MPI-CUDA-SART-solver − a software for tomographic reconstruction of 3D emission profiles in fusion devices

Table of Contents

Tomographic reconstruction of plasma emissivity is a computationally challenging problem. Ray transfer matrices (RTMs), pre-computed using ray tracing, are dense matrices of extremely large size, since they take into account the light reflected by the walls of fusion devices. Operations with RTM require tens or even hundreds of GB of RAM. MPI-CUDA-SART-solver is a specialized tomographic reconstruction software optimized for GPU-accelerated high-performance clusters (HPCs). The software uses MPI and Nvidia CUDA technologies. The software implements the well-known Simultaneous Algebraic Reconstruction Technique (SART) tomographic algorithm, with or without regularization, as well as a variant of the SART algorithm with logarithmic residual minimization. Unlike most SART implementations, this software is optimized for dense (not sparse) matrices. Input and output of data is in HDF5 file format, and parameters are set via the command line interface (CLI). The software is written in C++ and depends on the MPI, CUDA, cuBLAS, HDF5 API, and the argparse library.

The problem statement and SART algorithm

All optical measuring instruments (e.g. filtered cameras, linear line of sight arrays connected to spectrometers or photomultipliers) are considered to be absolutely calibrated and tuned to the same wavelength. The input data is the radiation intensity in absolute units W/(m^2 sr), or photon/(s m^2 sr) at each of the detectors. The data from the individual detectors are combined into an array called a *measurement vector*. Detector is defined here as a measuring device responsible for an individual value in the measurement vector, this can be either a single pixel on a CCD matrix or a group of such pixels measuring radiation coming from a certain observation cone.

The inverse problem of reconstructing the volumetric distribution of emissivity for a given wavelength of radiation on a fixed spatial grid of light sources, called *voxels*, is formulated as follows:

$$\|\mathbf{Hf}-\mathbf{g}\|+\|\mathbf{Lf}\| \xrightarrow[\mathbf{f}]{} \min, \tag{1}$$

where **H** is the RTM, each element of which contains the average path traversed by rays through a given voxel in any direction and hitting a given detector, **g** is the measurement vector, **L** is the regularization matrix, **f** is the *solution vector*, containing the sought-after emissivity profile.

The SART algorithm solves the problem (1) iteratively. At the step k+1 the solution is expressed in terms of the solution at step k in standard and logarithmic variants of SART as Eq. (2) and (3), respectively:

$$f_i^{(k+1)} = f_i^{(k)} + \frac{\alpha}{\sum_{j'} H_{ij'}} \sum_j H_{ij} \frac{\left(g_j - \sum_{i'} H_{i'j} f_{i'}^{(k)}\right)}{\sum_{i'} H_{i'j}} - \beta \sum_{i'} L_{ii'} f_{i'}^{(k)}, \tag{2}$$

$$f_i^{(k+1)} = f_i^{(k)} \left(\sum_j \frac{H_{ij} g_j}{\sum_{i'} H_{i'j}} \Bigg/ \sum_j \frac{H_{ij} \sum_{i'} H_{i'j} f_{i'}^{(k)}}{\sum_{i'} H_{i'j}} \right)^{\alpha} e^{-\beta \sum_{i'} L_{ii'} \ln\left(f_{i'}^{(k)}\right)}, \tag{3}$$

where k is the iteration step, i is the voxel index, j is the detector index, α is the relaxation parameter, β is the regularization weight. Parameters α and β are controlled by the user. Eq. (4) is used as initial guess for f_i^0 thus accelerating convergence compared to $f_i^0 = const$. When processing a time series of measurements, the user can choose whether to start the iteration with (4), or with the solution found in the previous time moment.

$$f_i^{(0)} = \sum_j \frac{H_{ij} g_j}{\sum_{i'} H_{i'j}} \tag{4}$$

The algorithm stops when the condition (5) is met or when the maximum number of iterations is reached:

$$\frac{\left| \sum_j \left(\sum_{i'} H_{i'j} f_{i'}^{(k+1)} \right)^2 - \sum_j \left(\sum_{i'} H_{i'j} f_{i'}^{(k)} \right)^2 \right|}{\sum_j \left(g_j \right)^2} < C_{tol}, \tag{5}$$

where C_{tol} is the user controlled relative convergence limit.

Some voxels may not be directly observed by any of the detectors, and their light reaches the detectors only through the reflections. For such voxels, the value of $\sum_{j'} H_{ij'}$ is much smaller than the respective values for directly observed voxels. Similarly, there may be detectors that do not directly observe any of the voxels, and for them the value of $\sum_{i'} H_{i'j}$ is much smaller than the respective values for the detectors directly observing the voxels. In most cases such voxels and detectors must be excluded from consideration, since they carry little useful information, but may increase the errors, because the above values stand in (2) and (3) in the denominator. In addition, the measurement vector may contain data from saturated detectors, for which the exact intensity value is unknown. To exclude saturated detectors from consideration, their values g_j can be masked with negative numbers. The software processes only those detectors and voxels for which the following conditions are met:

$$\sum_{i'} H_{i'j} > h_l, \quad \sum_{j'} H_{ij'} > h_d, \quad g_j \geq 0, \tag{6}$$

where h_d is the minimum allowable average distance passed through the voxel by rays that hit at least one of the detectors and h_l the minimum allowable average distance passed through at least one voxel by rays that hit the detector. The values of h_d and h_l are controlled by the user.

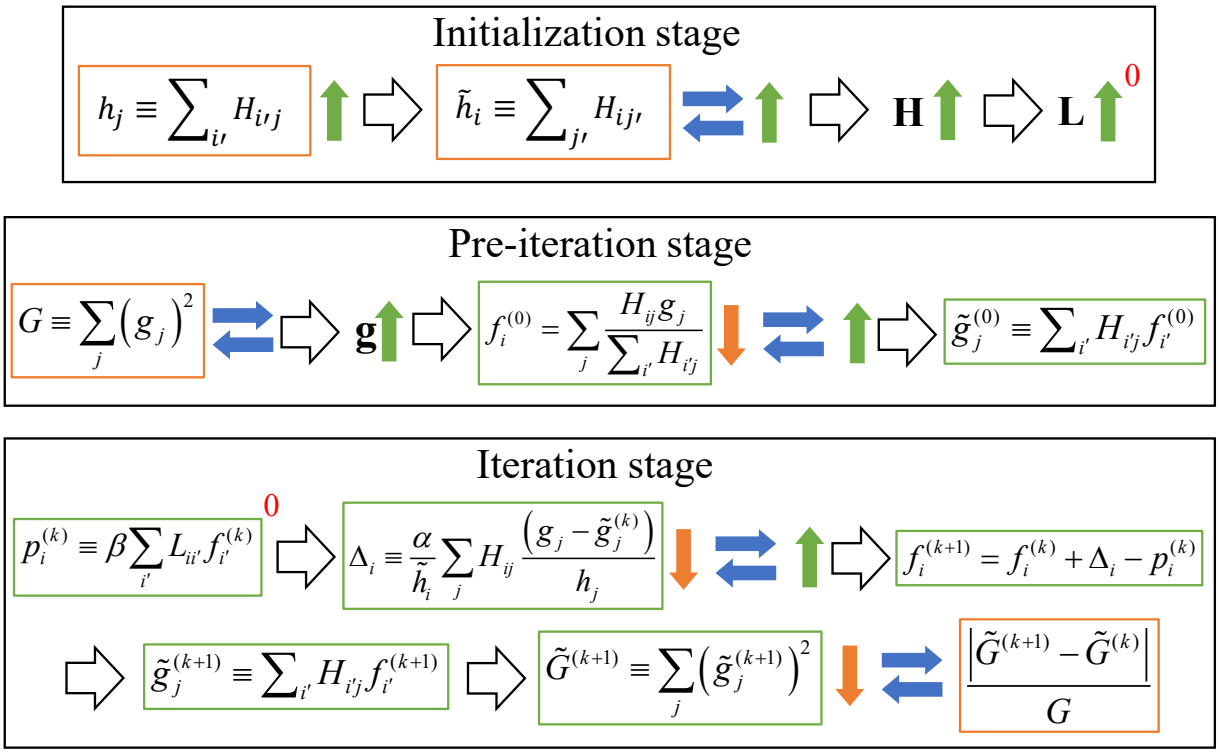
Parallel implementation of the SART algorithm with distributed storage of the ray transfer matrix in RAM

In most tomographic tasks, **H** is a sparse matrix, that is, it contains mostly zeros. This happens when radiation is either not reflected from surfaces or reflected specularly. Storing sparse matrices in the form (index, value) decreases the amount of used RAM. However, the walls of fusion devices reflect the radiation diffusely, and the matrix **H** is dense. For example, in the problem of reconstructing the emissivity in beryllium spectral lines from synthetic images of filtered cameras, the size of the matrix **H** was 85.6 GB, while the size of the corresponding sparse matrix calculated without taking into account reflections was only a few MB. Matrices of this size do not fit in the memory of modern GPUs. In some cases, with more detailed spatial grid, the matrix **H** will not fit even in the RAM of individual nodes of the computing cluster. Using the MPI and CUDA technology, the software implements a uniformly distributed storage of the matrix in the RAM of the computing nodes and the GPUs.

Each parallel MPI process loads only a part of the matrix **H** for only a part of the detectors, but for all voxels, and only a respective segment of the measurement vector **g**. Then, unless the user specifies that the calculations should be performed on the CPU, these partial data are loaded into the GPU's memory. Each MPI process works with a dedicated GPU. Thus, if the user has reserved N cluster computing nodes, each of which has K GPUs with M GB of memory, the

maximum matrix size that the software can handle will be $N \cdot K\ M$ GB. However, this approach requires synchronization and data exchange between parallel MPI processes at each iteration. Figure 1 shows the calculation scheme for the standard SART algorithm (Eq. (2)) with GPU acceleration. The scheme for the logarithmic variant of the algorithm looks similar. Operations at the *initialization* stage are performed only once during the program startup. Operations at the *pre-iteration* stage are performed at the beginning of processing the next measurement vector in the series, operations at the *iteration* stage are performed at each iteration step.

Here is an explanation of the *initialization* stage in the scheme in Figure 1: CPU calculates $\sum_{i'} H_{i'j}$ and loads the result into GPU memory, then CPU calculates $\sum_{j'} H_{ij'}$, but since each MPI process contains only a segment of matrix **H**, synchronization of MPI processes is required to sum up their results and send the total result back to all processes. The result is then loaded into GPU memory, followed by loading matrix **H.** Then the root MPI process (with index 0) loads matrix **L** into GPU memory. Although MPI processes must be synchronized twice at each iteration, when the size of matrix **H** is tens of GB, much less time is required for synchronization compared to the computation time, so synchronization has no significant impact on performance.



Legend: Orange frame – calculating on CPU, green frame – calculating on GPU, green up arrow – transferring to GPU memory, orange down arrow – transferring to CPU, horizontal blue arrows – summing up the results of all MPI-processes and broadcasting, red index in the upper right corner – calculations performed by the MPI-process with a specified index

Figure 1. Calculation scheme for the standard SART algorithm (Eq. (2)) with GPU acceleration on initialization, pre-iteration and iteration stages.

As mentioned above, the software processes a series of measurements and the user can specify the time intervals of interest by setting the start, T_1, and end, T_2, moments of each interval, the time step, Δt, and the synchronicity limit, $\Delta\tau$. The last two parameters are optional. The software takes into account that the measuring devices whose data are processed jointly do not necessarily make measurements synchronously. Let's denote T_k as a total set of time moments when the k-th device makes the measurements. Then only those falling in the interval $t_k \subset T_k$: $T_1 \leq T_k \leq T_2$ are selected. The beginning of the processed time series is the time moment $T_{st} = \min_k \mathbf{t_k}$. If the step Δt is not specified by user, it is defined as $\Delta t = \max_k\{\min(\mathbf{t}_{k+1} - \mathbf{t}_k)\}$ and if no synchronicity limit is specified, it is defined as $\Delta\tau = \Delta t$. Starting from the moment T_{st} with step Δt, a series of measurement vectors is formed from the available sets. If for the moment $t_n = T_{st} + n\Delta t$ at least one measurement falling within the interval $t_n \pm \Delta\tau$ is present in all sets t_k, the measurement vector corresponding to this moment is added to the series, and for each device the measurement made at the moment closest to t_n is selected. If for at least one device there are no measurements falling within $t_n \pm \Delta\tau$, the time moment t_n is skipped.

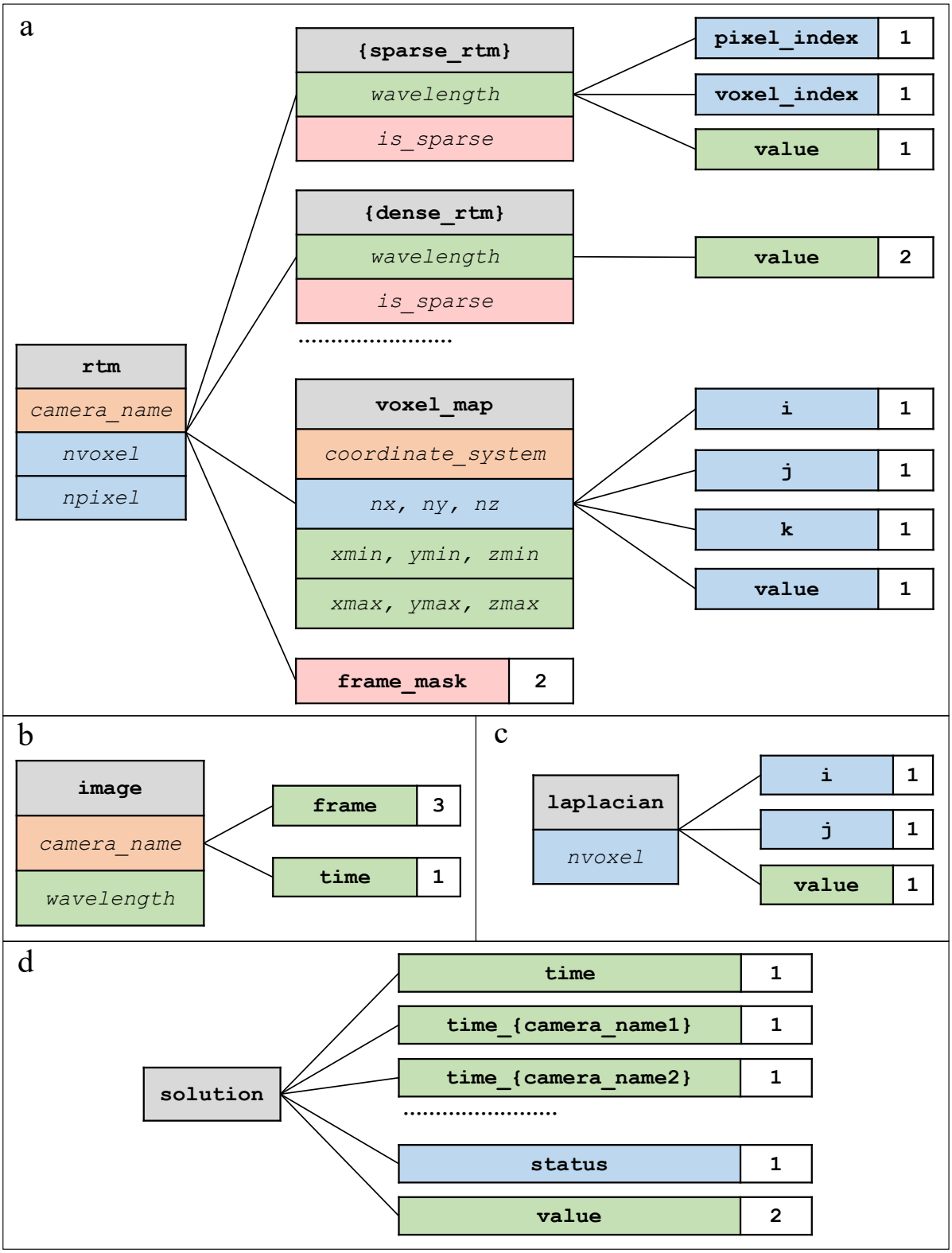
To speed up I/O operations when reading and writing to files, the software caches measurements and solutions from the time series. That is, reading and writing to files is not carried out at every time step, but once in several steps. The maximum number of cached measurements and solutions is set by the user.

Data format in HDF5 files

For input and output data, the software uses the HDF5 file format. There are three types of input files containing: RTMs for measuring devices, measurements, and the regularization matrix **L**. Each measuring device corresponds to one file containing measurements, and one, or several files containing RTMs. In the case of multiple RTM files per device, the files contain RTM segments for non-overlapping voxel groups. The output files contain the solutions found.

Figure 2 shows the hierarchical structure of the input and output files. The root group in the RTM files (Figure 2 (a)) is called **rtm** and contains attributes: *camera_name* - name of the measuring device (camera), *nvoxel* - number of voxels in RTM, *npixel* - number of detectors in RTM. One file can contain several RTM, calculated for the same voxels and the same measuring device, but for different characteristics, like the light reflection properties, or the radiation wavelength. The user specifies the name of the group containing the RTM to be used. The group containing the RTM has two attributes: *wavelength* – radiation wavelength in nm, for which the RTM is calculated, and *is_sparse*, defining if the matrix is saved as sparse, or not. For sparse matrices, the group contains three data arrays: **pixel_index** for detector indices, **voxel_index** for

voxel indices, and **value** for RTM values. For dense matrices, the group contains only a two-dimensional array **value**, of size *npixel* × *nvoxel*.



Legend: *Bold font – the name of the group or the data array, Italic – the name of the attribute; grey background – the group, green – float data, blue – integer data, red – Boolean data, orange – strings; the numbers are the dimensions of data arrays*

Figure 2. Hierarchical data structure of the input files with: (a) RTM, (b) measurements, (c) regularization matrix **L** and the output file with the reconstructed emission profile (d).

In addition to the groups with RTMs, the root **rtm** group contains a **voxel_map** group with parameters of a three-dimensional spatial grid with a constant step along each direction, as well as a voxel map, which maps the voxels for which the RTM is obtained to the cells of this spatial grid. The **voxel_map** group has the following attributes: *coordinate_system* - coordinate system of the spatial grid ("cartesian", "cylindrical" etc.); *nx*, *ny*, *nz* - number of cells in the spatial grid along the 1st, 2nd and 3rd axes respectively; *xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax* - spatial grid boundaries along respective axes in meters. The voxel map is saved as a sparse matrix, so the **voxel_map** group contains four data arrays: indexes of grid cells **i**, **j**, **k** and **value**. The value array contains indexes of the RTM voxels corresponding to the cells of the spatial grid. One voxel may occupy multiple cells of the spatial grid, so the **value** array may contain repeating values. The number of unique values in the value array must be equal to the value of the *nvoxel* attribute of the **rtm** group. Voxel map provides compatibility with RayTransferObject classes from Cherab Spectroscopy Modelling Framework, which can be used to calculate the RTM.

The RTM can contain information only for a part of the detectors of the measuring device, for example, if only a part of the camera image is used for emission profile reconstruction. Therefore, the **rtm** group contains a two-dimensional data array **frame_mask**, which for each detector of this measuring device contains the value 1 (detector is active) or 0 (detector is not active). It is assumed that the detectors of the measuring device are organized in a two-dimensional array. The number of active detectors in the **frame_mask** array must be equal to the value of the *npixel* attribute of the **rtm** group. The order of the detectors in the RTM corresponds to the order of the active detectors in the flattened (in row-major order) **frame_mask** array.

RTM for one measuring device can be distributed among several files. Such partial RTMs contain information for the same detectors (and share the same **frame_mask** array), but for different voxels. Partial RTMs are easier to calculate on the HPC, because splitting the RTM for a single measuring device into segments allows to use multiple HPC nodes. All **voxel_map** attributes in these partial RTM files must be the same, but the voxel maps themselves must not overlap, that is, they must not contain the same combinations of indices (i, j, k).

The root group in the measurement files (Figure 2 (b)), is called **image** and has two attributes: *camera_name* for the name of the measuring device and *wavelength* for the wavelength in nm for which the measurements were taken. The group contains two data arrays: **time** - the time moments for each measurement in seconds and **frame** - a three-dimensional array of measurements, where the 1st dimension corresponds to time, and the 2nd and 3rd correspond to detectors organized in a two-dimensional array. The size of the **frame** array along the 1st axis must match the size of the **time** array, and the sizes along the 2nd and 3rd dimensions must match the sizes of the **frame_mask** array from the RTM file for this measuring device.

The root group in files containing the regularization matrix (Figure 2 (c)) is called **laplacian** (since the regularization is linear, the matrix is the Laplacian operator) and has an attribute *nvoxel* whose value should be equal to the number of voxels in the full RTM. The regularization matrix is stored in a sparse from, and the Laplacian group contains two arrays with voxel indices **i** and **j** and the **value** array.

When launched, the software checks the consistency of the input data, in particular, it checks the correspondence of wavelengths for the RTM and measurements within the user-specified threshold, the equality of total voxel maps for the RTM of different measuring devices, the equality of the size of the regularization matrix to the number of voxels in the RTM, etc.

The root group in the output file with the reconstructed emissivity profile (Figure 2 (d)) is called **solution** and contains the following data arrays: **time** - the time moments for which the emissivity was reconstructed, ***time_{camera_name}*** (the name of the corresponding measuring device is substituted instead of ***{camera_name}***) - the exact time moments when the device measurements were taken, **status** - the status of the solution of the inverse problem (0 - the convergence criterion is met, -1 - the maximum number of iterations is reached), **value** - a two-dimensional array with the reconstructed emissivity, where the 1st dimension corresponds to time, and the second one to voxels. The output file also contains a **voxel_map** group allowing to map the solution into the spatial grid.

Command line interface

Table 1. Command line arguments with description

Argument	Description	Default value
`-o, --output_file`	Filename to save the solution to.	solution.h5
`-t, --time_range`	Coma separated time intervals (in seconds) to process in a form: start:stop:(step):(synch_threshold). The step and the synchronization threshold are optional. Example: “10:20:0.1:0.05, 30:50:0.5, 55:70”.	0:∞
`-w, --wavelength_threshold`	An RTM is considered valid if its wavelength is within this threshold of the image wavelength (in nm).	50
`-d, --ray_density_threshold`	The value of threshold h_d in Eq. (6).	1e-6
`-r,`	The value of threshold h_l in Eq. (6).	1e-6

`--ray_lenght_threshold`		
`-m, --max_iterations`	Maximum number of SART iterations.	2000
`-c, --conv_tolerance`	The value of relative tolerance C_{tol} in (5).	1e-5
`-l, -- laplacian_file`	The path to the file with the regularization matrix. **L**. No regularization is used if not specified.	
`-b, --beta_laplace`	Regularization weight β in Eq. (2) and (3).	0.05
`-R, --relaxation`	Relaxation parameter α in Eq. (2) and (3).	1
`-n, --raytransfer_name`	The name of the group in the HDF5 file containing the RTM.	with_reflections
`-L, --logarithmic`	Use logarithmic SART solver of Eq. (3) instead of Eq. (2).	
`--max_cached_frames`	Maximum number of cached measurement frames.	100
`--max_cached_solutions`	Maximum number of cached solutions (reconstructed emissivity profiles).	100
`--no_guess`	If specified, will not use solution found on previous time moment as initial guess for the next one (always use Eq. (4)).	
`--use_cpu`	If specified, performs calculations on CPUs instead of GPUs.	
`--parallel_read`	If specified, all MPI processes read RTM data in parallel. Use only for high-IOPS storages.	
`input_files`	Trailing list of RTM and measurements hdf5 files. Files can be in any order; the software will sort them automatically.	