in the same python environment where bmiptools is installed) in order to permanently install the plugin. Once that this is done, one cna check the installation by looking at the `LOCAL_PLUGINs` dictionary, i.e.

```
from bmiptools.setting.installed_plugins import LOCAL_PLUGINS

print(LOCAL_PLUGINS)
```

Warning: Once a plugin is installed, bmiptools will always look for that plugin at the path specified during the installation. As such it is recommend to do change the position of the script of the custom plugins once installed.

CHAPTER

TWENTYEIGHT

CASE OF STUDY: DESTRIPER OPTIMIZATION

Here the optimization routine of the destriper plugin is briefly discussed with a practical example. In the *Implementation details* section of the Destriper plugin was discussed how the filter used by the plugin works, and how it can be optimized.

Note: The following imports are needed to run the code snapshots below.

```
import numpy as np
import pywt

from scipy import fftpack
```

From a practical point of view, the filter $f_{w,l,\sigma}$ used in the destriper plugin can be implemented with the function below.

```
def destripe_slice(x, wavelet_name, sigma, level):
    """
    Core destriper transformation which is applied to a slice of a stack.

    :param x: (ndarray) array containing the image to process.
    :param wavelet_name: (str) name of the wavelet to be used.
    :param sigma: (float) standard deviation of the gaussian filter used to remove␣
↪vertical lines.
    :param level: (int) decomposition level of the wavelet transform
    :return: (ndarray) the destriped image.
    """

    res = pywt.wavedec2(x, wavelet_name, level=level)
    filtred_res = []
    for coeff in res:

        if type(coeff) is tuple:

            cV = coeff[1]
            fft2_cV = fftpack.fftshift(fftpack.fft2(cV))
            size_y, size_x = fft2_cV.shape

            x_hat = (np.arange(-size_y, size_y, 2) + 1) / 2
            filter_transfer_function = -np.expm1(-x_hat ** 2 / (2 * sigma ** 2))
            filter_transfer_function = np.tile(filter_transfer_function, (size_x, 1)).T
            fft2_cV = fft2_cV * filter_transfer_function
```

(continues on next page)

(continued from previous page)

```
            cV = fftpack.ifft2(fftpack.ifftshift(fft2_cV))
            filtred_res.append((coeff[0], np.real(cV), coeff[2]))

        else:

            filtred_res.append(coeff)

    return pywt.waverec2(filtred_res, wavelet_name)
```

This function is exactly equal to the current filter function implementation of the `Destriper` class.

28.1 The optimization routine

The parameter space used for the optimization routine is defined as follow

- all the wavelet available in bmiptools should be tested;
- the values of σ tested are the ones between 0.01 and 50 with a step of 1;
- the decomposition level l is set to be equal to the maximum compatible with the image input shape and the wavelet used.

The function below can be used to generate the parameter space, according to the 3 conditions above. The function takes as argument the input image shape, the list of the possible values of σ, and the list of the possible wavelets names.

```
def generate_destriper_parameter_space(image_shape,sigma_range,wavelet_range):
    """
    Compute all the possible parameters combinations given the individual parameters␣
↪ranges.

    :param image_shape: (tuple) shape of the image used for the loss computation.
    :param sigma_range: (list[float]) list of values of possible sigma.
    :param wavelet_range: (list[str]) list of possible wavelet names.
    :return: (list[list]) the parameter space.
    """
    param_space = []
    for wname in wavelet_range:

        Lmax = pywt.dwtn_max_level(image_shape, wname)
        for sigma in sigma_range:

            param_space.append([wname,sigma,Lmax])

    return param_space
```

The list of all the available wavelets in bmiptools is stored in the variable `SUPPORTED_WAVELET`, and can be obtained as showed below.

```
from bmiptools.transformation import SUPPORTED_WAVELET
```

Assume to apply the optimization routine on the slice `sl` of some stack. In total, with the parameter space boundaries chosen, there are 5300 possible combinations to test.

```
sl =  ... # slice of stack on which the destriper optimization is applied

# define parameter space boundaries
sigma_range = np.arange(0.01,50,1)
wavelet_range = SUPPORTED_WAVELET
image_shape = sl.shape

# generate parameter space
pspace = generate_destriper_parameter_space(image_shape,sigma_range,wavelet_range)
N_param_comb = len(pspace)
print('Total number of parameters combinations: ',N_param_comb)
```

For the optimization, the loss function $\mathcal{L}[w, l, \sigma]$ can be computed with the function below.

```
def self_supervised_decurtaining_loss(x,destriped):
    """
    Self-supervised loss used for the parameter search.

    :param x: (ndarray) input slice;
    :param destriped: (ndarray) destriped slice;
    :return: (float) loss value for the given parameters.
    """
    stripes = x-destriped
    R = np.mean(np.abs(np.gradient(stripes,axis=0)))
    Q = np.mean(np.abs(np.gradient(stripes,axis=1)-np.gradient(x,axis=1)))
    return 2*R+Q
```

The research of the best parameter combination can done by using a simple grid search. The loss value for all the possible combinations of parameters in the parameter space generated, can be computed with the code below.

```
# optimization routine
L = []
for p in pspace:

    # initialize filter with a set of parameters
    filter = lambda x: destripe_slice(x,*p)

    # apply filter
    dest_sl = filter(sl)

    # compute loss
    L_p = self_supervised_decurtaining_loss(sl,dest_sl)
    L.append(L_p)
```

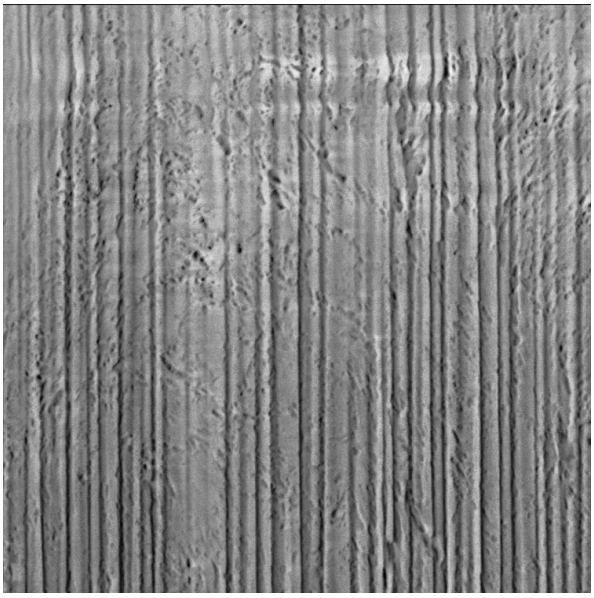
Clearly, the best parameters are the one corresponding to the lowest value of the loss.

```
# print results
best_idx = np.argmin(L)
print('Best parameters: ', pspace[best_idx])
print('Loss value: ', L[best_idx])
```

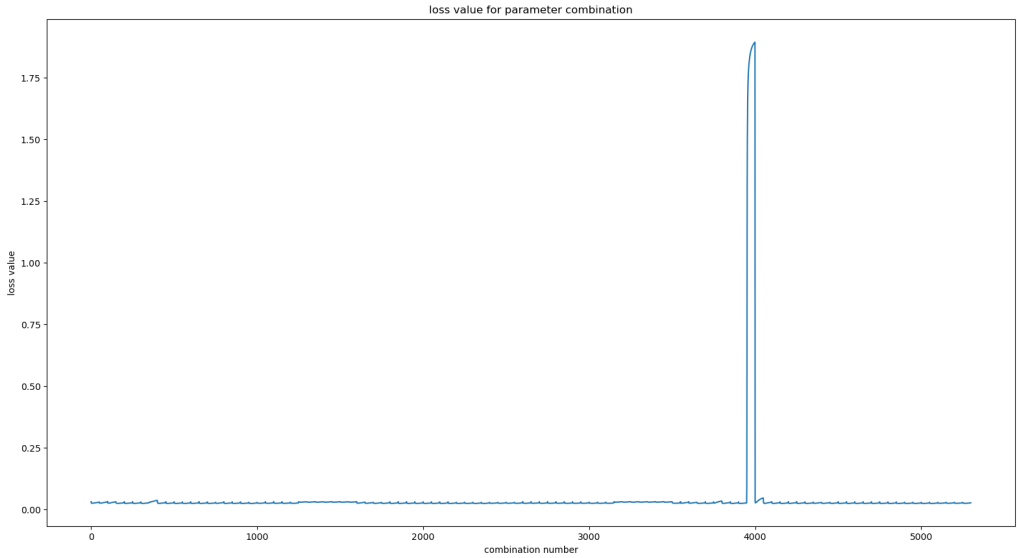
The optimization routine described here, contains all the essential steps which are present in the `Destriper` class.

28.2 Results

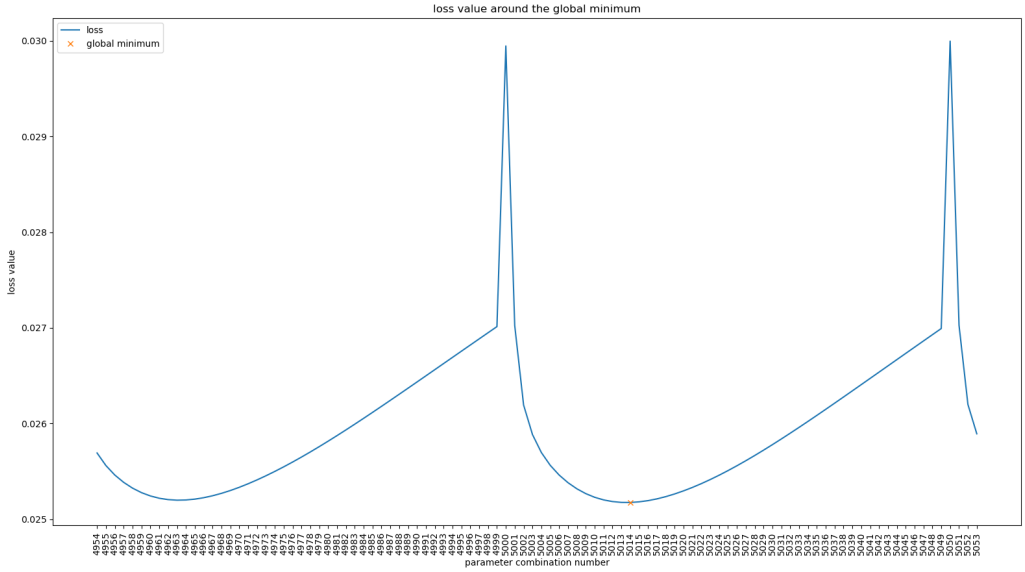
Consider the image below as input for the algorithm. The striping artifacts are clearly visible.



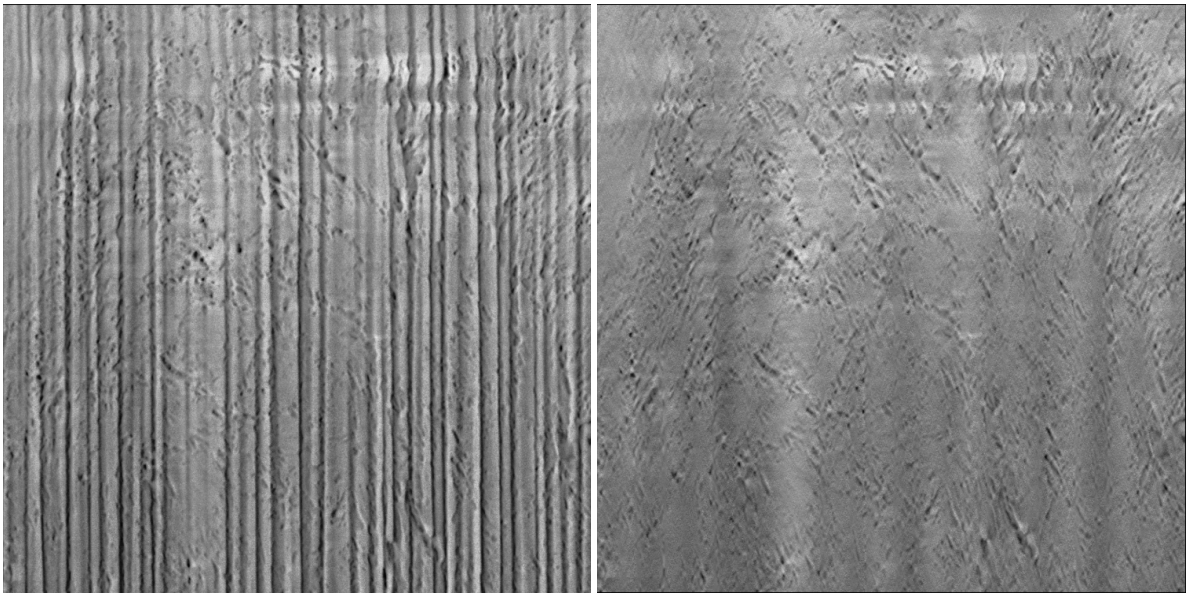
The loss value for all the 5300 combinations is showed in the graph below.



Being not completely clear how the loss function look like, it can be useful to zoom around the global minimum of the loss, as showed in the graph below.



The best parameter combination corresponds to the loss value $\mathcal{L}[w, \sigma, l] = 0.025176$, which gives the visual result below

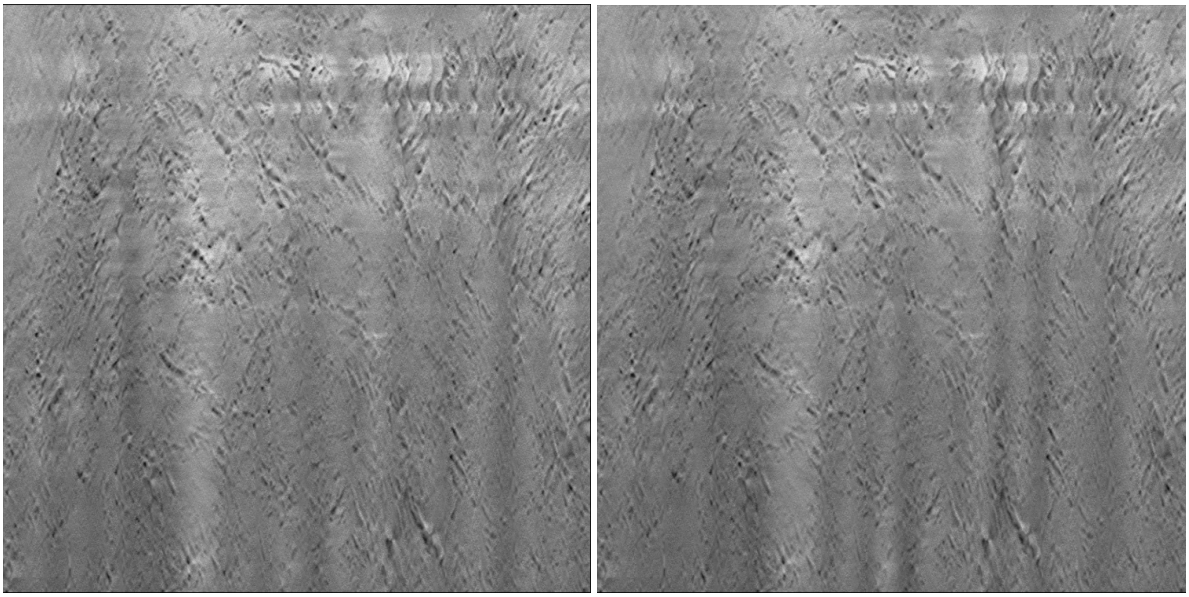


Clearly, different values of the loss correspond to different level of destriping. The animation below show how the filter quality changes in different points of the loss, confirming empirically that the loss function used is able to capture the idea of image without vertical stripes.

To give a closer look at the different visual results, the different images showed above compared with the one obtained with the best parameter combination are available below.

Global minimum vs Local minimum

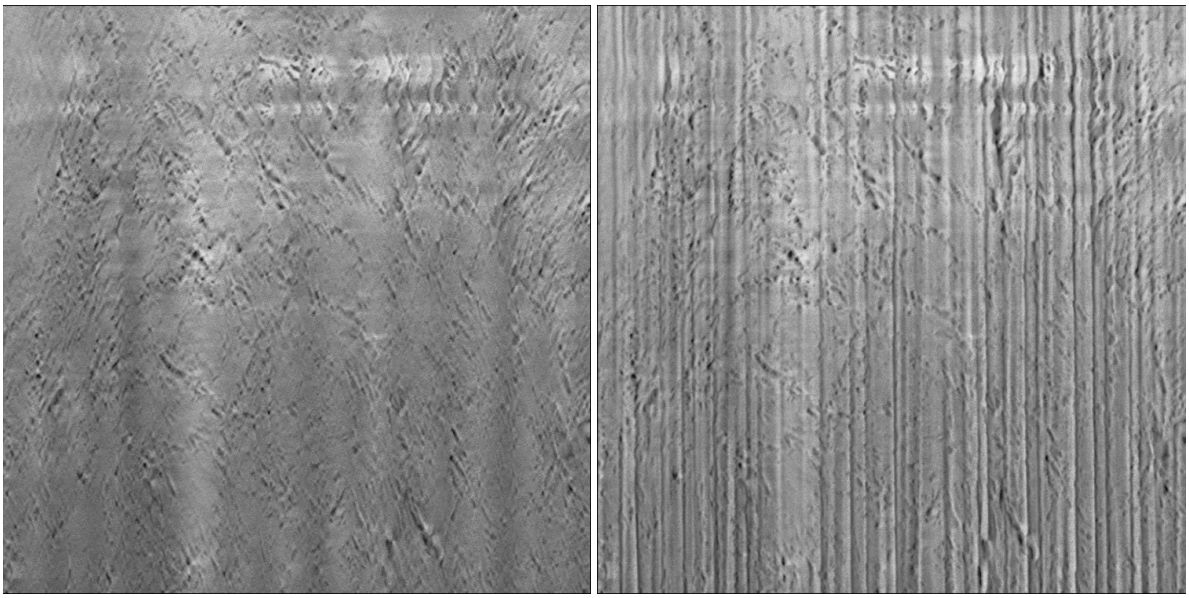
On the right, one can see the result produced with the parameter combination corresponding to a local minimum of the loss ($\mathcal{L}[w, \sigma, l] = 0.025200$)



There is not much difference between the one corresponding to the global and local minimum.

Global minimum vs Local maximum

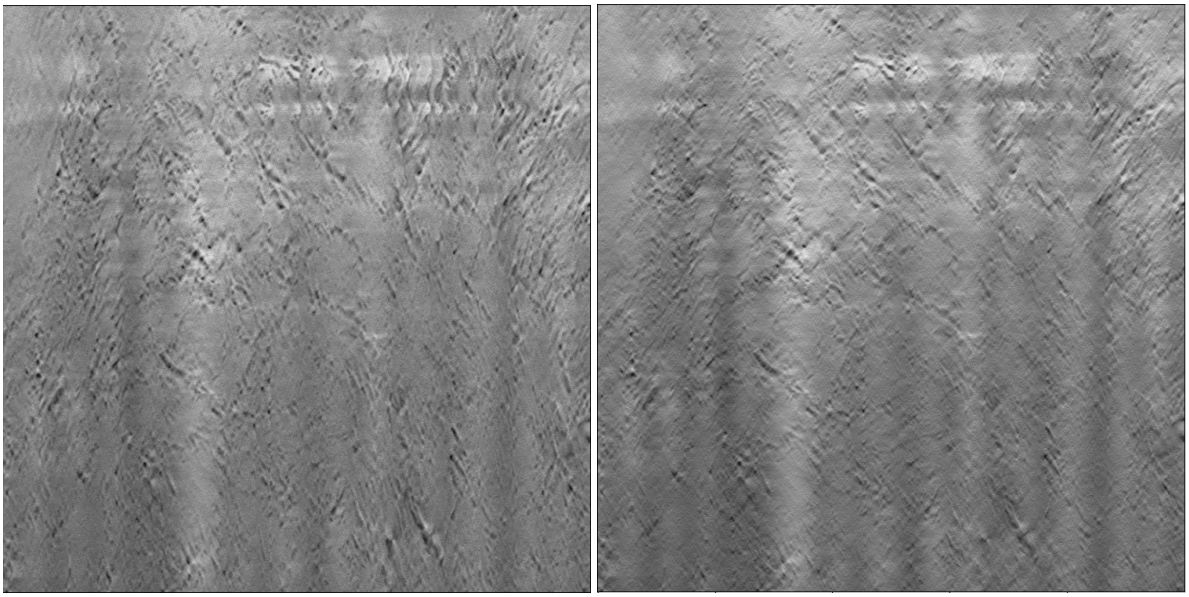
On the right, one can see the result produced with the parameter combination corresponding to a local maximum of the loss ($\mathcal{L}[w, \sigma, l] = 0.029996$)



Being completely far away from the global minimum, the stripes are still visible, as expected.

Global minimum vs Away from minimum

On the right, one can see the result produced with the parameter combination corresponding to a value in between a local maximum and the global minimum of the loss ($\mathcal{L}[w, \sigma, l] = 0.026342$).



Note that the image on the right, appear too blurred and with diagonal details which seem amplified. This is a clear sign that the filtering of the vertical component of the wavelet decomposition is too much.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

CHAPTER

TWENTYNINE

CASE OF STUDY: DECHARGER OPTIMIZATION

Here the optimization routine for the *local GF2RBGF algorithm* of the decharger plugin is briefly discussed with a practical example. In the *Implementation details* section of the Decharger plugin was discussed how the local GF2RBGF algorithm works, and how it can be optimized.

Note: The following imports are needed to run the code snapshots below.

```
import numpy as np
import psutil as psu

from joblib import Parallel,delayed
from scipy.ndimage.morphology import binary_dilation,binary_fill_holes
from skimage.morphology import reconstruction
from skimage.restoration import rolling_ball
from skimage.measure import label,regionprops

from bmiptools.transformation.basic.filters import gaussian_filter2d
from bmiptools.transformation.restoration._restoration_shared import generate_parameter_
↪space
```

As explained in the *Implementation details* section of the Decharger plugin, the local GF2RBGF consist essentially in two steps: first identification of the regions containing the charging artifact, then correct the image in this region by subtracting an estimated charging contribution. The function below implement the first step, produce correction mask $M(j,i)$ plus other data useful for the next correction step.

```
def find_charging(x, gf1_sigma, color_shift, inverse,A_threshold=50,n_available_cpu=psu.
↪cpu_count()):
    """
    Function used to estimate the regions o the image where charging is present. It is␣
↪based on a down-hill filter
    followed by a threshold operator.

    :param x: (np.array) slice to correct, in which charging is estimated.
    :param gf1_sigma: (float) sigma of the first gaussian filter.
    :param color_shift: (float in [0,1]) shift used in the downhill filter for the␣
↪estimation of the charged
                        regions.
    :param inverse: (bool) if True the charged region is estimated from the inverse␣
↪image.
    :return: (np.array) correction mask, (np.array) flattened image, (np.array) Low-
```

(continues on next page)

(continued from previous page)

```
↪Frequency image used to invert
            flattening.
    """
    LFx = gaussian_filter2d(x, gf1_sigma)
    flattened_x = x - LFx
    if inverse:

        xt = 1 - flattened_x

    else:

        xt = flattened_x

    seed = np.copy(xt - color_shift)
    seed[1:-1, 1:-1] = xt.min()
    mask = xt
    filled = reconstruction(seed, mask, method='dilation')
    candidate_mask = (xt - filled) > color_shift
    candidate_mask = binary_fill_holes(candidate_mask)
    labeled_parts = label(candidate_mask)
    props = regionprops(labeled_parts)
    fmask = np.zeros(candidate_mask.shape)

    def func_to_par(p, fmask):

        if p.area > A_threshold:

            fmask += (labeled_parts == p.label).astype(np.uint8)

        return fmask

    Parallel(n_jobs=n_available_cpu, require='sharedmem')(delayed(func_to_par)(p, fmask)␣
↪for p in props)
    return fmask, flattened_x, LFx
```

The final decharged image for the local GF2RBGF $I_{decharged}(j,i)$ is produced by the function below.

```
def correct_charging(flattened_x,correction_mask,LFx,gf2_sigma,RB_radius,gf3_sigma,n_px_
↪border=10):
    """
    Function applying the charging correction from the results obtained from the␣
↪function '_find_charging'.

    :param flattened_x: (np.array) flattened slice.
    :param correction_mask: (np.array) binary mask containing the regions to correct.
    :param LFx: (np.array) low-frequency image used to invert the flattening.
    :param gf2_sigma: (float) sigma of the second gaussian filter.
    :param RB_radius: (int) radius parameter of the rolling ball algorithm used for the␣
↪background estimation.
    :param gf3_sigma: (float) sigma of the gaussian filter used to estimate the␣
↪correction mask.
    :param n_px_border: (int) number of pixel at the border used to fade the correction␣
```

(continues on next page)

(continued from previous page)

```
↪away in the input slice.
    :return: (np.array) corrected slice.
    """
    # define corrections zone
    dilations = [correction_mask]
    for i in range(n_px_border):

        dilations.append(binary_dilation(dilations[-1],iterations=1).astype(np.uint8))

    borders = []
    for i in range(n_px_border,0,-1):

        lmbda = (n_px_border-i)/(n_px_border+1)
        borders.append(lmbda*(dilations[i]-dilations[i-1]).astype(np.float32))

    borders = np.array(borders)
    border_region = dilations[-1]-correction_mask
    increasing_borders = borders.sum(0)
    decreasing_borders = (1-borders.sum(0))*border_region

    # corrector
    if gf2_sigma > 0:

        LFflattened_x = gaussian_filter2d(flattened_x,gf2_sigma)

    else:

        LFflattened_x = 0

    bkg_corr = rolling_ball(flattened_x - LFflattened_x,radius=RB_radius)
    gf_bkg_corr = gaussian_filter2d(bkg_corr,gf3_sigma)
    decharged_x = flattened_x*(1-border_region+decreasing_borders-correction_mask) + \
                  (flattened_x-gf_bkg_corr)*(correction_mask+increasing_borders)

    return decharged_x+LFx
```

The full local GF2RBGF algorithm consist practically in the application in sequence of the two functions presented above. A standardization/destandardization step is added to make the algorithm more robust to variations in the dynamics of the input image.

```
def local_GF2RBGF(x,param):
    """
    Local_GF2RBGF methods for the charging correction/reduction.

    :param x: (np.array) the slice to correct.
    :param param: (tuple) tuple containing all the algorithm parameter.
    :return: (np.array) decharged slice.
    """

    #get parameters
    gf1_sigma, color_shift, gf2_sigma, RB_radius, gf3_sigma, inverse = param
```

(continues on next page)

(continued from previous page)

```
    # 0/1 standardize
    M = x.max()
    m = x.min()
    stand_x = standardizer(x,'0/1')

    # identify and correct
    correction_mask,flattened_x,LFx = find_charging(stand_x,gf1_sigma,color_shift,
↪inverse)
    decharged_x = correct_charging(flattened_x,correction_mask,LFx,gf2_sigma,RB_radius,
↪gf3_sigma)

    # destandardize
    decharged_x = (M-m)*decharged_x+m

    return decharged_x
```

The three functions described here, are very close to the corresponding one are present in the `Decharger` class.

29.1 The optimization routine

The parameter space used for the optimization routine is defined as follow

- the σ_{GF1} and σ_{GF2} parameter are tested for 3 possible values: 40, 80, and 120;
- the values of c_{shift} tested are 0.05, 0.1, and 0.2;
- the rolling ball radius r is tested for a value of 2, 10, and 50;
- the σ_{GF3} parameter is tested for 3 possible values: 4, 25, and 50.

In total the number of possible combinations tested are 243. To generate the parameter space from these setting one can use the function `generate_parameter_space`.

```
from bmiptools.transformation.restoration._restoration_shared import generate_parameter_
↪space

# define parameter space boundaries
gf1_sigma_range = [40,80,120]
color_shift_range = [0.05,0.1,0.2]
gf2_sigma_range = [40,80,120]
RB_radius_range = [2,10,50]
gf3_sigma_range = [4,25,50]

# generate parameter space
parameter_space,_ = generate_parameter_space({'gf1_sigma': gf1_sigma_range,
                                              'color_shift': color_shift_range,
                                              'gf2_sigma': gf2_sigma_range,
                                              'RB_radius': RB_radius_range,
                                              'gf3_sigma': gf3_sigma_range,
                                              'inverse': [True]})
print('Total number of parameters combination: ',len(parameter_space))
```

Note: The example used here is a case of inverse charging: the shift in brightness is towards lower brightness level

rather than the opposite, as in the usual charging. That is way the parameter space contains also the `inverse` parameter which is always `True`.

According to the optimization procedure presented in the *Implementation details* section of this plugin, pairs of charged-decharged regions (Q, Q^s) used in to compute the loss, can be obtained using the function below.

```
def get_loss_optimization_mask_pairs(cmask, dilation_iteration=10, N_charged_regions_for_
↪optimization=20):
    """
    Compute the pairs charged region / uncharged region pairs needed for the computation␣
↪of the Decharger loss.

    :param cmask: (np.array) mask with all the estimated charged regions
    :param dilation_iteration: (int) umber of dilation done to correction mask in order␣
↪define the region in which
                               charging is not present but the histogram is still␣
↪comparable with the one obtained
                               from the region in  which charging is present.
    :param N_charged_regions_for_optimization: (int) maximum number of regions␣
↪considered in a correction mask for
                                               the loss computation. The region selected␣
↪are the one with the
                                               biggest areas.
    :return: (list of np.array) list of couples of masks for the charged region and its␣
↪surrounding uncharged region.
    """
    labeled_cmask, Nlabels = label(cmask, return_num=True)
    prop_cmask = regionprops(labeled_cmask)
    area_label_pair = [[item.area, item.label] for item in prop_cmask]
    sorted_area_label_pair = sorted(area_label_pair, reverse=True)
    masks_pairs = []
    for _, l in sorted_area_label_pair[:N_charged_regions_for_optimization]:

        sel_cmask = (labeled_cmask == l).astype(np.int8)
        s1 = binary_dilation(sel_cmask, iterations=dilation_iteration).astype(np.int8)
        s2 = binary_dilation(s1, iterations=2 * dilation_iteration).astype(np.int8)
        sel_gmask = s2 - s1
        sel_gmask = ((sel_gmask-binary_dilation(cmask,
                                                iterations=dilation_iteration).astype(np.
↪int8))>0).astype(np.int8)
        masks_pairs.append((sel_cmask, sel_gmask))

    return masks_pairs
```

Given the pairs (Q, Q^s), the loss $\mathcal{L}[c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}](I, Q, Q^s)$ is computed with the function below.

```
def l1(p, q):
    """
    l1 distance between normalized p and q.
    """

    return np.sum(np.abs(p / np.sum(p) - q / np.sum(q)))
```

(continues on next page)

(continued from previous page)

```
def compute_loss(sl, mask_pairs):
    """
    Compute the loss function for the decharger.

    :param sl: (np.array) slice on which the loss is evaluated.
    :param mask_pairs: (list of np.arrays) masks of the charged/uncharged region pairs␣
↪used to compute the loss.
    :return: (float) the loss value.
    """
    sl_norm = (sl - sl.min()) / (sl.max() - sl.min()) * 256
    loss = 0
    for m_c, m_b in mask_pairs:

        val_c = sl_norm.flatten()[m_c.flatten() == 1]
        h_c, _ = np.histogram(val_c, bins=256, range=(0, 256), density=True)
        val_b = sl_norm.flatten()[m_b.flatten() == 1]
        h_b, _ = np.histogram(val_b, bins=256, range=(0, 256), density=True)
        loss += l1(h_c, h_b)

    return loss / len(mask_pairs)
```

Assuming to apply the optimization routine on the slice `sl` of some stack. The code below should be used to compute the loss function for each parameter combination.

```
sl = ...     # slice used for the decharger optimization routine

# compute the loss
loss = []
for p in parameter_space:

    # get parameter combination
    gf1_sigma, color_shift, gf2_sigma, RB_radius, gf3_sigma,inverse = p

    # estimate correction mask
    cmask, flat_slice, LFslice = find_charging(sl, gf1_sigma, color_shift, inverse)

    # correct slice and compute the loss value
    if np.sum(cmask) != 0:

        mask_pairs = get_loss_optimization_mask_pairs(cmask)
        corrected_slice = correct_charging(flat_slice, cmask, LFslice, gf2_sigma, RB_
↪radius, gf3_sigma)
        l = compute_loss(corrected_slice, mask_pairs)
        loss.append(l)

    else:

        loss.append(np.inf)
```

Clearly, the best parameters are the one corresponding to the lowest value of the loss.

```
# print best parameters
```

(continues on next page)

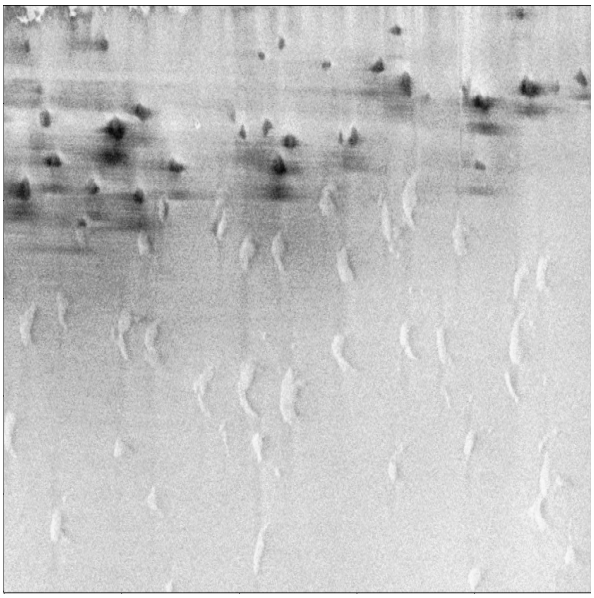
(continued from previous page)

```
best_idx = np.argmin(loss)
print('Best parameter combinations: ',parameter_space[best_idx])
print('Loss = {}'.format(loss[best_idx]))
```

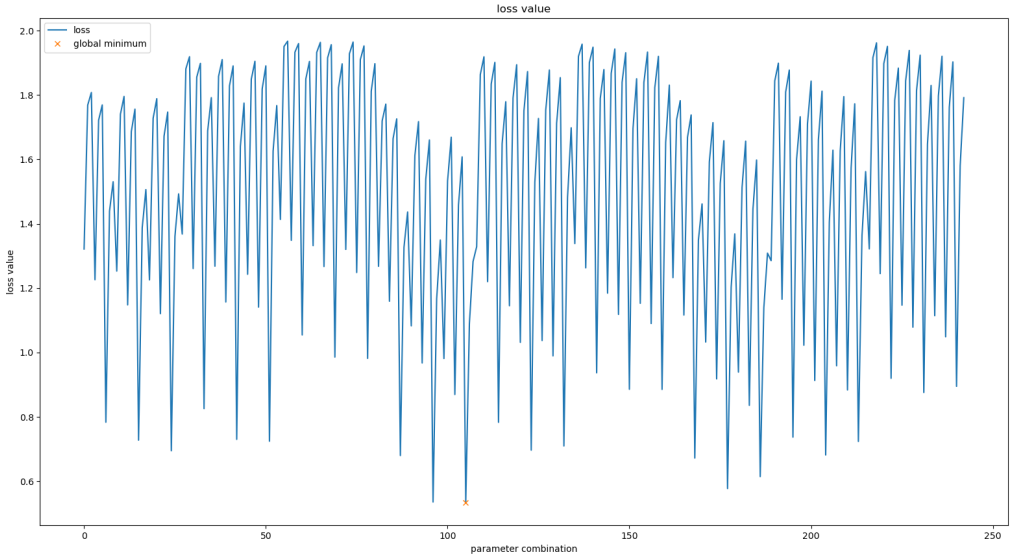
The optimization routine described here, contains all the essential steps which are present in the `Decharger` class.

29.2 Results

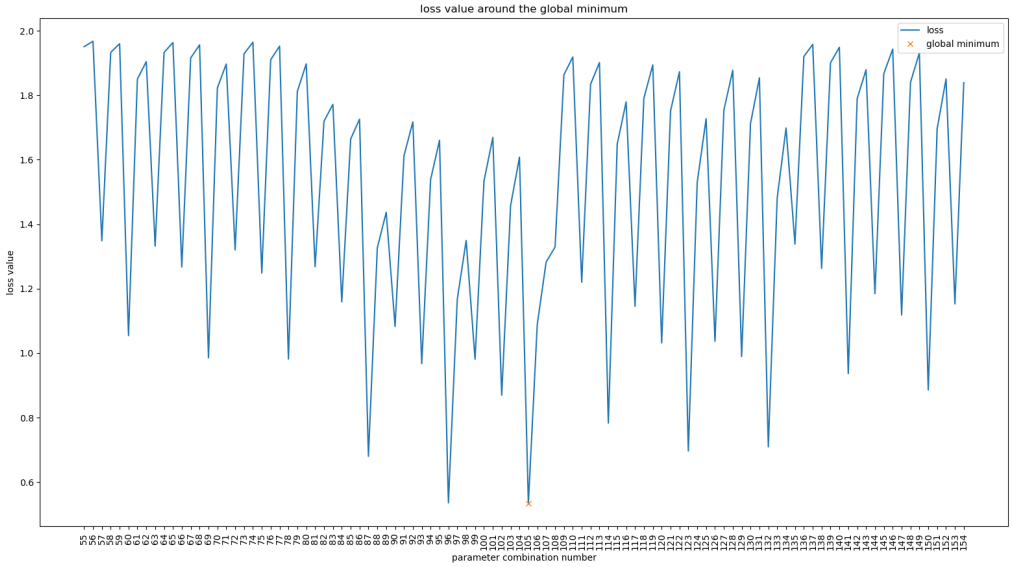
Consider the image below as input for the algorithm. The inverse charging artifacts are clearly visible in the upper half of the image.



The loss value for all the 243 combinations is showed in the graph below.



Being not completely clear how the loss function look like, it can be useful to zoom around the global minimum of the loss, as showed in the graph below.



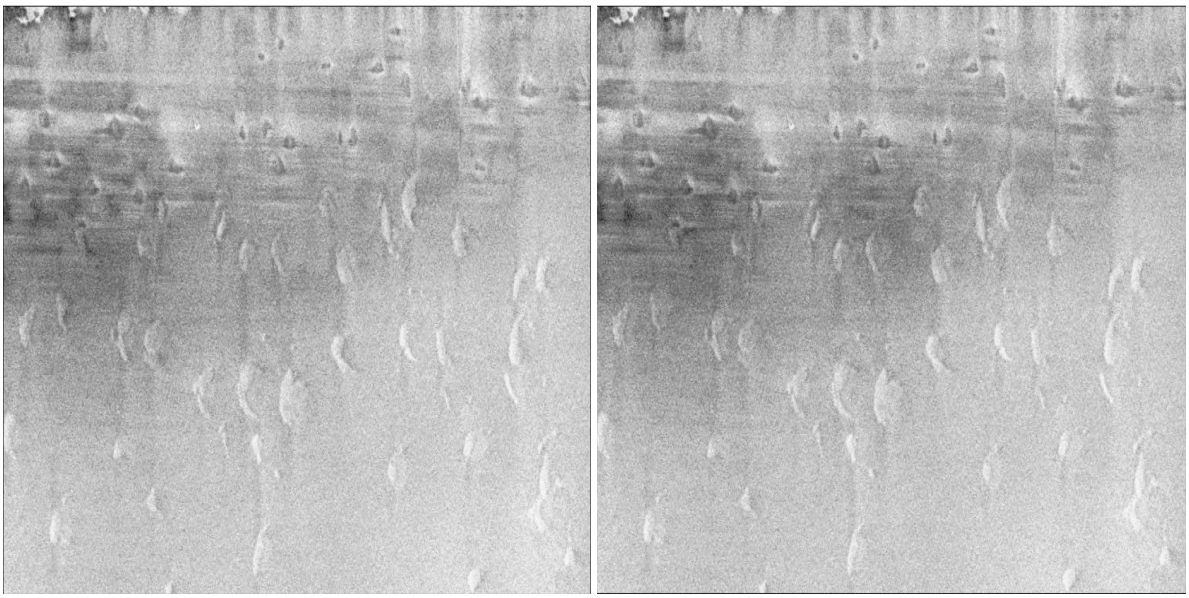
The best parameter combination corresponds to the loss value $\mathcal{L}[c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}] = 0.5344$, which gives the visual result below.

Clearly, different values of the loss correspond to different level of decharging. The animation below show how the filter quality changes in different points of the loss, confirming empirically that the loss function used is able to capture the idea of image wit less level of (inverse, in this case) charging .

To give a closer look at the different visual results, the different images showed above compared with the one obtained with the best parameter combination are available below.

Global minimum vs Local minimum

On the right, one can see the result produced with the parameter combination corresponding to a local minimum of the loss ($\mathcal{L}[c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}] = 0.6950$).

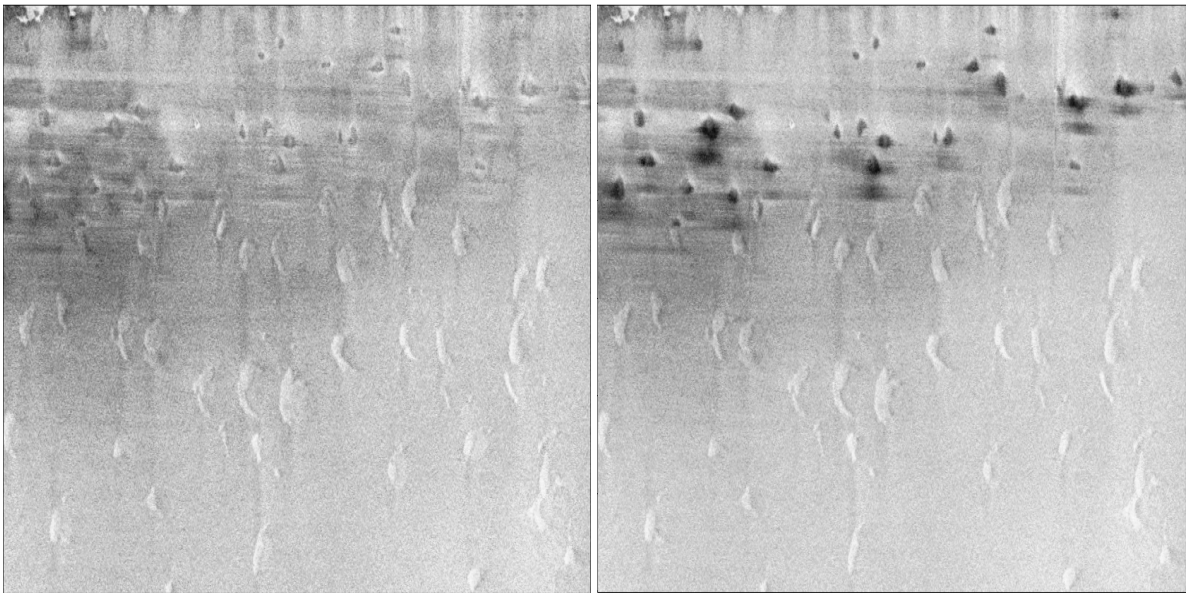


There is not much difference between the one corresponding to the global and local minimum.

Global minimum vs Away from minimum

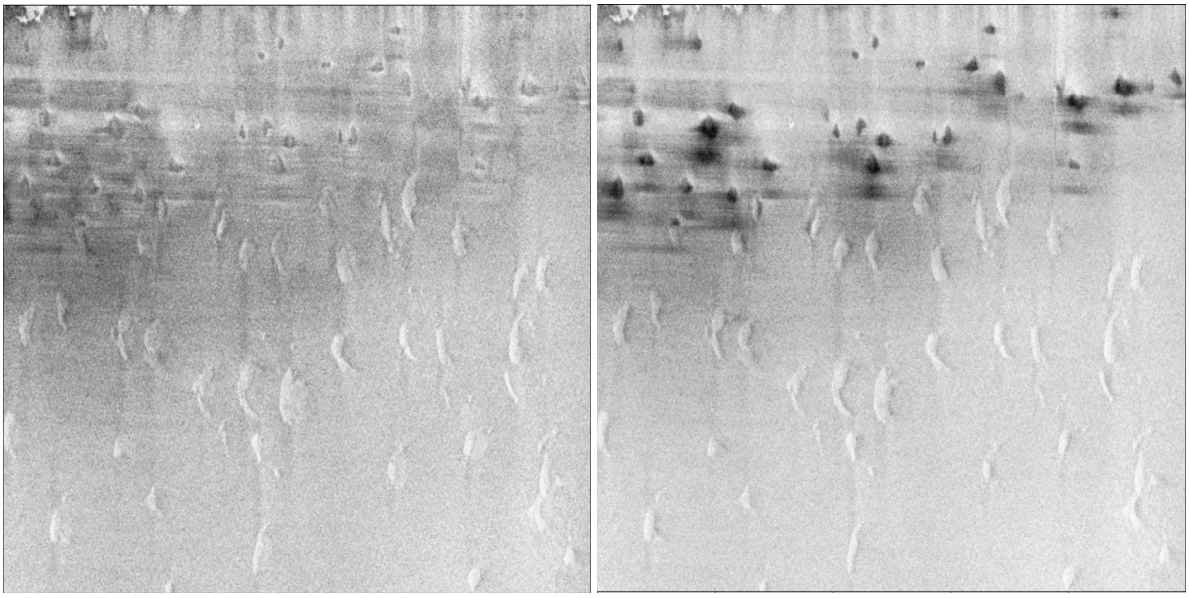
On the right, one can see the result produced with the parameter combination corresponding to a value in between a

local maximum and the global minimum of the loss ($\mathcal{L}[c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}] = 1.4365$).



Global minimum vs Local maximum

On the right, one can see the result produced with the parameter combination corresponding to a local maximum of the loss ($\mathcal{L}[c_{shift}, \sigma_{GF_1}, \sigma_{GF_2}, r, \sigma_{GF_3}] = 1.9672$)



Being completely far away from the global minimum, the charging artifact is practically left unchanged.

Note: The script used to produce the images displayed can be found here. To reproduce the images showed above one may consult the examples/documentation_scritps folder, where is explained how to run the example scripts and where one can find all the necessary input data.

CHAPTER

THIRTY

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Chang2000] "Adaptive wavelet thresholding for image denoising and compression." - Chang, S. Grace, Bin Yu, and Martin Vetterli -Image Processing, IEEE Transactions on 9.9 (2000): 1532-1546. DOI:10.1109/83.862633

[Donoho1994] "Ideal spatial adaptation by wavelet shrinkage." - D. L. Donoho and I. M. Johnstone. - Biometrika 81.3 (1994): 425-455. DOI:10.1093/biomet/81.3.425

[Gupta2015] "Image Denoising Using Bayes Shrink Method Based On Wavelet Transform" - Payal Gupta and Amit Garg - International Journal of Electronic and Electrical Engineering. Volume 8, Number 1 (2015), pp. 33-40

[Chambolle2004] "An algorithm for total variation minimization and applications" - Chambolle, A. - Journal of Mathematical Imaging and Vision. 20 (2004): 89–97

[Goldstein2009] "The split Bregman method for L1-regularized problems." - Goldstein, Tom, and Stanley Osher - SIAM journal on imaging sciences 2.2 (2009): 323-343.

[Bush2011] "Bregman algorithms" - Bush J. - Senior Thesis (2011), https://web.math.ucsb.edu/~cgarcia/UGProjects/BregmanAlgorithms_JacquelineBush.pdf

[Paris2009] "Bilateral Filtering: Theory and Applications" - Paris S., Kornprobst P., Tumblin J., Durand F. - Foundations and Trends® in Computer Graphics and Vision: Vol. 4: No. 1 (2009), pp 1-73. http://dx.doi.org/10.1561/0600000020

[Buades2011] "Non-Local Means Denoising" - Antoni Buades, Bartomeu Coll, and Jean-Michel Morel - Image Processing On Line, 1 (2011), pp. 208–212. https://doi.org/10.5201/ipol.2011.bcm_nlm

[Darbon2008] "Fast nonlocal filtering applied to electron cryomicroscopy" - J. Darbon, A. Cunha, T. F. Chan, S. Osher and G. J. Jensen, 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008, pp. 1331-1334, doi: 10.1109/ISBI.2008.4541250.

[Batson2019] "Noise2Self: Blind Denoising by Self-Supervision" - Joshua Batson, Loic Royer Proceedings of the 36th International Conference on Machine Learning, PMLR 97:524-533, 2019.

[Lehtinen2018] "Noise2Noise: Learning Image Restoration without Clean Data" - Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, Timo Aila - https://arxiv.org/abs/1803.04189

[Krull2019] "Noise2Void - Learning Denoising from Single Noisy Images" - Alexander Krull, Tim-Oliver Buchholz, Florian Jug - https://arxiv.org/pdf/1811.10980.pdf

[Batson2019] "Noise2Self: Blind Denoising by Self-Supervision" - Joshua Batson, Loic Royer Proceedings of the 36th International Conference on Machine Learning, PMLR 97:524-533, 2019.

[Beat2009] "Stripe and ring artifact removal with combined wavelet—Fourier filtering" - Beat Münch, Pavel Trtik, Federica Marone, and Marco Stampanoni - https://doi.org/10.1364/OE.17.008567

[Sternberg1983] "Biomedical Image Processing" - Sternberg S. R. - Computer - Volume: 16, Issue: 1, Jan 1983 - doi: 10.1109/MC.1983.1654163.

[Spehner2020] "Cryo-FIB-SEM as a promising tool for localizing proteins in 3D" - Spehner D., Steyer A. M., Bertinetti L., Orlov I., Benoit L., Pernet-Gallay K., Schertel A., Schultz P. - J. Struct. Biol. 2020 Jul 1;211(1):107528,doi: 10.1016/j.jsb.2020.107528.

[Robinson2004] "Efficient morphological reconstruction: A downhill filter" - Robinson K., Whelan P. F. - Pattern Recognition Letters 25(15):1759-1767, November 2004 - doi:10.1016/j.patrec.2004.07.002.

[Evangelidis2008] "Parametric Image Alignment using Enhanced Correlation Coefficient Maximization" - G.D. Evangelidis, E.Z. Psarakis - IEEE Trans. on PAMI, vol. 30, no. 10, 2008

[Reddy1996] "An FFT-based technique for translation, rotation, and scale-invariant image registration." - Reddy B.S., and B. N. Chatterji. - IEEE transactions on image processing : a publication of the IEEE Signal Processing Society 5 8 (1996): 1266-71 .

[LeBesnerais2005] "Dense optical flow by iterative local window registration" - G. Le Besnerais and F. Champagnat, - IEEE International Conference on Image Processing 2005, 2005, pp. I-137, doi: 10.1109/ICIP.2005.1529706.

INDEX

Symbols

B

F

G

I

P

S

T

V

W