SIMPA

version 0.4.0

CAMI (Computer Assisted Medical Interventions), DKFZ, Heidelberg and Cancer Research UK, Cambridge Institute (CRUK CI)

June 03, 2021

Contents

Welcome to the SIMPA documentation!

SIMPA

Simulation and Image Processing
for Photoacoustic Imaging

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. Please note that there might be compatiblity issues using mcx-cl with the MCX Adapter as this use case is not being tested and supported by the SIMPA developers.

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 (https://github.com/CAMI-DKFZ/simpa/). As such, only the current master and develop branch of the repository are open source.

To make us aware of an issue, please create an issue on the SIMPA github repository.

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 obtain the meeting links from the core developer team. We also have a Slack workspace that you can join if you are interested to contribute.

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 description details 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`, `digital_device_twin: simpa.core.device_digital_twins.digital_device_twin_base.DigitalDeviceTwinBase`)

This method constitutes the staring point for the simulation pipeline of the SIMPA toolkit.

Parameters:

- **simulation_pipeline** – a list of callable functions
- **settings** – settings dictionary containing the simulation instructions
- **digital_device_twin** – a digital device twin of an imaging device as specified by the DigitalDeviceTwinBase class.

Raises:

- **TypeError** – if one of the given parameters is not of the correct type
- **AssertionError** – if the digital device twin is not able to simulate the settings specification

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_module.`**`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` **(**`device`**)**

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

Model-based volume creation

class `simpa.core.volume_creation_module.volume_creation_module_model_based_adapter.`**`VolumeCreationModelModelBasedAdapter`** (`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

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.volume_creation_module.volume_creation_module_segmentation_based_adapter.`**`VolumeCreationModuleSegmentationBasedAdapter`** `(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 modelling

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.optical_simulation_module.`**`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`, `illumination_geometry`, `probe_position_mm`**)**

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
- **illumination_geometry** – A device that represents a detection geometry

Returns: Fluence in units of J/cm^2

run (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

mcx integration

class `simpa.core.optical_simulation_module.optical_forward_model_mcx_adapter.`**`OpticalForwardModelMcxAdapter`** (`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, illumination_geometry, probe_position_mm`)

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
- **illumination_geometry** – A device that represents a detection geometry

Returns: Fluence in units of J/cm^2

Acoustic forward modelling

class `simpa.core.acoustic_forward_module.`**`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`** (`detection_geometry`) → 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` (`digital_device_twin`)

Call this method to invoke the simulation process.

Parameters: **digital_device_twin** –

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

k-Wave integration

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.acoustic_forward_module.acoustic_forward_module_k_wave_adapter.`**`AcousticForwardModelKWaveAdapter`** (`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` (`detection_geometry`) → 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

Image reconstruction

class `simpa.core.reconstruction_module.`**`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`**,** `detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase`**)** → numpy.ndarray

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

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

Returns: a reconstructed photoacoustic image

`run` **(**`device`**)**

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

`simpa.core.reconstruction_module.reconstruction_utils.`**`apply_b_mode`** `(data: numpy.ndarray = None, method: str = None)` → numpy.ndarray

Applies B-Mode specified method to data. Method is either envelope detection using hilbert transform (Tags.RECONSTRUCTION_BMODE_METHOD_HILBERT_TRANSFORM), absolute value (Tags.RECONSTRUCTION_BMODE_METHOD_ABS) or none if nothing is specified is performed.

Parameters:
- **data** – (numpy array) data used for applying B-Mode method
- **method** – (str) Tags.RECONSTRUCTION_BMODE_METHOD_HILBERT_TRANSFORM or Tags.RECONSTRUCTION_BMODE_METHOD_ABS

Returns: (numpy array) data with B-Mode method applied, all

`simpa.core.reconstruction_module.reconstruction_utils.`**`bandpass_filtering`** `(data: None._VariableFunctionsClass.tensor = None, time_spacing_in_ms: float = None, cutoff_lowpass: int = 8000000, cutoff_highpass: int = 100000, tukey_alpha: float = 0.5)` → None._VariableFunctionsClass.tensor

Apply a bandpass filter with cutoff values at *cutoff_lowpass* and *cutoff_highpass* MHz and a tukey window with alpha value of *tukey_alpha* inbetween on the *data* in Fourier space.

Parameters:
- **data** – (torch tensor) data to be filtered
- **time_spacing_in_ms** – (float) time spacing in milliseconds, e.g. 2.5e-5
- **cutoff_lowpass** – (int) Signal above this value will be ignored (in MHz)
- **cutoff_highpass** – (int) Signal below this value will be ignored (in MHz)
- **tukey_alpha** – (float) transition value between 0 (rectangular) and 1 (Hann window)

Returns: (torch tensor) filtered data

`simpa.core.reconstruction_module.reconstruction_utils.`**`get_apodization_factor`** `(apodization_method: str = 'BoxApodization', dimensions: tuple = None, n_sensor_elements=None, device: torch.device = 'cpu')` → None._VariableFunctionsClass.tensor

Construct apodization factors according to *apodization_method* [hann, hamming or box apodization (default)] for given dimensions and *n_sensor_elements*.

Parameters:

- **apodization_method** – (str) Apodization method, one of Tags.RECONSTRUCTION_APODIZATION_HANN, Tags.RECONSTRUCTION_APODIZATION_HAMMING and Tags.RECONSTRUCTION_APODIZATION_BOX (default)
- **dimensions** – (tuple) size of each dimension of reconstructed image as int, might have 2 or 3 entries.
- **n_sensor_elements** – (int) number of sensor elements
- **device** – (torch device) PyTorch tensor device

Returns: (torch tensor) tensor with apodization factors which can be multipied with DAS values

`simpa.core.reconstruction_module.reconstruction_utils.`**`reconstruction_mode_transformation`**`(time_series_sensor_data: None._VariableFunctionsClass.tensor = None, mode: str = 'pressure')` → None._VariableFunctionsClass.tensor

Transformes *time_series_sensor_data* for other modes, for example *Tags.RECONSTRUCTION_MODE_DIFFERENTIAL*. Default mode is *Tags.RECONSTRUCTION_MODE_PRESSURE*.

Parameters:

- **time_series_sensor_data** – (torch tensor) Time series data to be transformed
- **mode** – (str) reconstruction mode: Tags.RECONSTRUCTION_MODE_PRESSURE (default) or Tags.RECONSTRUCTION_MODE_DIFFERENTIAL

Returns: (torch tensor) potentially transformed tensor

Backprojection

class `simpa.core.reconstruction_module.reconstruction_module_delay_and_sum_adapter.`**`ImageReconstructionModuleDelayAndSumAdapter`**`(global_settings: simpa.utils.settings.Settings)`

`reconstruction_algorithm` **(**`time_series_sensor_data, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase`**)**

Applies the Delay and Sum beamforming algorithm [1] to the time series sensor data (2D numpy array where the first dimension corresponds to the sensor elements and the second to the recorded time steps) with the given beamforming settings (dictionary). A reconstructed image (2D numpy array) is returned. This implementation uses PyTorch Tensors to perform computations and is able to run on GPUs.

[1] T. Kirchner et al. 2018, “Signed Real-Time Delay Multiply and Sum Beamforming for Multispectral Photoacoustic Imaging”, https://doi.org/10.3390/jimaging4100121

`simpa.core.reconstruction_module.reconstruction_module_delay_and_sum_adapter.`**`reconstruct_dela`**`(time_series_sensor_data: numpy.ndarray, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase, settings: dict = None, sound_of_speed: int = 1540, time_spacing: float = 2.5e-08, sensor_spacing: float = 0.1)` → numpy.ndarray

Convenience function for reconstructing time series data using Delay and Sum algorithm implemented in PyTorch

Parameters:

- **time_series_sensor_data** – (2D numpy array) sensor data of shape (sensor elements, time steps)
- **detection_geometry** – The DetectionGeometryBase that should be used to reconstruct the given time series data
- **settings** – (dict) settings dictionary: by default there is none and the other parameters are used instead, but if parameters are given in the settings those will be used instead of parsed arguments)
- **sound_of_speed** – (int) speed of sound in medium in meters per second (default: 1540 m/s)
- **time_spacing** – (float) time between sampling points in seconds (default: 2.5e-8 s which is equal to 40 MHz)
- **sensor_spacing** – (float) space between sensor elements in millimeters (default: 0.1 mm)

Returns: (2D numpy array) reconstructed image as 2D numpy array

Delay-Multiply-And-Sum (DMAS)

class `simpa.core.reconstruction_module.reconstruction_module_delay_multiply_and_sum_adapter.`**`ImageReconstructionModuleDelayMultiplyAndSumAdapter`**`(global_settings: simpa.utils.settings.Settings)`

`reconstruction_algorithm``(time_series_sensor_data, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase)`

Applies the Delay Multiply and Sum beamforming algorithm [1] to the time series sensor data (2D numpy array where the first dimension corresponds to the sensor elements and the second to the recorded time steps) with the given beamforming settings (dictionary). A reconstructed image (2D numpy array) is returned. This implementation uses PyTorch Tensors to perform computations and is able to run on GPUs.

[1] T. Kirchner et al. 2018, “Signed Real-Time Delay Multiply and Sum Beamforming for Multispectral Photoacoustic Imaging”, https://doi.org/10.3390/jimaging4100121

`simpa.core.reconstruction_module.reconstruction_module_delay_multiply_and_sum_adapter.`**`reconst`**`(time_series_sensor_data: numpy.ndarray, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase, settings: dict = None, sound_of_speed: int = 1540, time_spacing: float = 2.5e-08, sensor_spacing: float = 0.1)` → numpy.ndarray

Convenience function for reconstructing time series data using Delay and Sum algorithm implemented in PyTorch

Parameters:

- **time_series_sensor_data** – (2D numpy array) sensor data of shape (sensor elements, time steps)
- **detection_geometry** – The DetectioNGeometryBase to use for the reconstruction of the given time series data
- **settings** – (dict) settings dictionary: by default there is none and the other parameters are used instead, but if parameters are given in the settings those will be used instead of parsed arguments)
- **sound_of_speed** – (int) speed of sound in medium in meters per second (default: 1540 m/s)
- **time_spacing** – (float) time between sampling points in seconds (default: 2.5e-8 s which is equal to 40 MHz)
- **sensor_spacing** – (float) space between sensor elements in millimeters (default: 0.1 mm)

Returns: (2D numpy array) reconstructed image as 2D numpy array

signed Delay-Multiply-And-Sum (sDMAS)

class `simpa.core.reconstruction_module.reconstruction_module_signed_delay_multiply_and_sum_adapter.`**`ImageReconstructionModuleSignedDelayMultiplyAndSumAdapter`**`(global_settings: simpa.utils.settings.Settings)`

`reconstruction_algorithm``(time_series_sensor_data, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase)`

Applies the signed Delay Multiply and Sum beamforming algorithm [1] to the time series sensor data (2D numpy array where the first dimension corresponds to the sensor elements and the second to the recorded time steps) with the given beamforming settings (dictionary). A reconstructed image (2D numpy array) is returned. This implementation uses PyTorch Tensors to perform computations and is able to run on GPUs.

[1] T. Kirchner et al. 2018, "Signed Real-Time Delay Multiply and Sum Beamforming for Multispectral Photoacoustic Imaging", https://doi.org/10.3390/jimaging4100121

`simpa.core.reconstruction_module.reconstruction_module_signed_delay_multiply_and_sum_adapter.`
`(time_series_sensor_data: numpy.ndarray, detection_geometry: simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.DetectionGeometryBase, settings: dict = None, sound_of_speed: int = 1540, time_spacing: float = 2.5e-08, sensor_spacing: float = 0.1)` → numpy.ndarray

Convenience function for reconstructing time series data using Delay and Sum algorithm implemented in PyTorch

Parameters:

- **time_series_sensor_data** – (2D numpy array) sensor data of shape (sensor elements, time steps)
- **settings** – (dict) settings dictionary: by default there is none and the other parameters are used instead, but if parameters are given in the settings those will be used instead of parsed arguments)
- **sound_of_speed** – (int) speed of sound in medium in meters per second (default: 1540 m/s)
- **time_spacing** – (float) time between sampling points in seconds (default: 2.5e-8 s which is equal to 40 MHz)
- **sensor_spacing** – (float) space between sensor elements in millimeters (default: 0.1 mm)

Returns: (2D numpy array) reconstructed image as 2D numpy array

Time Reversal

class `simpa.core.reconstruction_module.reconstruction_module_time_reversal_adapter.`**`ReconstructionModuleTimeReversalAdapter`**`(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, detection_geometry)`

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`, `detection_geometry`)

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

Parameters:

- **time_series_sensor_data** – the time series sensor data
- **detection_geometry** –

Returns: a reconstructed photoacoustic image

`reorder_time_series_data` (`time_series_sensor_data`, `detection_geometry`)

Reorders the time series data to match the order that is assumed by kwave during image reconstruction with TimeReversal.

The main issue here is, that, while forward modelling allows for the definition of 3D cuboid bounding boxes for the detector elements, TimeReversal does not implement this feature. Instead, a binary mask is given and these are indexed in a column-row-wise manner in the output. The default np.argsort() method does not yield the same result as expected by k-Wave. Hence, this workaround.

Processing Components

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.processing_components.`**`ProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Defines a simulation component, which can be used to pre- or post-process simulation data.

Noise Models

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.processing_components.noise.gamma_noise.`**`GammaNoiseProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Applies Gaussian noise to the defined data field. The noise will be applied to all wavelengths.

Component Settings:

```
Tags.NOISE_SHAPE (default: 2)
Tags.NOISE_SCALE (default: 2)
Tags.NOISE_MODE (default: Tags.NOISE_MODE_ADDITIVE)
Tags.DATA_FIELD (required)
```

`run` (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.processing_components.noise.gaussian_noise.`**`GaussianNoiseProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Applies Gaussian noise to the defined data field. The noise will be applied to all wavelengths. Component Settings:

```
Tags.NOISE_MEAN (default: 0)
Tags.NOISE_STD (default: 1)
Tags.NOISE_MODE (default: Tags.NOISE_MODE_ADDITIVE)
Tags.NOISE_NON_NEGATIVITY_CONSTRAINT (default: False)
Tags.DATA_FIELD (required)
```

run (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

class `simpa.core.processing_components.noise.poisson_noise.`**`PoissonNoiseProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Applies Gaussian noise to the defined data field. The noise will be applied to all wavelengths.
Component Settings:

```
Tags.NOISE_MEAN (default: 3)
Tags.NOISE_MODE (default: Tags.NOISE_MODE_ADDITIVE)
Tags.DATA_FIELD (required)
```

run (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

class `simpa.core.processing_components.noise.salt_and_pepper_noise.`**`SaltAndPepperNoiseProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Applies salt and pepper noise to the defined data field. The noise will be applied to all wavelengths.
The noise will be 50% salt and 50% pepper noise, but both can be set to the same value using the NOISE_MIN and NOISE_MAX fields.
Component Settings:

```
Tags.NOISE_MIN (default: min(data_field))
Tags.NOISE_MAX (default: max(data_field))
Tags.NOISE_FREQUENCY (default: 0.01)
Tags.DATA_FIELD (required)
```

run (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

class `simpa.core.processing_components.noise.uniform_noise.`**`UniformNoiseProcessingComponent`** (`global_settings`, `component_settings_key: str`)

Applies uniform noise to the defined data field. The noise will be applied to all wavelengths.
The noise will be uniformly distributed between [min, max[.
Component Settings:

```
Tags.NOISE_MIN (default: 0)
Tags.NOISE_MAX (default: 1)
Tags.NOISE_MODE (default: Tags.NOISE_MODE_ADDITIVE)
Tags.DATA_FIELD (required)
```

run (`device`)

Executes the respective simulation module

Parameters: **digital_device_twin** – The digital twin that can be used by the digital device_twin.

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. Additionally, we have included detection geometries and illumination geometries that can be used to create custom photoacoustic devices for simulation.

Detection Geometries

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.device_digital_twins.devices.detection_geometries.detection_geometry_base.`**`DetectionGeometryBase`** (`number_detector_elements`, `detector_element_width_mm`, `detector_element_length_mm`, `center_frequency_hz`, `bandwidth_percent`, `sampling_frequency_mhz`, `probe_width_mm`, `device_position_mm: numpy.ndarray = None`)

This class represents an illumination geometry

abstract **`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` **()** → 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_accounting_for_field_of_view` **()** → 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

abstract **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.

class simpa.core.device_digital_twins.devices.detection_geometries.curved_array.**CurvedArrayDetectionGeometry** (pitch_mm=0.5, radius_mm=40, number_detector_elements=256, detector_element_width_mm=0.24, detector_element_length_mm=13, center_frequency_hz=3960000.0, bandwidth_percent=55, sampling_frequency_mhz=40, angular_origin_offset=3.141592653589793, device_position_mm=None)

This class represents a digital twin of a ultrasound detection device with a curved detection geometry

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_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_field_of_view_extent_mm () → numpy.ndarray

Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

class simpa.core.device_digital_twins.devices.detection_geometries.linear_array.**LinearArrayDetectionGeometry** (pitch_mm=0.5, number_detector_elements=100, detector_element_width_mm=0.24, detector_element_length_mm=0.5, center_frequency_hz=3960000.0, bandwidth_percent=55, sampling_frequency_mhz=40, device_position_mm: numpy.ndarray = None)

This class represents a digital twin of a ultrasound detection device with a linear detection geometry.

`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_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_field_of_view_extent_mm` () → numpy.ndarray

Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

class `simpa.core.device_digital_twins.devices.detection_geometries.planar_array.`**`PlanarArrayDetectionGeometry`** (`pitch_mm=0.5`, `number_detector_elements_x=100`, `number_detector_elements_y=100`, `detector_element_width_mm=0.24`, `detector_element_length_mm=0.5`, `center_frequency_hz=3960000.0`, `bandwidth_percent=55`, `sampling_frequency_mhz=40`, `device_position_mm: numpy.ndarray = None`)

This class represents a digital twin of a ultrasound detection device with a linear detection geometry.

`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_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_field_of_view_extent_mm` **()** → numpy.ndarray
Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

Illumination Geometries

class `simpa.core.device_digital_twins.devices.illumination_geometries.illumination_geometry_base.`**`IlluminationGeometryBase`**
This class represents an illumination geometry.

`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.

abstract **`get_mcx_illuminator_definition`** **(**`global_settings: simpa.utils.settings.Settings`, `probe_position_mm: numpy.ndarray`**)** → dict
IMPORTANT: This method creates a dictionary that contains tags as they are expected for the mcx simulation tool to represent the illumination geometry of this device.

Parameters:

- **global_settings** – The global_settings instance containing the simulation instructions
- **probe_position_mm** – the position of the probe in the volume

Returns: Dictionary that includes all parameters needed for mcx.

class `simpa.core.device_digital_twins.devices.illumination_geometries.pencil_array_illumination.`**`PencilArrayIlluminationGeometry`** `(pitch_mm=0.5, number_illuminators_x=100, number_illuminators_y=100)`
This class represents a slit illumination geometry. The device position is defined as the middle of the slit.

`get_field_of_view_extent_mm` **()** → numpy.ndarray
Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

`get_mcx_illuminator_definition` **(**`global_settings: simpa.utils.settings.Settings`, `probe_position_mm`**)** → dict
IMPORTANT: This method creates a dictionary that contains tags as they are expected for the mcx simulation tool to represent the illumination geometry of this device.

Parameters:

- **global_settings** – The global_settings instance containing the simulation instructions
- **probe_position_mm** – the position of the probe in the volume

Returns: Dictionary that includes all parameters needed for mcx.

class `simpa.core.device_digital_twins.devices.illumination_geometries.pencil_beam_illumination.`**`PencilBeamIlluminationGeometry`**

This class represents a pencil beam illumination geometry. The device position is defined as the middle of the slit.

`get_field_of_view_extent_mm` **()** → numpy.ndarray

Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

`get_mcx_illuminator_definition` **(**`global_settings: simpa.utils.settings.Settings`**,** `probe_position_mm`**)** → dict

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

Parameters:

- **global_settings** – The global_settings instance containing the simulation instructions
- **probe_position_mm** – the position of the probe in the volume

Returns: Dictionary that includes all parameters needed for mcx.

class `simpa.core.device_digital_twins.devices.illumination_geometries.slit_illumination.`**`SlitIlluminationGeometry`** (`slit_vector_mm: list = None`, `direction_vector_mm: list = None`)

This class represents a slit illumination geometry. The device position is defined as the middle of the slit.

`get_field_of_view_extent_mm` **()** → numpy.ndarray

Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

`get_mcx_illuminator_definition` **(**`global_settings: simpa.utils.settings.Settings`**,** `probe_position_mm`**)** → dict

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

Parameters:

- **global_settings** – The global_settings instance containing the simulation instructions
- **probe_position_mm** – the position of the probe in the volume

Returns: Dictionary that includes all parameters needed for mcx.

Models of real world devices

class `simpa.core.device_digital_twins.devices.pa_devices.ithera_msot_acuity.`**`MSOTAcuityEcho`** (`device_position_mm: numpy.ndarray = None`)

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.
```

`update_settings_for_use_of_model_based_volume_creator` **(**`global_settings: simpa.utils.settings.Settings`**)**
This method can be overwritten by a photoacoustic device if the device poses special constraints to the volume that should be considered by the model-based volume creator.

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.device_digital_twins.devices.pa_devices.ithera_msot_invision.`**`InVision256TF`**`(device_position_mm: numpy.ndarray = None)`
This class represents a digital twin of the InVision 256-TF, manufactured by iThera Medical, Munich, Germany (https://www.ithera-medical.com/products/msot-invision/). 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 InVision series devices are:

```
Joseph, James, et al. "Evaluation of precision in optoacoustic tomography
for preclinical imaging in living subjects."
Journal of Nuclear Medicine 58.5 (2017): 807-814.

Mer■ep, Elena, et al. "Whole-body live mouse imaging by hybrid
reflection-mode ultrasound and optoacoustic tomography."
Optics letters 40.20 (2015): 4643-4646.
```

`get_field_of_view_extent_mm` **()** → numpy.ndarray
Returns the field of view extent of this imaging device. It is defined as a numpy array of the shape [xs, xe, ys, ye, zs, ze], where x, y, and z denote the coordinate axes and s and e denote the start and end positions. These coordinates are defined relatively to the probe position.

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.core.device_digital_twins.devices.pa_devices.ithera_rsom.`**`RSOMExplorerP50`**`(element_spacing_mm=0.02, number_elements_x=10, number_elements_y=10, device_position_mm: numpy.ndarray = None)`
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.
```

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.

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

`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.`**`min_max_normalization`**`(data: numpy.ndarray = None)` → numpy.ndarray

Normalizes the given data by applying min max normalization. The resulting array has values between 0 and 1 inclusive.

Parameters: **data** – (numpy array) data to be normalized

Returns: (numpy array) normalized array

`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_matrix_between_vectors`**`(a, b)`

Returns the rotation matrix from a to b

Parameters:
- **a** – 3D vector to rotate
- **b** – 3D target vector

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

`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:

- **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

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

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_forward_module

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_module

ACOUSTIC_MODEL_K_WAVE = *'kwave'*
Corresponds to the kwave simulaiton.
Usage: module acoustic_forward_module, 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_forward_module, 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_module, 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 core.processing_components

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

DETECTOR_ELEMENT_WIDTH_MM = *'detector_element_width_mm'*
Width of a detector element. Corresponds to the pitch - the distance between two detector element borders.
Usage: module acoustic_forward_module, naming convention

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

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

DIGITAL_DEVICE_MSOT_INVISION = *'digital_device_invision'*
Corresponds to the InVision 256-TF 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

DIGITAL_DEVICE_SLIT_ILLUMINATION_LINEAR_DETECTOR = *'digital_device_slit_illumination_linear_detector'*
Corresponds to a PA device with a slit as illumination and a linear array as detection geometry.

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

DOWNSCALE_FACTOR = *('downscale_factor', (<class 'int'>, <class 'float'>, <class 'numpy.int32'>, <class 'numpy.float64'>))*
Downscale factor of the resampling in the qPAI reconstruction
Usage: module algorithms (iterative_qPAI_algorithm.py)

ELLIPTICAL_TUBULAR_STRUCTURE = *'EllipticalTubularStructure'*
Corresponds to the EllipticalTubularStructure in the structure_library.
Usage: module volume_creation_module, 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_module, 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_IPASC_DEFINITION = *'ipasc'*
Corresponds to a source definition in mcx.
Usage: adapter mcx_adapter, naming convention

ILLUMINATION_TYPE_MSOT_ACUITY_ECHO = *'msot_acuity_echo'*
s 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_MSOT_INVISION = *'invision'*
Corresponds to a source definition in mcx.
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

IMAGE_PROCESSING = *'image_processing'*
Location of the image algorithms outputs in the SIMPA output file.
Usage: naming convention

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

Usage: adapter segmentation_based_volume_creator

ITERATIVE_RECONSTRUCTION_CONSTANT_REGULARIZATION = *('constant_regularization', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, the fluence regularization will be constant.
Usage: module algorithms (iterative_qPAI_algorithm.py)

ITERATIVE_RECONSTRUCTION_MAX_ITERATION_NUMBER = *('maximum_iteration_number', (<class 'int'>, <class 'numpy.integer'>))*
Maximum number of iterations performed in iterative reconstruction if stopping criterion is not reached.
Usage: module algorithms (iterative_qPAI_algorithm.py)

ITERATIVE_RECONSTRUCTION_REGULARIZATION_SIGMA = *('regularization_sigma', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Sigma value used for constant regularization of fluence.
Usage: module algorithms (iterative_qPAI_algorithm.py)

ITERATIVE_RECONSTRUCTION_SAVE_INTERMEDIATE_RESULTS = *('save_intermediate_results', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If True, a list of all intermediate absorption updates (middle slices only) will be saved in a numpy file.
Usage: module algorithms (iterative_qPAI_algorithm.py)

ITERATIVE_RECONSTRUCTION_STOPPING_LEVEL = *('iteration_stopping_level', (<class 'int'>, <class 'numpy.integer'>, <class 'float'>, <class 'float'>))*
Ratio of improvement and preceding error at which iteration method stops. Usage: module algorithms (iterative_qPAI_algorithm.py)

ITERATIVE_qPAI_RESULT = *'iterative_qpai_result'*
Name of the data field in which the iterative qPAI result will be stored.
Usage: naming convention

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_module

LINEAR_UNMIXING_RESULT = *'linear_unmixing_result'*
Name of the data field in which the linear unmixing result will be stored.
Usage: naming convention

LOAD_AND_SAVE_HDF5_FILE_AT_THE_END_OF_SIMULATION_TO_MINIMISE_FILESIZE = *('minimize_file_size', (<class 'bool'>, <class 'bool'>, <class 'numpy.bool_'>))*
If not set to False, the HDF5 file will be optimised after the simulations are done. Usage: simpa.core.simulation.simulate

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

MCX_ASSUMED_ANISOTROPY = *('mcx_seed', (<class 'float'>, <class 'float'>))*

The anisotropy that should be assumed for the mcx simulations. If not set, a default value of 0.9 will be assumed. Usage: module optical_modelling, adapter mcx_adapter

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_module

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

NOISE_FREQUENCY = *'noise_frequency'*
Frequency of the noise model.
Usage: module core.processing_components.noise

NOISE_MAX = *'noise_max'*
Max of a noise model.
Usage: module core.processing_components.noise

NOISE_MEAN = *'noise_mean'*
Mean of a noise model.
Usage: module core.processing_components.noise

NOISE_MIN = *'noise_min'*
Min of a noise model.
Usage: module core.processing_components.noise

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 core.processing_components.noise

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

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

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

NOISE_SCALE = *'noise_scale'*
Scale of a noise model.
Usage: module core.processing_components.noise

NOISE_SHAPE = *'noise_shape'*
Shape of a noise model.
Usage: module core.processing_components.noise

NOISE_STD = *'noise_std'*
Standard deviation of a noise model.
Usage: module core.processing_components.noise

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

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_module

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_JSON_FILE = *('optical_model_illumination_geometry_json_file', <class 'str'>)*
Absolute path of the location of the JSON file containing the IPASC-formatted optical forward model illumination geometry.
Usage: module optical_simulation_module

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_module, 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_module

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_module, 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_module, 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/m^3.
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_INTRINSIC_EULER_ANGLE = *'intrinsic_euler_angle'*
Intrinsic euler angles of the detector elements in the kWaveArray.
Usage: adapter KwaveAcousticForwardModel, adapter TimeReversalAdapter, 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 reconstruction_module

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

RECONSTRUCTION_ALGORITHM_DMAS = *'DMAS'*
Corresponds to the reconstruction algorithm DMAS with the MitkBeamformingAdapter.
Usage: module reconstruction_module, naming convention

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

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

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

RECONSTRUCTION_ALGORITHM_TIME_REVERSAL = *'time_reversal'*
Corresponds to the reconstruction algorithm Time Reversal with TimeReversalAdapter.
Usage: module reconstruction_module, 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_module, 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_ELEMENT_POSITIONS = *'sensor_element_positions'*
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_forward_module, 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_forward_module, naming convention

SENSOR_PITCH_MM = *'sensor_pitch_mm'*

Pitch of detector elements of the used PA device.
Usage: adapter AcousticForwardModelKWaveAdapter, naming convention

SENSOR_RADIUS_MM = *'sensor_radius_mm'*
Radius of a concave geometry of the used PA device.
Usage: adapter AcousticForwardModelKWaveAdapter, 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_module, naming_convention

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

Usage: module volume_creation_module

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_module, 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_module

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_module, naming convention

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

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

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, module device_digital_twins

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

`VOLUME_CREATOR_VERSATILE` = *'volume_creator_versatile'*
Corresponds to the ModelBasedVolumeCreator.
Usage: module volume_creation_module, 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

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.`**`Spectrum`** (`spectrum_name: str`, `wavelengths: numpy.ndarray`, `values: numpy.ndarray`)

An instance of this class represents the absorption spectrum over wavelength for a particular

`get_value_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_value_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` **(**`oxygenation=None`**)**

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=1e-10`**,** `mus=1e-10`**,** `g=1e-10`**)**

TODO

`dermis` **(**`background_oxy=None`**,** `blood_volume_fraction=None`**)**

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

`epidermis` **(**`melanosom_volume_fraction=None`**)**

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

`get_blood_volume_fractions` **(**`total_blood_volume_fraction=1e-10`**,** `oxygenation=1e-10`**)**

TODO

`muscle` **(**`background_oxy=None`**,** `blood_volume_fraction=None`**)**

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.

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

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

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

`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`**

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.io_handling.serialization.`**`SIMPAJSONSerializer`**
TODO

default (`_object: object`**)**
TODO

class `simpa.io_handling.serialization.`**`SIMPASerializer`**
TODO

serialize (`_object: object`**)**

Module: log

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI) SPDX-License-Identifier: MIT

class `simpa.log.file_logger.`**`Logger`**

The SIMPA Logger. The purpose of this class is to guarantee that the logging Config has been set and that logging strings are written to the same file throughout the entire simulation pipeline. Per default, the log file is located in the home directory as defined by Path.home().

The log levels are defined the same way they are in the python logging module: DEBUG: Detailed information, typically of interest only when diagnosing problems. INFO: Confirmation that things are working as expected. WARNING: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected. ERROR: Due to a more serious problem, the software has not been able to perform some function. CRITICAL: A serious error, indicating that the program itself may be unable to continue running.

`critical` **(**`msg`**)**

Logs a critical message to the logging system. CRITICAL: A serious error, indicating that the program itself may be unable to continue running.

Parameters: **msg** – the message to log

`debug` **(**`msg`**)**

Logs a debug message to the logging system. DEBUG: Detailed information, typically of interest only when diagnosing problems.

Parameters: **msg** – the message to log

`error` **(**`msg`**)**

Logs an error message to the logging system. ERROR: Due to a more serious problem, the software has not been able to perform some function.

Parameters: **msg** – the message to log

`info` **(**`msg`**)**

Logs an info message to the logging system. INFO: Confirmation that things are working as expected.

Parameters: **msg** – the message to log

`warning` **(**`msg`**)**

Logs a warning message to the logging system. WARNING: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.

Parameters: **msg** – the message to log

Examples

Performing an optical forward simulation

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

```python
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
SPDX-License-Identifier: MIT
"""

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 import *
import numpy as np
from simpa.core import VolumeCreationModelModelBasedAdapter, OpticalForwardModelMcxAdapter,
    GaussianNoiseProcessingComponent

from simpa.utils.path_manager import PathManager

# FIXME temporary workaround for newest Intel architectures
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

# 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()

VOLUME_TRANSDUCER_DIM_IN_MM = 60
VOLUME_PLANAR_DIM_IN_MM = 30
VOLUME_HEIGHT_IN_MM = 60
SPACING = 0.5
RANDOM_SEED = 471
VOLUME_NAME = "MyVolumeName_"+str(RANDOM_SEED)

# 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.constant(1e-4, 1e-4, 0
    background_dictionary[Tags.STRUCTURE_TYPE] = Tags.BACKGROUND

    muscle_dictionary = Settings()
    muscle_dictionary[Tags.PRIORITY] = 1
    muscle_dictionary[Tags.STRUCTURE_START_MM] = [0, 0, 10]
    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,
                                                    10,
                                                    VOLUME_HEIGHT_IN_MM/2]
    vessel_1_dictionary[Tags.STRUCTURE_END_MM] = [VOLUME_TRANSDUCER_DIM_IN_MM/2,
                                                  12,
                                                  VOLUME_HEIGHT_IN_MM/2]
    vessel_1_dictionary[Tags.STRUCTURE_RADIUS_MM] = 3
    vessel_1_dictionary[Tags.MOLECULE_COMPOSITION] = TISSUE_LIBRARY.blood()
    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, 9]
    epidermis_dictionary[Tags.STRUCTURE_END_MM] = [0, 0, 10]
    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

# 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)

general_settings = {
    # These parameters set the general propeties of the simulated volume
    Tags.RANDOM_SEED: RANDOM_SEED,
    Tags.VOLUME_NAME: VOLUME_NAME,
    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.WAVELENGTHS: [798],
    Tags.DIGITAL_DEVICE_POSITION: [VOLUME_TRANSDUCER_DIM_IN_MM/2,
                                   VOLUME_PLANAR_DIM_IN_MM/2,
                                   0],
    Tags.LOAD_AND_SAVE_HDF5_FILE_AT_THE_END_OF_SIMULATION_TO_MINIMISE_FILESIZE: True
}

settings = Settings(general_settings)

settings.set_volume_creation_settings({
    Tags.SIMULATE_DEFORMED_LAYERS: True,
    Tags.STRUCTURES: create_example_tissue()
})
settings.set_optical_settings({
    Tags.OPTICAL_MODEL_NUMBER_PHOTONS: 1e7,
    Tags.OPTICAL_MODEL_BINARY_PATH: path_manager.get_mcx_binary_path(),
    Tags.OPTICAL_MODEL: Tags.OPTICAL_MODEL_MCX,
    Tags.ILLUMINATION_TYPE: Tags.ILLUMINATION_TYPE_PENCIL,
    Tags.LASER_PULSE_ENERGY_IN_MILLIJOULE: 50
})
settings["noise_model_1"] = {
    Tags.NOISE_MEAN: 1.0,
    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
}

pipeline = [
    VolumeCreationModelModelBasedAdapter(settings),
    OpticalForwardModelMcxAdapter(settings),
    GaussianNoiseProcessingComponent(settings, "noise_model_1")
]

class ExampleDeviceSlitIlluminationLinearDetector(PhotoacousticDevice):
    """
    This class represents a digital twin of a PA device with a slit as illumination next to

    """

    def __init__(self):
        super().__init__(device_position_mm=np.asarray([VOLUME_TRANSDUCER_DIM_IN_MM/2,
                                                         VOLUME_PLANAR_DIM_IN_MM/2, 0]))
        self.set_detection_geometry(LinearArrayDetectionGeometry())
        self.add_illumination_geometry(SlitIlluminationGeometry(slit_vector_mm=[20, 0, 0],
                                                                direction_vector_mm=[0, 0, 5

device = ExampleDeviceSlitIlluminationLinearDetector()

simulate(pipeline, settings, device)

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
                   log_scale=True)
```

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

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

```
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
SPDX-License-Identifier: MIT
"""

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
import numpy as np
from simpa.utils.path_manager import PathManager
from simpa.core import ImageReconstructionModuleDelayAndSumAdapter, GaussianNoiseProcessingC
    OpticalForwardModelMcxAdapter, AcousticForwardModelKWaveAdapter, VolumeCreationModelMode
    FieldOfViewCroppingProcessingComponent
from simpa.core.device_digital_twins import MSOTAcuityEcho


# 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.constant(1e-10, 1e-10,
    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()
    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, 0.5]
    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


# 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,
            Tags.GPU: True,

            # 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: False,
    Tags.ACOUSTIC_MODEL_BINARY_PATH: path_manager.get_matlab_binary_path(),
    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.ACOUSTIC_MODEL_BINARY_PATH: path_manager.get_matlab_binary_path(),
    Tags.ACOUSTIC_SIMULATION_3D: True,
    Tags.PROPERTY_ALPHA_POWER: 1.05,
    Tags.TUKEY_WINDOW_ALPHA: 0.5,
    Tags.BANDPASS_CUTOFF_LOWPASS: int(8e6),
    Tags.BANDPASS_CUTOFF_HIGHPASS: int(0.1e6),
    Tags.RECONSTRUCTION_BMODE_AFTER_RECONSTRUCTION: True,
    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,
    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["noise_initial_pressure"] = {
    Tags.NOISE_MEAN: 1,
    Tags.NOISE_STD: 0.01,
    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: 1,
    Tags.NOISE_MODE: Tags.NOISE_MODE_ADDITIVE,
    Tags.DATA_FIELD: Tags.TIME_SERIES_DATA
}

device = MSOTAcuityEcho(device_position_mm=np.array([VOLUME_TRANSDUCER_DIM_IN_MM/2,
                                                     VOLUME_PLANAR_DIM_IN_MM/2,
                                                     0]))

device.update_settings_for_use_of_model_based_volume_creator(settings)

SIMUATION_PIPELINE = [
    VolumeCreationModelModelBasedAdapter(settings),
    OpticalForwardModelMcxAdapter(settings),
    GaussianNoiseProcessingComponent(settings, "noise_initial_pressure"),
    AcousticForwardModelKWaveAdapter(settings),
    GaussianNoiseProcessingComponent(settings, "noise_time_series"),
    FieldOfViewCroppingProcessingComponent(settings),
    ImageReconstructionModuleDelayAndSumAdapter(settings)
]

simulate(SIMUATION_PIPELINE, settings, device)

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=True,
                   show_initial_pressure=True,
                   show_absorption=False,
                   show_segmentation_map=False,
                   show_tissue_density=False,
                   show_reconstructed_data=True,
                   show_fluence=False,
                   log_scale=False)
```

Defining custom tissue structures and properties

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

```
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
SPDX-License-Identifier: MIT
"""

from simpa.utils import MolecularCompositionGenerator
from simpa.utils import MOLECULE_LIBRARY
from simpa.utils import Molecule
from simpa.utils import Spectrum
from simpa.utils.libraries.spectra_library import ScatteringSpectrumLibrary, AnisotropySpect
import numpy as np


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


def create_custom_chromophore(volume_fraction: float = 1.0):
    chromophore = Molecule(
            absorption_spectrum=create_custom_absorber(),
            volume_fraction=volume_fraction,
            scattering_spectrum=ScatteringSpectrumLibrary.CONSTANT_SCATTERING_ARBITRARY(40.0
            anisotropy_spectrum=AnisotropySpectrumLibrary.CONSTANT_ANISOTROPY_ARBITRARY(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
```

Defining a custom digital device twin class

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

```
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
SPDX-License-Identifier: MIT
"""

# FIXME temporary workaround for newest Intel architectures
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
from simpa.core.device_digital_twins.digital_device_twin_base import PhotoacousticDevice
from simpa.core.device_digital_twins.devices.illumination_geometries.slit_illumination impor
from simpa.core.device_digital_twins.devices.detection_geometries.linear_array import Linear

from simpa.utils import Settings, Tags
import numpy as np


class ExampleDeviceSlitIlluminationLinearDetector(PhotoacousticDevice):
    """
    This class represents a digital twin of a PA device with a slit as illumination next to

    """

    def __init__(self):
        super().__init__()
        self.set_detection_geometry(LinearArrayDetectionGeometry())
        self.add_illumination_geometry(SlitIlluminationGeometry())


if __name__ == "__main__":
    device = ExampleDeviceSlitIlluminationLinearDetector()
    settings = Settings()
    settings[Tags.DIM_VOLUME_X_MM] = 20
    settings[Tags.DIM_VOLUME_Y_MM] = 50
    settings[Tags.DIM_VOLUME_Z_MM] = 20
    settings[Tags.SPACING_MM] = 0.5
    settings[Tags.STRUCTURES] = {}

    x_dim = int(round(settings[Tags.DIM_VOLUME_X_MM]/settings[Tags.SPACING_MM]))
    z_dim = int(round(settings[Tags.DIM_VOLUME_Z_MM]/settings[Tags.SPACING_MM]))

    positions = device.get_detection_geometry().get_detector_element_positions_accounting_fo
    detector_elements = device.get_detection_geometry().get_detector_element_orientations(gl
    positions = np.round(positions/settings[Tags.SPACING_MM]).astype(int)
    import matplotlib.pyplot as plt
    plt.scatter(positions[:, 0], positions[:, 2])
    plt.quiver(positions[:, 0], positions[:, 2], detector_elements[:, 0], detector_elements[
    plt.show()
```

Defining custom tissue types

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

```
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
SPDX-License-Identifier: MIT
```

```
"""

from simpa.utils import MolecularCompositionGenerator
from simpa.utils import MOLECULE_LIBRARY
from simpa.utils import Molecule
from simpa.utils import Spectrum
from simpa.utils.libraries.spectra_library import ScatteringSpectrumLibrary, AnisotropySpect
import numpy as np


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


def create_custom_chromophore(volume_fraction: float = 1.0):
    chromophore = Molecule(
            absorption_spectrum=create_custom_absorber(),
            volume_fraction=volume_fraction,
            scattering_spectrum=ScatteringSpectrumLibrary.CONSTANT_SCATTERING_ARBITRARY(40.0
            anisotropy_spectrum=AnisotropySpectrumLibrary.CONSTANT_ANISOTROPY_ARBITRARY(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
```

Load a segmentation mask and use it to simulate

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

```
"""
SPDX-FileCopyrightText: 2021 Computer Assisted Medical Interventions Group, DKFZ
SPDX-FileCopyrightText: 2021 VISION Lab, Cancer Research UK Cambridge Institute (CRUK CI)
```

```
SPDX-License-Identifier: MIT
"""

from simpa.core.simulation import simulate
from simpa.utils.settings import Settings
from simpa.utils import Tags, SegmentationClasses
import numpy as np
from skimage.data import shepp_logan_phantom
from simpa.utils.libraries.tissue_library import TISSUE_LIBRARY
from simpa.utils.libraries.molecule_library import MOLECULE_LIBRARY
from simpa.utils.libraries.tissue_library import MolecularCompositionGenerator
from simpa.visualisation.matplotlib_data_visualisation import visualise_data
from scipy.ndimage import zoom
from simpa.utils.path_manager import PathManager
from simpa.core.device_digital_twins import RSOMExplorerP50
from simpa.core import VolumeCreationModuleSegmentationBasedAdapter, OpticalForwardModelMcxA

# FIXME temporary workaround for newest Intel architectures
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

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

# 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()

target_spacing = 1.0

label_mask = shepp_logan_phantom()

label_mask = np.digitize(label_mask, bins=np.linspace(0.0, 1.0, 11), right=True)

label_mask = np.reshape(label_mask, (400, 1, 400))

input_spacing = 0.2
segmentation_volume_tiled = np.tile(label_mask, (1, 128, 1))
segmentation_volume_mask = np.round(zoom(segmentation_volume_tiled, input_spacing/target_spa
                                         order=0)).astype(int)

def segmention_class_mapping():
    ret_dict = dict()
    ret_dict[0] = TISSUE_LIBRARY.heavy_water()
    ret_dict[1] = TISSUE_LIBRARY.blood()
    ret_dict[2] = TISSUE_LIBRARY.epidermis()
    ret_dict[3] = TISSUE_LIBRARY.muscle()
    ret_dict[4] = TISSUE_LIBRARY.mediprene()
    ret_dict[5] = TISSUE_LIBRARY.ultrasound_gel()
    ret_dict[6] = TISSUE_LIBRARY.heavy_water()
    ret_dict[7] = (MolecularCompositionGenerator()
                   .append(MOLECULE_LIBRARY.oxyhemoglobin(0.01))
                   .append(MOLECULE_LIBRARY.deoxyhemoglobin(0.01))
                   .append(MOLECULE_LIBRARY.water(0.98))
                   .get_molecular_composition(SegmentationClasses.COUPLING_ARTIFACT))
    ret_dict[8] = TISSUE_LIBRARY.heavy_water()
    ret_dict[9] = TISSUE_LIBRARY.heavy_water()
    ret_dict[10] = TISSUE_LIBRARY.heavy_water()
    return ret_dict
```

```
settings = Settings()
settings[Tags.SIMULATION_PATH] = path_manager.get_hdf5_file_save_path()
settings[Tags.VOLUME_NAME] = "SegmentationTest"
settings[Tags.RANDOM_SEED] = 1234
settings[Tags.WAVELENGTHS] = [700]
settings[Tags.SPACING_MM] = target_spacing
settings[Tags.DIM_VOLUME_X_MM] = 400 / (target_spacing / input_spacing)
settings[Tags.DIM_VOLUME_Y_MM] = 128 / (target_spacing / input_spacing)
settings[Tags.DIM_VOLUME_Z_MM] = 400 / (target_spacing / input_spacing)

settings.set_volume_creation_settings({
    Tags.INPUT_SEGMENTATION_VOLUME: segmentation_volume_mask,
    Tags.SEGMENTATION_CLASS_MAPPING: segmention_class_mapping(),

})

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,
})

pipeline = [
    VolumeCreationModuleSegmentationBasedAdapter(settings),
    OpticalForwardModelMcxAdapter(settings)
]

simulate(pipeline, settings, RSOMExplorerP50(element_spacing_mm=1.0))

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

if VISUALIZE:
    visualise_data(path_manager.get_hdf5_file_save_path() + "/" + "SegmentationTest" + ".hdf
```

Index

Python Module Index