STEMsalabim Documentation

Release 4.0.2

Jan Oliver Oelerich

Mar 23, 2018

Getting Started

CHAPTER 1

Installing STEMsalabim

1.1 Downloading the source code

We recommend you download the latest stable release (4.0.2) from the Releases page. If you want the latest features and/or bugfixes, you can also clone the repository using

```
$ git clone https://gitlab.com/STRL/STEMsalabim.git
$ git checkout devel # only if you want the devel code.
```

1.2 Requirements

The following libraries and tools are needed to successfully compile the code:

- A C++11 compiler (such as gcc/g++). The Intel compiler suite is currently _not_ supported!
- CMake > 3.3
- NetCDF
- libConfig >= 1.5
- FFTW3
- An MPI implementation (such as OpenMPI)

The following libraries are *optional* and are needed only to enable additional features:

- libCurl (required for HTTP POST status announcements)

Note: You may find some of the requirements in the repositories of your Linux distribution, at least the compiler, CMake, libCurl and OpenMPI. On Debian or Ubuntu Linux, for example, you can simply run the following command to download and install all the requirements:

```
$ apt-get install build-essential      \
                  cmake                \
                  libconfig++-dev      \
                  libfftw3-dev         \
                  libnetcdf-dev        \
                  libcurl4-openssl-dev \
```

```
                doxygen             \
                libopenmpi-dev      \
                openmpi-bin
```

Tip: As the main work of the STEM image simulations is carried out by the FFTW3 library, you may reach best performance when you compile the library yourself with all available CPU level optimizations enabled.

1.3 Building STEMsalabim

Extract the code archive to some folder on your hard drive, e.g.

```
$ cd /tmp
$ tar xzf stemsalabim-VERSION.tar.gz
```

Then, create a build directory and run CMake to generate the build configuration:

```
$ mkdir /tmp/stemsalabim-build
$ cd /tmp/stemsalabim-build
$ cmake ../stemsalabim-VERSION
```

Please refer to the CMake documentation for instructions how to specify library paths and other environment variables, in case the above commands failed. When your libraries exist at non-standard places in your file system, you can specify the search paths as follows:

```
$ cmake ../stemsalabim-VERSION                                          \
    -DFFTW_ROOT=/my/custom/fftw/                                        \
    -DLIBCONFIG_ROOT=/my/custom/libconfig/                              \
    -DNETCDF_INCLUDE_DIR=/my/custom/netcdf/include                      \
    -DCMAKE_INSTALL_PREFIX=/usr/local                                   \
    -DCMAKE_EXE_LINKER_FLAGS='-Wl,-rpath,/my/custom/lib64:/my/custom/lib' \
    -DCMAKE_CXX_COMPILER=/usr/bin/g++
```

In the above example, some custom library paths are specified and the program's run-time search path is modified. If cmake doesn't detect the correct compiler automatically, you can specify it with `-DCMAKE_CXX_COMPILER=`.

Having generated the necessary build files with CMake, simply compile the program using `make` and move it to the install location with `make install`:

```
$ make -j8          # use 8 cores for compilation
$ make install      # move the binaries and libraries to the INSTALL_PREFIX path
```

You are now ready to execute your first simulation.

1.4 Building with Intel MKL, Intel compiler (and Intel MPI)

It is possible to use the Intel® Parallel Studio for compilation, which includes the Intel® Math Kernel Library (MKL) that *STEMsalabim* can use for discrete fourier transforms instead of FFTW3. If the Intel® MPI Library is also available, it can be used as the MPI implementation in *STEMsalabim*.

Note: We have tested compiling and running *STEMsalabim* only with Parallel Studio 2017 so far.

STEMsalabim's CMake files try to find the necessary libraries themselves, when the folling conditions are true:

1. Either the environment variable `MKLROOT` is set to a valid install location of the MKL, or the CMake variable `MKL_ROOT` (pointing at the same location) is specified.
2. The CMake variable `GCCDIR` points to the install directory of a C++11 compatible GCC compiler. This is important, because the `libstdc++` from a GCC install is required for the Intel compilers to use modern C++ features.

For example, let's say the Intel suite is installed in `/opt/intel` and we have GCC 6.3 installed in `/opt/gcc-6.3`, then CMake could be invoked like this:

```
$ export PATH=$PATH:/opt/intel/... # mpicxx and icpc should be in the path!
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/gcc-6.3/lib64 \
    cmake ../source -DMKL_ROOT=/opt/intel -DCMAKE_CXX_COMPILER=icpc -DGCCDIR=/opt/
↪gcc-6.3
```

Depending on how your environment variables are set, you may be able to skip the `LD_LIBRARY_PATH=..` part. When *STEMsalabim* is executed, you may again need to specify the library path of the `libstdc++`, using

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/gcc-6.3/lib64 mpirun -np ... /path/to/
↪stemsalabim -p ...
```

Note: Some fiddling with paths and environment variables is probably necessary. It may help to know basic CMake syntax and have a look at `/path/to/stemsalabim/cmake/FindMKL.cmake` if CMake is unable to find something.

CHAPTER 2

Executing STEMsalabim

STEMsalabim is executed on the command line and configured via input configuration files in libConfig syntax. To learn about the structure of the configuration files, please read *Parameter files*.

Note: Some of configuration parameters can be changed via command line parameters, which are described in *Command line arguments*.

STEMsalabim supports both threaded (shared memory) and MPI (distributed memory) parallelization. For most efficient resource usage we recommend a hybrid approach, where one MPI task is run per node that spawns a bunch of threads to parallelize the work within the node. (See *Hybrid Parallelization model* for more information on how *STEMsalabim* is parallelized.)

2.1 Thread-only parallelization

You can execute *STEMsalabim* on a single multi-core computer as follows:

```
$ stemsalabim --params=./my_config_file.cfg --num-threads=32
```

This will run the simulation configured in `my_config_file.cfg` on 32 cores, of which 31 are used as workers.

2.2 MPI only parallelization

For pure MPI parallelization without spawning additional threads, *STEMsalabim* must be called via `mpirun` or `mpiexec`, depending on the MPI implementation available on your machine:

```
$ mpirun -n 32 stemsalabim --params=./my_config_file.cfg --num-threads=1 --package-
↪size=10
```

This command will run the simulation in parallel on 32 MPI processors without spawning additional threads.

Note: We chose a *work package size* ten times the number of threads on each MPI processor (which is 1 here). This is so that each thread calculates (on average) ten pixels until results are communicated via the network. This

reduces management overhead but increases the amount of data sent via the network.

2.3 Hybrid parallelization

Hybrid parallelization is the recommended mode to run *STEMsalabim*.

For hybrid parallelization, make sure that on each node only a single MPI process is spawned and that there is no CPU pinning active, i.e., *STEMsalabim* needs to be able to spawn threads on different cores.

For example, if we wanted to run a simulation in parallel on 32 machines using OpenMPI and on each machine use 16 cores, we would run

```
$ mpirun -n 32 --bind-to none --map-by ppr:1:node:pe=16 \
    stemsalabim                                         \
        --params=./my_config_file.cfg                   \
        --num-threads=16                                \
        --package-size=160
```

The options `--bind-to none --map-by ppr:1:node:pe=16` tell OpenMPI not to bind the process to anything and to reserve 16 threads for each instance. Please refer to the manual of your MPI implementation to figure out how exactly to run the software. On HPC clusters it is wise to contact the admin team for optimizing the simulation performance.

2.4 Running the `Si 001` example

In the source code archive you find an `examples/Si_001` folder that contains a simple example that you can execute to get started. The file `Si_001.xyz` describes a 2x2x36 unit cell Si sample. Please see *Crystal file format* for the format description.

In the file `Si_001.cfg` we find the simulation configuration / parameters. The file contains all available parameters, regardless of whether they have their default value. We recommend to always specify a complete set of simulation parameters in the configuration files.

You can now run the simulation:

```
$ /path/to/stemsalabim --params Si_001.cfg --num-threads=8
```

After the simulation finished (about 3 hours on an Intel i7 CPU with 8 cores) you can analyze the results found in `Si_001.nc`. Please see the next page (*Visualization of crystals and results*) for details.

CHAPTER 3

Visualization of crystals and results

Now that our simulation finished successfully, we can continue with visualizing the results.

3.1 Visualize the supercell with Ovito

The *STEMsalabim* output files (somewhat) comply with the AMBER specifications to visualize the specimen structure. However, all the dimensions and variables of AMBER live within the NetCDF group `/AMBER` in the NC file. This means that *STEMsalabim* output files **will not be compatible to visualization programs requiring the pure AMBER specs!**

However, the authors of the excellent cross-platform Ovito software, which is a visualization program for atomic structures (and much more), have added support for the `/AMBER` sub-group, so that *STEMsalabim* NetCDF result files can be visualized seamlessly in Ovito.

What you will see is the atomic structure of the input specimen. In addition to the positional coordinates of each atom, you find the mean square displacement (`msd`), the slice ID (`slice`), the equilibrium coordinates (`lattice_coordinates`), elements (`elements`) and atomic radii (`radii`) as variables. Each frozen lattice configuration is one `frame` in the AMBER specs, so you can see the atoms wiggling around if you use the frame slider of ovito.

3.2 Generate an ADF STEM image

In the `examples/Si_001` folder you will find the two files `make_haadf.m` and `make_haadf.py`. Both extract an HAADF image from the simulated file. Please have a look at the code to get an idea of how to work with the NetCDF result files.

- MATLAB® uses HDF5 for its `.mat` format for a couple of versions now, and is therefore perfectly capable of reading *STEMsalabim* result files. For quick analysis and image generation is a great tool.
- Python with the NetCDF4 module is also a great tool to analyze and visualize *STEMsalabim* result files, especially combined with numerical libraries such as numpy or pandas.

3.3 What now?

You have now completed your first simulation and looked at some of its results. In order to use *STEMsalabim* for your research you should dig deeper into this documentation and read the following documents:

3.3.1 General information

Warning: This documentation is not really complete (yet).

Throughout this documentation we assume that you are familiar with the theoretical background behind the scanning transmission electron microscope (STEM) to some degree. Also, we assume that you have some knowledge about the UNIX/Linux command line and parallelized computation. *STEMsalabim* is currently not intended to be run on a desktop computer. While that is possible and works, the main purpose of the program is to be used in a highly parallelized multi-computer environment.

We took great care of making *STEMsalabim* easy to install. You can find instructions at *Installing STEMsalabim*. However, if you run into technical problems you should seek help from an administrator of your computer cluster first.

Structure of a simulation

There essence of *STEMsalabim* is to model the interaction of a focused electron beam with a bunch of atoms, typically in the form of a crystalline sample. Given the necessary input files, the simulation crunches numbers for some time, after which all of the calculated results can be found in the output file. Please refer to *Executing STEMsalabim* for notes how to start a simulation.

Input files

All information about the specimen are listed in the *Crystal file format*, which is one of the two required input files for *STEMsalabim*. It contains each atom's species (element), coordinates, and mean square displacement as it appears in the Debye-Waller factors.

In addition, you need to supply a *Parameter files* for each simulation, containing information about the microscope, detector, and all required simulation parameters. All these parameters are given in a specific syntax in the *Parameter files* that are always required for starting a STEMsalabim* simulation.

Output files

The complete output of a *STEMsalabim* simulation is written to a NetCDF file. NetCDF is a binary, hierarchical file format for scientific data, based on HDF5. NetCDF/HDF5 allow us to compress the output data and store it in machine-readable, organized format while still only having to deal with a single output file.

You can read more about the output file structure at *Output file format*.

Hybrid Parallelization model

STEMsalabim simulations can be parallelized both via POSIX threads and via message passing interface (MPI). A typical simulation will use both schemes at the same time: MPI is used for communication between the computing nodes, and threads are used for intra-node parallelization, the usual multi-cpu/multi-core structure.

Hint: A high performance computation cluster is an array of many (equal) computing *nodes*. Typical highly-parallelized software uses more than one of the nodes for parallel computations. There is usually no memory

that is shared between the nodes, so all information required for the management of parallel computing needs to be explicitely communicated between the processes on the different machines. The quasi-standard for that is the message passing interface (MPI).

Let us assume a simulation that runs on M computers and each of them spawns N threads.

There is a single, special *master thread* (the thread 0 of the MPI process with rank 0) that orchestrates the simulation, i.e., manages and distributes work packages. All other threads ($(M \times N) - 1$) participate in the simulation. In MPI mode, each MPI process writes results to its own temporary file, and after each frozen lattice configuration the results are merged. Merging is carried out sequentially by each individual MPI processor, so that no race condition is ran into. The parameter `output.tmp_dir` (see *Parameter files*) should be set to a directory that is local to each MPI processor (e.g., `/tmp`).

A typical *STEMsalabim* simulation is composed of many independent multi-slice simulations that differ only in the position of the scanning probe. Hence, parallelization is done on the level of these multi-slice simulations, with each thread performing them independently from other threads. In order to reduce the number of MPI messages being sent around, only the main thread of each of the M MPI processors communicates with the master thread. The master thread sends a *work package* containing some number of probe pixels to be calculated to an MPI process, which then carries out all the calculations in parallel on its N threads. When a work package is finished, it requests another work package from the master MPI process until there is no work left. In parallel, the worker threads of the MPI process with rank 0 also work on emptying the work queue.

Note: Within one MPI processor, the threads can share their memory. As the main memory consumption comes from storing the weak phase objects of the slices in the multi-slice simulation, which don't change during the actual simulation, this greatly reduces memory usage as compared to MPI only parallelization. You should therefore always aim for hybrid parallelization!

3.3.2 Simulation Parameters

A *STEMsalabim* simulation is mainly configured via *parameter files*, with a few exceptions where configuration options may be overriden by *command line arguments*.

Parameter files

The configuration file that *STEMsalabim* expects for the command line parameter `--params` is formatted using the simple JSON-like syntax of libConfig syntax. Below all available parameters are tabulated. Each section (`block` in the libConfig syntax) is described separately below.

`application`

The `application` block contains general settings regarding the simulation.

```
application: {
    random_seed = 0;              # the random seed. 0 -> generate
    skip_simulation = false;      # skip the actual multi-slice simulation (for␣
↪debugging)
}
```

application.random_seed *unsigned int [default: '0']*

In the frozen lattice approximation, the atoms are randomly dislocated from their equilibrium position. The random seed for that can be specified here. If set to 0, a random seed is generated by the program. For reproduction of previous results, the seed can be set to a specific value.

application.skip_simulation *boolean [default: 'false']*

Do not carry out the actual multi-slice simulation, but only write the crystal information to the output file. This is for debugging purposes.

simulation

Some general settings specific to this simulation.

```
simulation: {
    title = "benchmarksmall";     # title of the simulation
    bandwidth_limiting = true;    # bandwidth limit the wave functions?
    normalize_always = false;     # normalize wave functions after each slice?
    output_file = "out.nc";       # output file name
    tmp_dir = "/tmp/";            # directory for temporary files.
    output_compress = "false";    # compress output data. This reduces file sizes,␣
↪but increases IO times.
}
```

simulation.title *string [default: ‘‘]*

The title of the simulation, as saved in the output NC file.

simulation.bandwidth_limiting *boolean [default: ‘true‘]*

Enforce cylindrical symmetry on all wave functions/operations. This results in all wave functions to be clipped significantly in momentum space. However, the benefit of reduced artefacts due to periodic boundary conditions is usually more important.

simulation.normalize_always *boolean [default: ‘false‘]*

Renormalize the electronic wave function after each slice of the multi-slice approximation, so that lost intensitiy due to limited k space grid is compensated for. Usually, this is not necessary, so the default is `false`.

simulation.output_file *string [default: ‘‘]*

Path to the result NetCDF file. Please read *Output file format* to learn about its format. Relative paths are interpreted relative to the working directory of the simulation.

simulation.tmp_dir *string [default: ‘folder of ouput file‘]*

When MPI parallelized, all MPI processors will write their results to temporary binary files in this directory, and only at the end of each frozen lattice configuration, the results are merged. We recommend using a local directory here (local to each MPI processor) so that access is fast. By default, the directory of the output file is used, which typically is **not local** but on a network file system. This can have a profound effect when CBED results are collected, i.e., output becomes large.

simulation.output_compress *boolean [default: ‘false‘]*

The written output data can be compressed by HDF5. This obviously takes more I/O time, but reduces output file sizes.

probe

Parameters of the STEM probe.

```
probe: {
    c5 = 5e6;                     # Fifth order spherical aberrations coefficient.␣
↪in nm
    cs = 2e3;                     # Third order spherical aberrations coefficient.␣
↪in nm
    defocus = -2.0;               # defocus value in nm
    fwhm_defoci = 6.0;            # FWHM of the defocus distribution when␣
↪simulating a defocus series for Cc.
```

```
    num_defoci = 1;                 # number of the defoci when simulating a defocus␣
↪series for Cc. Should be odd.
    astigmatism_ca = 0;             # Two-fold astigmatism. in nm
    astigmatism_angle = 0;          # Two-fold astigmatism angle. in mrad
    min_apert = 0.0;                # Minimum numerical aperture of the objective. in␣
↪mrad
    max_apert = 24.0;               # Maximum numerical aperture of the objective. in␣
↪mrad
    beam_energy = 200.0;            # Electron beam energy. in kV
    scan_density = 40;              # The sampling density for the electron probe␣
↪scanning. in 1/nm
}
```

probe.c5 *number (nm) [default: '5e7']*

Fifth order spherical aberrations coefficient.

probe.cs *number (nm) [default: '2e4']*

Third order spherical aberrations coefficient.

probe.defocus *number (nm) [default: '2.0']*

Probe defocus.

probe.fwhm_defoci *number (nm) [default: '6.0']*

STEMsalabim can calculate a defocus series to model chromatic aberrations. In that case, this parameter is the full-width-half-maximum of the normal distribution of defocus spread in `nm`.

probe.fwhm_defoci *number [default: '1']*

STEMsalabim can calculate a defocus series to model chromatic aberrations. In that case, this parameter is the number of defoci calculated.

probe.astigmatism_ca *number (nm) [default: '0']*

Two-fold astigmatism.

probe.astigmatism_angle *number (mrad) [default: '0']*

Two-fold astigmatism angle.

probe.min_apert *number (mrad) [default: '0']*

Minimum numerical aperture of the objective.

probe.max_apert *number (mrad) [default: '24']*

Maximum numerical aperture of the objective.

probe.beam_energy *number (keV) [default: '200']*

Electron beam energy.

probe.scan_density *number (1/nm) [default: '40.0']*

The sampling density for the electron probe scanning. This number multiplied by the supercell size in x and y direction gives the number of positions, i.e., scan points, that are simulated.

specimen

Settings for the specimen and potentials.

```
specimen: {
    max_potential_radius = 0.3;  # potential cut-off radius. in nm
    crystal_file = "input.xyz";  # xyz file with columns [Element, x, y, z, MSD]
}
```

specimen.max_potential_radius *number (nm) [default: '0.3']*

Distance, after which the atomic potentials are cut off during generation of the slices' transmission functions.

specimen.crystal_file *string [default: '']*

Path to the file containing the atomic coordinates. Please see *Crystal file format* to learn about its format. Relative paths are interpreted relative to the working directory of the simulation.

grating

Settings that describe the multi-slice algorithm, i.e., the density of the discretization and the slice thickness.

```
grating: {
    density = 360.0;              # The density for the real space and fourier␣
↪space grids. in 1/nm
    slice_thickness = 0.2715;     # Multi-slice slice thickness. in nm
}
```

grating.density *number (1/nm) [default: '360.0']*

The density for the real space and fourier space grids. This number multiplied by the supercell size in x and y direction gives the number of sampling grid points for the calculation. This also determines the maximum angle $\alpha = k\lambda$ that is described by the k-space grids.

grating.slice_thickness *number (nm) [default: '0.2715']*

The thickness of the slices in the multi-slice algorithm.

adf

Settings for collection of ADF data.

```
adf: {
    enabled = true;               # enable calculation and collection of ADF␣
↪intensities
    x = (0.0, 1.0);               # [min, max] where min and max are in relative␣
↪units
    y = (0.0, 1.0);               # [min, max] where min and max are in relative␣
↪units
    detector_min_angle = 1.0;     # inner detector angle in mrad
    detector_max_angle = 300.0;   # outer detector angle in mrad
    detector_num_angles = 300;    # number of bins of the ADF detector.
    detector_interval_exponent = 1.0;      # possibility to set non-linear detector␣
↪bins.
    save_slices_every = 0;        # save only every n slices. 0 -> only the sample␣
↪bottom is saved.
    average_configurations = true; # average the frozen phonon configurations in␣
↪the output file
    average_defoci = true;        # average the defoci in the output file
}
```

adf.enabled *boolean [default: 'true']*

Enable or disable calculation of ADF intensities completely.

adf.x *array [default: '[0.0,1.0]']*

The relative coordinates, between which the supercell is scanned by the electron probe. When `[0.0, 1.0]`, the whole `x` width of the supercell is scanned.

adf.y *array [default: '[0.0,1.0]']*

The relative coordinates, between which the supercell is scanned by the electron probe. When `[0.0, 1.0]`, the whole `y` width of the supercell is scanned.

adf.detector_min_angle *number (mrad) [default: '1.0']*

Inner ADF detector angle in mrad.

adf.detector_max_angle *number (mrad) [default: '300.0']*

Outer ADF detector angle in mrad.

adf.detector_num_angles *number [default: '300']*

Number of ADF detector angle bins.

adf.detector_interval_exponent *number [default: '1.0']*

A non-linear grid spacing of the detector grid can be chosen to increase the sampling at smaller angles. The `ith` grid point between θ_{min} and θ_{max} is calculated by $\theta_i = \theta_{min} + (i/N)^p(\theta_{max} - \theta_{min})$, where p is the interval exponent and N is the number of grid points as given by **detector.angles**. When set to `1.0`, the grid is linear.

Note: A fundamental lower limit for the grid spacing is given by the **grating.density** density. Finer sampling than the grating density will result in odd artefacts!

---

adf.save_slices_every *integer [default: '1']*

The ADF intensity is calculated and stored after every slice that is a multiple of this parameter. The default value of `1` results in every slice to be saved, higher numbers skip slices. The value `0` corresponds to the intensity only being saved after the last slice, i.e., at the sample bottom.

adf.average_configurations *boolean [default: 'true']*

Whether or not to perform in-place averaging over frozen lattice configurations during writing to the output file.

adf.average_defoci *boolean [default: 'true']*

Whether or not to perform in-place (weighted) averaging over defoci during writing to the output file.

cbed

Settings for collection of CBEDs.

Note: Storing CBEDs will increase the output file size drastically. Moreover, the information contained in the CBEDs may be redundant to the ADF data.

```
cbed: {
    enabled = true;              # enable calculation and collection of CBED␣
↪intensities
    x = (0.0, 1.0);              # [min, max] where min and max are in relative␣
↪units
    y = (0.0, 1.0);              # [min, max] where min and max are in relative␣
↪units
    size = [128, 128];           # When provided, this parameter determines the␣
↪size of CBEDs saved
                                 # to the output file. The CBEDs are resized using␣
↪bilinear interpolation.
    save_slices_every = 0;       # save only every n slices. 0 -> only the sample␣
↪bottom is saved.
    average_configurations = true; # average the frozen phonon configurations in␣
↪the output file
```

```
    average_defoci = true;        # average the defoci in the output file
}
```

cbed.enabled *boolean [default: 'false']*

Enable or disable calculation of CBED intensities completely.

cbed.x *array [default: '[0.0,1.0]']*

The relative `x` coordinates, between which CBED images are to be saved. When `[0.0, 1.0]`, the whole `y` width of the supercell is scanned.

cbed.y *array [default: '[0.0,1.0]']*

The relative y coordinates, between which CBED images are to be saved. When `[0.0, 1.0]`, the whole `y` width of the supercell is scanned.

cbed.size *array [default: '[0.0,0.0]']*

Width and height of the CBEDs. Saving CBEDs can become very disk-space intensive, especially when the `k` grids are huge (as required for big samples). When this parameter is given, one can choose the size of the CBEDs that are saved to the output file. The images are reduced using bilinear interpolation. The total intensity is preserved. Leave the parameter out or set any direction to 0 to use the original size.

cbed.save_slices_every *integer [default: '1']*

The STEM intensity is calculated and stored after every slice that is a multiple of this parameter. The default value of `1` results in every slice to be saved, higher numbers skip slices. The value `0` corresponds to the intensity only being saved after the last slice, i.e., at the sample bottom.

cbed.average_configurations *boolean [default: 'true']*

Whether or not to perform in-place averaging over frozen lattice configurations during writing to the output file.

cbed.average_defoci *boolean [default: 'true']*

Whether or not to perform in-place (weighted) averaging over defoci during writing to the output file.

`frozen_phonon`

Settings for the `frozen_phonon` algorithm to simulate TDS.

```
frozen_phonon: {
    number_configurations = 15;   # Number of frozen phonon configurations to␣
↪calculate
    fixed_slicing = true;         # When this is true, the z coordinate is not␣
↪varied during phonon vibrations.
}
```

frozen_phonon.number_configurations *integer [default: '1']*

Number of frozen phonon configurations to calculate.

frozen_phonon.fixed_slicing *boolean [default: 'true']*

When true, the `z` coordinates (beam direction) of the atoms is not varied, resulting in fixed slicing between subsequent frozen phonon configurations.

`http_reporting`

Settings for HTTP reporting of simulation progress.

```
http_reporting: {                   # send POST status requests of the simulation to␣
↪some HTTP endpoint.
    reporting = true;               # reporting can be disabled with this parameter.
    url = "http://my_url";          # The URL to POST to.
    auth_user = "my_user";          # username for HTTP basic auth
    auth_pass = "my_pass";          # password for HTTP basic auth
    parameters: {                   # All parameters in this http_reporting.
↪parameters are sent as query.
                                    # in addition to the status information. Use␣
↪whatever your API needs.
        my_login = "username";
        my_token = "abcdef";
    }
}
```

http_reporting.reporting *boolean [default: 'false']*

Should the simulation status be announced via HTTP POST requests? See *HTTP Status reporting* for details.

http_reporting.url *string [default: '']*

The HTTP API endpoint for the status reporting. See *HTTP Status reporting* for details.

http_reporting.auth_user *string [default: '']*

The username for HTTP Basic Authentication. Leave empty to not authenticate.

http_reporting.auth_pass *string [default: '']*

The password for HTTP Basic Authentication.

http_reporting.params *libConfig block [default: '{}']*

Additional parameters to include in the HTTP POST JSON requests. See *HTTP Status reporting* for details.

Command line arguments

–help, -h *flag*

Display a help message with a brief description of available command line parameters.

–params, -p *string (required)*

Path to the configuration file as explained above in *Parameter files* files.

–num-threads *integer [default: '1']*

Number of threads per MPI core. Note, that *STEMsalabim* will do nothing if only parallelized via threads and `--num-threads=1`, as thread `0` of the master MPI process does not participate in the calculation. See *Hybrid Parallelization model* for details.

–package-size *integer [default: '10']*

The number of tasks that are sent to an MPI process. This should scale with the number of threads each MPI process spawns. A good value is $10\times$ the value of **–num_threads**.

–skip-simulation *flag*

Override configuration parameter **application.skip_simulation** to true when this flag is set.

–num-configurations *integer*

Override configuration parameter **frozen_phonon.number_configurations**.

–defocus, -d *float*

Override the value of the **probe.defocus** setting. If specified exactly once, a single defocus is calculated. If specified *exactly three times*, the functionality is similar to the **probe.defocus** parameter.

–tmp-dir *string*

Override the value of the **output.tmp_dir** setting.

–output-file, -o *string*

Override the value of the **output.output_file** setting.

–crystal-file, -c *string*

Override the value of the **specimen.crystal_file** setting.

–title, -t *string*

Override the value of the **simulation.title** setting.

3.3.3 File formats

A *STEMsalabim* simulation is set-up via **input files** and its results are stored in an **output file**. The file for configuring a simulation is described in detail at *Parameter files*. Here, we describe the format of the **crystal file**, i.e., the atomic information about the specimen, and the **output file**, in which the results are stored.

Crystal file format

The crystal file is expected to be in XYZ format.

1. The **first line** contains the number of atoms.
2. The **second line** contains the cell dimension in x,y,z direction as floating point numbers in units of `nm`, separated by a space. Optionally, it can contain the x, y, z dimensions in the .exyz format:

   ```
   Lattice="lx 0.0 0.0 0.0 ly 0.0 0.0 0.0 lz"
   ```

3. The atomic information is given from the **third line onwards**, with each line corresponding to a single atom. Each line must have *exactly 5 columns*:
   - The atomic species as elemental abbreviation (e.g., `Ga` for gallium)
   - the x,y,z coordinates as floating point numbers in units of `nm`
   - the mean square displacement for the frozen lattice dislocations in units of nm^2.
   - (**optional**) The id of the slice this atom belongs to. This can be used to do custom slicing.

Below is a very brief, artificial example (without custom slicing):

```
5
1.0 2.0 10.0
Ga   0.0   0.0    0.0    1e-5
P    0.2   0.1    0.0    2e-5
Ga   0.0   0.0    1.0    1e-5
P    1.2   0.1    0.0    2e-5
O    1.0   2.0   10.0    0.0
```

Note: Atomic coordinates outside of the cell are periodically wrapped in `x` and `y` and clipped to the simulation box in `z` direction!

Output file format

All results of a *STEMsalabim* simulation are written to a binary NetCDF file. The NetCDF format is based on the Hierarchical Data Format and there are libraries to read the data for many programming languages.

The structure of NetCDF files is hierarchical and organized in groups. The following groups are written by *STEMsalabim*:

runtime

This group contains information about the program and the simulation, such as version, UUID and so on.

AMBER

This group contains the atomic coordinates, species, displacements, radii, etc. for the complete crystal for each single calculated frozen lattice configuration, as well as for each calculated defocus value. The AMBER group content is compatible with the AMBER specifications. A *STEMsalabim* NetCDF file can be opened seamlessly with the Ovito crystal viewer.

params

All simulation parameters are collected in the `params` group as attributes.

adf

This group contains the simulated ADF intensities, the coordinates of the electron probe beam during scanning, the detector angle grid that is used, and coordinates of the slices as used in the multi-slice algorithm.

cbed

This group contains the simulated CBED intensities, the coordinates of the electron probe beam during scanning, k-space grid, and coordinates of the slices as used in the multi-slice algorithm.

Reading NC Files

For a detailed view of the structure, we suggest using the ncdump utility: `ncdump -h some_results_file.nc`. As the underlying file format of NetCDF is HDF5, you may use any other HDF5 viewer to have a look at the results.

There are NetCDF bindings for most popular programming languages.

1. In MATLAB, we recommend using `h5read()` and the other HDF5 functions.
2. For Python, use the netCDF4 module.

3.3.4 HTTP Status reporting

STEMsalabim simulations may take a long time, even when running them in parallel on many processors. In order to ease tracking of the status of running simulations, we built reporting via HTTP POST requests into the program.

In order to use that feature, the libCURL library has to be installed and *STEMsalabim* needs to be linked against it.

Configure HTTP reporting

To configure HTTP reporting, please add the `http_reporting: {}` block to your simulations's parameter file, containing at least `reporting = true;` and the url to report to, `url = "http://my_server_address:port/path";`.

If you want to use HTTP basic authentication, you may also specify the options `auth_user = "your_user";` and `auth_pass = "your_pass";`. Note, that HTTP basic auth will be enabled as soon as `auth_user` is not empty. You should therefore only fill in that field when you want to use authentication.

Additional, custom payload for the HTTP requests may be specified in the sub-block `parameters: {}`. Each key-value pair in this block is translated into JSON and appended to each request. This allows you to use custom authentication techniques, such as token-based authentication etc.

An example configuration block with HTTP basic authentication may look like this:

```
http_reporting: {
    reporting = true;
    url = "http://my_api_endpoint:8000/stemsalabim-reporting";
    auth_user = "my_user";
    auth_pass = "my_pass";
    parameters: {
        simulation_category = "suitable for many Nature papers";
    }
}
```

The status requests

In each request that *STEMsalabim* sends, some JSON payload is common. In addition to the JSON values specified in the parameter file (see *Configure HTTP reporting*), the following parameters are always reported:

```
time: // The currrent date and time
id: // the UUID of the simulation
num_threads: // the number of threads of each MPI processor
num_processors: // the number of MPI processors
num_defoci: // the total number of defoci to calculate
num_configurations: // the total number of frozen phonon configurations to␣
↪calculate
event: // A code for what event is reported. see below.
```

The following four different `event` codes, each for a different event, are reported:

`START_SIMULATION`: A simulation is started

This request is sent at the beginning of a simulation. Additional key/value pairs sent are:

```
event: "START_SIMULATION"
version: // program version
git_commit: // git commit hash of the program version
title: // simulation title
```

`START_DEFOCUS`: A defocus iteration is started

This request is sent at the beginning of a defocus iteration. Additional key/value pairs sent are:

```
event: "START_DEFOCUS"
defocus: // the defocus value in nm
defocus_index: // the index of the defocus, between 0 and num_defoci
```

`START_FP_CONFIGURATION`: A frozen phonon iteration is started

This request is sent at the beginning of a frozen phonon configuration. Additional key/value pairs sent are:

```
event: "START_FP_CONFIGURATION"
defocus: // the defocus value in nm
defocus_index: // the index of the defocus, between 0 and num_defoci
configuration_index: // the index of the configuration, between 0 and num_
↪configurations
```

`PROGRESS`: Progress report

This request is sent during the calculation, typically after each integer percent of the simulation finished. Additional key/value pairs sent are:

```
event: "START_CONFIGURATION"
defocus: // the defocus value in nm
defocus_index: // the index of the defocus, between 0 and num_defoci
configuration_index: // the index of the configuration, between 0 and num_
↪configurations
progress: // progress between 0 and 1 of this configuration iteration within this␣
↪defocus iteration
```

`FINISH`: Simulation finished

This request is sent when the simulation finished. Additional key/value pairs sent are:

```
event: "FINISH"
```

How to process the reports

Obviously, in order to register the requests, an HTTP(S) server needs to be running on the target machine. For example, a very simple server in python using the http://flask.pocoo.org/ package, that only echos the requests, can be implemented as:

```
#!/usr/bin/env python

from flask import Flask
from flask import request
import json

app = Flask(__name__)

@app.route('/', methods=['POST'])
def echo():
    content = request.get_json()
    print(json.dumps(content, indent=4))
    return ""

if __name__ == "__main__":
    app.run()
```

Run the script and then start a *STEMsalabim* simulation to see requests imcoming.

3.3.5 Frequently Asked Questions

What about the name STEMsalabim?

STEMsalabim stems from the word *Simsalabim* which is known in Germany as what a magician says when casting spells (see here for a german dictionary entry). Furthermore, the phrase appears in the famous German children's song Auf einem Baum ein Kuckuck, where the phrase to our knowledge doesn't really have any meaning. (The song is about a cuckoo sitting on a tree and being shot by a huntsman...) Most germans will recognize the word. There is no contextual relation between STEM and the word Simsalabim.

When we first started writing the code we came up with it as a working title and we then simply agreed on keeping it.

CHAPTER 4

What's new

4.1 STEMsalabim 4.0.1, 4.0.2

March 23rd, 2018 March 21st, 2018

- Bug fixes
- Changed chunking of the ADF variable

4.2 STEMsalabim 4.0

March 9th, 2018

IMPORTANT

I'm releasing this version as 4.0.0, but neither the input nor output files changed. The parameter *precision* has become deprecated and there is a parameter *tmp-dir*. Please see the documentation.

- Removed option for double precision. When requested, this may be re-introduced, but it slowed down compilation times and made the code significantly more complicated. The multislice algorithm with all its approximations, including the scattering factor parametrization, is not precise enough to make the difference between single and double precision significant.
- Improved the Wave class, so that some important parts can now be vectorized by the compiler.
- Introduced some more caches, so that performance could greatly be improved. STEMsalabim should now be about twice as fast as before.
- Results of the MPI processors are now written to temporary files and merged after each configuration is finished. This removes many MPI calls which tended to slow down the simulation. See the *–tmp-dir* parameter.
- Moved the *Element*, *Atom*, and *Scattering* classes to their own (isolated) library *libatomic*. This is easier to maintain.
- Simplified MPI communication by getting rid of serialization of C++ objects into char arrays. This is too error-prone anyway.
- Added compatibility with the Intel parallel studio (Compilers, MKL for FFTs, Intel MPI). Tested with Intel 17 only.

- Some minor fixes and improvements.

4.3 STEMsalabim 3.1.0, 3.1.1, 3.1.2, 3.1.3, 3.1.4

February 23nd, 2018

- Added GPL-3 License
- Moved all the code to Gitlab
- Moved documentation to readthedocs.org
- Added Gitlab CI

4.4 STEMsalabim 3.0.1 and 3.0.2

February 22nd, 2018

- Fixed a few bugs
- Improved the CMake files for better build process

4.5 STEMsalabim 3.0.0

January 3rd, 2018

- Reworked input/output file format.
- Reworked CBED storing. Blank areas due to bandwidth limiting are now removed.
- Changes to the configuration, mainly to defocus series.
- Compression can be switched on and off via config file now.
- Prepared the project for adding a Python API in the future.
- Added tapering to smoothen the atomic potential at the edges as explained in I. Lobato, et al, Ultramicroscopy 168, 17 (2016).
- Added analysis scripts for Python and MATLAB to the Si 001 example.

4.6 STEMsalabim 2.0.0

August 1st, 2017

- Changed Documentation generator to Sphinx
- Introduced a lot of memory management to prevent memory fragmentation bugs
- split STEMsalabim into a core library and binaries to ease creation of tools
- Added diagnostics output with –print-diagnostics
- Code cleanup and commenting

4.7 STEMsalabim 2.0.0-beta2

April 20th, 2017

- Added possibility to also save CBEDs, i.e., the kx/ky resolved intensities in reciprocal space.
- Improved documentation.
- Switched to NetCDF C API. Dependency on NetCDF C++ is dropped.
- Switched to distributed (parallel) writing of the NC files, which is required for the CBED feature. This requires NetCDF C and HDF5 to be compiled with MPI support.

4.8 STEMsalabim 2.0.0-beta

March 27th, 2017

- Lots of code refactoring and cleanup
- Added Doxygen doc strings
- Added Markdown documentation and `make doc` target to build this website.
- Refined the output file structure
- Added HTTP reporting feature
- Added `fixed_slicing` option to fix each atom's slice througout the simulation
- Got rid of the boost libraries to ease compilation and installation

4.9 STEMsalabim 1.0

November 18th, 2016

- Initial release.

CHAPTER 5

Citing STEMsalabim

A technical paper introducing STEMsalabim is published in Ultramicroscopy journal [1].

If you use our program or its results, please cite us. You may use the following bibTeX entry:

```
@article{Oelerich2017,
    title = "STEMsalabim: A high-performance computing cluster friendly code for scanning transmission electron microscopy image simulations of thin specimens",
    journal = "Ultramicroscopy",
    volume = "177",
    number = "",
    pages = "91 - 96",
    year = "2017",
    note = "",
    issn = "0304-3991",
    doi = "http://dx.doi.org/10.1016/j.ultramic.2017.03.010",
    url = "http://www.sciencedirect.com/science/article/pii/S030439911630300X",
    author = "Jan Oliver Oelerich and Lennart Duschek and Jürgen Belz and Andreas Beyer and Sergei D. Baranovskii and Kerstin Volz",
    keywords = "Multislice simulations",
    keywords = "Electron scattering factors",
    keywords = "MPI",
    keywords = "Phonons"
}
```

[1]: http://dx.doi.org/10.1016/j.ultramic.2017.03.010

CHAPTER 6

Research done with STEMsalabim

6.1 2018

- Composition determination of multinary III/V semiconductors via STEM HAADF multislice simulations L. Duschek, A. Beyer, J. O. Oelerich, K. Volz

6.2 2017

- Surface relaxation of strained Ga(P,As)/GaP heterostructures investigated by HAADF STEM A. Beyer, L. Duschek, J. Belz, J. O. Oelerich, K. Jandieri, K. Volz
- Atomic structure of 'W'-type quantum well heterostructures investigated by aberration-corrected STEM P. Kuekelhan, A. Beyer, C. Fuchs, M.J. Weseloh, S.W. Koch, W. Stolz, K. Volz
- Influence of surface relaxation of strained layers on atomic resolution ADF imaging A. Beyer, L. Duschek, J. Belz, J. O. Oelerich, K. Jandieri, K. Volz
- Local Bi ordering in MOVPE grown Ga(As,Bi) investigated by high resolution scanning transmission electron microscopy A. Beyer, N. Knaub, P. Rosenow, K. Jandieri, P. Ludewig, L. Bannow, S. W.Koch, R. Tonner, K. Volz
- Influence of surface relaxation of strained layers on atomic resolution ADF imaging A. Beyer, L. Duschek, J. Belz, J. O. Oelerich, K. Jandieri, K. Volz
- STEMsalabim: A high-performance computing cluster friendly code for scanning transmission electron microscopy image simulations of thin specimens J. O. Oelerich, L. Duschek, J. Belz, A. Beyer, S. D. Baranovskii, K. Volz