Supporting Information - 3 - Software Setup: *Complete Simulation of timsTOF PASEF Raw Datasets with Timsim Enables Precise Benchmarking of False Discovery and Phosphosite Localization Error Rates*

David Teschner[1,2], Zixuan Xiao[3], Tim Maier[1,2], David Gomez-Zepeda[4,5], Mateusz K. Łącki[6], Michał P. Startek[7], Ute Distler[6], Tanja Ziesmann[6], Mathias Wilhelm[3,8], Andreas Hildebrandt[1,2], and Stefan Tenzer[4,5,6]

[1]Institute of Computer Science, Johannes Gutenberg University, 55128 Mainz, Germany
[2]Institute for Quantitative and Computational Biosciences (IQCB), Johannes Gutenberg University, 55128 Mainz, Germany
[3]Computational Mass Spectrometry, School of Life Sciences, Technical University of Munich, Freising 85354, Germany
[4]Helmholtz Institute for Translational Oncology, Mainz, Germany
[5]German Cancer Research Center (DKFZ), 69120 Heidelberg, Germany
[6]University Medical Center, Johannes Gutenberg University, 55131 Mainz, Germany
[7]Department of Mathematics, Informatics, and Mechanics, University of Warsaw, 02-097 Warsaw, Poland
[8]Munich Data Science Institute (MDSI), Technical University of Munich, Garching 85748, Germany

March 3, 2026

1 Installation

1.1 Installing `timsim` via pip

We recommend using a Linux distribution, preferably Ubuntu $\geqq$ 22.04, for optimal compatibility and performance. The `timsim` tool is provided by the `imspy-simulation` package, which can be installed directly from the Python Package Index (PyPI). This installation method is recommended for most users and allows for quick setup in a Python virtual environment (Python 3.11 required).

```
pip install imspy-simulation
```

This pulls in all required dependencies, including `imspy-core` and `imspy-predictors`. The deep-learning models use PyTorch, which ships with CUDA support by default. If a CPU-only installation is preferred, install the CPU-only variant of PyTorch before installing the package (see `https://pytorch.org/get-started/locally/`).

1.2 Installing `timsim` via Docker

We provide a pre-built Docker image on the GitHub Container Registry (GHCR) for reproducible and isolated deployment. The image includes the full `rustims` stack with CUDA support, all Python packages, and pre-cached pretrained models. To pull and run the image:

```
docker pull ghcr.io/thegreatherrlebert/rustims:latest
docker run --rm --gpus all ghcr.io/thegreatherrlebert/rustims:latest \
    timsim --help
```

To mount a local data directory into the container for running simulations:

```
docker run --rm --gpus all \
    -v /path/to/data:/workspace \
    ghcr.io/thegreatherrlebert/rustims:latest \
    timsim /workspace/sim-config.toml
```

The Dockerfile is maintained in the main repository[1] and images are automatically built and published on each release.

1.3 Get `timsim` by building from source

Advanced users or developers may wish to build the entire `rustims` stack from source. This allows for customization and integration of the latest features not yet published on PyPI. The process involves building both the Rust backend and Python frontend components.

1. Build Rust Backend

Ensure you have a working Rust installation (https://www.rust-lang.org/tools/install). Then build the Rust crates:

```
cd rustims/mscore && cargo build --release
cd ../rustdf && cargo build --release
```

2. Build Python Bindings (`imspy_connector`)

The Rust-Python bindings are built using `maturin`. Install it via pip:

```
pip install maturin[patchelf]
```

Then build the wheel from the `imspy_connector` directory:

[1] `https://github.com/theGreatHerrLebert/rustims`

```
cd rustims/imspy_connector
maturin build --release
```

Install the generated wheel:

```
pip install --force-reinstall ./target/wheels/[YOUR_WHEEL].whl
```

3. Install Python Packages

Install the modular Python packages from the `packages/` directory:

```
cd rustims/packages
pip install -e ./imspy-core
pip install -e ./imspy-predictors
pip install -e ./imspy-simulation
```

2 Running *timsim*

2.1 Command-Line Interface

`timsim` is most easily used via the command-line interface (CLI), which becomes available after installing the `imspy-simulation` package. Users can explore the available options by running `timsim -help` in a terminal. Although direct invocation with command-line arguments is possible, we strongly recommend using TOML-based configuration files. These files simplify complex simulation setups and ensure reproducibility by clearly documenting all parameter choices: `timsim -config /path/to/sim-config.toml`. Example configurations are bundled with the package and can also be found at GitHub[2]. Real timsTOF Blank samples for dia-PASEF, dda-PASEF, and thunder-PASEF, can be found at Zenodo [3].

2.2 Graphical User Interface Mode

`timsim` also supports a Graphical User Interface (GUI) mode, designed to make the tool more accessible to researchers with limited programming experience. The GUI is included in the `imspy-simulation` package. After installing `imspy-simulation`, ideally within a dedicated Python virtual environment, the GUI can be launched by activating the environment and executing `timsim_gui` in a terminal.

As shown in Figure 1, the GUI provides full access to all simulation parameters available in the configuration file. Users can create, modify, and save configuration files directly within the interface. These files can be reused for command-line simulations or reloaded into the GUI for further editing. Simulations can also be run directly from the GUI by clicking the `Run` button, which executes the underlying script and displays progress in the built-in console.

2.3 Jupyter Notebook Interactive Mode

For advanced users who wish to gain deeper insight into the simulation process or customize specific components, `timsim` can also be used in interactive Jupyter notebooks. In this mode, the CLI tool is broken down into

[2] `https://github.com/theGreatHerrLebert/rustims/tree/main/packages/imspy-simulation/src/imspy\_simulation/timsim/configs`

[3] `https://zenodo.org/uploads/15740932`

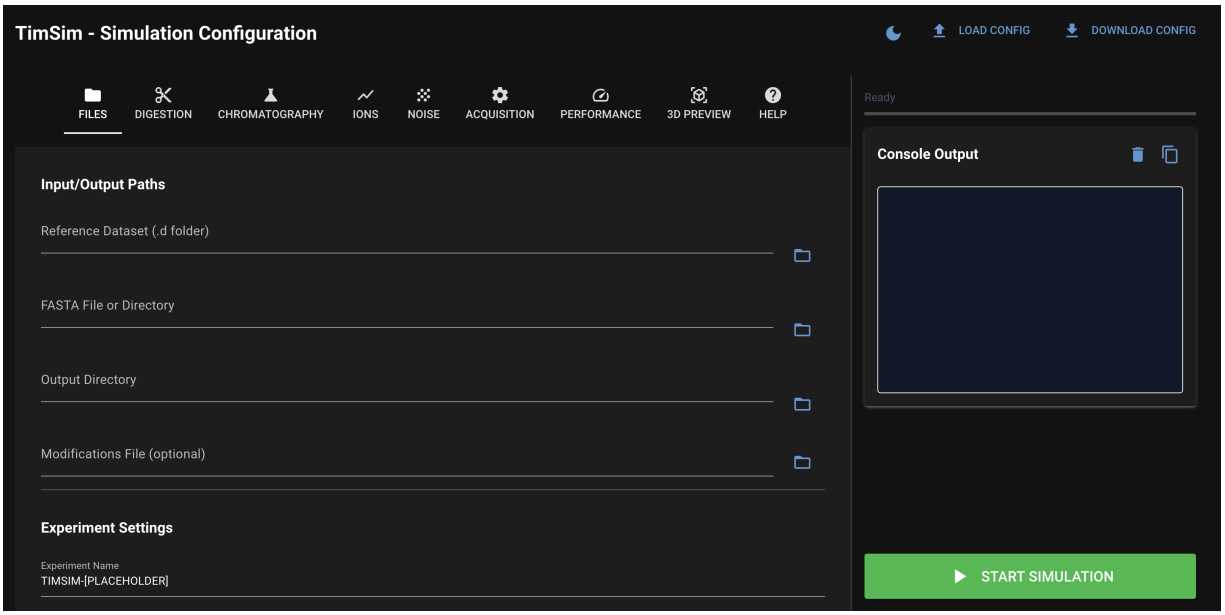

Figure 1: **The `timsim` simulation tool in GUI mode.** Users can interactively modify all parameters available in a `timsim` configuration file, load or save configurations, and choose to run simulations either via the command line or directly within the GUI. Simulation progress is displayed in an integrated console. An exhaustive explanation of all parameters is available in the *Documentation* tab.

its core building blocks, enabling users to explore, modify, and execute each step of the simulation pipeline in a controlled, transparent manner. The `imspy-simulation` package exposes all building blocks as importable Python modules, allowing users to compose custom simulation workflows. These are intended for users who want partial or full control over the simulation workflow, and serve as a starting point for custom simulations, development of new features, or integration with other tools in the computational proteomics ecosystem.

2.4 Integration Testing Mode

To facilitate systematic validation of both the simulator and third-party analysis tools, `timsim` provides an automated integration testing pipeline consisting of two command-line utilities. The first, `timsim-integration-sim`, generates synthetic timsTOF datasets from predefined test configurations, producing raw `.d` files together with a ground-truth SQLite database that records all simulated peptides, their retention times, ion mobilities, charge states, and fragment intensities. The second, `timsim-integration-eval`, executes one or more analysis tools on the simulated data and validates their output against the known ground truth.

The evaluation phase currently supports three external tools: DIA-NN, FragPipe, and Sage. Each tool is executed with standardized parameters appropriate for the test scenario, and its reported identifications are matched back to the simulation database. For each tool, the pipeline computes a set of quantitative metrics including the identification rate (fraction of simulated peptides recovered), the Pearson correlation between predicted and observed retention times, and the Pearson correlation between predicted and observed ion mobilities. For mixed-species experiments, per-species abundance ratios are compared against the known dilution factors to assess quantification accuracy. For phosphoproteomics experiments, the fraction of correctly localized modification sites is evaluated.

Because the supported analysis tools are subject to their own licensing terms, they are not bundled with

`timsim` and must be installed independently by the user. The pipeline locates each tool through a TOML-based environment configuration file that specifies absolute paths to the respective executables and any required auxiliary resources. For DIA-NN, the user provides the path to the `diann` binary. For FragPipe, the configuration includes the path to the `fragpipe` executable, the accompanying tools folder, workflow definition files for each acquisition and modification type (e.g., separate workflows for dia-PASEF, dda-PASEF, and their phosphoproteomics variants), and optionally the paths to a Python interpreter and a DIA-NN binary used internally by FragPipe. For Sage, the user may specify the binary path explicitly or rely on automatic discovery from common installation locations. The environment file additionally specifies paths to reference datasets, FASTA files (including pre-generated decoy databases where required by the tool), and performance settings such as the number of threads and per-tool timeout limits. Each tool is invoked as an external subprocess with its tool-specific command-line interface, and the pipeline captures both standard output and error streams for logging and debugging purposes.

A collection of predefined test scenarios is provided as TOML configuration files, covering the major acquisition modes and sample types supported by the simulator. These include dia-PASEF and dda-PASEF HeLa experiments, a mixed human–yeast–*E. coli* (HYE) quantification scenario with paired conditions at different dilution ratios, a phosphoproteomics scenario with paired datasets varying in modification site placement, and an HLA class I immunopeptidomics scenario using thunder-dda-PASEF acquisition. Each test configuration specifies its own pass/fail thresholds (e.g., minimum identification rate of 0.18–0.28 depending on the scenario, minimum retention time and ion mobility correlations of 0.90–0.95, maximum species ratio error of 0.20, and minimum PTM site localization accuracy of 0.80).

Upon completion, the pipeline generates per-test validation reports comprising detailed JSON metric files and visual HTML reports with diagnostic plots. A top-level summary report aggregates the pass/fail status across all tests and tools, including tool version information and elapsed run times. The complete output is bundled into a timestamped archive for record-keeping. This infrastructure serves three primary use cases: regression testing of the simulator during development, validation of new versions of third-party analysis tools against a perfectly defined ground truth, and providing software developers with a controlled test space where the true positive set is known exactly.

2.5 Simulation Parameters

The table below provides an exhaustive collection of all settable hyperparameters that can be set to alter the behaviour of the `timsim` simulation pipeline.

Parameter	**Description**
Main Settings	
`save_path`	Path where the simulated dataset will be saved.
`reference_path`	Path to a real timsTOF dataset (optimally a blank sample) to call to translation functions, extract window layout (dia), and optionally add noise peaks.
`fasta_path`	Input FASTA file used for peptide generation.
`experiment_name`	Label for the experiment, saved into metadata.
`acquisition_type`	selects acquisition mode (dda, dia, synchro, slice, midia).

use_reference_layout	If true, use quadrupole selection scheme from reference.
reference_in_memory	Load reference data into memory instead of reading from disk.
sample_peptides	If true, sample a fixed number of peptides from the proteome.
add_decoys	If true, add decoy peptides to the dataset.
proteome_mix	If true, simulate mixed-proteome sample (expects A, B dilution scheme per proteome).
silent_mode	Disable logging to console.
from_existing	Resume simulation from a previously stored SQLite blueprint, potentially with a different acquisition scheme.
existing_path	Path to existing simulation blueprint database.
apply_fragmentation	If false, only simulate MS1 scans (no fragments).
Peptide Digestion	
num_sample_peptides	Number of peptides to simulate (if sampling).
missed_cleavages	Number of allowed missed cleavages in digestion.
min_len, max_len	Minimum and maximum peptide length.
cleave_at	Cleavage residues for protease (e.g., KR for trypsin).
restrict	Restriction at cleavage site (e.g., not before proline).
Peptide Intensity	
intensity_mean, intensity_min, intensity_max	Parameters for log-intensity sampling.
sample_occurrences	If true, sample peptide occurrences from a distribution.
intensity_value	Fixed value if not sampling.
Isotopic Pattern	
isotope_k	Number of isotopologues to simulate.
isotope_min_intensity	Minimum relative intensity for an isotopologue to be included.
isotope_centroid	If true, use centroided (not profile) peaks.
LC & IM Distribution Settings	
gradient_length	Total LC gradient length in seconds.
sigma_alpha_rt, sigma_beta_rt	Beta distribution parameters for elution peak width.
k_lower_rt, k_upper_rt, k_alpha_rt, k_beta_rt	Beta parameters for EMG skew.
z_score, target_p	Control how much of the EMG distribution is kept.
sampling_step_size	Step size for evaluating elution profiles.
use_inv_mob_std_mean	If true, set fixed mobility standard deviation.
inv_mob_std	Mean inverse mobility standard deviation (if used).

Phosphorylation Settings	
phospho_mode	If true, create phosphopeptides in simulation (expects A, B dataset generation from common blueprint).
Noise Settings	
add_noise_to_signals	Add noise to theoretical peak positions.
mz_noise_precursor, fragment	Enable m/z noise for precursor or fragment ions.
precursor_noise_ppm, fragment_noise_ppm	Magnitude of noise in ppm.
mz_noise_uniform	Use uniform noise instead of Gaussian.
add_real_data_noise	Mix in real noise peaks from reference file.
reference_noise_intens	Cap on intensity of reference peaks.
down_sample_factor	Down sample fragment peaks by this factor, take probability of a peak is inversely proportional to its expected intensity contribution.
Charge State Probabilities	
p_charge	Base protonation probability for binomial model.
min_charge_contrib	Minimum contribution from each charge state.
binomial_charge_model	If true, binomial charge state predictor will be used instead of deep charge state predictor.
DDA Settings	
precursors_every	How often to select new precursors (in frames).
max_precursors	Maximum number of precursors per MS2 frame.
exclusion_width	Number of frames a selected precursor should not be re-targeted during on-the-fly selection of ions.
precursor_intens_thresh	Minimum intensity a precursor needs to have to be considered for selection.
selection_mode	Precursor selection algorithm (e.g., `topN`, `random`).
Property Variation Settings	
rt_variation_std	RT noise to be applied to apex values of the retention time distribution during re-acquisition of a simulation blueprint (standard deviation in seconds).
inv_mob_variation_std	Ion mobility noise to be applied to the apex of the inverse ion mobility distribution during re-acquisition of a simulation blueprint (standard deviation).
intensity_variation_std	Log-intensity noise (e.g., 0.02) to be applied to the occurrence of a peptide during re-acquisition of a simulation blueprint (standard deviation).

Performance Settings	
`num_threads`	Number of CPU threads to use. Set to -1 for all.
`batch_size`	Number of frames to be build in parallel during raw data generation.

Table 1: Detailed explanation of all simulation parameters used for configuring `timsim`.