SIMPA

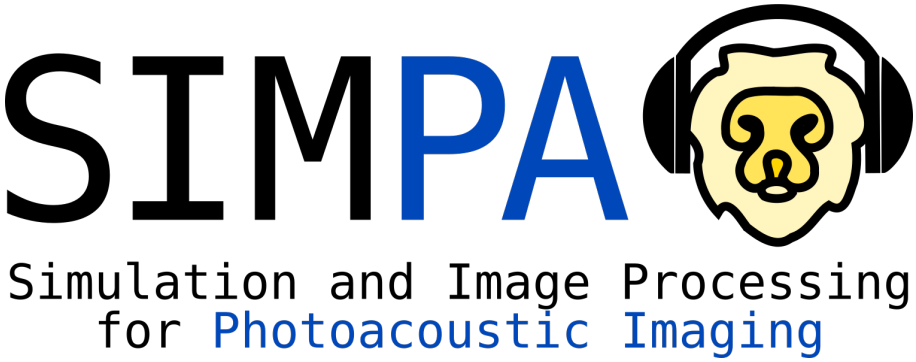
version 0.3.1

CAMI (Computer Assisted Medical Interventions), DKFZ, Heidelberg

April 13, 2021

Contents

Welcome to the SIMPA documentation!



README

The Simulation and Image Processing for Photoacoustic Imaging (SIMPA) toolkit.

SIMPA Install Instructions

You can install simpa with pip. Simply run:

```
pip install simpa
```

For a manual installation from the code, please follow steps 1 - 3:

1. `git clone https://github.com/CAMI-DKFZ/simpa.git`
2. `git checkout master`
3. `git pull`

Now open a python instance in the 'simpa' folder that you have just downloaded. Make sure that you have your preferred virtual environment activated

1. `cd simpa`
2. `pip install -r requirements.txt`
3. `python -m setup.py install`
4. Test if the installation worked by using `python` followed by `import simpa` then `exit()`

If no error messages arise, you are now setup to use simpa in your project.

You also need to manually install the pytorch library to use all features of SIMPA. To this end, use the pytorch website tool to figure out which version to install: `https://pytorch.org/get-started/locally/`

Building the documentation

When the installation went fine and you want to make sure that you have the latest documentation you should do the following steps in a command line:

1. Navigate to the `simpa` source directory (same level where the setup.py is in)
2. Execute the command `sphinx-build -b pdf -a simpa_documentation/src simpa_documentation`
3. Find the PDF file in `simpa_documentation/simpa_documantation.pdf`

External Tools installation instructions

mcx (Optical Forward Model)

Either download suitable executables or build yourself from the following sources:

http://mcx.space/

In order to obtain access to all custom sources that we implemented, please build mcx yourself from the following mcx Github fork: https://www.github.com/jgroehl/mcx

For the installation, please follow the instructions from the original repository.

k-Wave (Acoustic Forward Model)

Please follow the following steps and use the k-Wave install instructions for further (and much better) guidance under:

http://www.k-wave.org/

1. Install MATLAB with the core and parallel computing toolboxes activated at the minimum.
2. Download the kWave toolbox
3. Add the kWave toolbox base bath to the toolbox paths in MATLAB
4. If wanted: Download the CPP and CUDA binary files and place them inthe k-Wave/binaries folder
5. Note down the system path to the `matlab` executable file.

Overview

The main use case for the simpa framework is the simulation of photoacoustic images. However, it can also be used for image processing.

Simulating photoacoustic images

A basic example on how to use simpa in you project to run an optical forward simulation is given in the samples/minimal_optical_simulation.py file.

Path Management

As a pipelining tool that serves as a communication layer between different numerical forward models and processing tools, SIMPA needs to be configured with the paths to these tools on your local hard drive. To this end, we have implemented the `PathManager` class that you can import to your project using `from simpa.utils import PathManager`. The PathManager looks for a `path_config.env` file (just like the one we provided in the `simpa_examples`) in the following places in this order:

1. The optional path you give the PathManager
2. Your $HOME$ directory
3. The current working directory
4. The SIMPA home directory path

How to contribute

Please find a more detailed description of how to contribute as well as code style references in our **developer_guide.md**

The SIMPA code is written and maintained on a closed git repository that is hosted on a server of the German Cancer Research Center (DKFZ), Heidelberg, Germany and changes to the develop or master branch are mirrored on Github. As such, only the current master and develop branch of the repository are open source.

To contribute to SIMPA, please fork the SIMPA github repository and create a pull request with a branch containing your suggested changes. The core team developers will then review the suggested changes and integrate these into the code base.

Please make sure that you have included unit tests for your code and that all previous tests still run through.

There is a regular SIMPA status meeting every Friday on even calendar weeks at 10:00 CET/CEST and you are very welcome to participate and raise any issues or suggest new features. You can join the meeting using the following link:

https://meet.google.com/rze-bxej-cvj

Please see the github guidelines for creating pull requests: https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

Performance profiling

Do you wish to know which parts of the simulation pipeline cost the most amount of time? If that is the case then you can use the following commands to profile the execution of your simulation script. You simply need to replace the `myscript` name with your script name.

```
python -m cProfile -o myscript.cprof myscript.py
```

```
pyprof2calltree -k -i myscript.cprof
```

Developer Guide

Dear SIMPA developers, Dear person who wants to contribute to the SIMPA toolkit,

First of all: Thank you for your participation and help! It is much appreciated! This Guide is meant to be used as a collection of How-To's to contribute to the framework. In case you have any questions, do not hesitate to get in touch with the members of the core development team:

Kris K. Dreher (k.dreher@dkfz-heidelberg.de)

Janek M. Groehl (janek.grohl@cruk.cam.ac.uk)

How to contribute

The SIMPA code is written and maintained on a closed repository that is hosted on a server of the German Cancer Research Center and changes to the develop or master branch are mirrored on Github. As such, only the current master and develop branch of the repository are open source.

To contribute to SIMPA, please fork the SIMPA github repository and create a pull request with a branch containing your suggested changes. The core team developers will then review the suggested changes and integrate these into the code base.

Please make sure that you have included unit tests for your code and that all previous tests still run through.

There is a regular SIMPA status meeting every Friday on even calendar weeks at 10:00 CET/CEST and you are very welcome to participate and raise any issues or suggest new features. You can join the meeting using the following link:

https://meet.google.com/rze-bxej-cvj

Please see the github guidelines for creating pull requests: https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests

Coding style

When writing code for SIMPA, please use the PEP 8 python coding conventions (https://www.python.org/dev/peps/pep-0008/) and consider using the following structures in your code in order to make a new developer or someone external always know exactly what to expect.

- Classnames are written in camel-case notation `ClassName`
- Function names are written in small letter with _ as the delimiter `function_name`
- Function parameters are always annotated with their type `arg1: type = default`
- Only use primitive types as defaults. If a non-primitive type is used, then the default should be `None` and the parameter should be initialized in the beginning of a function.

- A single line of code should not be longer than 120 characters.
- Functions should follow the following simple structure:
    1. Input validation (arguments all not `None`, correct type, and acceptable value ranges?)
    2. Processing (clean handling of errors that might occur)
    3. Output generation (sanity checking of the output before handing it off to the caller)

Documenting your code

Only documented code will appear in the sphinx generated documentation.

A class should be documented using the following syntax:

```
class ClassName(Superclass):
    """
    Explain how the class is used and what it does.
    """
```

For functions, a lot of extra attributes can be added to the documentation:

```
def function_name(self, arg1:type = default, arg2:type = default) -> return_type:
    """
    Explain how the function is used and what it does.

    :param arg1: type, value range, Null acceptable?
    :param arg2: type, value range, Null acceptable?
    :returns: type, value range, does it return Null?
    :raises ExceptionType: explain when and why this exception is raised
    """
```

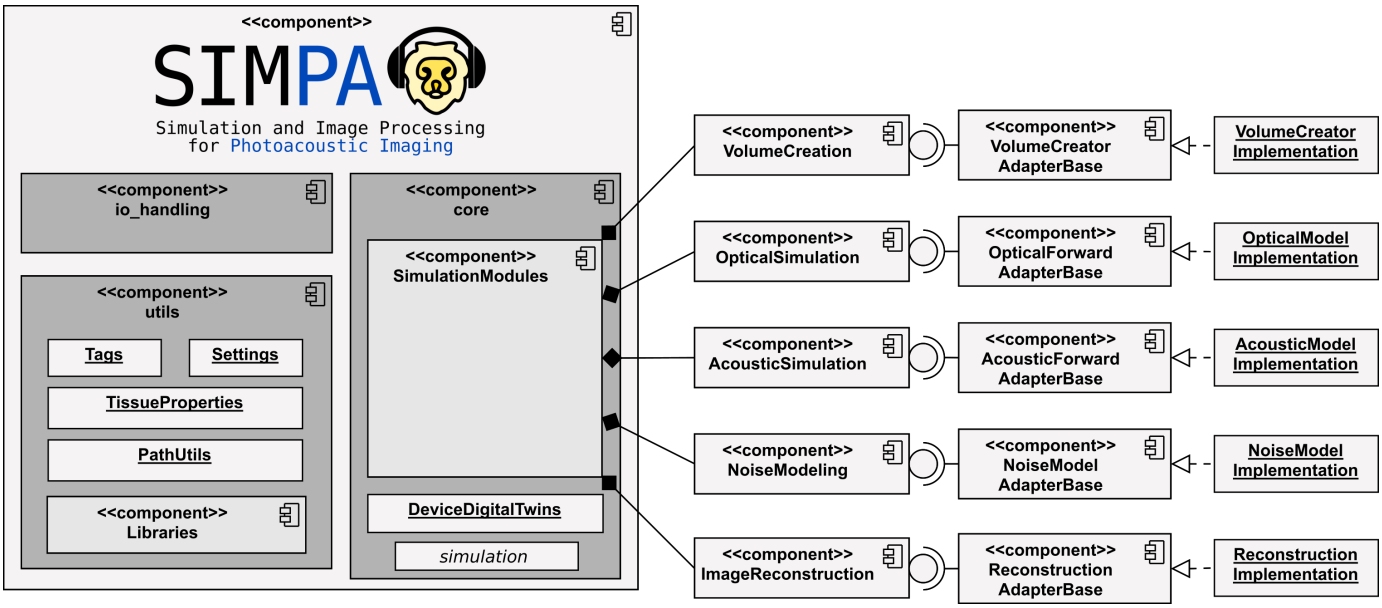
Adding literature absorption spectra

The central point, where absorption spectra are collected and handled is in `simpa.utils.libraries.spectra_library.py`. The file comprises the class `AbsorptionSpectrumLibrary`, in which the new absorption spectra can be added using the following two steps:

1. In the beginning of the class, there is a bunch of constants that define spectra using the `AbsorptionSpectrum` class. Add a new constant here: `NEW_SPECTRUM = AbsorptionSpectrum(absorber_name, wavelengths, absorptions)`. By convention, the naming of the constant should be the same as the `absorber_name` field. The `wavelengths` and `absorptions` arrays must be of the same length and contain corresponding values.
2. In the `__init__` method of the `AbsorptionSpectrumLibrary` class, the class constants are added to an internal list. This has the benefit of enabling the Library class to be iterable. Add your newly added constant field to the list here.
3. Your absorption spectrum is now usable throughout all of simpa and is accessible using the `SPECTRAL_LIBRARY` sngleton that can be imported using `from simpa.utils import SPECTRAL_LIBRARY`.

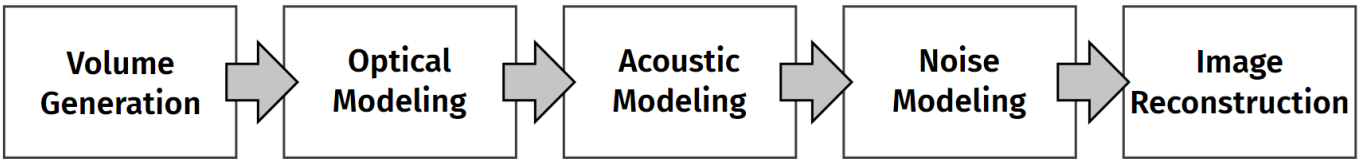
Class references

This component diagram shows the three principle modules of the SIMPA toolkit and gives an insight into their constituents. The core is concerned with providing interfaces for the simulation tools, while the utils module contains many scripts and classes to facilitate the use of the simulation pipeline.



Module: core

The purpose of the core module is to provide interfaces that facilitate the integration of toolboxes and code for photoacoustic modeling into a single continuous pipeline:



simpa.core.simulation.**simulate**(simulation_pipeline: list, settings: simpa.utils.settings.Settings)

This method constitutes the staring point for the simulation pipeline of the SIMPA toolkit. It calls all relevant and wanted simulation modules in the following pre-determined order:

```
def simulation(settings):
    for wavelength in settings[Tags.WAVELENGTHS]:

        simulation_data = volume_creator.create_simulation_volumes(settings)
        if optical_simulation in settings:
            optical_model.simulate(simulation_data, settings)
        if acoustic_simulation in settings:
            acoustic_model.simulate(simulation_data, settings)
        if noise_simulation in settings:
            noise_model.simulate(simulation_data, settings)
        if image_reconstruction in settings:
            reconstruction_model.simulate(simulation_data, settings)

        io_handler.save_hdf5(simulation_data, settings)
```

Parameters:

- **simulation_pipeline** – a list of callable functions
- **settings** – settings dictionary containing the simulation instructions

Returns: list with the save paths of the simulated data within the HDF5 file.

Volume creation

The core contribution of the SIMPA toolkit is the creation of in silico tissue-mimicking phantoms. This feature is represented by the volume_creation module, that two main volume creation modules: | Model-based creation of volumes using a set of rules | Segmentation-based creation of volumes

class simpa.core.volume_creation.**VolumeCreatorModuleBase** (global_settings: simpa.utils.settings.Settings)

Use this class to define your own volume creation adapter.

abstract **create_simulation_volume ()** → dict

This method will be called to create a simulation volume.

run ()

This method is the main entry point of volume creation for the SIMPA framework. It uses the Tags.VOLUME_CREATOR tag to determine which of the volume creators should be used to create the simulation phantom.

Parameters: **global_settings** – the settings dictionary that contains the simulation instructions

class simpa.core.volume_creation.segmentation_based_volume_creator.**SegmentationBasedVolumeCreator** (global_settings: simpa.utils.settings.Settings)

This volume creator expects a np.ndarray to be in the settigs under the Tags.INPUT_SEGMENTATION_VOLUME tag and uses this array together with a SegmentationClass mapping which is a dict defined in the settings under Tags.SEGMENTATION_CLASS_MAPPING.
With this, an even greater utility is warranted.

create_simulation_volume () → dict

This method will be called to create a simulation volume.

class simpa.core.volume_creation.versatile_volume_creator.**ModelBasedVolumeCreator** (global_settings: simpa.utils.settings.Settings)

The model-based volume creator uses a set of rules how to generate structures to create a simulation volume. These structures are added to the dictionary and later combined by the algorithm:

```
# Initialise settings dictionaries
simulation_settings = Settings()
all_structures = Settings()
structure = Settings()

# Definition of en example structure.
# The concrete structure parameters will change depending on the
# structure type
structure[Tags.PRIORITY] = 1
structure[Tags.STRUCTURE_START_MM] = [0, 0, 0]
structure[Tags.STRUCTURE_END_MM] = [0, 0, 100]
structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle()
structure[Tags.CONSIDER_PARTIAL_VOLUME] = True
structure[Tags.ADHERE_TO_DEFORMATION] = True
structure[Tags.STRUCTURE_TYPE] = Tags.HORIZONTAL_LAYER_STRUCTURE

all_structures["arbitrary_identifier"] = structure

simulation_settings[Tags.STRUCTURES] = all_structures

# ...
# Define further simulation settings
# ...

simulate(simulation_settings)
```

create_simulation_volume () → dict

This method creates a in silico respresentation of a tissue as described in the settings file that is given.

Parameters: **settings** – a dictionary containing all relevant Tags for the simulation to be able to instantiate a tissue.

Returns: a path to a npz file containing characteristics of the simulated volume: absorption, scattering, anisotropy, oxygenation, and a segmentation mask. All of these are given as 3d numpy arrays.

Model-based volume creation

class simpa.core.volume_creation.versatile_volume_creator.**ModelBasedVolumeCreator**(global_settings: simpa.utils.settings.Settings)

The model-based volume creator uses a set of rules how to generate structures to create a simulation volume. These structures are added to the dictionary and later combined by the algorithm:

```
# Initialise settings dictionaries
simulation_settings = Settings()
all_structures = Settings()
structure = Settings()

# Definition of en example structure.
# The concrete structure parameters will change depending on the
# structure type
structure[Tags.PRIORITY] = 1
structure[Tags.STRUCTURE_START_MM] = [0, 0, 0]
structure[Tags.STRUCTURE_END_MM] = [0, 0, 100]
structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle()
structure[Tags.CONSIDER_PARTIAL_VOLUME] = True
structure[Tags.ADHERE_TO_DEFORMATION] = True
structure[Tags.STRUCTURE_TYPE] = Tags.HORIZONTAL_LAYER_STRUCTURE

all_structures["arbitrary_identifier"] = structure

simulation_settings[Tags.STRUCTURES] = all_structures

# ...
# Define further simulation settings
# ...

simulate(simulation_settings)
```

create_simulation_volume () → dict

This method creates a in silico respresentation of a tissue as described in the settings file that is given.

Parameters: **settings** – a dictionary containing all relevant Tags for the simulation to be able to instantiate a tissue.

Returns: a path to a npz file containing characteristics of the simulated volume: absorption, scattering, anisotropy, oxygenation, and a segmentation mask. All of these are given as 3d numpy arrays.

Segmentation-based volume creation

class simpa.core.volume_creation.segmentation_based_volume_creator.**SegmentationBasedVolumeCreator**(global_settings: simpa.utils.settings.Settings)

This volume creator expects a np.ndarray to be in the settigs under the Tags.INPUT_SEGMENTATION_VOLUME tag and uses this array together with a SegmentationClass mapping which is a dict defined in the settings under Tags.SEGMENTATION_CLASS_MAPPING.
With this, an even greater utility is warranted.

create_simulation_volume () → dict

This method will be called to create a simulation volume.

Optical forward modeling

class simpa.core.optical_simulation.**OpticalForwardModuleBase**(global_settings: simpa.utils.settings.Settings)

Use this class as a base for implementations of optical forward models.

abstract **forward_model**(absorption_cm, scattering_cm, anisotropy)

A deriving class needs to implement this method according to its model.

Parameters:
- **absorption_cm** – Absorption in units of per centimeter
- **scattering_cm** – Scattering in units of per centimeter
- **anisotropy** – Dimensionless scattering anisotropy

Returns: Fluence in units of J/cm^2

run()

Call this method to invoke the simulation process.
A adapter that implements the forward_model method, will take optical properties of absorption, scattering, and scattering anisotropy as input and return the light fluence as output.

Parameters:
- **optical_properties_path** – path to a .npz file that contains the following tags: Tags.PROPERTY_ABSORPTION_PER_CM -> contains the optical absorptions in units of one per centimeter Tags.PROPERTY_SCATTERING_PER_CM -> contains the optical scattering in units of one per centimeter Tags.PROPERTY_ANISOTROPY -> contains the dimensionless optical scattering anisotropy
- **settings** –

Returns:

simpa.core.optical_simulation.illumination_definition.**define_illumination**(global_settings, optical_simulation_settings, nx, ny, nz)

This method creates a dictionary that represents the illumination geometry in a way that it can be used with the respective illumination framework.

Parameters:
- **global_settings** – The top-level settings dictionary
- **optical_simulation_settings** – The settings file containing the simulation instructions
- **nx** – number of voxels along the x dimension of the volume
- **ny** – number of voxels along the y dimension of the volume
- **nz** – number of voxels along the z dimension of the volume

simpa.core.optical_simulation.illumination_definition.**define_illumination_mcx**(global_settings, optical_simulation_settings, nx, ny, nz) → dict

This method creates a dictionary that contains tags as they are expected for the mcx simulation tool to represent the illumination geometry.

Parameters:
- **global_settings** – The top-level settings dictionary
- **optical_simulation_settings** – The settings file containing the simulation instructions
- **nx** – number of voxels along the x dimension of the volume
- **ny** – number of voxels along the y dimension of the volume
- **nz** – number of voxels along the z dimension of the volume

mcx integration

class simpa.core.optical_simulation.mcx_adapter.**McxAdapter** (global_settings: simpa.utils.settings.Settings)

This class implements a bridge to the mcx framework to integrate mcx into SIMPA. MCX is a GPU-enabled Monte-Carlo model simulation of photon transport in tissue:

```
Fang, Qianqian, and David A. Boas. "Monte Carlo simulation of photon migration in 3D
turbid media accelerated by graphics processing units."
Optics express 17.22 (2009): 20178-20190.
```

forward_model(absorption_cm, scattering_cm, anisotropy)

A deriving class needs to implement this method according to its model.

Parameters:
- **absorption_cm** – Absorption in units of per centimeter
- **scattering_cm** – Scattering in units of per centimeter
- **anisotropy** – Dimensionless scattering anisotropy

Returns: Fluence in units of J/cm^2

Acoustic forward modeling

class simpa.core.acoustic_simulation.**AcousticForwardModelBaseAdapter** (global_settings: simpa.utils.settings.Settings)

This method is the entry method for running an acoustic forward model. It is invoked in the *simpa.core.simulation.simulate* method, but can also be called individually for the purposes of performing acoustic forward modeling only or in a different context.
The concrete will be chosen based on the:

```
Tags.ACOUSTIC_MODEL
```

tag in the settings dictionary.

Parameters: **settings** – The settings dictionary containing key-value pairs that determine the simulation. Here, it must contain the Tags.ACOUSTIC_MODEL tag and any tags that might be required by the specific acoustic model.

Raises: **AssertionError** – an assertion error is raised if the Tags.ACOUSTIC_MODEL tag is not given or points to an unknown acoustic forward model.

abstract **forward_model**() → numpy.ndarray

This method performs the acoustic forward modeling given the initial pressure distribution and the acoustic tissue properties contained in the settings file. A deriving class needs to implement this method according to its model.

Returns: time series pressure data

run()

Call this method to invoke the simulation process.

Parameters: **global_settings** – the settings dictionary containing all simulation parameters.

Returns: a numpy array containing the time series pressure data per detection element

k-Wave integration

class simpa.core.acoustic_simulation.k_wave_adapter.**KwaveAcousticForwardModelAdapter** (global_settings: simpa.utils.settings.Settings)

The KwaveAcousticForwardModel adapter enables acoustic simulations to be run with the k-wave MATLAB toolbox. k-Wave is a free toolbox (http://www.k-wave.org/) developed by Bradley Treeby and Ben Cox (University College London) and Jiri Jaros (Brno University of Technology).
In order to use this toolbox, MATLAB needs to be installed on your system and the path to the MATLAB binary needs to be specified in the settings dictionary.

In order to use the toolbox from with SIMPA, a number of parameters have to be specified in the settings dictionary:

```
The initial pressure distribution:
    Tags.OPTICAL_MODEL_INITIAL_PRESSURE
Acoustic tissue properties:
    Tags.PROPERTY_SPEED_OF_SOUND
    Tags.PROPERTY_DENSITY
    Tags.PROPERTY_ALPHA_COEFF
The digital twin of the imaging device:
    Tags.DIGITAL_DEVICE
Other parameters:
    Tags.PERFORM_UPSAMPLING
    Tags.SPACING_MM
    Tags.UPSCALE_FACTOR
    Tags.PROPERTY_ALPHA_POWER
    Tags.GPU
    Tags.PMLInside
    Tags.PMLAlpha
    Tags.PlotPML
    Tags.RECORDMOVIE
    Tags.MOVIENAME
    Tags.ACOUSTIC_LOG_SCALE
    Tags.SENSOR_DIRECTIVITY_PATTERN
```

Many of these will be set automatically by SIMPA, but you may use the simpa.utils.settings_generator convenience methods to generate settings files that contain sensible defaults for these parameters.
Please also refer to the simpa_examples scripts to see how the settings file can be parametrized successfully.

forward_model () → numpy.ndarray

This method performs the acoustic forward modeling given the initial pressure distribution and the acoustic tissue properties contained in the settings file. A deriving class needs to implement this method according to its model.

Returns: time series pressure data

Noise modeling

Image reconstruction

class simpa.core.image_reconstruction.**ReconstructionAdapterBase** (global_settings: simpa.utils.settings.Settings)

This class is the main entry point to perform image reconstruction using the SIMPA toolkit. All information necessary for the respective reconstruction method must be contained in the respective settings dictionary.

abstract **reconstruction_algorithm (**time_series_sensor_data**)** → numpy.ndarray

A deriving class needs to implement this method according to its model.

Parameters: **time_series_sensor_data** – the time series sensor data

Returns: a reconstructed photoacoustic image

run ()

Executes the respective simulation module

Parameters: **global_settings** – The global SIMPA settings dictionary

Backprojection

Time Revearsal

class simpa.core.image_reconstruction.TimeReversalAdapter.**TimeReversalAdapter**(global_settings: simpa.utils.settings.Settings)

The time reversal adapter includes the time reversal reconstruction algorithm implemented by the k-Wave toolkit into SIMPA.

Time reversal reconstruction uses the time series data and computes the forward simulation model backwards in time:

```
Treeby, Bradley E., Edward Z. Zhang, and Benjamin T. Cox.
"Photoacoustic tomography in absorbing acoustic media using
time reversal." Inverse Problems 26.11 (2010): 115003.
```

get_acoustic_properties(input_data: dict)

This method extracts the acoustic tissue properties from the settings dictionary and amends the information to the input_data.

Parameters:

- **global_settings** – the settings dictionary containing key value pairs with the simulation instructions.
- **input_data** – a dictionary containing the information needed for time reversal.

reconstruction_algorithm(time_series_sensor_data)

A deriving class needs to implement this method according to its model.

Parameters: **time_series_sensor_data** – the time series sensor data

Returns: a reconstructed photoacoustic image

Digital device twins

At every step along the forward simulation, knowledge of the photoacoustic device that is used for the measurements is needed. This is important to reflect characteristic artefacts and challenges for the respective device.

To this end, we have included digital twins of commonly used devices into the SIMPA core.

MSOT Acuity Echo

class simpa.core.device_digital_twins.msot_devices.**MSOTAcuityEcho**

This class represents a digital twin of the MSOT Acuity Echo, manufactured by iThera Medical, Munich, Germany (https://www.ithera-medical.com/products/msot-acuity/). It is based on the real specifications of the device, but due to the limitations of the possibilities how to represent a device in the software frameworks, constitutes only an approximation.

Some important publications that showcase the use cases of the MSOT Acuity and Acuity Echo device are:

```
Regensburger, Adrian P., et al. "Detection of collagens by multispectral optoacoustic
tomography as an imaging biomarker for Duchenne muscular dystrophy."
Nature Medicine 25.12 (2019): 1905-1915.

Knieling, Ferdinand, et al. "Multispectral Optoacoustic Tomography for Assessment of
Crohn's Disease Activity."
The New England journal of medicine 376.13 (2017): 1292.
```

adjust_simulation_volume_and_settings(global_settings: simpa.utils.settings.Settings)

In case that the PAI device needs space for the arrangement of detectors or illuminators in the volume, this method will update the volume accordingly.

check_settings_prerequisites(global_settings: simpa.utils.settings.Settings) → bool

It might be that certain device geometries need a certain dimensionality of the simulated PAI volume, or that it required the existence of certain Tags in the global global_settings. To this end, a PAI device should use this method to inform the user about a mismatch of the desired device and throw a ValueError if that is the case.

Raises: **ValueError** – raises a value error if the prerequisites are not matched.

Returns: True if the prerequisites are met, False if they are not met, but no exception has been raised.

get_detector_element_orientations (global_settings: simpa.utils.settings.Settings) → numpy.ndarray

This method yields a normalised orientation vector for each detection element. The length of this vector is the same as the one obtained via the position methods:

```
get_detector_element_positions_base_mm
get_detector_element_positions_accounting_for_device_position_mm
```

Returns: a numpy array that contains normalised orientation vectors for each detection element

get_detector_element_positions_accounting_for_device_position_mm (global_settings: simpa.utils.settings.Settings) → numpy.ndarray

Similar to:

```
get_detector_element_positions_base_mm
```

This method returns the absolute positions of the detection elements relative to the device position in the imaged volume, where the device position is defined by the following tag:

```
Tags.DIGITAL_DEVICE_POSITION
```

Returns: A numpy array containing the coordinates of the detection elements

get_detector_element_positions_base_mm () → numpy.ndarray

Defines the abstract positions of the detection elements in an arbitraty coordinate system. Typically, the center of the field of view is defined as the origin.

To obtain the positions in an interpretable coordinate system, please use the other method:

```
get_detector_element_positions_accounting_for_device_position_mm
```

Returns: A numpy array containing the position vestors of the detection elements.

get_illuminator_definition (global_settings: simpa.utils.settings.Settings)

Defines the illumination geometry of the device in the settings dictionary.

RSOM Explorer P50

class simpa.core.device_digital_twins.rsom_device.**RSOMExplorerP50**(element_spacing_mm=0.02)

This class represents an approximation of the Raster-scanning Optoacoustic Mesoscopy (RSOM) device built by iThera Medical (Munich, Germany). Please refer to the companie's website for more information (https://www.ithera-medical.com/products/rsom-explorer-p50/).

Since simulating thousands of individual forward modeling steps to obtain a single raster-scanned image is computationally not feasible, we approximate the process with a device design that has detection elements across the entire field of view. Because of this limitation we also need to approximate the light source with a homogeneous illumination across the field of view.

The digital device is modeled based on the reported specifications of the RSOM Explorer P50 system. Technical details of the system can be found in the dissertation of Mathias Schwarz (https://mediatum.ub.tum.de/doc/1324031/1324031.pdf) and you can find more details on use cases of the device in the following literature sources:

```
Yew, Yik Weng, et al. "Raster-scanning optoacoustic mesoscopy (RSOM) imaging
as an objective disease severity tool in atopic dermatitis patients."
Journal of the American Academy of Dermatology (2020).

Hindelang, B., et al. "Non-invasive imaging in dermatology and the unique
```

```
potential of raster-scan optoacoustic mesoscopy."
Journal of the European Academy of Dermatology and Venereology
33.6 (2019): 1051-1061.
```

adjust_simulation_volume_and_settings (global_settings: simpa.utils.settings.Settings)
In case that the PAI device needs space for the arrangement of detectors or illuminators in the volume, this method will update the volume accordingly.

check_settings_prerequisites (global_settings: simpa.utils.settings.Settings) → bool
It might be that certain device geometries need a certain dimensionality of the simulated PAI volume, or that it required the existence of certain Tags in the global global_settings. To this end, a PAI device should use this method to inform the user about a mismatch of the desired device and throw a ValueError if that is the case.

Raises: **ValueError** – raises a value error if the prerequisites are not matched.

Returns: True if the prerequisites are met, False if they are not met, but no exception has been raised.

get_detector_element_orientations (global_settings: simpa.utils.settings.Settings)
This method yields a normalised orientation vector for each detection element. The length of this vector is the same as the one obtained via the position methods:

```
get_detector_element_positions_base_mm
get_detector_element_positions_accounting_for_device_position_mm
```

Returns: a numpy array that contains normalised orientation vectors for each detection element

get_detector_element_positions_accounting_for_device_position_mm (global_settings: simpa.utils.settings.Settings)
Similar to:

```
get_detector_element_positions_base_mm
```

This method returns the absolute positions of the detection elements relative to the device position in the imaged volume, where the device position is defined by the following tag:

```
Tags.DIGITAL_DEVICE_POSITION
```

Returns: A numpy array containing the coordinates of the detection elements

get_detector_element_positions_base_mm ()
Defines the abstract positions of the detection elements in an arbitraty coordinate system. Typically, the center of the field of view is defined as the origin.
To obtain the positions in an interpretable coordinate system, please use the other method:

```
get_detector_element_positions_accounting_for_device_position_mm
```

Returns: A numpy array containing the position vestors of the detection elements.

get_illuminator_definition (global_settings: simpa.utils.settings.Settings)
Defines the illumination geometry of the device in the settings dictionary.

Module: utils

The utils module contains several general-purpose utility functions whose purpose it is to facilitate the use of SIMPA. The most important of these is the Tags class, which defines the strings and data types that have to be used for the keys and values of the settings dictionary.

class simpa.utils.tags.**Tags**

This class contains all ‘Tags’ for the use in the settings dictionary as well as strings that are used in SIMPA as naming conventions. Every Tag that is intended to be used as a key in the settings dictionary is represented by a tuple. The first element of the tuple is a string that corresponds to the name of the Tag. The second element of the tuple is a data type or a tuple of data types. The values that are assigned to the keys in the settings should match these data types. Their usage within the SIMPA package is divided in “SIMPA package”, “module X”, “adapter Y”, “class Z” and “naming convention”.

ACOUSTIC_LOG_SCALE = *('acoustic_log_scale', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, the movie of the kwave simulation will be recorded in a log scale.
Usage: adapter KwaveAcousticForwardModel

ACOUSTIC_MODEL = *('acoustic_model', <class 'str'>)*
Choice of the used acoustic model.
Usage: module acoustic_simulation

ACOUSTIC_MODEL_BINARY_PATH = *('acoustic_model_binary_path', <class 'str'>)*
Absolute path of the location of the acoustic forward model binary.
Usage: module optical_simulation

ACOUSTIC_MODEL_K_WAVE = *'kwave'*
Corresponds to the kwave simulaiton.
Usage: module acoustic_simulation, naming convention

ACOUSTIC_MODEL_OUTPUT_NAME = *'acoustic_forward_model_output'*
Name of the acoustic forward model output field in the SIMPA output file.
Usage: naming convention

ACOUSTIC_MODEL_SETTINGS = *('acoustic_model_settings', <class 'dict'>)*
Acoustic model settings.

ACOUSTIC_MODEL_TEST = *'simpa_tests'*
Corresponds to an adapter for testing purposes only.
Usage: module acoustic_simulation, naming convention

ACOUSTIC_SIMULATION_3D = *('acoustic_simulation_3d', <class 'bool'>)*
If True, simulates the acoustic forward model in 3D.
Usage: SIMPA package

ADHERE_TO_DEFORMATION = *('adhere_to_deformation', <class 'bool'>)*
If True, a structure will be shifted according to the deformation.
Usage: adapter versatile_volume_creation

BACKGROUND = *'Background'*
Corresponds to the name of a structure.
Usage: adapter versatile_volume_creation, naming convention

BANDPASS_CUTOFF_HIGHPASS = *('bandpass_cuttoff_highpass', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sets the cutoff threshold in MHz for highpass filtering, i.e. lower limit of the tukey filter. Default is 0.1 MHz
Usage: adapter PyTorchDASAdapter

BANDPASS_CUTOFF_LOWPASS = *('bandpass_cuttoff_lowpass', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sets the cutoff threshold in MHz for lowpass filtering, i.e. upper limit of the tukey filter. Default is 8 MHz
Usage: adapter PyTorchDASAdapter

CIRCULAR_TUBULAR_STRUCTURE = *'CircularTubularStructure'*
Corresponds to the CircularTubularStructure in the structure_library.
Usage: module volume_creation, naming_convention

CONSIDER_PARTIAL_VOLUME = *('consider_partial_volume', <class 'bool'>)*

If True, the structure will be generated with its edges only occupying a partial volume of the voxel.
Usage: adapter versatile_volume_creation

DATA_FIELD = *'data_field'*
Defines which data field a certain function shall be applied to.
Usage: module processing

DEFORMATION_X_COORDINATES_MM = *'deformation_x_coordinates'*
Mesh that defines the x coordinates of the deformation.
Usage: adapter versatile_volume_creation, naming convention

DEFORMATION_Y_COORDINATES_MM = *'deformation_y_coordinates'*
Mesh that defines the y coordinates of the deformation.
Usage: adapter versatile_volume_creation, naming convention

DEFORMATION_Z_ELEVATIONS_MM = *'deformation_z_elevation'*
Mesh that defines the z coordinates of the deformation.
Usage: adapter versatile_volume_creation, naming convention

DEFORMED_LAYERS_SETTINGS = *('deformed_layers_settings', <class 'dict'>)*
Settings that contain the functional which defines the deformation of the layers.
Usage: adapter versatile_volume_creation

DIGITAL_DEVICE = *('digital_device', <class 'str'>)*
Digital device that is chosen as illumination source and detector for the simulation.
Usage: SIMPA package

DIGITAL_DEVICE_INVISION = *'digital_device_invision'*
Corresponds to the InVision 256-TF device.
Usage: SIMPA package, naming convention

DIGITAL_DEVICE_MSOT = *'digital_device_msot'*
Corresponds to the MSOTAcuityEcho device.
Usage: SIMPA package, naming convention

DIGITAL_DEVICE_POSITION = *('digital_device_position', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Position in [x, y, z] coordinates of the device in the generated volume.
Usage: SIMPA package

DIGITAL_DEVICE_RSOM = *'digital_device_rsom'*
Corresponds to the RSOMExplorerP50 device.
Usage: SIMPA package, naming convention

DIM_VOLUME_X_MM = *('volume_x_dim_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Extent of the x-axis of the generated volume.
Usage: SIMPA package

DIM_VOLUME_Y_MM = *('volume_y_dim_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Extent of the y-axis of the generated volume.
Usage: SIMPA package

DIM_VOLUME_Z_MM = *('volume_z_dim_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Extent of the z-axis of the generated volume.
Usage: SIMPA package

DL_MODEL_PATH = *('dl_model_path', <class 'str'>)*
Absolute path to the deep learning model used for the deep learning upsampling.
Usage: module processing

ELLIPTICAL_TUBULAR_STRUCTURE = *'EllipticalTubularStructure'*

Corresponds to the EllipticalTubularStructure in the structure_library.
Usage: module volume_creation, naming_convention

GPU = *('gpu', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, uses all available gpu options of the used modules.
Usage: SIMPA package

HORIZONTAL_LAYER_STRUCTURE = *'HorizontalLayerStructure'*
Corresponds to the HorizontalLayerStructure in the structure_library.
Usage: module volume_creation, naming_convention

ILLUMINATION_DIRECTION = *('illumination_direction', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Direction of the photon source as [x, y, z] vector used in mcx.
Usage: module optical_modelling, adapter mcx_adapter

ILLUMINATION_PARAM1 = *('illumination_param1', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
First parameter group of the specified illumination type as [x, y, z, w] vector used in mcx.
Usage: module optical_modelling, adapter mcx_adapter

ILLUMINATION_PARAM2 = *('illumination_param2', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Second parameter group of the specified illumination type as [x, y, z, w] vector used in mcx.
Usage: module optical_modelling, adapter mcx_adapter

ILLUMINATION_POSITION = *('illumination_position', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Position of the photon source in [x, y, z] coordinates used in mcx.
Usage: module optical_modelling, adapter mcx_adapter

ILLUMINATION_TYPE = *('optical_model_illumination_type', <class 'str'>)*
Type of the illumination geometry used in mcx.
Usage: module optical_modelling, adapter mcx_adapter

ILLUMINATION_TYPE_DISK = *'disk'*
Corresponds to disk source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_DKFZ_PAUS = *'pasetup'*
Corresponds to pasetup source in mcx. The geometrical definition is described in:
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_FOURIER = *'fourier'*
Corresponds to fourier source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_FOURIER_X = *'fourierx'*
Corresponds to fourierx source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_FOURIER_X_2D = *'fourierx2d'*
Corresponds to fourierx2d source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_GAUSSIAN = *'gaussian'*
Corresponds to gaussian source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_MSOT_ACUITY_ECHO = *'msot_acuity_echo'*
Corresponds to msot_acuity_echo source in mcx. The device is manufactured by iThera Medical, Munich, Germany (https: // www.ithera-medical.com / products / msot-acuity /).
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_PATTERN = *'pattern'*

Corresponds to pattern source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_PATTERN_3D = *'pattern3d'*
Corresponds to pattern3d source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_PENCIL = *'pencil'*
Corresponds to pencil source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_PENCILARRAY = *'pencilarray'*
Corresponds to pencilarray source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_PLANAR = *'planar'*
Corresponds to planar source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_RING = *'ring'*
Corresponds to ring source in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_SLIT = *'slit'*
Corresponds to slit source in mcx.
Usage: adapter mcx_adapter, naming convention

INPUT_SEGMENTATION_VOLUME = *('input_segmentation_volume', <class 'numpy.ndarray'>)*
Array that defines a segmented volume.
Usage: adapter segmentation_based_volume_creator

K_WAVE_SPECIFIC_DT = *('dt_acoustic_sim', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Temporal resolution of kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter

K_WAVE_SPECIFIC_NT = *('Nt_acoustic_sim', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Total time steps simulated by kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter

LASER_PULSE_ENERGY_IN_MILLIJOULE = *('laser_pulse_energy_in_millijoule', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>, <class 'list'>, <class 'range'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Laser pulse energy used in the optical simulation.
Usage: module optical_simulation

MAX_DEFORMATION_MM = *'max_deformation'*
Maximum deformation in z-direction.
Usage: adapter versatile_volume_creation, naming convention

MCX_SEED = *('mcx_seed', (<class 'int'>, <class 'numpy.integer'>))*
Specific seed for random initialisation in mcx.
if not set, Tags.RANDOM_SEED will be used instead. Usage: module optical_modelling, adapter mcx_adapter

MEDIUM_TEMPERATURE_CELCIUS = *('medium_temperature', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Temperature of the simulated volume.
Usage: module noise_simulation

MOLECULE_COMPOSITION = *('molecule_composition', <class 'list'>)*
List that contains all the molecules within a structure.

Usage: module volume_creation

MOVIENAME = *('movie_name', <class 'str'>)*
Name of the movie recorded by kwave.
Usage: adapter KwaveAcousticForwardModel

NOISE_MEAN = *'noise_mean'*
Mean of a noise model.
Usage: module processing.noise_models

NOISE_MODE = *'noise_mode'*
The mode tag of a noise model is used to differentiate between Tags.NOISE_MODE_ADDITIVE and Tags.NOISE_MODE_MULTIPLICATIVE.
Usage: module processing.noise_models

NOISE_MODE_ADDITIVE = *'noise_mode_additive'*
A noise model shall be applied additively s_n = s + n.
Usage: module processing.noise_models

NOISE_MODE_MULTIPLICATIVE = *'noise_mode_multiplicative'*
A noise model shall be applied multiplicatively s_n = s * n.
Usage: module processing.noise_models

NOISE_NON_NEGATIVITY_CONSTRAINT = *'noise_non_negativity_constraint'*
Defines if after the noise model negative values shall be allowed.
Usage: module processing.noise_models

NOISE_STD = *'noise_std'*
Standard deviation of a noise model.
Usage: module processing.noise_models

OPTICAL_MODEL = *('optical_model', <class 'str'>)*
Choice of the used optical model.
Usage: module optical_simulation

OPTICAL_MODEL_BINARY_PATH = *('optical_model_binary_path', <class 'str'>)*
Absolute path of the location of the optical forward model binary.
Usage: module optical_simulation

OPTICAL_MODEL_FLUENCE = *'fluence'*
Name of the optical forward model output fluence field in the SIMPA output file.
Usage: naming convention

OPTICAL_MODEL_ILLUMINATION_GEOMETRY_XML_FILE = *('optical_model_illumination_geometry_xml_file', <class 'str'>)*
Absolute path of the location of the optical forward model illumination geometry.
Usage: module optical_simulation

OPTICAL_MODEL_INITIAL_PRESSURE = *'initial_pressure'*
Name of the optical forward model output initial pressure field in the SIMPA output file.
Usage: naming convention

OPTICAL_MODEL_MCX = *'mcx'*
Corresponds to the mcx simulation.
Usage: module optical_simulation, naming convention

OPTICAL_MODEL_MCXYZ = *'mcxyz'*
Corresponds to the mcxyz simulation.
Usage: module optical_simulation, naming convention

OPTICAL_MODEL_NUMBER_PHOTONS = *('optical_model_number_of_photons', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Number of photons used in the optical simulation.
Usage: module optical_simulation

OPTICAL_MODEL_OUTPUT_NAME = *'optical_forward_model_output'*
Name of the optical forward model output field in the SIMPA output file.
Usage: naming convention

OPTICAL_MODEL_SETTINGS = *('optical_model_settings', <class 'dict'>)*
Optical model settings

OPTICAL_MODEL_TEST = *'simpa_tests'*
Corresponds to an adapter for testing purposes only.
Usage: module optical_simulation, naming convention

OPTICAL_MODEL_UNITS = *'units'*
Name of the optical forward model output units field in the SIMPA output file.
Usage: naming convention

ORIGINAL_DATA = *'original_data'*
Name of the simulation outputs as original data in the SIMPA output file.
Usage: naming convention

PARALLELEPIPED_STRUCTURE = *'ParallelepipedStructure'*
Corresponds to the ParallelepipedStructure in the structure_library.
Usage: module volume_creation, naming_convention

PMLAlpha = *('pml_alpha', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Alpha coefficient of the “perfectly matched layer” (PML) around the simulated volume in kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PMLInside = *('pml_inside', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, the “perfectly matched layer” (PML) in kwave is located inside the volume.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PMLSize = *('pml_size', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Size of the “perfectly matched layer” (PML) around the simulated volume in kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PRIORITY = *('priority', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Number that corresponds to a priority of the assigned structure. If another structure occupies the same voxel in a volume, the structure with a higher priority will be preferred.
Usage: adapter versatile_volume_creator

PROPERTY_ABSORPTION_PER_CM = *'mua'*
Optical absorption of the generated volume/structure in 1/cm.
Usage: SIMPA package, naming convention

PROPERTY_ALPHA_COEFF = *'alpha_coeff'*
Acoustic attenuation of kwave of the generated volume/structure in dB/cm/MHz.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PROPERTY_ALPHA_POWER = *('medium_alpha_power', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Exponent of the exponential acoustic attenuation law of kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PROPERTY_ANISOTROPY = *'g'*
Optical scattering anisotropy of the generated volume/structure.
Usage: SIMPA package, naming convention

PROPERTY_DENSITY = *'density'*
Density of the generated volume/structure in kg/m³.
Usage: SIMPA package, naming convention

PROPERTY_DIRECTIVITY_ANGLE = *'directivity_angle'*
Directionality of the sensors in kwave of the used PA device.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PROPERTY_GRUNEISEN_PARAMETER = *'gamma'*
We define PROPERTY_GRUNEISEN_PARAMETER to contain all wavelength-independent constituents of the PA signal. This means that it contains the percentage of absorbed light converted into heat. Naturally, one could make an argument that this should not be the case, however, it simplifies the usage of this tool.
Usage: SIMPA package, naming convention

PROPERTY_OXYGENATION = *'oxy'*
Oxygenation of the generated volume/structure.
Usage: SIMPA package, naming convention

PROPERTY_SCATTERING_PER_CM = *'mus'*
Optical scattering (NOT REDUCED SCATTERING mus'! mus'=mus*(1-g)) of the generated volume/structure in 1/cm.
Usage: SIMPA package, naming convention

PROPERTY_SEGMENTATION = *'seg'*
Segmentation of the generated volume/structure.
Usage: SIMPA package, naming convention

PROPERTY_SENSOR_MASK = *'sensor_mask'*
Sensor mask of kwave of the used PA device.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

PROPERTY_SPEED_OF_SOUND = *'sos'*
Speed of sound of the generated volume/structure in m/s.
Usage: SIMPA package, naming convention

PlotPML = *('plot_pml', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, the "perfectly matched layer" (PML) around the simulated volume in kwave is plotted.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

RANDOM_SEED = *('random_seed', (<class 'int'>, <class 'numpy.integer'>))*
Random seed for numpy and torch.
Usage: SIMPA package

RECONSTRUCTED_DATA = *'reconstructed_data'*
Name of the reconstructed data field in the SIMPA output file.
Usage: naming convention

RECONSTRUCTED_DATA_NOISE = *'reconstructed_data_noise'*
Name of the reconstructed data with applied noise field in the SIMPA output file.
Usage: naming convention

RECONSTRUCTION_ALGORITHM = *('reconstruction_algorithm', <class 'str'>)*
Choice of the used reconstruction algorithm.
Usage: module image_reconstruction

RECONSTRUCTION_ALGORITHM_DAS = *'DAS'*
Corresponds to the reconstruction algorithm DAS with the MitkBeamformingAdapter.
Usage: module image_reconstruction, naming convention

RECONSTRUCTION_ALGORITHM_DMAS = *'DMAS'*
Corresponds to the reconstruction algorithm DMAS with the MitkBeamformingAdapter.

Usage: module image_reconstruction, naming convention

RECONSTRUCTION_ALGORITHM_PYTORCH_DAS = *'PyTorch_DAS'*
Corresponds to the reconstruction algorithm DAS with the PyTorchDASAdapter.
Usage: module image_reconstruction, naming convention

RECONSTRUCTION_ALGORITHM_SDMAS = *'sDMAS'*
Corresponds to the reconstruction algorithm sDMAS with the MitkBeamformingAdapter.
Usage: module image_reconstruction, naming convention

RECONSTRUCTION_ALGORITHM_TEST = *'TEST'*
Corresponds to an adapter for testing purposes only.
Usage: module image_reconstruction, naming convention

RECONSTRUCTION_ALGORITHM_TIME_REVERSAL = *'time_reversal'*
Corresponds to the reconstruction algorithm Time Reversal with TimeReversalAdapter.
Usage: module image_reconstruction, naming convention

RECONSTRUCTION_APODIZATION_BOX = *'BoxApodization'*
Corresponds to the box window function for apodization.
Usage: adapter PyTorchDASAdapter, naming convention

RECONSTRUCTION_APODIZATION_HAMMING = *'HammingApodization'*
Corresponds to the Hamming window function for apodization.
Usage: adapter PyTorchDASAdapter, naming convention

RECONSTRUCTION_APODIZATION_HANN = *'HannApodization'*
Corresponds to the Hann window function for apodization.
Usage: adapter PyTorchDASAdapter, naming convention

RECONSTRUCTION_APODIZATION_METHOD = *('reconstruction_apodization_method', <class 'str'>)*
Choice of the apodization method used, i.e. window functions .
Usage: adapter PyTorchDASAdapter

RECONSTRUCTION_BMODE_AFTER_RECONSTRUCTION = *'Envelope_Detection_after_Reconstruction'*
Specifies whether an envelope detection should be performed after reconstruction, default is False Usage: adapter PyTorchDASAdapter, naming convention

RECONSTRUCTION_BMODE_BEFORE_RECONSTRUCTION = *'Envelope_Detection_before_Reconstruction'*
Specifies whether an envelope detection should be performed before reconstruction, default is False Usage: adapter PyTorchDASAdapter, naming convention

RECONSTRUCTION_BMODE_METHOD = *('reconstruction_bmode_method', <class 'str'>)*
Choice of the B-Mode method used in the Mitk Beamforming.
Usage: adapter MitkBeamformingAdapter

RECONSTRUCTION_BMODE_METHOD_ABS = *'Abs'*
Corresponds to the absolute value as the B-Mode method used in the Mitk Beamforming.
Usage: adapter MitkBeamformingAdapter, naming convention

RECONSTRUCTION_BMODE_METHOD_HILBERT_TRANSFORM = *'EnvelopeDetection'*
Corresponds to the Hilbert transform as the B-Mode method used in the Mitk Beamforming.
Usage: adapter MitkBeamformingAdapter, naming convention

RECONSTRUCTION_INVERSE_CRIME = *('reconstruction_inverse_crime', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, the Time Reversal reconstruction will commit the “inverse crime”.
Usage: TimeReversalAdapter

RECONSTRUCTION_MITK_BINARY_PATH = *('reconstruction_mitk_binary_path', <class 'str'>)*
Absolute path to the Mitk Beamforming script.

Usage: adapter MitkBeamformingAdapter

RECONSTRUCTION_MITK_SETTINGS_XML = *('reconstruction_mitk_settings_xml', <class 'str'>)*
Absolute path to the Mitk Beamforming script settings.
Usage: adapter MitkBeamformingAdapter

RECONSTRUCTION_MODE = *('reconstruction_mode', <class 'str'>)*
Choice of the reconstruction mode used in the Backprojection.
Usage: adapter BackprojectionAdapter

RECONSTRUCTION_MODEL_SETTINGS = *('reconstruction_model_settings', <class 'dict'>)*
” Reconstruction Model Settings

RECONSTRUCTION_MODE_DIFFERENTIAL = *'differential'*
Corresponds to the differential mode used in the Backprojection.
Usage: adapter BackprojectionAdapter, naming_convention

RECONSTRUCTION_MODE_FULL = *'full'*
Corresponds to the full mode used in the Backprojection.
Usage: adapter BackprojectionAdapter, naming_convention

RECONSTRUCTION_MODE_PRESSURE = *'pressure'*
Corresponds to the pressure mode used in the Backprojection.
Usage: adapter BackprojectionAdapter, naming_convention

RECONSTRUCTION_OUTPUT_NAME = *('reconstruction_result', <class 'str'>)*
Absolute path of the image reconstruction result.
Usage: adapter MitkBeamformingAdapter

RECONSTRUCTION_PERFORM_BANDPASS_FILTERING = *('reconstruction_perform_bandpass_filtering', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
Whether bandpass filtering should be applied or not. Default should be True
Usage: adapter PyTorchDASAdapter

RECORDMOVIE = *('record_movie', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, a movie of the kwave simulation will be recorded.
Usage: adapter KwaveAcousticForwardModel

RECTANGULAR_CUBOID_STRUCTURE = *'RectangularCuboidStructure'*
Corresponds to the RectangularCuboidStructure in the structure_library.
Usage: module volume_creation, naming_convention

SEGMENTATION_CLASS_MAPPING = *('segmentation_class_mapping', <class 'dict'>)*
Mapping that assigns every class in the INPUT_SEGMENTATION_VOLUME a MOLECULE_COMPOSITION.
Usage: adapter segmentation_based_volume_creator

SENSOR_BANDWIDTH_PERCENT = *('sensor_bandwidth', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sensor bandwidth in kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_CENTER_FREQUENCY_HZ = *('sensor_center_frequency', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sensor center frequency in kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_CONCAVE = *'concave'*
Indicates that the geometry of the used PA device in the Mitk Beamforming is concave.
Usage: adapter MitkBeamformingAdapter, naming convention

SENSOR_DIRECTIVITY_PATTERN = *'sensor_directivity_pattern'*

Sensor directivity pattern of the sensor in kwave. Default should be “pressure”.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_DIRECTIVITY_SIZE_M = *('sensor_directivity_size', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Size of each detector element in kwave.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_LINEAR = *'linear'*
Indicates that the geometry of the used PA device in the Mitk Beamforming is linear.
Usage: adapter MitkBeamformingAdapter, naming convention

SENSOR_NUM_ELEMENTS = *('sensor_num_elements', (<class 'int'>, <class 'numpy.integer'>))*
Number of detector elements for kwave if no device was selected.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_NUM_USED_ELEMENTS = *('sensor_num_used_elements', (<class 'int'>, <class 'numpy.integer'>))*
Number of detector elements that fit into the generated volume if the dimensions and/or spacing of the generated volume were not highly resolved enough to be sufficient for the selected PA device.
Usage: module acoustic_simulation, naming convention

SENSOR_RADIUS_MM = *'sensor_radius_mm'*
Radius of a concave geometry of the used PA device in the Mitk Beamforming.
Usage: adapter MitkBeamformingAdapter, naming convention

SENSOR_RECORD = *('sensor_record', <class 'str'>)*
Sensor Record mode of the sensor in kwave. Default should be “p”.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SENSOR_SAMPLING_RATE_MHZ = *('sensor_sampling_rate_mhz', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sampling rate of the used PA device.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, naming convention

SETTINGS = *'settings'*
Location of the simulation settings in the SIMPA output file.
Usage: naming convention

SETTINGS_JSON = *('settings_json', (<class 'bool'>, <class 'numpy.bool_'>))*
If True, the SIMPA settings are saved in a .json file.
Usage: SIMPA package

SETTINGS_JSON_PATH = *('settings_json_path', <class 'str'>)*
Absolute path to a .json file if SETTINGS_JSON is set to True. Usage: SIMPA package

SIMPA_OUTPUT_NAME = *'simpa_output.hdf5'*
Default filename of the SIMPA output if not specified otherwise.
Usage: SIMPA package, naming convention

SIMPA_OUTPUT_PATH = *('simpa_output_path', <class 'str'>)*
Default path of the SIMPA output if not specified otherwise.
Usage: SIMPA package

SIMULATE_DEFORMED_LAYERS = *('simulate_deformed_layers', <class 'bool'>)*
If True, the horizontal layers are deformed according to the DEFORMED_LAYERS_SETTINGS.
Usage: adapter versatile_volume_creation

SIMULATIONS = *'simulations'*
Location of the simulation outputs in the SIMPA output file.
Usage: naming convention

SIMULATION_PATH = *('simulation_path', <class 'str'>)*
Absolute path to the folder where the SIMPA output is saved.
Usage: SIMPA package

SIMULATION_PROPERTIES = *'simulation_properties'*
Location of the simulation properties in the SIMPA output file.
Usage: naming convention

SPACING_MM = *('voxel_spacing_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Isotropic extent of one voxels in mm in the generated volume.
Usage: SIMPA package

SPHERICAL_STRUCTURE = *'SphericalStructure'*
Corresponds to the SphericalStructure in the structure_library.
Usage: module volume_creation, naming_convention

STRUCTURES = *('structures', <class 'dict'>)*
Settings dictionary which contains all the structures that should be generated inside the volume.
Usage: module volume_creation

STRUCTURE_BIFURCATION_LENGTH_MM = *('structure_bifurcation_length_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Length after which a VesselStructure will bifurcate.
Usage: adapter versatile_volume_creation, class VesselStructure

STRUCTURE_CURVATURE_FACTOR = *('structure_curvature_factor', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Factor that determines how strongly a vessel tree is curved.
Usage: adapter versatile_volume_creation, class VesselStructure

STRUCTURE_DIRECTION = *('structure_direction', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Direction as [x, y, z] vector starting from STRUCTURE_START_MM in which the vessel will grow.
Usage: adapter versatile_volume_creation, class VesselStructure

STRUCTURE_ECCENTRICITY = *('structure_excentricity', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>, <class 'numpy.ndarray'>))*
Eccentricity of the structure.
Usage: adapter versatile_volume_creation, class EllipticalTubularStructure

STRUCTURE_END_MM = *('structure_end', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Ending of the structure as [x, y, z] coordinates in the generated volume.
Usage: adapter versatile_volume_creation, class GeometricalStructure

STRUCTURE_FIRST_EDGE_MM = *('structure_first_edge_mm', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Edge of the structure as [x, y, z] vector starting from STRUCTURE_START_MM in the generated volume.
Usage: adapter versatile_volume_creation, class ParallelepipedStructure

STRUCTURE_RADIUS_MM = *('structure_radius', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>, <class 'numpy.ndarray'>))*
Radius of the structure.
Usage: adapter versatile_volume_creation, class GeometricalStructure

STRUCTURE_RADIUS_VARIATION_FACTOR = *('structure_radius_variation_factor', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Factor that determines how strongly a the radius of vessel tree varies.
Usage: adapter versatile_volume_creation, class VesselStructure

STRUCTURE_SECOND_EDGE_MM = *('structure_second_edge_mm', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Edge of the structure as [x, y, z] vector starting from STRUCTURE_START_MM in the generated volume.

Usage: adapter versatile_volume_creation, class ParallelepipedStructure

STRUCTURE_SEGMENTATION_TYPE = *'structure_segmentation_type'*
Defines the structure segmentation type to one segmentation type in SegmentationClasses.
Usage: module volume_creation, naming convention

STRUCTURE_START_MM = *('structure_start', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Beginning of the structure as [x, y, z] coordinates in the generated volume.
Usage: adapter versatile_volume_creation, class GeometricalStructure

STRUCTURE_THIRD_EDGE_MM = *('structure_third_edge_mm', (<class 'list'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Edge of the structure as [x, y, z] vector starting from STRUCTURE_START_MM in the generated volume.
Usage: adapter versatile_volume_creation, class ParallelepipedStructure

STRUCTURE_TYPE = *('structure_type', <class 'str'>)*
Defines the structure type to one structure in the structure_library.
Usage: module volume_creation

STRUCTURE_X_EXTENT_MM = *('structure_x_extent_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
X-extent of the structure in the generated volume.
Usage: adapter versatile_volume_creation, class RectangularCuboidStructure

STRUCTURE_Y_EXTENT_MM = *('structure_y_extent_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Y-extent of the structure in the generated volume.
Usage: adapter versatile_volume_creation, class RectangularCuboidStructure

STRUCTURE_Z_EXTENT_MM = *('structure_z_extent_mm', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Z-extent of the structure in the generated volume.
Usage: adapter versatile_volume_creation, class RectangularCuboidStructure

TIME_SERIES_DATA = *'time_series_data'*
Name of the time series data field in the SIMPA output file.
Usage: naming convention

TIME_SERIES_DATA_NOISE = *'time_series_data_noise'*
Name of the time series data with applied noise field in the SIMPA output file.
Usage: naming convention

TIME_STEP = *('time_step', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Temporal resolution of mcx.
Usage: adapter mcx_adapter

TISSUE_PROPERTIES_OUPUT_NAME = *'properties'*
Name of the simulation properties field in the SIMPA output file.
Usage: naming convention

TOTAL_TIME = *('total_time', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Total simulated time in mcx.
Usage: adapter mcx_adapter

TUKEY_WINDOW_ALPHA = *('tukey_window_alpha', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sets alpha value of Tukey window between 0 (similar to box window) and 1 (similar to Hann window). Default is 0.5
Usage: adapter PyTorchDASAdapter

UNITS_ARBITRARY = *'arbitrary_unity'*

Define arbitrary units if no units were given in the settings.
Usage: module optical_simulation, naming convention

UNITS_PRESSURE = *'newton_per_meters_squared'*
Standard units used in the SIMPA framework.
Usage: module optical_simulation, naming convention

UPSAMPLED_DATA = *'upsampled_data'*
Name of the simulation outputs as upsampled data in the SIMPA output file.
Usage: naming convention

UPSAMPLING_METHOD = *('upsampling_method', <class 'str'>)*
Choice of the upsampling method used in the image processing.
Usage: module processing

UPSAMPLING_METHOD_BILINEAR = *'bilinear'*
Corresponds to the bilinear upsampling method used in the image processing.
Usage: module processing, naming concention

UPSAMPLING_METHOD_DEEP_LEARNING = *'deeplearning'*
Corresponds to deep learning as the upsampling method used in the image processing.
Usage: module processing, naming concention

UPSAMPLING_METHOD_LANCZOS2 = *'lanczos2'*
Corresponds to lanczos with kernel size 2 as the upsampling method used in the image processing.
Usage: module processing, naming concention

UPSAMPLING_METHOD_LANCZOS3 = *'lanczos3'*
Corresponds to lanczos with kernel size 3 as the upsampling method used in the image processing.
Usage: module processing, naming concention

UPSAMPLING_METHOD_NEAREST_NEIGHBOUR = *'nearestneighbour'*
Corresponds to nearest neighbour as the upsampling method used in the image processing.
Usage: module processing, naming concention

UPSAMPLING_SCRIPT = *('upsampling_script', <class 'str'>)*
Name of the upsampling script used for the lanczos upsampling.
Usage: module processing

UPSCALE_FACTOR = *('upscale_factor', (<class 'int'>, <class 'float'>, <class 'numpy.int32'>, <class 'numpy.float64'>))*
Upscale factor of the upsampling in the image processing.
Usage: module processing

US_GEL = *('us_gel', <class 'bool'>)*
If True, us gel is placed between the PA device and the simulated volume.
Usage: SIMPA package

VOLUME_CREATION_MODEL_SETTINGS = *('volume_creation_model_settings', <class 'dict'>)*
" Volume Creation Model Settings

VOLUME_CREATOR = *('volume_creator', <class 'str'>)*
Choice of the volume creator adapter.
Usage: module volume_creation, module device_digital_twins

VOLUME_CREATOR_SEGMENTATION_BASED = *'volume_creator_segmentation_based'*
Corresponds to the SegmentationBasedVolumeCreator.
Usage: module volume_creation, naming convention

VOLUME_CREATOR_VERSATILE = *'volume_creator_versatile'*
Corresponds to the ModelBasedVolumeCreator.

Usage: module volume_creation, naming convention

VOLUME_NAME = *('volume_name', <class 'str'>)*
Name of the SIMPA output file.
Usage: SIMPA package

WAVELENGTH = *('wavelength', (<class 'int'>, <class 'numpy.integer'>))*
Single wavelength used for the current simulation.
Usage: SIMPA package

WAVELENGTHS = *('wavelengths', (<class 'list'>, <class 'range'>, <class 'tuple'>, <class 'numpy.ndarray'>))*
Iterable of all the wavelengths used for the simulation.
Usage: SIMPA package

simpa.utils.constants.**EPS** = *1e-20*
Defines the smallest increment that should be considered by SIMPA.

class simpa.utils.constants.**SaveFilePaths**
The save file paths specify the path of a specific data structure in the dictionary of the simpa output hdf5. All of these paths have to be used like: SaveFilePaths.PATH + “data_structure”

class simpa.utils.constants.**SegmentationClasses**
The segmentation classes define which “tissue types” are modelled in the simulation volumes.

simpa.utils.deformation_manager.**create_deformation_settings**(bounds_mm, maximum_z_elevation_mm=1, filter_sigma=1, cosine_scaling_factor=4)
FIXME

simpa.utils.deformation_manager.**get_functional_from_deformation_settings**(deformation_settings: dict)
FIXME

simpa.utils.dict_path_manager.**generate_dict_path**(data_field, wavelength: (<class 'int'>, <class 'float'>) = None) → str
Generates a path within an hdf5 file in the SIMPA convention

Parameters:
- **settings** – SIMPA Settings dictionary.
- **data_field** – Data field that is supposed to be stored in an hdf5 file.
- **wavelength** – Wavelength of the current simulation.

Returns: String which defines the path to the data_field.

class simpa.utils.path_manager.**PathManager**(environment_path=None)
As a pipelining tool that serves as a communication layer between different numerical forward models and processing tools, SIMPA needs to be configured with the paths to these tools on your local hard drive. To this end, we have implemented the *PathManager* class that you can import to your project using *from simpa.utils import PathManager*. The PathManager looks for a *path_config.env* file (just like the one we provided in the *simpa_examples*) in the following places in this order:

1. The optional path you give the PathManager
2. Your $HOME$ directory
3. The current working directory
4. The SIMPA home directory path

detect_local_path_config()
This methods looks in the default local paths for a path_config.env file.

class simpa.utils.settings.**Settings**(dictionary: dict = None)
The Settings class is a dictionary that contains all relevant settings for running a simulation in the SIMPA toolkit. It includes an automatic sanity check for input parameters using the simpa.utils.Tags class.

Usage: Seetings({Tags.KEY1: value1, Tags.KEY2: value2, …})

get_acoustic_settings ()
” Returns the settings for the acoustic forward model that are saved in this settings dictionary

get_optical_settings ()
” Returns the settings for the optical forward model that are saved in this settings dictionary

get_reconstruction_settings ()
” Returns the settings for the reconstruction model that are saved in this settings dictionary

get_volume_creation_settings ()
” Returns the settings for the optical forward model that are saved in this settings dictionary

set_acoustic_settings (`acoustic_settings: dict`**)**
Replaces the currently stored acoustic forward model settings with the given dictionary

Parameters: **acoustic_settings** – a dictionary containing the acoustic model settings

set_optical_settings (`optical_settings: dict`**)**
Replaces the currently stored optical settings with the given dictionary

Parameters: **optical_settings** – a dictionary containing the optical settings

set_reconstruction_settings (`reconstruction_settings: dict`**)**
Replaces the currently stored reconstruction model settings with the given dictionary

Parameters: **reconstruction_settings** – a dictionary containing the reconstruction model settings

set_volume_creation_settings (`volume_settings: dict`**)**
Replaces the currently stored volume creation settings with the given dictionary

Parameters: **volume_settings** – a dictionary containing the volume creator settings

`simpa.utils.calculate.`**`calculate_gruneisen_parameter_from_temperature`**
(`temperature_in_celcius`)
This function returns the dimensionless gruneisen parameter based on a heuristic formula that was determined experimentally:

```
@book{wang2012biomedical,
    title={Biomedical optics: principles and imaging},
    author={Wang, Lihong V and Wu, Hsin-i},
    year={2012},
    publisher={John Wiley \& Sons}
}
```

Parameters: **temperature_in_celcius** – the temperature in degrees celcius

Returns: a floating point number, if temperature_in_celcius is a number or a float array, if temperature_in_celcius is an array

`simpa.utils.calculate.`**`calculate_oxygenation`** (`molecule_list`)

Returns: an oxygenation value between 0 and 1 if possible, or None, if not computable.

`simpa.utils.calculate.`**`create_spline_for_range`** (`xmin_mm=0`, `xmax_mm=10`, `maximum_y_elevation_mm=1`, `spacing=0.1`)
Creates a functional that simulates distortion along the y position between the minimum and maximum x positions. The elevation can never be smaller than 0 or bigger than maximum_y_elevation_mm.

Parameters:
- **xmin_mm** – the minimum x axis value the return functional is defined in
- **xmax_mm** – the maximum x axis value the return functional is defined in
- **maximum_y_elevation_mm** – the maximum y axis value the return functional will yield

Returns: a functional that describes a distortion field along the y axis

`simpa.utils.calculate.`**`randomize_uniform`**(`min_value: float`, `max_value: float`)
returns a uniformly drawn random number in [min_value, max_value[

Parameters:
- **min_value** – minimum value
- **max_value** – maximum value

Returns: random number in [min_value, max_value[

`simpa.utils.calculate.`**`rotation`**(`angles`)
Rotation matrix around the x-, y-, and z-axis with angles [theta_x, theta_y, theta_z].

Parameters: **angles** – Angles through which the matrix is supposed to rotate in the form of [theta_x, theta_y, theta_z].

Returns: rotation matrix

`simpa.utils.calculate.`**`rotation_x`**(`theta`)
Rotation matrix around the x-axis with angle theta.

Parameters: **theta** – Angle through which the matrix is supposed to rotate.

Returns: rotation matrix

`simpa.utils.calculate.`**`rotation_y`**(`theta`)
Rotation matrix around the y-axis with angle theta.

Parameters: **theta** – Angle through which the matrix is supposed to rotate.

Returns: rotation matrix

`simpa.utils.calculate.`**`rotation_z`**(`theta`)
Rotation matrix around the z-axis with angle theta.

Parameters: **theta** – Angle through which the matrix is supposed to rotate.

Returns: rotation matrix

class `simpa.utils.tissue_properties.`**`TissueProperties`**

Libraries

Another important aspect of the utils class is the libraries that are being provided. These contain compilations of literature values for the acoustic and optical properties of commonly used tissue.

class `simpa.utils.libraries.molecule_library.`**`MolecularComposition`**(`segmentation_type=None`, `molecular_composition_settings=None`)

`update_internal_properties`()
FIXME

class `simpa.utils.libraries.literature_values.`**`MorphologicalTissueProperties`**
This class contains a listing of morphological tissue parameters as reported in literature. The listing is not the result of a meta analysis, but rather uses the best fitting paper at the time pf implementation. Each of the fields is annotated with a literature reference or a descriptions of how the particular values were derived for tissue modelling.

class `simpa.utils.libraries.literature_values.`**`OpticalTissueProperties`**

This class contains a listing of optical tissue parameters as reported in literature. The listing is not the result of a meta analysis, but rather uses the best fitting paper at the time pf implementation. Each of the fields is annotated with a literature reference or a descriptions of how the particular values were derived for tissue modelling.

class simpa.utils.libraries.literature_values.**StandardProperties**
This class contains a listing of default parameters that can be used. These values are sensible default values but are generally not backed up by proper scientific references, or are rather specific for internal use cases.

class simpa.utils.libraries.spectra_library.**AbsorptionSpectrum** (spectrum_name: str, wavelengths: numpy.ndarray, absorption_per_centimeter: numpy.ndarray)
An instance of this class represents the absorption spectrum over wavelength for a particular

get_absorption_for_wavelength (wavelength: int**)** → float

Parameters: **wavelength** – the wavelength to retrieve a optical absorption value for [cm^{-1}]. Must be an integer value between the minimum and maximum wavelength.

Returns: the best matching linearly interpolated absorption value for the given wavelength.

get_absorption_over_wavelength ()

Returns: numpy array with the available wavelengths and the corresponding absorption properties

simpa.utils.libraries.spectra_library.**view_absorption_spectra** (save_path=None)
Opens a matplotlib plot and visualizes the available absorption spectra.

Parameters: **save_path** – If not None, then the figure will be saved as a png file to the destination.

class simpa.utils.libraries.tissue_library.**MolecularCompositionGenerator**
The MolecularCompositionGenerator is a helper class to facilitate the creation of a MolecularComposition instance.

class simpa.utils.libraries.tissue_library.**TissueLibrary**
TODO

blood_arterial ()

Returns: a settings dictionary containing all min and max parameters fitting for full blood.

blood_generic (oxygenation=None**)**

Returns: a settings dictionary containing all min and max parameters fitting for full blood.

blood_venous ()

Returns: a settings dictionary containing all min and max parameters fitting for full blood.

bone ()

Returns: a settings dictionary containing all min and max parameters fitting for full blood.

constant (mua**,** mus**,** g**)**
TODO

dermis (background_oxy=0.5**)**

Returns: a settings dictionary containing all min and max parameters fitting for dermis tissue.

epidermis ()

Returns: a settings dictionary containing all min and max parameters fitting for epidermis tissue.

get_blood_volume_fractions (total_blood_volume_fraction**,** oxygenation**)**
TODO

muscle (background_oxy=0.5)

Returns: a settings dictionary containing all min and max parameters fitting for generic background tissue.

subcutaneous_fat (background_oxy=0.5)

Returns: a settings dictionary containing all min and max parameters fitting for subcutaneous fat tissue.

class simpa.utils.libraries.structure_library.**Background** (global_settings: simpa.utils.settings.Settings, background_settings: simpa.utils.settings.Settings = None)

Defines a background that fills the whole simulation volume. It is always given the priority of 0 so that other structures can overwrite it when necessary. Example usage:

background_dictionary = Settings() background_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.constant(0.1, 100.0, 0.9) background_dictionary[Tags.STRUCTURE_TYPE] = Tags.BACKGROUND

get_enclosed_indices ()

Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings)

Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings () → dict

Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**CircularTubularStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)

Defines a circular tube which is defined by a start and end point as well as a radius. This structure implements partial volume effects. The tube can be set to adhere to a deformation defined by the simpa.utils.deformation_manager. The start and end points of the tube will then be shifted along the z-axis accordingly. Example usage:

single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 9 structure[Tags.STRUCTURE_START_MM] = [50, 0, 50] structure[Tags.STRUCTURE_END_MM] = [50, 100, 50] structure[Tags.STRUCTURE_RADIUS_MM] = 5 structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood() structure[Tags.CONSIDER_PARTIAL_VOLUME] = True structure[Tags.ADHERE_TO_DEFORMATION] = True structure[Tags.STRUCTURE_TYPE] = Tags.CIRCULAR_TUBULAR_STRUCTURE

get_enclosed_indices ()

Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings)

Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()

Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**EllipticalTubularStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)

Defines a elliptical tube which is defined by a start and end point as well as a radius and an eccentricity. The elliptical geometry corresponds to a circular tube of the specified radius which is compressed along the z-axis until it reaches the specified eccentricity under the assumption of a constant volume. This structure implements partial volume effects. The tube can be set to adhere to a deformation defined by the simpa.utils.deformation_manager. The start and end points of the tube will then be shifted along the z-axis accordingly. Example usage:

```
# single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 9 structure[Tags.STRUCTURE_START_MM] = [50, 0, 50] structure[Tags.STRUCTURE_END_MM] = [50, 100, 50] structure[Tags.STRUCTURE_RADIUS_MM] = 5 structure[Tags.STRUCTURE_ECCENTRICITY] = 0.8 structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood() structure[Tags.CONSIDER_PARTIAL_VOLUME] = True structure[Tags.ADHERE_TO_DEFORMATION] = True structure[Tags.STRUCTURE_TYPE] = Tags.ELLIPTICAL_TUBULAR_STRUCTURE
```

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**GeometricalStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Base class for all model-based structures for ModelBasedVolumeCreator. A GeometricalStructure has an internal representation of its own geometry. This is represented by self.geometrical_volume which is a 3D array that defines for every voxel within the simulation volume if it is enclosed in the GeometricalStructure or if it is outside. Most of the GeometricalStructures implement a partial volume effect. So if a voxel has the value 1, it is completely enclosed by the GeometricalStructure. If a voxel has a value between 0 and 1, that fraction of the volume is occupied by the GeometricalStructure. If a voxel has the value 0, it is outside of the GeometricalStructure.

fill_internal_volume ()
Fills self.geometrical_volume of the GeometricalStructure.

***abstract* get_enclosed_indices ()**
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

***abstract* get_params_from_settings (**single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

properties_for_wavelength (wavelength**)** → simpa.utils.tissue_properties.TissueProperties
Returns the values corresponding to each optical/acoustic property used in SIMPA. :param wavelength: Wavelength of the queried properties :return: optical/acoustic properties

***abstract* to_settings ()** → simpa.utils.settings.Settings
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**HorizontalLayerStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Defines a Layer structure which spans the xy-plane in the SIMPA axis convention. The thickness of the layer is defined along the z-axis. This layer can be deformed by the simpa.utils.deformation_manager. Example usage:

single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 10 structure[Tags.STRUCTURE_START_MM] = [0, 0, 0] structure[Tags.STRUCTURE_END_MM] = [0, 0, 100] structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.epidermis() structure[Tags.CONSIDER_PARTIAL_VOLUME] = True structure[Tags.ADHERE_TO_DEFORMATION] = True structure[Tags.STRUCTURE_TYPE] = Tags.HORIZONTAL_LAYER_STRUCTURE

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**ParallelepipedStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Defines a parallelepiped which is defined by a start point and three edge vectors which originate from the start point. This structure currently does not implement partial volume effects. Example usage:

single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 9 structure[Tags.STRUCTURE_START_MM] = [25, 25, 25] structure[Tags.STRUCTURE_FIRST_EDGE_MM] = [5, 1, 1] structure[Tags.STRUCTURE_SECOND_EDGE_MM] = [1, 5, 1] structure[Tags.STRUCTURE_THIRD_EDGE_MM] = [1, 1, 5] structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle() structure[Tags.STRUCTURE_TYPE] = Tags.PARALLELEPIPED_STRUCTURE

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**RectangularCuboidStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Defines a rectangular cuboid (box) which is defined by a start point its extent along the x-, y-, and z-axis. This structure implements partial volume effects. The box can be set to adhere to a deformation defined by the simpa.utils.deformation_manager. The start point of the box will then be shifted along the z-axis accordingly. Example usage:

single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 9 structure[Tags.STRUCTURE_START_MM] = [25, 25, 25] structure[Tags.STRUCTURE_X_EXTENT_MM] = 40 structure[Tags.STRUCTURE_Y_EXTENT_MM] = 50 structure[Tags.STRUCTURE_Z_EXTENT_MM] = 60 structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle() structure[Tags.CONSIDER_PARTIAL_VOLUME] = True structure[Tags.ADHERE_TO_DEFORMATION] = True structure[Tags.STRUCTURE_TYPE] = Tags.RECTANGULAR_CUBOID_STRUCTURE

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**SphericalStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Defines a sphere which is defined by a start point and a radius. This structure implements partial volume effects. The sphere can be set to adhere to a deformation defined by the simpa.utils.deformation_manager. The start point of the sphere will then be shifted along the z-axis accordingly. Example usage:

```
# single_structure_settings initialization structure = Settings()

structure[Tags.PRIORITY] = 9 structure[Tags.STRUCTURE_START_MM] = [50, 50, 50] structure[Tags.STRUCTURE_RADIUS_MM] = 10 structure[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood() structure[Tags.CONSIDER_PARTIAL_VOLUME] = True structure[Tags.ADHERE_TO_DEFORMATION] = True structure[Tags.STRUCTURE_TYPE] = Tags.SPHERICAL_STRUCTURE
```

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (single_structure_settings**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

class simpa.utils.libraries.structure_library.**Structures** (settings: simpa.utils.settings.Settings, volume_creator_settings: dict)
TODO

class simpa.utils.libraries.structure_library.**VesselStructure** (global_settings: simpa.utils.settings.Settings, single_structure_settings: simpa.utils.settings.Settings = None)
Defines a vessel tree that is generated randomly in the simulation volume. The generation process begins at the start with a specified radius. The vessel grows roughly in the specified direction. The deviation is specified by the curvature factor. Furthermore, the radius of the vessel can vary depending on the specified radius variation factor. The bifurcation length defines how long a vessel can get until it will bifurcate. This structure implements partial volume effects. Example usage:

```
# single_structure_settings initialization structure_settings = Settings()

structure_settings[Tags.PRIORITY] = 10 structure_settings[Tags.STRUCTURE_START_MM] = [50, 0, 50] structure_settings[Tags.STRUCTURE_DIRECTION] = [0, 1, 0] structure_settings[Tags.STRUCTURE_RADIUS_MM] = 4 structure_settings[Tags.STRUCTURE_CURVATURE_FACTOR] = 0.05 structure_settings[Tags.STRUCTURE_RADIUS_VARIATION_FACTOR] = 1 structure_settings[Tags.STRUCTURE_BIFURCATION_LENGTH_MM] = 70 structure_settings[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood() structure_settings[Tags.CONSIDER_PARTIAL_VOLUME] = True structure_settings[Tags.STRUCTURE_TYPE] = Tags.VESSEL_STRUCTURE
```

fill_internal_volume ()
Fills self.geometrical_volume of the GeometricalStructure.

get_enclosed_indices ()
Gets indices of the voxels that are either entirely or partially occupied by the GeometricalStructure. :return: mask for a numpy array

get_params_from_settings (`single_structure_settings`**)**
Gets all the parameters required for the specific GeometricalStructure. :param single_structure_settings: Settings which describe the specific GeometricalStructure. :return: Tuple of parameters

to_settings ()
Creates a Settings dictionary which contains all the parameters needed to create the same GeometricalStructure again. :return : A tuple containing the settings key and the needed entries

Module: io_handling

`simpa.io_handling.io_hdf5.`**load_hdf5** (`file_path`, `file_dictionary_path='/'`)
Loads a dictionary from an hdf5 file.

Parameters:
- **file_path** – Path of the file to load the dictionary from.
- **file_dictionary_path** – Path in dictionary structure of hdf5 file to lo the dictionary in.

Returns: Dictionary

`simpa.io_handling.io_hdf5.`**save_hdf5** (`save_item`, `file_path: str`, `file_dictionary_path: str = '/'`, `file_compression: str = None`)
Saves a dictionary with arbitrary content or an item of any kind to an hdf5-file with given filepath.

Parameters:
- **save_item** – Dictionary to save.
- **file_path** – Path of the file to save the dictionary in.
- **file_dictionary_path** – Path in dictionary structure of existing hdf5 file to store the dictionary in.
- **file_compression** – possible file compression for the hdf5 output file. Values are: gzip, lzf and szip.

Returns: `Null`

Examples

Performing a complete forward simulation with acoustic modeling, optical modeling, as well as image reconstruction

The file can be found in simpa_examples/minimal_optical_simulation.py:

```
from simpa.utils import Tags, TISSUE_LIBRARY

from simpa.core.simulation import simulate
from simpa.utils.settings import Settings
from simpa.visualisation.matplotlib_data_visualisation import visualise_data
from simpa.core.device_digital_twins.msot_devices import MSOTAcuityEcho
import numpy as np
from simpa.utils.path_manager import PathManager

from simpa.core import *

# FIXME temporary workaround for newest Intel architectures
```

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

VOLUME_TRANSDUCER_DIM_IN_MM = 75
VOLUME_PLANAR_DIM_IN_MM = 20
VOLUME_HEIGHT_IN_MM = 25
SPACING = 0.25
RANDOM_SEED = 4711

# TODO: Please make sure that a valid path_config.env file is located in your home directory
#  point to the correct file in the PathManager().
path_manager = PathManager()

# If VISUALIZE is set to True, the simulation result will be plotted
VISUALIZE = True


def create_example_tissue():
    """
    This is a very simple example script of how to create a tissue definition.
    It contains a muscular background, an epidermis layer on top of the muscles
    and a blood vessel.
    """
    background_dictionary = Settings()
    background_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle()
    background_dictionary[Tags.STRUCTURE_TYPE] = Tags.BACKGROUND

    muscle_dictionary = Settings()
    muscle_dictionary[Tags.PRIORITY] = 1
    muscle_dictionary[Tags.STRUCTURE_START_MM] = [0, 0, 0]
    muscle_dictionary[Tags.STRUCTURE_END_MM] = [0, 0, 100]
    muscle_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.muscle()
    muscle_dictionary[Tags.CONSIDER_PARTIAL_VOLUME] = True
    muscle_dictionary[Tags.ADHERE_TO_DEFORMATION] = True
    muscle_dictionary[Tags.STRUCTURE_TYPE] = Tags.HORIZONTAL_LAYER_STRUCTURE

    vessel_1_dictionary = Settings()
    vessel_1_dictionary[Tags.PRIORITY] = 3
    vessel_1_dictionary[Tags.STRUCTURE_START_MM] = [VOLUME_TRANSDUCER_DIM_IN_MM/2,
                                                    0, 10]
    vessel_1_dictionary[Tags.STRUCTURE_END_MM] = [VOLUME_TRANSDUCER_DIM_IN_MM/2, VOLUME_PLAN
    vessel_1_dictionary[Tags.STRUCTURE_RADIUS_MM] = 3
    vessel_1_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood_generic()
    vessel_1_dictionary[Tags.CONSIDER_PARTIAL_VOLUME] = True
    vessel_1_dictionary[Tags.STRUCTURE_TYPE] = Tags.CIRCULAR_TUBULAR_STRUCTURE

    epidermis_dictionary = Settings()
    epidermis_dictionary[Tags.PRIORITY] = 8
    epidermis_dictionary[Tags.STRUCTURE_START_MM] = [0, 0, 0]
    epidermis_dictionary[Tags.STRUCTURE_END_MM] = [0, 0, 1]
    epidermis_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.epidermis()
    epidermis_dictionary[Tags.CONSIDER_PARTIAL_VOLUME] = True
    epidermis_dictionary[Tags.ADHERE_TO_DEFORMATION] = True
    epidermis_dictionary[Tags.STRUCTURE_TYPE] = Tags.HORIZONTAL_LAYER_STRUCTURE

    tissue_dict = Settings()
    tissue_dict[Tags.BACKGROUND] = background_dictionary
    tissue_dict["muscle"] = muscle_dictionary
    tissue_dict["epidermis"] = epidermis_dictionary
    tissue_dict["vessel_1"] = vessel_1_dictionary
```

```
    return tissue_dict


def add_msot_specific_settings(settings: Settings):
    volume_creator_settings = Settings(settings.get_volume_creation_settings())
    device = MSOTAcuityEcho()
    probe_size_mm = device.probe_height_mm
    mediprene_layer_height_mm = device.mediprene_membrane_height_mm
    heavy_water_layer_height_mm = probe_size_mm - mediprene_layer_height_mm

    new_volume_height_mm = settings[Tags.DIM_VOLUME_Z_MM] + mediprene_layer_height_mm + \
                           heavy_water_layer_height_mm

    # adjust the z-dim to msot probe height
    settings[Tags.DIM_VOLUME_Z_MM] = new_volume_height_mm

    # adjust the x-dim to msot probe width
    # 1 mm is added (0.5 mm on both sides) to make sure no rounding errors lead to a detecto
    # of the simulated volume.

    if settings[Tags.DIM_VOLUME_X_MM] < round(device.probe_width_mm) + 1:
        width_shift_for_structures_mm = (round(device.probe_width_mm) + 1 - settings[Tags.DI
        settings[Tags.DIM_VOLUME_X_MM] = round(device.probe_width_mm) + 1
        device.logger.debug(f"Changed Tags.DIM_VOLUME_X_MM to {settings[Tags.DIM_VOLUME_X_MM
    else:
        width_shift_for_structures_mm = 0

    device.logger.debug(volume_creator_settings)

    for structure_key in volume_creator_settings[Tags.STRUCTURES]:
        device.logger.debug("Adjusting " + str(structure_key))
        structure_dict = volume_creator_settings[Tags.STRUCTURES][structure_key]
        if Tags.STRUCTURE_START_MM in structure_dict:
            structure_dict[Tags.STRUCTURE_START_MM][0] = structure_dict[Tags.STRUCTURE_START
                                                         0] + width_shift_for_structures
            structure_dict[Tags.STRUCTURE_START_MM][2] = structure_dict[Tags.STRUCTURE_START
                                                         2] + device.probe_height_mm
        if Tags.STRUCTURE_END_MM in structure_dict:
            structure_dict[Tags.STRUCTURE_END_MM][0] = structure_dict[Tags.STRUCTURE_END_MM]
                                                       0] + width_shift_for_structures_m
            structure_dict[Tags.STRUCTURE_END_MM][2] = structure_dict[Tags.STRUCTURE_END_MM]
                                                       2] + device.probe_height_mm

    if Tags.US_GEL in volume_creator_settings and volume_creator_settings[Tags.US_GEL]:
        us_gel_thickness = np.random.normal(0.4, 0.1)
        us_gel_layer_settings = Settings({
            Tags.PRIORITY: 5,
            Tags.STRUCTURE_START_MM: [0, 0,
                                      heavy_water_layer_height_mm - us_gel_thickness + medip
            Tags.STRUCTURE_END_MM: [0, 0, heavy_water_layer_height_mm + mediprene_layer_heig
            Tags.CONSIDER_PARTIAL_VOLUME: True,
            Tags.MOLECULE_COMPOSITION: TISSUE_LIBRARY.ultrasound_gel(),
            Tags.STRUCTURE_TYPE: Tags.HORIZONTAL_LAYER_STRUCTURE
        })

        volume_creator_settings[Tags.STRUCTURES]["us_gel"] = us_gel_layer_settings
    else:
        us_gel_thickness = 0

    mediprene_layer_settings = Settings({
```

```
        Tags.PRIORITY: 5,
        Tags.STRUCTURE_START_MM: [0, 0, heavy_water_layer_height_mm - us_gel_thickness],
        Tags.STRUCTURE_END_MM: [0, 0, heavy_water_layer_height_mm - us_gel_thickness + medip
        Tags.CONSIDER_PARTIAL_VOLUME: True,
        Tags.MOLECULE_COMPOSITION: TISSUE_LIBRARY.mediprene(),
        Tags.STRUCTURE_TYPE: Tags.HORIZONTAL_LAYER_STRUCTURE
    })

    volume_creator_settings[Tags.STRUCTURES]["mediprene"] = mediprene_layer_settings

    background_settings = Settings({
        Tags.MOLECULE_COMPOSITION: TISSUE_LIBRARY.heavy_water(),
        Tags.STRUCTURE_TYPE: Tags.BACKGROUND
    })
    volume_creator_settings[Tags.STRUCTURES][Tags.BACKGROUND] = background_settings

# Seed the numpy random configuration prior to creating the global_settings file in
# order to ensure that the same volume
# is generated with the same random seed every time.

np.random.seed(RANDOM_SEED)
VOLUME_NAME = "CompletePipelineTestMSOT_"+str(RANDOM_SEED)

general_settings = {
            # These parameters set the general properties of the simulated volume
            Tags.RANDOM_SEED: RANDOM_SEED,
            Tags.VOLUME_NAME: "CompletePipelineTestMSOT_" + str(RANDOM_SEED),
            Tags.SIMULATION_PATH: path_manager.get_hdf5_file_save_path(),
            Tags.SPACING_MM: SPACING,
            Tags.DIM_VOLUME_Z_MM: VOLUME_HEIGHT_IN_MM,
            Tags.DIM_VOLUME_X_MM: VOLUME_TRANSDUCER_DIM_IN_MM,
            Tags.DIM_VOLUME_Y_MM: VOLUME_PLANAR_DIM_IN_MM,
            Tags.VOLUME_CREATOR: Tags.VOLUME_CREATOR_VERSATILE,

            # Simulation Device
            Tags.DIGITAL_DEVICE: Tags.DIGITAL_DEVICE_MSOT,

            # The following parameters set the optical forward model
            Tags.WAVELENGTHS: [700]
        }
settings = Settings(general_settings)
np.random.seed(RANDOM_SEED)

settings.set_volume_creation_settings({
    Tags.STRUCTURES: create_example_tissue(),
    Tags.SIMULATE_DEFORMED_LAYERS: True
})

settings.set_optical_settings({
    Tags.OPTICAL_MODEL_NUMBER_PHOTONS: 1e7,
    Tags.OPTICAL_MODEL_BINARY_PATH: path_manager.get_mcx_binary_path(),
    Tags.ILLUMINATION_TYPE: Tags.ILLUMINATION_TYPE_MSOT_ACUITY_ECHO,
    Tags.LASER_PULSE_ENERGY_IN_MILLIJOULE: 50,
})

settings.set_acoustic_settings({
    Tags.ACOUSTIC_SIMULATION_3D: True,
    Tags.ACOUSTIC_MODEL_BINARY_PATH: path_manager.get_matlab_binary_path(),
    Tags.GPU: True,
    Tags.PROPERTY_ALPHA_POWER: 1.05,
```

```
    Tags.SENSOR_RECORD: "p",
    Tags.PMLInside: False,
    Tags.PMLSize: [31, 32],
    Tags.PMLAlpha: 1.5,
    Tags.PlotPML: False,
    Tags.RECORDMOVIE: False,
    Tags.MOVIENAME: "visualization_log",
    Tags.ACOUSTIC_LOG_SCALE: True
})

settings.set_reconstruction_settings({
    Tags.RECONSTRUCTION_PERFORM_BANDPASS_FILTERING: False,
    Tags.TUKEY_WINDOW_ALPHA: 0.5,
    Tags.BANDPASS_CUTOFF_LOWPASS: int(8e6),
    Tags.BANDPASS_CUTOFF_HIGHPASS: int(0.1e6),
    Tags.RECONSTRUCTION_BMODE_METHOD: Tags.RECONSTRUCTION_BMODE_METHOD_HILBERT_TRANSFORM,
    Tags.RECONSTRUCTION_APODIZATION_METHOD: Tags.RECONSTRUCTION_APODIZATION_BOX,
    Tags.RECONSTRUCTION_MODE: Tags.RECONSTRUCTION_MODE_DIFFERENTIAL
})

settings["noise_initial_pressure"] = {
    Tags.NOISE_MEAN: 1,
    Tags.NOISE_STD: 0.1,
    Tags.NOISE_MODE: Tags.NOISE_MODE_MULTIPLICATIVE,
    Tags.DATA_FIELD: Tags.OPTICAL_MODEL_INITIAL_PRESSURE,
    Tags.NOISE_NON_NEGATIVITY_CONSTRAINT: True
}

settings["noise_time_series"] = {
    Tags.NOISE_STD: 3,
    Tags.NOISE_MODE: Tags.NOISE_MODE_ADDITIVE,
    Tags.DATA_FIELD: Tags.TIME_SERIES_DATA
}

add_msot_specific_settings(settings)

SIMUATION_PIPELINE = [
    ModelBasedVolumeCreator(settings),
    McxAdapter(settings),
    GaussianNoiseModel(settings, "noise_initial_pressure"),
    KwaveAcousticForwardModelAdapter(settings),
    GaussianNoiseModel(settings, "noise_time_series"),
    DelayAndSumAdapter(settings)
]

simulate(SIMUATION_PIPELINE, settings)

if Tags.WAVELENGTH in settings:
    WAVELENGTH = settings[Tags.WAVELENGTH]
else:
    WAVELENGTH = 700

if VISUALIZE:
    visualise_data(path_manager.get_hdf5_file_save_path() + "/" + VOLUME_NAME + ".hdf5", WAV
                   show_time_series_data=False,
                   show_absorption=False,
                   show_segmentation_map=False,
                   show_tissue_density=False,
                   show_reconstructed_data=True,
                   show_fluence=False)
```

Reading the HDF5 simulation output

The file can be found in simpa_examples/access_saved_PAI_data.py:

Defining custom tissue structures and properties

The file can be found in simpa_examples/create_custom_tissues.py:

```
from simpa.utils import MolecularCompositionGenerator
from simpa.utils import MOLECULE_LIBRARY
from simpa.utils import Molecule
from simpa.utils import AbsorptionSpectrum
import numpy as np


def create_custom_absorber():
    wavelengths = np.linspace(200, 1500, 100)
    absorber = AbsorptionSpectrum(spectrum_name="random absorber",
                                  wavelengths=wavelengths,
                                  absorption_per_centimeter=np.random.random(
                                      np.shape(wavelengths)))
    return absorber


def create_custom_chromophore(volume_fraction: float = 1.0):
    chromophore = Molecule(
            spectrum=create_custom_absorber(),
            volume_fraction=volume_fraction,
            mus500=40.0,
            b_mie=1.1,
            f_ray=0.9,
            anisotropy=0.9
        )
    return chromophore


def create_custom_tissue_type():

    # First create an instance of a TissueSettingsGenerator
    tissue_settings_generator = MolecularCompositionGenerator()

    water_volume_fraction = 0.4
    blood_volume_fraction = 0.5
    custom_chromophore_volume_fraction = 0.1
    # The volume fraction within every tissue type should sum up to 1.

    oxygenation = 0.4

    # Then append chromophores that you want
    tissue_settings_generator.append(key="oxyhemoglobin",
                                     value=MOLECULE_LIBRARY.oxyhemoglobin(oxygenation * bloo
    tissue_settings_generator.append(key="deoxyhemoglobin",
                                     value=MOLECULE_LIBRARY.deoxyhemoglobin((1 - oxygenation
    tissue_settings_generator.append(key="water",
                                     value=MOLECULE_LIBRARY.water(water_volume_fraction))
    tissue_settings_generator.append(key="custom",
                                     value=create_custom_chromophore(custom_chromophore_volu

    return tissue_settings_generator
```

Index

S

Python Module Index

s