Current state of the openSTB sonar simulator

Blair Bonnett[1], Holger Schmaljohann[2], Sudhanshu Apte[1], and Thomas Fickenscher[1]

[1] Chair for Radio Frequency Engineering, Helmut Schmidt University, 22043 Hamburg, Germany
[2] Bundeswehr Technical Center for Ships and Naval Weapons, Maritime Technology and Research (WTD 71), 24340 Eckenförde, Germany

Corresponding author: *Blair Bonnett, Lehrstuhl für Hochfrequenztechnik, Fakultät für Elektrotechnik, Helmut-Schmidt-Universität, 22039 Hamburg, Germany; blair.bonnett@hsu-hh.de*

Abstract: *The development and implementation of algorithms for synthetic aperture sonar (SAS) processing requires data with ground truth for validation. In most instances, it is difficult or even impossible to collect real data with sufficiently accurate ground truth. Instead, simulated data is commonly used for this purpose. The requirements for these simulations may range from a simple setup with a few point targets to large models of realistic scenes requiring significant computational resources.*

We previously outlined plans to create an open-source sonar simulation framework to facilitate community-driven development and sharing of simulation techniques. This project, subsequently named the openSTB (open sonar toolboxes) simulator, is now under active development. It is written in Python utilising widely available numerical processing libraries such as NumPy, and has support for high-performance computing (HPC) environments built in. It is designed in a modular structure with each plugin having a specified interface. A simulation is then created by combining the desired plugins. As well as the actual simulation methods, this modular structure extends to the loading of inputs and saving of results, allowing the simulator to be integrated with existing data formats and other processing tools. Many common implementations of these plugins are included, while the defined interface allows users to write their own to customise the simulations to their needs.

In this paper we will present the current state of the openSTB simulator. This will include examples of different simulations that can be performed, including simulations utilising an HPC cluster. Future plans for further included plugins and simulation techniques will also be outlined.

Keywords: *synthetic aperture sonar, simulation*

1. INTRODUCTION

Developing, implementing and improving any processing algorithm requires data with *ground truth*, i.e., for which correct results is known, in order to evaluate its performance. It is often difficult or impossible to collect such data suitable for verifying sonar algorithms. For example, tracking the position of a vehicle while it is underwater typically relies on an intertial navigation system. This is not sufficiently accurate to generate a well-focused synthetic aperture sonar (SAS) image. Techniques such as the displaced phase-centre antenna (DPCA) algorithm are used to correct the navigation data based on the collected sonar data. By definition, real data with this problem does not have ground truth and so using it to analyse the performance of the correction algorithm can only be done with secondary statistics such as the sharpness of an image generated using the corrected navigation. Simulated data is therefore vital for evaluating the performance of such algorithms.

The complexity of the required simulations will vary based on application. Simulating a small number of ideal point targets can be sufficient for simple cases such as checking the point response of a processing chain. Bigger scenes can be modelled with a large number of point targets. Some algorithms may required more advanced techniques such as incorporating elastic scattering effects or modelling shadows cast by objects in the scene. Each increase in simulation complexity results in a corresponding increase in both the effort required to write and validate the simulation code and the computational resources required to perform the simulation in a timely manner.

In a presentation at the International Conference of Underwater Acoustics in 2024, we postulated that a significant majority of sonar researchers had at some time written a simulator of some form. We then proposed the creation of an open-source sonar simulation framework to refocus such individual development efforts into a single tool that would benefit the whole community. Access to a high-quality simulation tool would also reduce the cost of entry for new researchers in the community who need data to work with. A standard tool to create controlled, reproducible data could also be useful in evaluating and comparing different algorithms for a particular task.

This simulation framework, named the openSTB (open sonar toolboxes) simulator, is now under development using the BSD-2-Clause Plus Patent license. In this paper we will first list the general design principles used for the simulator and then the software stack it uses. An overview of how the simulator can be accessed and used along with its current features are then given, followed by some example simulations. We conclude with an overview of our ideas for future improvements to the simulator.

2. DESIGN PRINCIPLES

A general-purpose simulator needs to be readily adaptable to the needs of each user. To ensure this flexibility, the following design principles have been adopted for the openSTB simulator:

- **Everything should be a plugin.** All operations, from loading a configuration through modelling a physical process to scheduling computation of each piece of the simulation and saving the results in the desired format should be performed by plugins. A simulation is then defined as the set of plugins that should be combined to generate the output.

- **Plugins should be easy to write.** In order to customise a simulation, the user may wish to write their own plugins. The interface for each kind of plugin needs to be well-defined, and the expected inputs and outputs clearly documented.
- **Plugins should be easy to install and use.** Having written a plugin, the user needs be able to easily install it so that the simulator can find it, or be able to simply point the simulator to the file containing the plugin.
- **Batteries should be included.** Many operations will be common across a wide range of simulations. Plugins for such operations should be provided with the simulator.

3. SOFTWARE STACK

The Python programming language has been chosen as it is itself open-source and widely available, and has a large numerical and scientific programming ecosystem. We also believe that a large proportion of the target audience of the project will have some degree of familiarity with both Python and the scientific Python ecosystem. Although the intention is for the simulator to contain sufficient plugins for many use-cases, using a language with which users are likely to be familiar lowers the barrier to customising it with their own plugins.

As of the time of writing the external libraries the simulator depends on include the following:

- NumPy (`https://numpy.org/`) for array handling and numerical computation
- SciPy (`https://scipy.org/`) for various scientific computations
- Dask (`https://www.dask.org/`) and its distributed task scheduler (`https://distributed.dask.org/`) for parallel and distributed computing
- Zarr (`https://zarr.dev/`) for storing results
- Quaternionic (`https://quaternionic.readthedocs.io/`) to implement quaternion algebra for rotations
- Click (`https://click.palletsprojects.com/`) to generate a command-line interface to use the simulator

It is worth noting that the Numba just-in-time compiler (`https://numba.pydata.org/`) is included as a sub-dependency of these libraries. It is envisaged that in the future some parts of the simulator may utilise Numba for performance improvements.

Large simulations are computationally expensive, but the process is typically straightforward to parallelise as in many cases the final result for a collection of targets can be broken into a sum of the results obtained from subsets of those targets. The Dask library used by the simulator provides support for parallel computing by maintaining a graph of tasks to be performed and scheduling their execution. The standard Dask scheduler works on a single machine, utilising all the CPU cores (or a portion of the cores if desired) to perform a parallel computation. To utilise a high-performance computing (HPC) environment, the Dask distributed scheduler can be used. This distributes the computation across multiple nodes in a cluster, and can be used with many common HPC resource managers. The simulator includes support for both the single-machine and distributed schedulers.

It is intended to follow the scientific Python community's SPEC0 (Minimum Supported Dependencies) guideline [1] to determine which versions of Python and the libraries to support. Broadly, this corresponds to versions of Python released in the previous three years and versions of the libraries released in the previous two years. Exceptions may be made if required, for example to avoid a critical bug in a library.

4. USING THE SIMULATOR

The openSTB website at `https://openstb.dev` hosts documentation and other details about the project. Development of the simulator occurs on GitHub at `https://github.com/openSTB/simulator`. Users who wish to help with future development, or who want to test new features, can install it from that Git repository. Releases are made on the Python package index (PyPI) under the name `openstb-simulator` so that the simulator can be installed with the user's preferred Python package manager.

It is envisaged that many users will want a simple interface which they can supply with a configuration file and receive the results of the simulation. The simulator includes a simple command-line interface to provide this functionality. By default, it reads a configuration file in TOML format but, in keeping with the design principles, support for other configuration formats can be added as plugins (this could include support for loading the configuration from a network source). The configuration file defines the set of plugins, including all relevant parameters for those plugins, which make up the simulation.

Other users may wish to use the simulator as a library, for example to perform multiple simulations with varying parameters, to integrate it into an existing interface or simply to use the functionality of some plugins. The code is structured so as to facilitate such use, whether from a simple script or from a more interactive source such as a Jupyter notebook. It is commented, has full docstrings to describe the behaviour of each function and includes static typing information.

4.1. CURRENTLY INCLUDED PLUGINS

As mentioned in the design guidelines, the components of a simulation are implemented as plugins. In this section, we provide a non-exhaustive overview of the plugins that are included with the simulator at the time of writing. This is intended both to show the sorts of simulation that are possible 'out of the box', and to give an idea as to how simulations can be customised by replacing the appropriate components.

Ultimately, a sonar system can be described to the simulator as a transmitter, one or more receivers and the signal used by the transmitter. A transducer plugin gives details of the position, orientation and distorting factors (such as the beampattern) of both the transmitter and the receivers. The signal plugin is responsible for sampling the transmitted signal for use in the simulation. These samples can be windowed by an attached window plugin; currently a set of common windows are included. These plugins can be separately defined or, for convenience, a system plugin can be used to provide all details of a particular system.

The trajectory followed by the system during the simulation is specified by a plugin which reports the position, orientation and velocity of the system at a requested time. Plugins implementing idealised trajectories are included, and it is straightforward to write a custom plugin which loads this information from an external dataset. Similarly, the times at which a ping is started is provided by a plugin; implementations for a constant time interval between pings and a constant distance travelled between pings are included.

After a ping is transmitted, the simulator needs to calculate how long the signal takes to reach a particular target and return to the sonar. This is done by a travel time calculator plugin; implementations making the stop-and-hop assumption and iteratively calculating the travel time to include intra-ping motion (described in more detail in the example simulation section) are both included.

Distortions, either amplitude-only or frequency-dependent, model how the signal changes as it propagates. Currently included plugins include acoustic attenuation based on the model in [2], Doppler distortion of the signal due to the movement of the sonar, geometric spreading and beampattern effects of the transducers.

Higher-level operations are also designed as plugins. This includes how to load simulation configuration from a file (support for a TOML configuration file is included), how to utilise the other plugins to perform a simulation (a point target simulation is currently implemented) and how to store the results (support for both NumPy and MATLAB files are included).

5. EXAMPLE SIMULATIONS

The following simulations use idealised point targets to model a scene. Each of these targets has a reflectivity factor which is the fraction of incident energy that is scattered back to the sonar; note that there is no aspect dependence. This is a simple model of the behaviour of a target, but can suffice for many applications, and is also a useful approach to validate the general design of the simulator framework.

The simulation is performed in the frequency domain. This allows the time delay corresponding to the travel time to be applied without interpolation, and also enables frequency-dependent effects to be modelled. The steps taken to calculate the echo observed by a receiver from a target are:

1. Evaluate the spectrum of the transmitted signal at the frequencies corresponding to the desired length and sampling interval of the final simulated result.
2. Modify the spectrum with any transmit effects, for example the beampattern of the transmitter, spreading losses or acoustic attenuation.
3. Compute the two-way travel time it takes the transmitted pulse to reach the target and echo back to the receiver.
4. Apply this delay and the target reflectivity to the spectrum.
5. Apply any receive distortions to the spectrum.
6. Use an inverse Fourier transform to obtain the time-domain echo signal from the target.

Summing the echoes from each of the targets in the scene gives the final signal recorded by the receiver. Note that in practice this summation is performed in the frequency domain before taking the inverse Fourier transform. For large scenes, each worker can independently compute the echo from a subset of the targets with the final result then being the sum of all these intermediate results.

The system used for the examples consists of a 3 cm wide and 5 cm high transmitter and a 35-element linear array of 3 cm by 3 cm receivers with a 3.5 cm spacing between elements. A 10 ms long linear frequency-modulated chirp from 70 kHz to 90 kHz was transmitted. Spherical spreading, acoustic attenuation and the beampatterns of the transducers were included in the simulation, though it is worth noting that our processing toolchain compensates for these effects when reconstructing an image. Configuration files for all the following simulations are included as examples with the simulator.

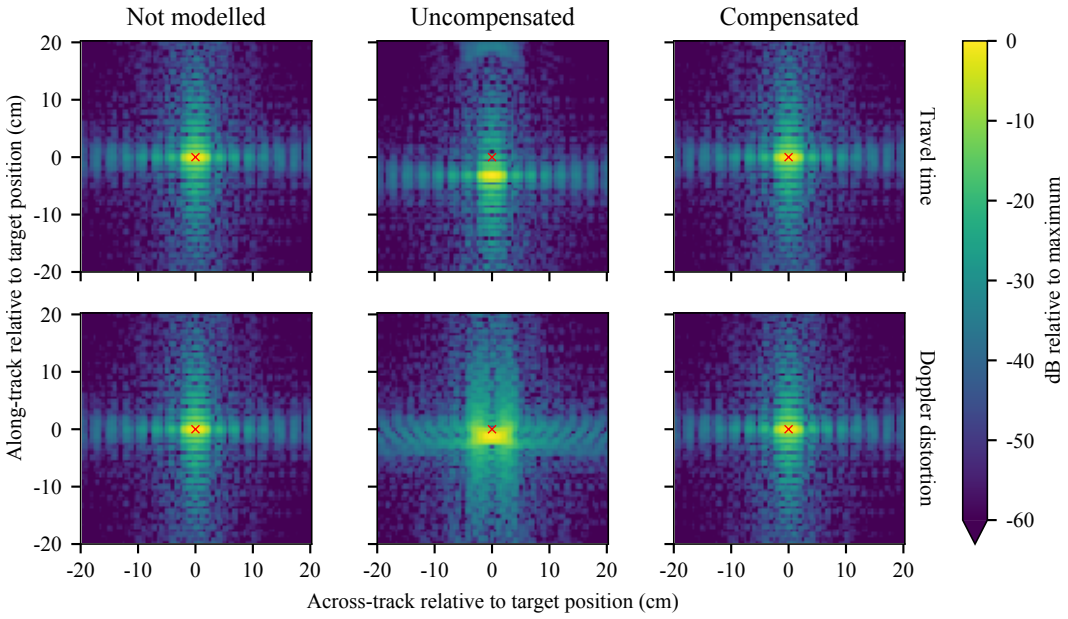


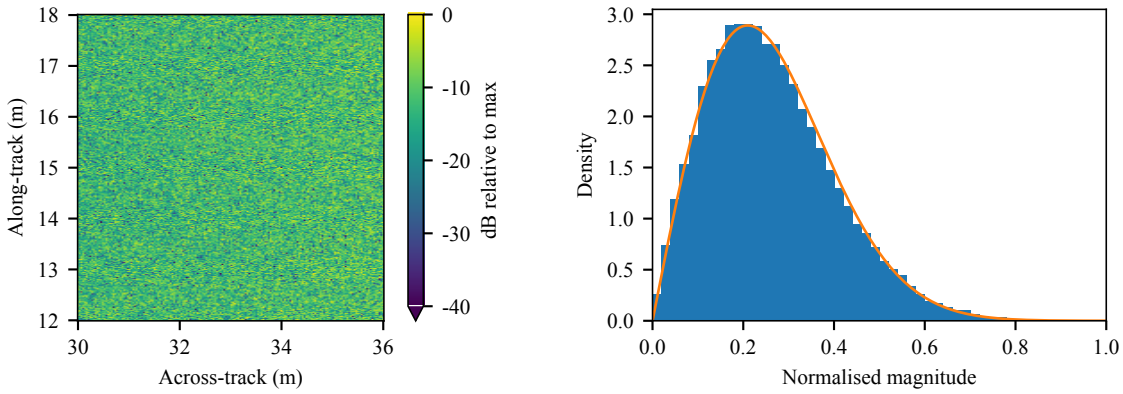
Figure 1: Point-spread function of images reconstructed from simulations of a single point target. The top row models the impact of the vehicle motion on the two-way travel time while the bottom row models the Doppler distortion of the signal. The left column has no motion effects, the middle column shows the impact of the uncompensated motion, and the right column applies compensation during reconstruction. The axes show position relative to the target location which is marked with a red cross.

5.1. INTRA-PING MOTION

It is not uncommon to use the stop-and-hop assumption that the vehicle is stationary for the duration of a ping and then instantaneously moves to the position of the next ping. From a simulation viewpoint, this both simplifies the computation of the two-way travel time of an acoustic pulse, and removes a source of distortion which may otherwise complicate the analysis of the results if the intra-ping motion is not relevant. However, in some cases it is desirable to model the continuous movement of the vehicle during ping to give a more realistic result.

One of the included travel time plugins supports finding the two-way travel time taking motion into account. The velocity at the time of transmission is used to approximate the receive position, and then the time of reception is iteratively adjusted until the time taken for the sonar to move to the proposed receive position matches (within some tolerance) the time the sound would take to move along the corresponding path to the target and back to that receive position. The upper row of Figure 1 shows (from left to right) the point-spread function (PSF) of a single target with the stop-and-hop approximation, the PSF with this iterative travel-time plugin in use and is not compensated during imaging, and the PSF if the motion is compensated during imaging. The intra-ping motion shifts the target, and the compensation corrects the shift.

The movement of the sonar also causes a Doppler shift to the signal. One of the plugins models this following the wide-band approach of [3] to calculate a scale factor η and modify the spectrum of the echo from each target accordingly. The lower row of Figure 1 shows a similar set of PSFs (Doppler not modelled, uncompensated and compensated). The distortion consists of both a shift and a blurring of the PSF, and can be compensated during imaging.



(a) *(b)*

Figure 2: (a) The intensity image and (b) the magnitude histogram of a piece of the flat, bland seafloor modelled by idealised point targets. The overlaid line in (b) is a Rayleigh distribution fit to the data.

5.2. BLAND SEAFLOOR

It is important to ensure that the available resources are being fully utilised during a simulation. To check this, a patch of bland seafloor was modelled by a rectangle measuring 30 m in along-track and 50 m in across-track containing uniformly distributed point targets at a density of 10 000 per m^2 or 3.44 per square wavelength at the centre frequency. The system followed a 30 m long trajectory at a 10 m height over ground along the edge of the rectangle. 66 pings were simulated, spaced 45 cm apart and each recording a trace of duration 0.25 s.

Five nodes of the HSUper cluster at Helmut Schmidt University were used to run the simulation. Each node had two Intel Xeon Platinum 8360Y processors and 256 GB of RAM, yielding a total of 360 CPU cores. The simulation, consisting of 2310 traces from each of the 15 million targets, was completed in 1 day and 20 hours of wall time using almost 690 days of CPU time. This successfully saturated the available cores, indicating the simulation controller and Dask scheduler were not causing bottlenecks. Figure 2 shows a section of an image reconstructed from the simulated data. This image exhibits the expected fully-developed speckle with the histogram of the magnitudes having the corresponding Rayleigh distribution.

6. FUTURE PLANS

Although point target modelling is useful in many cases, it is intended to add a facet-based simulation method. This would allow scattering models based on the target properties and the incident and scattering angles to be attached to each target in a scene. The shadows cast by objects in the scene could also be modelled with such a simulator, for example by following the methods in [4]. Further extensions, such as to model double-scattering within the scene, could also be investigated. Other simulation methods, for example the recently-published Fourier-domain wavefield rendering [5], could be considered in the future.

At this stage, the simulations only use CPU processing. It is desired to extend it to use available GPUs. This would include investigating how to optimally use a mixture of GPU and CPU resources, such as in a single workstation or in a cluster with many CPU nodes and few GPU nodes.

Some investigation into how the simulator behaves with very large scenes is required. This may necessitate some changes in how a simulation is divided into tasks to avoid memory problems. On a related note, the limits of the Dask task scheduler have not yet been explored. Each task requires a scheduler overhead of around 200 µs to handle [6]. Above a certain number of cores (dependent on the duration of the tasks), the scheduler will be saturated and cause a bottleneck. This upper limit is likely to be in the thousands of cores. It is suspected that this will not be a problem for most use-cases. Increasing the number of usable cores would require either increasing the size of each task (which has its own limits, for example, the available memory on a node) or moving to a custom scheduling solution, such as direct MPI communication.

It is hoped that plugins will be contributed by the community. Plugins which are likely to be widely useful and straightforward to maintain may be accepted into the main repository. It is intended to create an organisation to store and list user-contributed plugins that are not added to the main project (for those familiar with the Sphinx documentation generator, similar to the `sphinx-contrib` group for its community-provided addons).

The overall project (open sonar toolboxes) has been designed to be able to expanded to projects other than simulation. For example, a toolbox implementing common image reconstruction methods could be created, or one which performs post-processing operations such as autofocus. These would not be required to be in Python, and could serve both as useful starting points for new research and in benchmarks for comparing results between different systems or algorithms. Community involvement in this, as well as improvements to the simulation framework, would be greatly welcome.

ACKNOWLEDGEMENTS

Computational resources (HPC cluster HSUper) have been provided by the project hpc.bw, funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU.

REFERENCES

[1] Scientific Python. *SPEC 0 — Minimum Supported Dependencies*. URL: `https://scientific-python.org/specs/spec-0000/`.

[2] M. A. Ainslie and J. G. McColm. “A simplified formula for viscous and chemical absorption in sea water”. *The Journal of the Acoustical Society of America*, vol. 103, no. 3, 1998, pp. 1671–1672.

[3] D. W. Hawkins and P. T. Gough. “Temporal Doppler effects in SAS”. *Proceedings of the Institute of Acoustics: Sonar Signal Processing*, vol. 26, no. 5, 2004.

[4] B. Thomas, C. Sanford, and A. J. Hunter. “Occlusion Modeling for Coherent Echo Data Simulation: A Comparison Between Ray-Tracing and Convex-Hull Occlusion Methods”. *IEEE Journal of Oceanic Engineering*, vol. 49, no. 3, 2024, pp. 944–962.

[5] C. J. Sanford, B. W. Thomas, and A. J. Hunter. “Fourier-Domain Wavefield Rendering for Rapid Simulation of Synthetic Aperture Sonar Data”. *IEEE Journal of Oceanic Engineering*, vol. 49, no. 4, 2024, pp. 1501–1515.

[6] Dask. *FAQ - Dask documentation*. URL: `https://docs.dask.org/en/stable/faq.html` (visited on 05/27/2025).