SoftwareX 36 (2026) 103163

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

VIPR: Versatile inverse problem software framework

Jeyhun Rustamov[a], Sascha Creutzburg[b], Jens Bornschein[b], Michael Bussmann[c,a], Marina Ganeva[d], Alexander Gerlach[e], Stefan Häusler[d], Bernd Helm[b], Alexander Hinderhofer[e], Robert Juzak[b], Alexandros Koutsioubas[d], Rico Löser[g], Valentin Munteanu[e], Vedhas Pandit[a], Ulrich Schramm[a,f], Frank Schreiber[e], Nico Mothes[b], Jeffrey Kelling[a,h,*]

[a] Institut für Strahlenphysik, Helmholtz-Zentrum Dresden-Rossendorf, Bautzner Landstraße 400, 01328 Dresden, Germany
[b] Helm & Walter IT-Solutions GmbH, Einsteinstraße 6, 01069 Dresden, Germany
[c] Center for Advanced Systems Understanding (CASUS), Conrad-Schiedt-Straße 20, 02826 Görlitz, Germany
[d] Jülich Centre for Neutron Science (JCNS) at Heinz Maier-Leibnitz Zentrum (MLZ), Forschungszentrum Jülich GmbH, Lichtenbergstraße 1, 85748 Garching, Germany
[e] Institut für Angewandte Physik, Universität Tübingen, Auf der Morgenstelle 10, 72076 Tübingen, Germany
[f] Technische Universität Dresden, 01062 Dresden, Germany
[g] Fraunhofer Institute for Machine Tools and Forming Technology IWU, Reichenhainer Straße 88, 09126 Chemnitz, Germany
[h] Fakultät für Naturwissenschaften, Technische Universität Chemnitz, Straße der Nationen 62, 09111 Chemnitz, Germany

ARTICLE INFO

Keywords:
Inverse problems
Machine learning
Reflectometry
Plugin architecture
Reproducible workflows

ABSTRACT

VIPR (Versatile Inverse Problem Software Framework) is a plugin-based software framework for building reproducible machine-learning workflows for scientific inverse problems. These problems arise when researchers need to reconstruct physical properties of a sample from incomplete or indirect measurements, a common challenge in X-ray and neutron scattering experiments where only intensity measurements are available and phase information is lost. VIPR addresses such ill-posed reconstruction tasks where direct inversion is often impossible. VIPR implements a modular microkernel architecture with domain-specific plugins to produce configurable machine learning workflows, including both deterministic and probabilistic models. Workflows are defined via declarative YAML configurations and can be executed through a command-line interface or a containerized web application. For a given experimental dataset, VIPR produces standardized analysis artifacts, including visualizations and statistical summaries. This article demonstrates VIPR for X-ray and neutron reflectometry and for reconstructing PEMFC operating parameters from sensor data. Both applications use domain-specific plugins with the same core inference workflow.

Metadata

Code metadata.

Nr.	Code metadata description	Metadata
C1	Current code version	v1.0.0
C2	Permanent link to code/repository used for this code version	https://github.com/Photon-AI-Research/vipr_demonstrator
C3	Permanent link to Reproducible Capsule	https://doi.org/10.14278/rodare.4633
C4	Legal Code License	GNU Lesser General Public License v3.0 or later (LGPL-3.0-or-later)
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Python 3.10+, PyTorch, Pydantic 2, beartype, FastAPI, Celery, Redis, Nuxt.js, Vue 3, TypeScript, Vuetify, Docker
C7	Compilation requirements, operating environments & dependencies	Requires Docker and Docker Compose for deployment. Python 3.10+ for local CLI usage.
C8	If available Link to developer documentation/manual	https://vipr-docs.pages.dev
C9	Support email for questions	vipr@helmundwalter.de

* Corresponding author at: Institut für Strahlenphysik, Helmholtz-Zentrum Dresden-Rossendorf, Bautzner Landstraße 400, 01328 Dresden, Germany.
Email address: j.kelling@hzdr.de (J. Kelling).

https://doi.org/10.1016/j.softx.2026.103163
Received 20 July 2026; Received in revised form 16 October 2026; Accepted 19 October 2026
Available online 28 October 2026
2352-7110/© 2026 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).
Open access funding enabled and organized by Projekt DEAL.

1. Motivation and significance

Inverse problems arise in many experimental techniques in materials science and particle physics. For instance, in scattering experiments, such as neutron and X-ray reflectometry, the measured intensity corresponds to the squared magnitude of a complex wavefield. As a consequence of this measurement principle, phase information is lost and direct inversion of the measured intensity data is generally not possible without additional assumptions, as multiple parameters can lead to an identical intensity measurement. This phase problem [1] is a prime example making the mapping from the measurement space to the parameter space ambiguous and thereby the problem of recovering underlying physical parameters from the measurement non-convex and ill-posed. Similarly limited detector resolution or detector noise can also lead to a loss of information which can also cause an otherwise invertible problem to become ill-posed.

Manual or iterative fitting-based reconstructions can be applied to solve these problems given careful and time-consuming initialization by domain experts. Bayesian inference [2] can address this ambiguity by combining the data with prior knowledge to estimate a posterior distribution over the parameters rather than a single point estimate, making the analyses even more computationally demanding. Because of this, either approach is often impractical in large-scale research facilities for in-situ analysis where high-throughput data streams can be produced by detectors. Even in lower-throughput settings, timing is critical when experimental decisions need to be made during limited beamtime.

Machine learning-based inference that can deliver rapid analysis and uncertainty quantification during measurements would be especially valuable in such scenarios.

There are already open-source software packages that serve as powerful tools for model-based reflectometry analysis and partially address these challenges. *GenX* [3] employs the differential evolution algorithm as a robust optimization method to avoid local minima when fitting X-ray and neutron reflectivity data. *Refl1D* [4] offers fitting via a Bayesian uncertainty analysis framework, *Bumps* [5], which includes Markov chain Monte Carlo (MCMC) sampling to quantify parameter uncertainty and correlations. It has also been extended with a web-based interface to improve accessibility [6]. *refnx* [7] is another Python package that offers fitting methods based on least-squares and a Bayesian approach using the MCMC algorithm. It has also been extended to related techniques such as ellipsometry through *refellips* [8]. BornAgain [9] provides an open-source framework for simulating and fitting X-ray and neutron reflectometry, off-specular scattering, and grazing-incidence small-angle scattering data using analytical models and a range of numerical optimization methods, including stochastic global optimizers such as simulated annealing and genetic algorithms. Similarly, *anaklasis* [10] provides a Python-based software package for specular neutron and X-ray reflectivity modeling, with a particular focus on parameter-uncertainty quantification via bootstrap analysis and MCMC sampling. While these tools are widely used and highly effective, the reconstruction process remains iterative and computationally expensive for large parameter spaces. MCMC-based approaches can provide uncertainty estimates, but they are often not well suited for rapid analysis in high-throughput settings. This has motivated growing interest in generative approaches for inverse uncertainty quantification [11].

Recently, machine-learning-based approaches have also been applied to X-ray and neutron reflectivity [12–16]. However, such solutions often rely on project-specific code and analysis pipelines that are difficult to adapt, compare and reproduce across research facilities. In practice, applying a different method to the same experimental dataset would require substantial manual adaptation of complex pipelines. As a result, simply sharing trained models is not sufficient, as it does not capture important steps such as preprocessing transformations needed for accurate parameter estimation. More broadly, recent scientific software has increasingly emphasized modular, configuration-driven, and reproducible workflows for computational research [17,18], highlighting the need for reusable and interoperable analysis. Existing machine-learning solutions are typically published as software attached to an individual study and are tailored to that specific application; some achieve a higher degree of reuse within their domain, such as *Reflectorch* [19], but, to our knowledge, none provides a cross-domain layer that integrates different models behind a common interface, both within data pipelines and through a shared user interface. Rather than contributing a new machine-learning model, VIPR addresses this gap by providing such a platform.

VIPR is a modular, plugin-based framework for data-driven inverse problem workflows that provides a standardized interface for reproducible machine-learning inference pipelines.

VIPR's key contributions are:

- **Workflow interchangeability:** VIPR enables complete inference workflows (e.g., data loading, preprocessing, model execution, and postprocessing) to be defined through YAML configuration files. This facilitates the application of different methods to the same dataset without the need to modify the underlying code.
- **Probabilistic inverse modeling:** VIPR integrates probabilistic generative models that enable the prediction of conditional posterior distributions of physical parameters. This enables efficient uncertainty quantification and the detection of multiple solutions.
- **Modular workflow design:** Domain-specific logic, such as data loading, preprocessing, and model loading, can be encapsulated in plugins. The core framework, on the other hand, handles orchestration, configuration management, and deployment.
- **Reproducibility:** Entire workflows are defined via YAML configuration files and can be executed via a command-line interface or a web application. This ensures consistency across different computing environments.
- **Extensibility:** VIPR allows for straightforward incorporation of new experimental domains via a registry-based plugin mechanism.

The remainder of this article is structured as follows. Section 2 describes the VIPR software architecture and its functionality. Section 3 presents the use of VIPR with illustrative examples. Section 4 discusses the practical research impact of VIPR on scientific inverse problems. Finally, Section 5 provides the conclusion and outlook for future work.

2. Software description

VIPR is a modular, plugin-based framework for deep learning inference on scientific inverse problems. While originally developed for reflectometry, its architecture is domain-agnostic, separating the core orchestration logic from domain-specific physical models and data handling.

2.1. Software architecture

VIPR is built upon the Cement application framework [20] and NF4IP [21]. It follows a layered microkernel architecture (Fig. 1) that is divided into two primary layers:

1. **vipr-core**: Provides the fundamental infrastructure, including the command-line interface (CLI), API generation, configuration management, and the orchestration of the inference pipeline. It provides abstract handler base classes (data loader, model loader, predictor) and pipeline orchestration but contains no domain logic.
2. **Domain Plugins**: External Python packages (e.g., `vipr-reflectometry`) that register domain-specific handlers, filters, and hooks with the core.

The framework uses three primary mechanisms to ensure extensibility without modifying the core codebase:

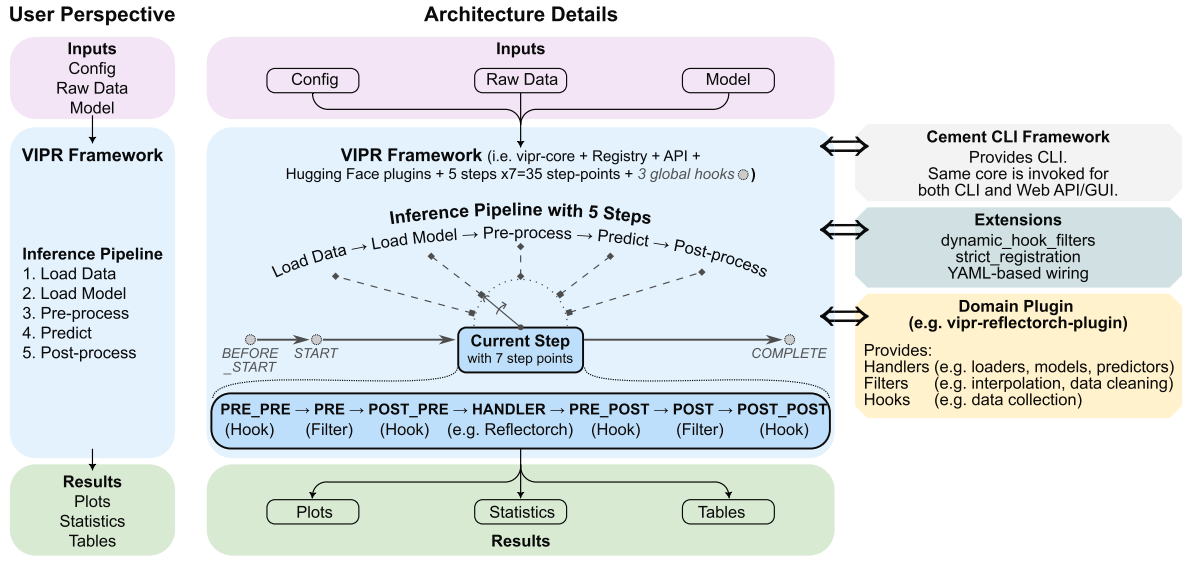


Fig. 1. Pictorial overview of the VIPR framework. Domain plugins register handlers, filters, and hooks at the core's extension points to define the scientific workflow.

- **Handlers** extend handler base classes (e.g., `DataLoaderHandler`, `ModelLoaderHandler`, `PredictorHandler`) to implement domain-specific data loading, model loading, and prediction.
- **Filters** are chainable functions that transform data passing through the pipeline (e.g., interpolation, normalization). They are executed in a deterministic, weight-sorted order.
- **Hooks** are callbacks used for non-transforming tasks such as logging, validation, or real-time visualization.

The `strict_registration` extension enforces fail-fast validation by rejecting registrations to undefined hook/filter namespaces. The `dynamic_hooks_filters` extension provides YAML-based wiring and validates configured callbacks and parameters against a global registry.

VIPR includes a built-in HuggingFace plugin that resolves repository definitions registered by domain plugins and synchronizes required model artifacts (configuration files and weights) into local model directories. When running the API service, this synchronization is automatically triggered during startup.

2.2. The inference pipeline

The core of VIPR is a generic, five-step inference pipeline designed to support reproducible scientific workflows. The pipeline ensures that data flows through a predictable sequence of operations:

1. **Load Data:** A handler reads experimental files (e.g., CSV, HDF5) and standardizes them into a unified `DataSet` structure containing input given as a vector of arguments (x), a mapped vector of values/results (y) optionally with uncertainties (dx, dy).
2. **Load Model:** A handler loads a pre-trained model and prepares it for inference on the configured device (CPU/GPU).
3. **Preprocess:** A chain of filters transforms the `DataSet` for model consumption. Filters are executed in deterministic weight-sorted order and configured via YAML, enabling operations such as normalization, data cleaning, outlier removal, or interpolation to the model's input grid.
4. **Predict:** The model generates predictions based on the preprocessed input.
5. **Postprocess:** Results are formatted, and hooks are triggered to generate and collect output artifacts such as plots and tables, which are persisted upon pipeline completion.

The unified `DataSet` container that flows through the pipeline is defined as:

```
class DataSet:
    x: np.ndarray              # Input arguments (batch_size, ...)
    y: Optional[np.ndarray]    # Mapped values/results (batch_size, ...)
    dx: Optional[np.ndarray]   # Argument uncertainties
    dy: Optional[np.ndarray]   # Value uncertainties
    metadata: dict             # Loader metadata, file info, etc.
```

The `DataSet` design abstracts over array data with arbitrary shapes. It supports both single and multiple independent samples in a unified container. Multiple samples are stacked along the first array axis (batch dimension), while a single sample is represented with a leading dimension of size one. For one-dimensional measurements (e.g., spectral curves, time series), the resulting array shapes are `(n_samples, n_points)`. The design extends naturally to higher-dimensional data (e.g., images, 3D volumes) with the same batch-first convention.

Each pipeline step exposes seven extension points arranged symmetrically around the handler execution in the center (as detailed in the lower panel of Fig. 1): two filter chains (pre- and post-execution) each bracketed by a pair of hooks. Combined with three workflow-level hooks (`INFERENCE_BEFORE_START_HOOK`, `INFERENCE_START_HOOK`, and `INFERENCE_COMPLETE_HOOK`), this yields a total of 38 extension points across all five pipeline steps ($5 \times 7 + 3$) that plugins use to insert domain-specific behavior without modifying the pipeline structure. Plugins write result artifacts (images, tables, diagrams) into a shared `DataCollector` at configured hook points throughout the pipeline; on the `INFERENCE_COMPLETE_HOOK`, the accumulated data is serialized to a UUID-addressed result directory for frontend consumption or offline inspection.

Implementing a domain plugin. A domain plugin is distributed as an installable Python package and registered through the `vipr.plugins` entry point. Its `load(app)` function registers the components required by the inference workflow. A `DataLoaderHandler` reads the domain-specific measurement format and maps the measurements to the common `DataSet` representation. A `ModelLoaderHandler` loads the configured inference model. A `PredictorHandler` receives the preprocessed `DataSet` and the loaded model and returns the raw model output. Filters transform the `DataSet`. At configured hook points, callbacks use the shared `DataCollector` API to record result tables, diagrams, and images. The registered components are selected and configured through YAML. For one data format and one inference method, the framework-specific integration comprises a data loader, a model loader, a predictor, filters

for the required data transformations, callbacks for result generation, and a YAML configuration. VIPR plugins implement domain-specific inference workflows. The development, training, and scientific validation of the underlying models are outside the scope of the framework. Appendix C provides a high-level implementation walkthrough and quantifies the scope of the reflectometry plugin.

2.3. Domain application: reflectometry

To demonstrate domain adaptation, we implemented the external `vipr-reflectometry` package. It maps the five-step workflow to X-ray and neutron reflectometry by registering domain-specific handlers, filters, and hooks.

For data loading, the plugin provides `csv_spectrareader` (column-based text files including ORSO-compatible formats [22]) and `hdf5_spectrareader` (HDF5 [23] for batch processing). An optional preprocessing filter removes non-positive intensities and, when uncertainties are available, can discard high-error points or truncate high-error regions. For Reflectorch, a separate filter interpolates the measurements onto the model's Q-grid. Three model families address different analysis needs. Reflectorch [19,24,25] produces deterministic point estimates from a neural network that receives the reflectivity curve together with user-defined prior bounds; predictions can optionally be polished via least-squares optimization. Normalizing-flow models [26–29] generate posterior samples from learned conditional distributions, enabling uncertainty quantification and multimodal solution detection. PANPE [30] additionally accepts user-specified prior bounds as network input at inference time, so that a single trained model covers varying experimental constraints without retraining. All handler selections and parameters are declared in YAML, enabling users to switch between deterministic and probabilistic workflows without code changes.

2.4. Domain application: system-state inference from sensor data

To demonstrate VIPR on an inverse problem beyond scattering, we applied it to simulated sensor data from a polymer electrolyte membrane fuel cell (PEMFC). A conditional invertible neural network (cINN) [31] infers probability distributions for PEMFC operating parameters from measurable or derived quantities [32].

The external `vipr-fuel-cell` plugin [33] maps this problem to the generic inference workflow described in Section 2.2 without requiring modifications to `vipr-core`. It registers domain-specific handlers for sensor-data ingestion, cINN model loading, and posterior sampling, together with a preprocessing filter and a result-collection hook. Preprocessing selects and scales the sensor quantities that serve as conditioning variables for the cINN. During prediction, samples drawn in the latent space are mapped through the inverse cINN and transformed back to physical parameter units, from which posterior summary statistics are calculated. During postprocessing, the result-collection hook writes the results to VIPR's shared `DataCollector`. These results include parameter summary tables and time series of posterior means. For selected sensor states, the hook also records marginal posterior distributions.

2.5. Deployment and web interface

In addition to the CLI, VIPR includes a containerized web application for interactive analysis. The deployment architecture consists of four services:

- **Frontend (Nuxt.js):** A reactive web UI that allows users to configure pipelines visually and interact with results.
- **API Backend (FastAPI):** Exposes the REST API and dispatches inference jobs to Celery workers.
- **Redis:** Message broker and result backend for asynchronous task execution.
- **Worker Pool (Celery):** Executes the inference pipeline asynchronously. This decoupling ensures that long-running inference tasks—such as generating thousands of posterior samples—do not block the user interface.

The API backend and the Celery worker share a persistent storage volume. A Docker Compose configuration for deploying the full stack alongside the source files of VIPR's developer documentation is maintained as the overarching `vipr-framework` package. A schematic overview of the containerized deployment is provided in Fig. A.7.

Redis queues concurrent inference requests, which a Celery worker processes sequentially (`concurrency=1`). Each task uses a unique identifier for its result directory.

Software quality. Automated tests for `vipr-core`, `vipr-api`, `vipr-frontend`, and `vipr-reflectometry` are run in their GitLab CI pipelines. Across these components, the test suites include unit, integration, and browser-based end-to-end tests. Within `vipr-core`, `beartype` checks annotated functions at runtime, while component parameters supplied through YAML are validated against their Pydantic schemas. Validation and cleaning of measurement data are domain-specific. The reflectometry plugin accepts column-based `.csv`, `.dat`, and `.txt` files and HDF5 files with `.h5` or `.hdf5` extensions. Its data loaders raise explicit errors if no input path is provided, the file does not exist, or another file extension is used. Measurement uncertainties are optional. Compatibility with `vipr-core` is managed through package dependency constraints. The developer documentation covers installation, configuration, the extension interfaces, and the implementation of domain plugins.

3. Illustrative examples

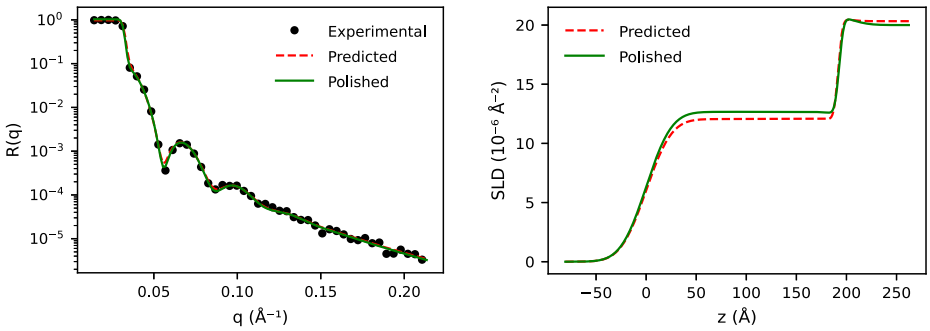
This section illustrates how VIPR can be applied to reflectometry analysis using both deterministic and probabilistic deep learning models. We first demonstrate the basic usage of VIPR with a deterministic workflow on X-ray reflectometry data in Section 3.1. We then present a probabilistic case study on representative experimental neutron reflectometry data from the MARIA reflectometer at the Heinz Maier-Leibnitz Zentrum (MLZ) [34] in Section 3.2. Next, we demonstrate how results from different inference runs can be compared within a unified workflow in Section 3.3. Finally, Section 3.4 presents an inverse problem from another scientific domain: reconstructing PEMFC operating parameters under steady-state conditions from simulated sensor data.

3.1. Deterministic workflow on experimental x-ray reflectometry data

As a first example, we consider experimental X-ray reflectivity data shown in Fig. 2(a). This measurement represents a two-layer film, where the substrate is silicon with a thin silicon oxide layer on top, followed by a perylene diimide (PTCDI-C3) layer. Listing B.1 shows an example YAML configuration specifying the workflow used here on this dataset. From a user's perspective, this configuration file drives the whole workflow for SLD profile prediction, by specifying all components of the inference pipeline, including data loading, preprocessing filters, model loading, prediction, and postprocessing. In order to change the pipeline, there is no need to make any code changes; only the swapping of components and modification of parameters in the configuration file as will be discussed in Section 3.2. This allows users to easily customize the workflow for different datasets, models, and analysis requirements. The specified workflow can be executed using the CLI command:

```
vipr --config <config.yaml> inference run
```

In this example configuration file (cf. Listing B.1), the `csv_spectrareader` handler reads reflectivity data from a text file and assigns the corresponding columns to the reflectivity intensity and momentum transfer variables. The preprocessing filter `preprocess_interpolate` interpolates the experimental curve to the specific Q-spacing required by the neural network. Then, the model handler loads a pre-trained Reflectorch model [19] for point estimation specified by `config_name` for reflectometry analysis.



(a) Experimental reflectivity curve with predicted and polished reflectivity curves.

(b) SLD profiles for predicted and polished parameters.

Fig. 2. Reflectivity analysis results for the PTCDI-C3 thin-film sample.

In the prediction stage, the `reflectorch_predictor` handler performs neural network inference to predict the physical parameters and compute the reflectivity curves and SLD profiles. The `prior_bounds` parameter allows researchers to input prior knowledge into the model during inference. This includes knowledge about parameter bounds as well as instrument correction parameters, such as a misalignment-induced offset in momentum transfer and a multiplicative intensity scaling factor applied to the reflectivity curve. Inclusion of prior knowledge helps the model eliminate potential ambiguities and produce a single estimate of the parameters. Inferred parameters can optionally initialize a nonlinear least-squares fit in which the reflectivity is calculated using the Abelès matrix formalism [19,35]. This *polishing* step improves agreement with the measured curve at small computational cost when model prediction is valid and conversely provides robustness against model prediction failures.

After completing the inference workflow, VIPR creates a directory to store all analysis results in a structured manner. This allows users not only to quantify but also to visualize the inference results. Fig. 2(a) and (b) display the predicted reflectivity curves and corresponding SLD profiles obtained for the PTCDI-C3 thin film dataset, respectively.

As described previously in Section 2.5, VIPR provides users not only with a text-based user interface (CLI) but also with a graphical user interface (GUI) through a containerized web application that follows the same configuration-driven inference pipeline depicted in Fig. 3. The web interface allows users to modify the workflow parameters, run inference, compare stored results, and inspect resulting plots in an interactive manner. It also provides an Auto Execute mode for on-the-fly parameter fitting: inference is re-run automatically whenever the configuration is adjusted in the UI, providing immediate visual feedback. Both the CLI and web application interface support interoperability across research facilities through the simple exchange of configuration files, ensuring reproducible and portable workflows.

3.2. Probabilistic workflow on experimental neutron reflectometry data

The deterministic workflow, described in Section 3.1, incorporates prior knowledge to produce a single estimate of the underlying structure. While this is effective for problems where prior constraints on the parameter bounds are available, it does not provide insight into uncertainty or the presence of multiple solutions.

To address this limitation, VIPR also offers probabilistic deep learning models, such as normalizing flows. These models learn the distribution of parameters instead of predicting a single estimate. Thus, at inference time, we can generate samples from the learned conditional posterior distributions of physical parameters, given the measured reflectivity curve. This enables direct analysis of uncertainty and multimodality.

To demonstrate these capabilities, we analyze neutron reflectometry data measured at the MARIA reflectometer at MLZ. The analyzed dataset corresponds to the spin-down (non-spin-flip) reflectivity curve of a two-layer (Pt/Fe) film at the air/MgO interface, originally presented in the supporting information of the anaklasis software paper [10]. We use a pre-trained normalizing-flow-based model (Neural Spline Flow [29]) with a 1D CNN [37] embedding network for the inference pipeline. Listing B.2 shows the example configuration for this setup.

To explore the multimodality of solutions, a Gaussian Mixture Model (GMM) clustering algorithm with two components is applied, resulting in a clustered corner plot as shown in Fig. 4. GMM clustering of the predicted posterior distribution over the eight physical parameters reveals two well-separated modes. These modes are most clearly separated in the joint distribution of the first- and second-layer thickness parameters (denoted d_{L1} and d_{L2} in Fig. 4). The dominant mode, which accounts for 86% of the samples, corresponds to a structure with a relatively thick first layer and a thinner second layer. On the other hand, the secondary mode (14%) corresponds to the inverse configuration, with a thinner first layer and a thicker second layer. VIPR validates these results by computing reflectivity curves corresponding to the cluster centroids and comparing them with the experimental curve shown in Fig. 5(a). As can be seen from the figure, both parameter combinations produce good agreement with the measured data. The corresponding SLD profiles are shown in Fig. 5(b). The ambiguity arises from the invariance of the measurement process, which loses phase information, under phase shifts in the reflected neutron radiation as a result of which different SLD profiles can produce identical, or very similar, reflectivity curves.

This example illustrates the ill-posed nature of inverse problems in reflectometry. It demonstrates how VIPR facilitates the identification of multiple possible solutions and provides a comprehensive overview of parameter uncertainty. While traditional optimization can recover multiple solutions through systematic exploration with different priors and initializations, this typically requires multiple independent fitting runs. VIPR's probabilistic approach achieves full posterior characterization systematically in a single inference step, making it well-suited for near real-time experimental feedback.

3.3. Comparing results from multiple inference runs

To relate the two posterior modes identified in Section 3.2 to results from deterministic inference, VIPR provides a compare workflow that aggregates results from multiple independent runs into a unified comparison.

The workflow is invoked with:

```
vipr --config <compare_config.yaml> compare run
```

The configuration file specifies which stored results to compare. In the present example (cf. Listing B.5), the comparison aggregates three results: the normalizing-flow result from Section 3.2, from which the two cluster-centroid solutions are extracted, and two Reflectorch-based results obtained under different prior bounds on the first-layer thickness



Fig. 3. Web application interface of the VIPR framework.

(1–50 Å and 1–400 Å, respectively; the second layer was bounded to 1–400 Å in both cases). Listings B.3 and B.4 are the configuration files corresponding to the deterministic inferences. Notice that to change the prior bounds, the user only needs to minimally tweak the configuration file (cf. Listing B.4) and rerun the inference pipeline.

Fig. 6(a) and (b) show the resulting reflectivity curves and SLD profiles. The four polished solutions separate into two distinct sets: a thick first layer with a thinner second layer, and the inverse configuration. When the prior bounds are set broadly (1–400 Å for both layers), Reflectorch's point estimator converges to the dominant solution identified by the normalizing flow (86% posterior mass), which corresponds to a thick first layer. Constraining the first-layer thickness to 1–50 Å forces convergence to the secondary mode (14% posterior mass). Both families achieve comparable fit quality, as indicated by the metrics in Table 1. The choice of prior bounds thus determines which of the two solutions Reflectorch recovers—a degeneracy that the probabilistic model reveals directly. Ultimately, the compare workflow eliminates the need for custom analysis scripts when evaluating different methods. By generating joint plots and metric tables from a single configuration file, VIPR provides a simple way to directly compare how different models and prior constraints impact the final results. The compare workflow is also accessible through the web application, where stored inference results can be selected and compared interactively.

3.4. PEMFC operating-state reconstruction

For a reproducible cross-domain example, VIPR processes a simulated PEMFC operating profile with the cINN developed for Test Case 1 by Löser et al. [32]. It estimates nine operating parameters from eleven measurable or derived input quantities, including voltage losses, maximum current densities, and mass flow rates. Further details, including an example configuration and result, are provided in Appendix E.

The two applications use different data structures. A reflectometry measurement is a curve sampled on a momentum-transfer grid. A PEMFC sensor state contains eleven measured or derived quantities. VIPR represents both data types with the same `DataSet` container and processes them through the same five-step inference pipeline.

4. Impact

VIPR facilitates implementation of new approaches to scientific inverse problems by integrating machine-learning-based inference into reproducible and configurable workflows. The main practical advantage of VIPR is that complete inference workflows can be exchanged declaratively for the same dataset without modifying code. This allows researchers to switch easily between different inference approaches within a unified and reproducible workflow, eliminating the need to write extensive custom code when testing a new algorithm.

The PEMFC example in Section 3.4 further shows that the same inference pipeline can reconstruct operating parameters from sensor data through a separate domain plugin.

In particular, the framework supports the use of probabilistic models that can estimate the full posterior distribution of physical parameters, rather than providing a single-point estimate. This capability enables uncertainty quantification and the identification of multimodal solutions

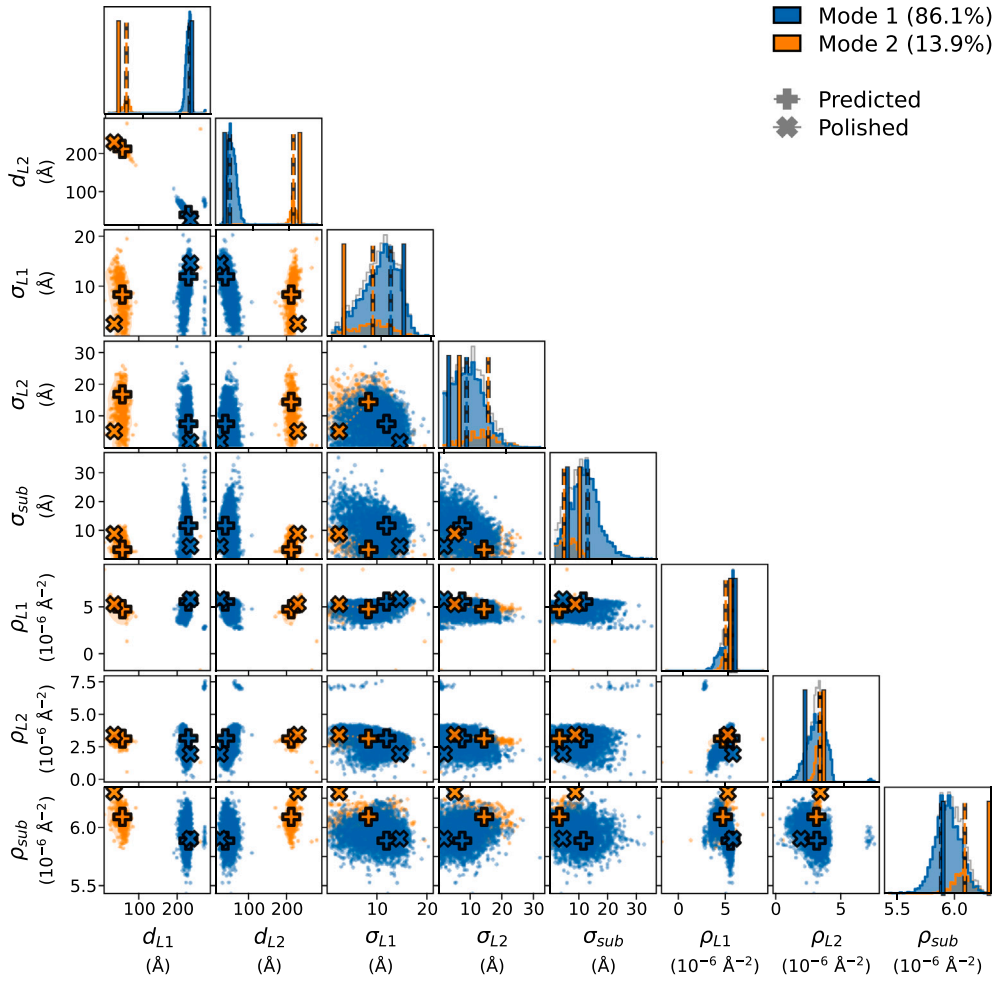
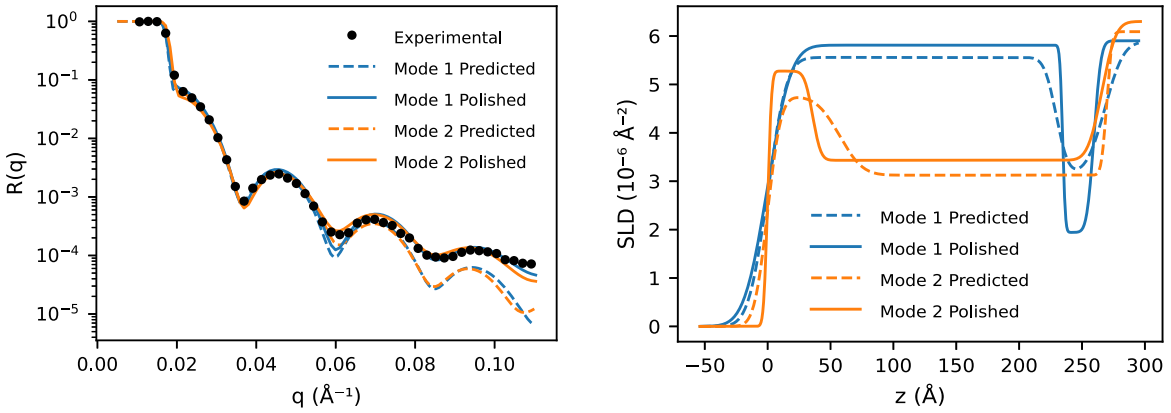


Fig. 4. Clustered corner plot of posterior samples inferred from the experimental neutron reflectivity curve of a Pt/Fe thin-film sample [36].



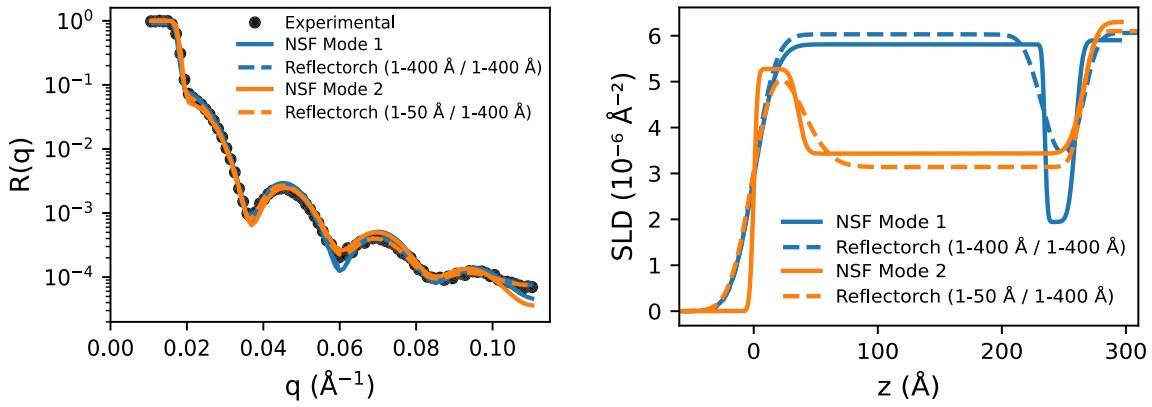
(a) Experimental reflectivity curve with predicted and polished reflectivity curves.

(b) Predicted and polished SLD profiles.

Fig. 5. Model predictions evaluated at the centroids of the two posterior modes for a Pt/Fe thin-film system.

in ill-posed inverse problems, which traditional iterative fitting-based methods lack.

By providing single-shot estimates or posterior samples, VIPR accelerates analysis compared to traditional iterative approaches. With probabilistic generative models, such as normalizing flows, it is possible to recover the full posterior parameter distributions within seconds, whereas traditional MCMC-based approaches would require significant computation time (Section D). Thus, the framework enables rapid inference using pre-trained models, delivering near real-time results. This capability makes VIPR suitable for high-throughput experiments



(a) Reflectivity curves for all four polished solutions together with the experimental data.

(b) SLD profiles corresponding to the four polished solutions.

Fig. 6. Comparison of polished reflectivity curves and SLD profiles for the Pt/Fe dataset. Solid lines: normalizing-flow cluster-centroid solutions (NSF Mode 1 and Mode 2). Dashed lines: Reflectorch solutions obtained under different prior bounds on the first-layer thickness.

Table 1
Fit metrics for the four polished solutions. MAE and RMSE are computed over the 92 measured Q points using the residuals $\log_{10} R_{\text{fit}} - \log_{10} R_{\text{exp}}$.

Solution	d_{L1}/d_{L2} bounds (Å)	MAE	RMSE
NSF Mode 1	–	0.0652	0.0846
NSF Mode 2	–	0.0718	0.0926
Reflectorch point estimator	1–400 / 1–400	0.0335	0.0478
Reflectorch point estimator	1–50 / 1–400	0.0278	0.0350

and in situ measurements at modern X-ray and neutron facilities. Furthermore, VIPR can supply neural networks with additional inputs, allowing researchers to incorporate prior knowledge about the samples and/or the instrument into the model.

These capabilities have been validated on experimental neutron reflectometry data acquired at the MARIA reflectometer at MLZ, as described in Section 3.2. In addition to standalone use, the VIPR framework is designed for integration into beamline data-acquisition workflows. For this purpose, the CLI enables automated execution within post-processing scripts triggered upon measurement completion, supporting fully automated, rapid analysis. The web application, on the other hand, serves as an interactive environment for configuring and validating the inference pipeline, lowering the barrier for experimental scientists to adopt this tool and benefit from state-of-the-art machine learning models for reflectometry and beyond. It also supports interactive comparison of stored inference results obtained with different workflows, without requiring custom analysis scripts.

Finally, VIPR provides portable, reproducible, and standardized analysis workflows. The overall pipeline of preprocessing, model selection, prediction, and result generation is defined in a single portable YAML configuration file. This facilitates reproduction of analyses across different computing environments by simply sharing configuration files.

5. Conclusions and future work

We presented VIPR, a modular, plugin-based framework for machine-learning-driven solutions to scientific inverse problems. The framework enables configurable, extensible, standardized, and reproducible inference workflows defined in YAML configuration files and executed via both command-line and web-based interfaces. This design supports both automated rapid analysis and user-friendly accessibility through an interactive graphical interface.

In this article, we demonstrated how VIPR can be applied to the domain of X-ray and neutron reflectometry through a dedicated domain plugin, showing how both deterministic and probabilistic deep learning models can be integrated within a straightforward pipeline to address ill-posed inverse problems. We also applied the same core inference pipeline to reconstruct PEMFC operating parameters from sensor data through a separate domain plugin. These two domain plugins show that VIPR can support different inverse problems without modifying the underlying core framework.

VIPR is currently deployed at the MARIA reflectometer at MLZ, where it provides fast data analysis with uncertainty estimation and near real-time parameter inference. In addition, we plan to expand the range of domain-specific plugins to support further inverse-problem applications. These developments aim to establish VIPR as a comprehensive platform for solving inverse problems using machine learning, enabling rapid, near real-time analysis during experiments.

CRediT authorship contribution statement

Jeyhun Rustamov: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Investigation, Formal analysis, Data curation. **Sascha Creutzburg:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Investigation, Formal analysis. **Jens Bornschein:** Writing – review & editing, Writing – original draft, Supervision, Software, Project administration. **Michael Bussmann:** Writing – review & editing, Supervision, Project administration, Funding acquisition. **Marina Ganeva:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Funding acquisition, Conceptualization. **Alexander Gerlach:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization. **Stefan Häusler:** Writing – review & editing, Writing – original draft, Validation, Project administration. **Bernd Helm:** Writing – review & editing, Supervision, Project administration, Funding acquisition. **Alexander Hinderhofer:** Writing – review & editing, Validation, Supervision, Project administration, Methodology. **Robert Juzak:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Alexandros Koutsioubas:** Writing – review & editing, Writing – original draft, Validation, Resources, Methodology, Formal analysis, Data curation. **Rico Löser:** Investigation, Methodology, Software, Writing – review &

editing. **Valentin Munteanu:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation. **Vedhas Pandit:** Writing – review & editing, Writing – original draft, Validation, Supervision, Investigation, Data curation. **Ulrich Schramm:** Writing – review & editing, Supervision, Project administration, Funding acquisition. **Frank Schreiber:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization. **Nico Mothes:** Writing – review & editing, Supervision, Software, Project administration, Methodology, Funding acquisition, Conceptualization. **Jeffrey Kelling:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Investigation, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

We used LLMs (Claude, DeepSeek) for linguistic refinement, including paraphrasing and word choice suggestions. All content underwent extensive manual editing. The research design, analyses, and conclusions are entirely the authors' original work.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered potential competing interests:

Jeffrey Kelling reports that financial support was provided by the Bundesministerium für Bildung und Forschung. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The project "Versatile Inverse Problem fRamework" (VIPR) was funded by the German Federal Ministry of Education and Research (Grant 05D23CJ1).

The VIPR Fuel Cell Plugin was developed by Helm & Walter IT-Solutions GmbH in collaboration with the Fraunhofer Institute for Machine Tools and Forming Technology IWU.

Appendix A. Deployment architecture

Appendix B. Code snippets

```yaml
vipr:
  inference:
    filters:
      INFERENCE_PREPROCESS_PRE_FILTER:
      - class: vipr_reflectometry.reflectorch.preprocess.
    interpolation_filter.InterpolationFilter
        enabled: true
        method: preprocess_interpolate
        parameters: null
        weight: 0
    load_data:
      handler: csv_spectrareader
      parameters:
        column_mapping:
          I: 1
          q: 0
        data_path: '@vipr_reflectometry/examples/data/PTCDI-C3.txt'
    load_model:
      handler: reflectorch
      parameters:
        config_name: g_mc_point_xray_intconv_standard_L2_InputQ_size1024
    postprocess:
      handler: ''
      parameters: {}
    prediction:
      handler: reflectorch_predictor
      parameters:
        calc_polished_sld_profile: true
        calc_pred_curve: true
        calc_pred_sld_profile: true
        clip_prediction: true
        polish_prediction: true
        prior_bounds:
          roughnesses: [[0, 20], [0, 15], [0, 15]]
          slds: [[10, 13], [20, 21], [20, 21]]
          thicknesses: [[1, 400], [1, 10]]
          q_shift: [-0.002, 0.002]
          r_scale: [0.9, 1.1]
        sld_profile_padding_left: 0.4
        sld_profile_padding_right: 1.3
        q_resolution: null
        upper_phase_sld: 0
        use_q_shift: false
    preprocess:
      handler: ''
      parameters: {}
  result_id: PTCDI-C3_Reflectorch
  config_name: PTCDI-C3_Reflectorch
```

Listing B.1. Example VIPR reflectometry inference configuration for a deterministic model applied to experimental X-ray data with polishing by least-squares fit enabled.

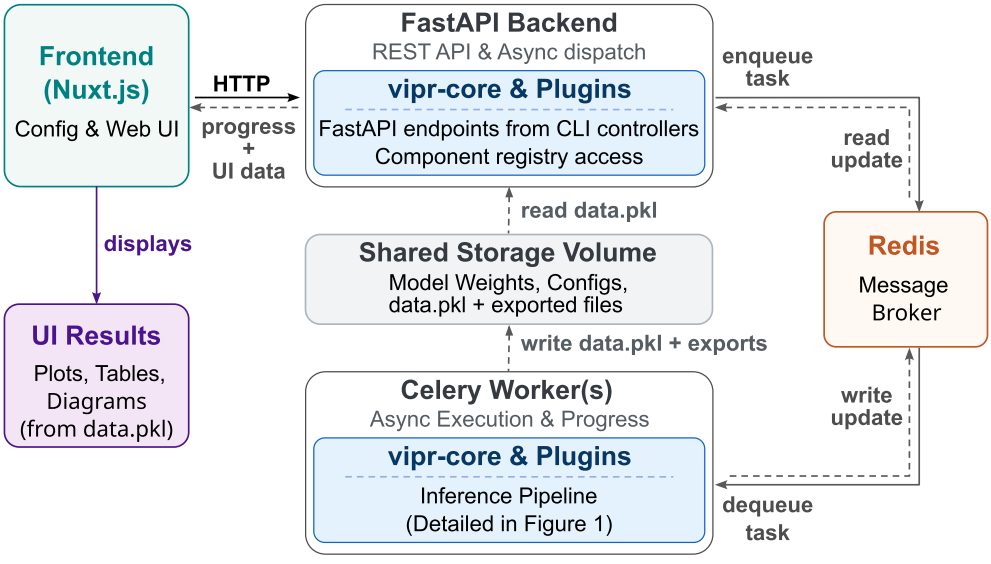


Fig. A.7. Deployment architecture with asynchronous worker decoupling via Redis.

```yaml
vipr:
  config_name: Fe_Pt_DN_NSF
  inference:
    filters:
      INFERENCE_PREDICTION_POST_FILTER:
      - class: vipr_reflectometry.flow_models.predict.
    physical_constraint_filter.PhysicalConstraintFilter
        enabled: true
        method: filter_unphysical_samples
        parameters: {}
        weight: -5
      INFERENCE_PREPROCESS_PRE_FILTER:
      - class: vipr_reflectometry.flow_models.preprocess.flow_preprocessor
    .FlowPreprocessor
        enabled: true
        method: preprocess_flow
        parameters: {}
        weight: 0
    hooks:
      INFERENCE_POSTPROCESS_PRE_PRE_FILTER_HOOK:
      - class: vipr_reflectometry.flow_models.postprocess.
    basic_corner_plot.BasicCornerPlot
        enabled: true
        method: create_basic_corner_plot
        parameters: {}
        weight: 0
      - class: vipr_reflectometry.flow_models.postprocess.cluster.
    model_selection.hook.ClusterDiagnosticsHook
        enabled: false
        method: cluster_model_selection
        parameters:
          hdbscan_mcs_max: 200
          hdbscan_mcs_min: 20
          hdbscan_mcs_step: 20
          hdbscan_min_samples: 10
          k_max: 10
          k_min: 1
          method: gmm
          n_init: 10
          seed: 42
          whiten: true
        weight: 0
      - class: vipr_reflectometry.flow_models.postprocess.cluster.
    clustering.hook.ClusterHook
        enabled: true
        method: cluster_posterior_samples
        parameters:
          covariance_type: full
          method: gmm
          min_cluster_size: 50
          min_samples: 10
          n_components: 2
          n_init: 10
          polish_centroids: true
          seed: 42
          weight_concentration_prior: null
          weight_concentration_prior_type: dirichlet_process
        weight: 0
    load_data:
      handler: csv_spectrareader
      parameters:
        column_mapping:
          I: 1
          q: 0
        data_path: '@vipr_reflectometry/examples/data/Fe_Pt_DN.dat'
    load_model:
      handler: flow_model_loader
      parameters:
        config_name: fxc34ran
        device: cpu
        model_dir: saved_models
        weights_format: pt
    postprocess:
      handler: ''
      parameters: {}
    prediction:
      handler: flow_predictor
      parameters:
        device: cpu
        num_samples: 4000
        q_resolution: 0.1
        seed: 42
    preprocess:
      handler: ''
      parameters: {}
  result_id: Fe_Pt_DN_NSF
```

Listing B.2. Example VIPR reflectometry inference configuration for a normalizing flow–based model applied to experimental neutron data with polishing by least-squares fit enabled.

```yaml
vipr:
  inference:
    filters:
      INFERENCE_PREPROCESS_PRE_FILTER:
      - enabled: true
        weight: -10
        parameters:
          filter_threshold: 0.3
          filter_remove_singles: true
          filter_remove_consecutives: true
          filter_consecutive: 3
          filter_q_start_trunc: 0.1
        class: vipr_reflectometry.reflectorch.preprocess.error_bar_filter.
    ErrorBarFilter
        method: preprocess_error_bar_filter
      - enabled: true
        weight: 0
        parameters: null
        class: vipr_reflectometry.reflectorch.preprocess.
    interpolation_filter.InterpolationFilter
        method: preprocess_interpolate
    load_data:
      handler: csv_spectrareader
      parameters:
        data_path: '@vipr_reflectometry/examples/data/Fe_Pt_DN.dat'
        column_mapping:
          q: 0
          I: 1
    load_model:
      handler: reflectorch
      parameters:
        config_name: NR-2layers-basic-v1
        weights_format: safetensors
        repo_id: valentinsingularity/reflectivity
        device: cpu
    prediction:
      handler: reflectorch_predictor
      parameters:
        calc_pred_curve: true
        clip_prediction: true
        polish_prediction: true
        upper_phase_sld: 0
        q_resolution: 0.1
        use_q_shift: false
        sld_profile_padding_left: 0.3
        sld_profile_padding_right: 1.3
        prior_bounds:
          thicknesses: [[1, 50], [1, 400]]
          roughnesses: [[1, 20], [1, 20], [1, 20]]
          slds: [[6, 6.5], [3, 3.2], [5.9, 6.1]]
          log10_background: [-10, -4]
          r_scale: [0.9, 1.1]
        calc_pred_sld_profile: true
        calc_polished_sld_profile: true
    preprocess:
      handler: ''
      parameters: {}
    postprocess:
      handler: ''
      parameters: {}
  result_id: Fe_Pt_DN_Reflectorch_narrow_bounds
  config_name: Fe_Pt_DN_Reflectorch_narrow_bounds
```

Listing B.3. Example VIPR reflectometry inference configuration for a deterministic model applied to experimental neutron data with narrow prior bounds on layer thicknesses.

The broad-bounds variant (`Fe_Pt_DN_Reflectorch_broad_bounds.yaml`) is identical to Listing B.3 except for the prior bounds on the first layer thickness, which are widened from 1–50 Å to 1–400 Å. The resulting diff is shown in Listing B.4.

```yaml
vipr:
  inference:
    # ... unchanged ...
    prediction:
      handler: reflectorch_predictor
      parameters:
        # ... unchanged ...
        prior_bounds:
          # ... unchanged ...
          thicknesses: [[1, 400], [1, 400]]
    # ... unchanged ...
  result_id: Fe_Pt_DN_Reflectorch_broad_bounds
  config_name: Fe_Pt_DN_Reflectorch_broad_bounds
```

Listing B.4. Differences between the narrow- and broad-bounds VIPR reflectometry inference configurations. Only the `vipr.inference.prediction.parameters.prior_bounds.thicknesses` entry differs operationally; the `result_id` and `config_name` entries are renamed to disambiguate runs. All other fields match Listing B.3.

```yaml
compare:
  results:
    - Fe_Pt_DN_Reflectorch_narrow_bounds
    - Fe_Pt_DN_Reflectorch_broad_bounds
    - Fe_Pt_DN_NSF
  overlay:
    series_kinds:
      - polished
    include_experimental: true
  result_id: Fe_Pt_DN_compare
```

Listing B.5. An example VIPR compare configuration that overlays the polished reflectivity and SLD profiles from the narrow-bounds, broad-bounds, and normalizing flow-based inference runs against the experimental neutron data.

Appendix C. Implementing a domain plugin

The following excerpts show how `vipr-reflectometry` registers its workflow components with VIPR. Further details are provided in the plugin-development guide, the inference workflow documentation, and the documentation of configurable hooks and filters.

Package discovery. The package exposes its `load(app)` function through the `vipr.plugins` entry-point group. VIPR invokes this function at start-up to register the plugin's domain components.

Data loading. The `csv_spectrareader` loader reads momentum transfer, reflectivity, and optional uncertainties from a column-based text file and returns them as a batch-first `DataSet`.

```toml
[project.entry-points."vipr.plugins"]
reflectometry = "vipr_reflectometry:load"
```

```python
# vipr_reflectometry/__init__.py
def load(app):
    from . import shared, reflectorch, flow_models

    shared.load(app)
    reflectorch.load(app)
    flow_models.load(app)
    # The optional PANPE integration is loaded in the same way.
```

Listing C.1. Plugin entry point and `load(app)` function in `vipr-reflectometry` (simplified).

```python
@discover_data_loader('csv_spectrareader', CSVSpectraReaderParams)
class CSVSpectraReaderDataLoader(DataLoaderHandler):
    def _load_data(self, **kwargs) -> DataSet:
        params = CSVSpectraReaderParams.model_validate(kwargs)
        spectrum = SpectraReader(
            params.data_path,
            column_mapping=params.column_mapping,
        ).list()[0]
        data = spectrum.resolve()

        return DataSet(
            x=data.q,
            y=data.I,
            dx=data.dQ,
            dy=data.dI,
            metadata={'source': params.data_path},
        )

def load(app):
    app.handler.register(CSVSpectraReaderDataLoader)
```

Listing C.2. Data-loader implementation in `vipr-reflectometry` (simplified).

```python
@discover_model_loader('reflectorch', ReflectorchModelLoaderParams)
class ReflectorchModelLoader(ModelLoaderHandler):
    def _load_model(self, **kwargs):
        params = ReflectorchModelLoaderParams.model_validate(kwargs)
        return InferenceModel(
            config_name=params.config_name,
            root_dir=str(get_model_family_root('reflectorch')),
            device=params.device,
        )

@discover_predictor(
    'reflectorch_predictor',
    parameters=ReflectorchPredictorParams,
)
class ReflectorchPredictor(PredictorHandler):
    def _predict(self, dataset, model, params):
        params = ReflectorchPredictorParams.model_validate(params)
        q_resolution = (
            dataset.dx[0]
            if dataset.dx is not None
            else params.q_resolution
        )
        return model.predict(
            reflectivity_curve=dataset.y[0],
            q_values=dataset.x[0],
            q_resolution=q_resolution,
            prior_bounds=params.prior_bounds,
            polish_prediction=params.polish_prediction,
            calc_pred_curve=params.calc_pred_curve,
            calc_pred_sld_profile=params.calc_pred_sld_profile,
            calc_polished_sld_profile=params.calc_polished_sld_profile,
        )

def load(app):
    app.handler.register(ReflectorchModelLoader)
    app.handler.register(ReflectorchPredictor)
```

Listing C.3. Reflectorch model-loader and predictor pair (simplified).

Connecting a model and its inference routine. Each inference method contributes a model loader and a predictor. For the Reflectorch workflow, the model loader initializes the configured Reflectorch inference model. The predictor calls this model with the preprocessed reflectivity curve, its momentum-transfer grid and resolution, the prior bounds, and options controlling polishing least-squares fit and the calculation of reflectivity curves and scattering-length-density profiles. The returned dictionary contains the inferred layer parameters and, if enabled, the polished parameters and calculated curves and profiles. The normalizing-flow and PANPE workflows implement the same pair of interfaces.

Preprocessing and result generation. Before prediction, the interpolation filter maps each measured curve to the input grid of the loaded Reflectorch model. After prediction, the `INFERENCE_POSTPROCESS_PRE_PRE_FILTER_HOOK` invokes `ReflectorchDataCollector.collect_prediction_results`. The

```python
class InterpolationFilter:
    @discover_filter(
        'INFERENCE_PREPROCESS_PRE_FILTER',
        enabled_in_config=True,
    )
    def preprocess_interpolate(self, data: DataSet, **kwargs):
        model = self.app.inference.model
        result = model.interpolate_data_to_model_q(
            data.x[0], data.y[0],
            sigmas_exp=data.dy[0] if data.dy is not None else None,
            q_res_exp=data.dx[0] if data.dx is not None else None,
            as_dict=True,
        )
        return DataSet(
            x=result['q_model'],
            y=result['reflectivity'],
            dx=result.get('q_resolution'),
            dy=result.get('sigmas'),
            metadata=data.metadata,
        )

class ReflectorchDataCollector:
    def __init__(self, app):
        self.spectrum_visualizer = SpectrumVisualizer(app)
        app.hook.register(
            'INFERENCE_POSTPROCESS_PRE_PRE_FILTER_HOOK',
            self.collect_prediction_results,
        )

    def collect_prediction_results(self, app, data=None, result=None):
        dc = app.datacollector.create_item_collector(0)
        self.spectrum_visualizer.add_parameter_table_for_spectrum(dc, ...)
        self.spectrum_visualizer.add_reflectivity_diagram_for_spectrum(dc,
     ...)
        self.spectrum_visualizer.add_sld_diagram_for_spectrum(dc, ...)
```

Listing C.4. Reflectometry preprocessing and result collection (simplified).

```yaml
vipr:
  inference:
    filters:
      INFERENCE_PREPROCESS_PRE_FILTER:
        - class: vipr_reflectometry.reflectorch.preprocess.
    interpolation_filter.InterpolationFilter
          method: preprocess_interpolate
          enabled: true
          weight: 0
          parameters: null
    load_data:
      handler: csv_spectrareader
      parameters:
        data_path: '@vipr_reflectometry/examples/data/Fe_Pt_DN.dat'
        column_mapping: {q: 0, I: 1}
    load_model:
      handler: reflectorch
      parameters:
        config_name: NR-2layers-basic-v1
        device: cpu
    prediction:
      handler: reflectorch_predictor
      parameters:
        polish_prediction: true
```

Listing C.5. Connection of the registered reflectometry components through YAML (simplified).

method adds parameter tables and reflectivity and scattering-length-density diagrams to the shared `DataCollector`.

Workflow configuration. YAML selects the handlers by their registered names and the filters by their classes and methods. Configured parameters are validated against the schemas declared by the respective components before execution.

Plugin scope. As a set of configurable components, `vipr-reflectometry` provides two data loaders, three model loaders, three predictors, three model builders, three preprocessing filters, one post-prediction filter, and four postprocessing hooks. The package also includes example configurations and tests for the supported workflows. VIPR uses these components to execute inference. Model development, training, and scientific validation remain outside the framework.

Appendix D. Posterior sampling: normalising flow vs. MCMC

In Section 3.2, we show the results of sampling the distribution of parameter estimates for a given neutron reflectometry measurement. In contrast to models like Reflectorch [19] for point estimation, which yield a single prediction, sampling the posterior distribution provides estimates of parameter uncertainties and can uncover ambiguous solutions. The distribution shown in Fig. 4 is sampled $N = 4000$ times from the posterior distribution learned by an NSF in VIPR's reflectometry plugin. Other Bayesian approaches include Markov chain Monte Carlo (MCMC) sampling. To cross-check these solutions, we use the *emcee* ensemble sampler [38], as implemented in the *refnx* package [7]. A quantitative comparison of learned neural posteriors against MCMC is beyond the scope of this work; for this we refer to [30]. Here, we only confirm the two solutions found by the NSF in Fig. 4 via MCMC and report indicative timings, without a strict benchmark comparison.

A key difference from MCMC, which explores the posterior for each input measurement, is that a normalizing flow must first be trained on the whole input parameter space; this cost is *amortized* over repeated, fast inference on many measurements. In VIPR's main use case as an *online* inference pipeline, the low latency until a prediction is available matters more than the one-time training cost.

The NSF model used in Section 3.2 was trained on a single NVIDIA Tesla V100-SXM2-32 GB GPU in $\lesssim 12$ h. Inference of $N = 4000$ posterior samples takes 0.1 s on the same GPU or 1.2 s on a CPU (Intel Xeon Gold 6136, 12 cores)—well within the time frame to support online decisions during experimental beamtimes.

Fig. D.1 shows a corner plot of the posterior distribution produced by *refnx* MCMC while covering the whole training parameter range of the NSF. This run was performed using the VIPR CLI and the VIPR-reflectometry domain plugin, which also includes limited support for *refnx* fitting (differential evolution and *emcee*). GMM clustering was applied to the MCMC samples after thinning to identify the centroids of two modes. The MCMC approach finds the same two solutions with exchanged layer thicknesses as predicted by the NSF in Fig. 4. Even this large MCMC run did not converge. Achieving reliable convergence in this setting would require substantially more computation; the run time and hardware are given in the caption of Fig. D.1.

The *emcee* method is not ideal for multi-modal posteriors. The walkers tend to remain near one mode and only rarely transition to the other. These rare transitions cause slow inter-mode mixing, which dominates the autocorrelation time; for the present run it is estimated as $\tau \approx 4077$ iterations. This estimate may not have stabilized even after 200 000 iterations and is therefore likely an underestimate. Transitions between the modes appear as trails of intermediate samples in the scatter plots. These samples increase the apparent background density and make the cluster centroids less stable. To obtain more stable cluster centroids, we applied a biased thinning procedure by selecting the highest-likelihood sample from each chain segment of length $\tau = 4077$. The resulting sample density in Fig. D.1 should therefore not be interpreted as the posterior density. The transition paths are particularly visible in the $d_{L2}(d_{L1})$ panel.

Instead of using wide prior bounds to expose multiple solutions, *refnx* + *emcee* is better used with narrow prior bounds, with MCMC providing a local uncertainty estimate for a single mode. In this case the autocorrelation time decreases to $\tau \approx 400$ iterations, hence a smaller run of 20 000 iterations (+2000 burn-in iterations) provides a sufficient chain length ($20\,000 \approx 50 \cdot \tau$, $\tau \approx 400$) for uncertainty estimation. This run with 24 walkers takes about 5 min on a single CPU (Intel Xeon Gold 6136, 12 cores), which is often too much latency for real-time experiment control.

While a similarly small run with wide prior bounds is insufficient to reliably identify the two solutions shown in Fig. D.1, a hint of modes can be seen when samples are not de-correlated. Taking this 5 min runtime as reference, amortization of the NSF model training cost would require less than 150 inferences. We want to remind the reader that the main limiting

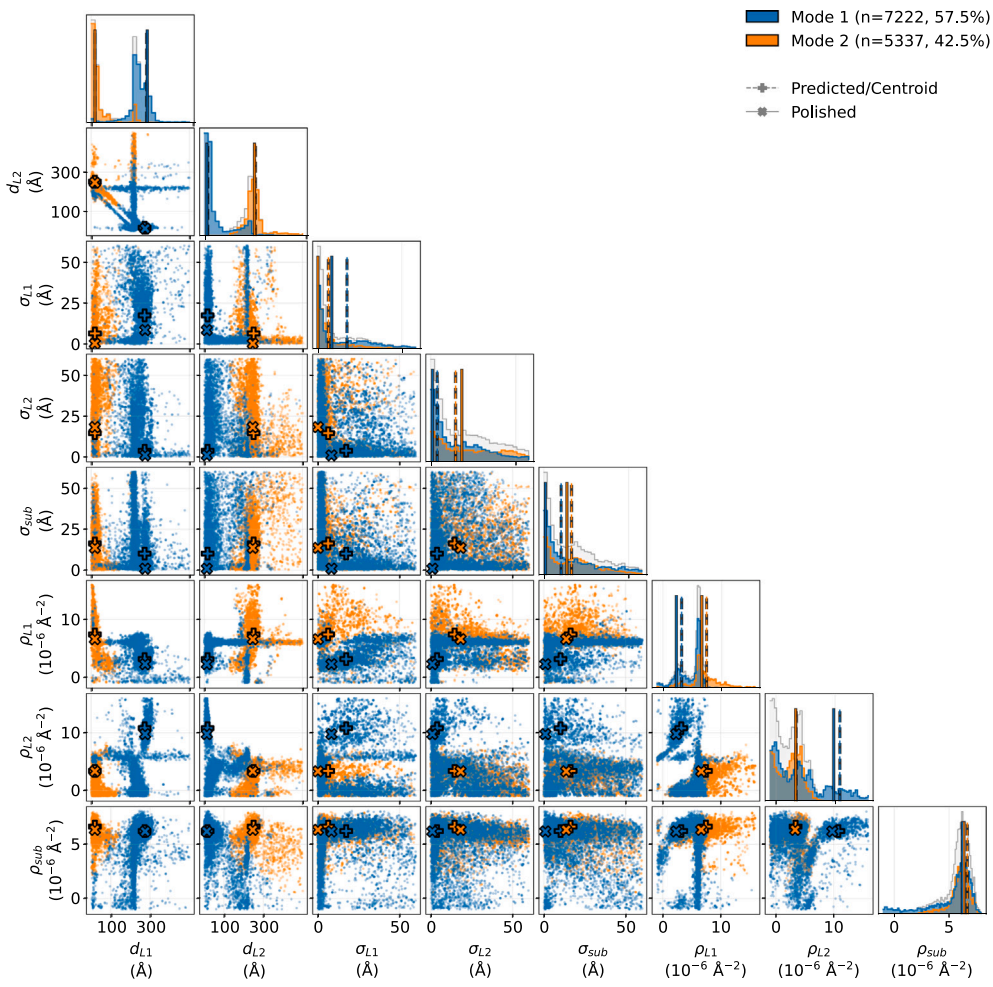


Fig. D.1. Corner plot of the posterior distribution produced by MCMC for the Fe/Pt neutron reflectometry dataset, with chains of 200 000 iterations and 256 walkers After 2000 burn-in iterations. The Autocorrelation time is estimated at $\tau \approx 4077$ iterations. Biased thinning was applied (maximum-likelihood sample from each τ-window), yielding 12 559 effective samples. The run took about 10.5 h on two AMD EPYC 7702 64-Core CPUs.

factor in VIPR's target applications is prediction latency. Reducing overall compute cost, including training, and hence amortization, is often not relevant.

Appendix E. PEMFC example configuration and output

Listing E.1 shows the complete YAML configuration used for the PEMFC example in Section 3.4. Preprocessing is implemented by the configured filter, while result collection is registered as a hook. The preprocess and postprocess handler entries therefore remain empty.

For each sensor state, VIPR generates 1000 joint samples from the conditional posterior distribution. For the sensor state at simulation step 41, Fig. E.1 shows histograms of the 1000 posterior samples for each of the nine reconstructed operating parameters. The dashed lines indicate the posterior mean values calculated from these samples.

```yaml
# Uncertainty-aware reconstruction of PEMFC operating parameters from a
# packaged sensor profile and the verified model bundle in this repository.
# No model registry is required.
vipr:
  inference:
    filters:
      INFERENCE_PREPROCESS_PRE_FILTER:
        - class: vipr_fuel_cell.preprocess.condition_preprocessor.PEMFCConditionPreprocessor
          method: preprocess_conditions
          enabled: true
          weight: 0
          parameters:
            invalid_rows: drop
            out_of_range: warn
    load_data:
      handler: pemfc_dataset
      parameters:
        data_path: "@vipr_fuel_cell/resources/datasets/operating_profile/sensor_data.csv"
        profile_path: "@vipr_fuel_cell/resources/datasets/operating_profile/profile.yaml"
    load_model:
      handler: pemfc_cinn
      parameters:
        model_dir: ../../../models/test_case_1
        device: cpu
    preprocess:
      handler: ""
      parameters: {}
    prediction:
      handler: pemfc_posterior
      parameters:
        num_samples: 1000
        seed: 42
        quantiles: [0.025, 0.5, 0.975]
        time_batch_size: 8
        common_latent_samples: true
        # Processed index 40 corresponds to simulation step 41 shown in the example.
        posterior_snapshot_indices: [40]
        histogram_bins: 30
    postprocess:
      handler: ""
      parameters: {}
  result_id: pemfc-operating-state-reconstruction
```

Listing E.1. VIPR configuration for PEMFC operating-state reconstruction.

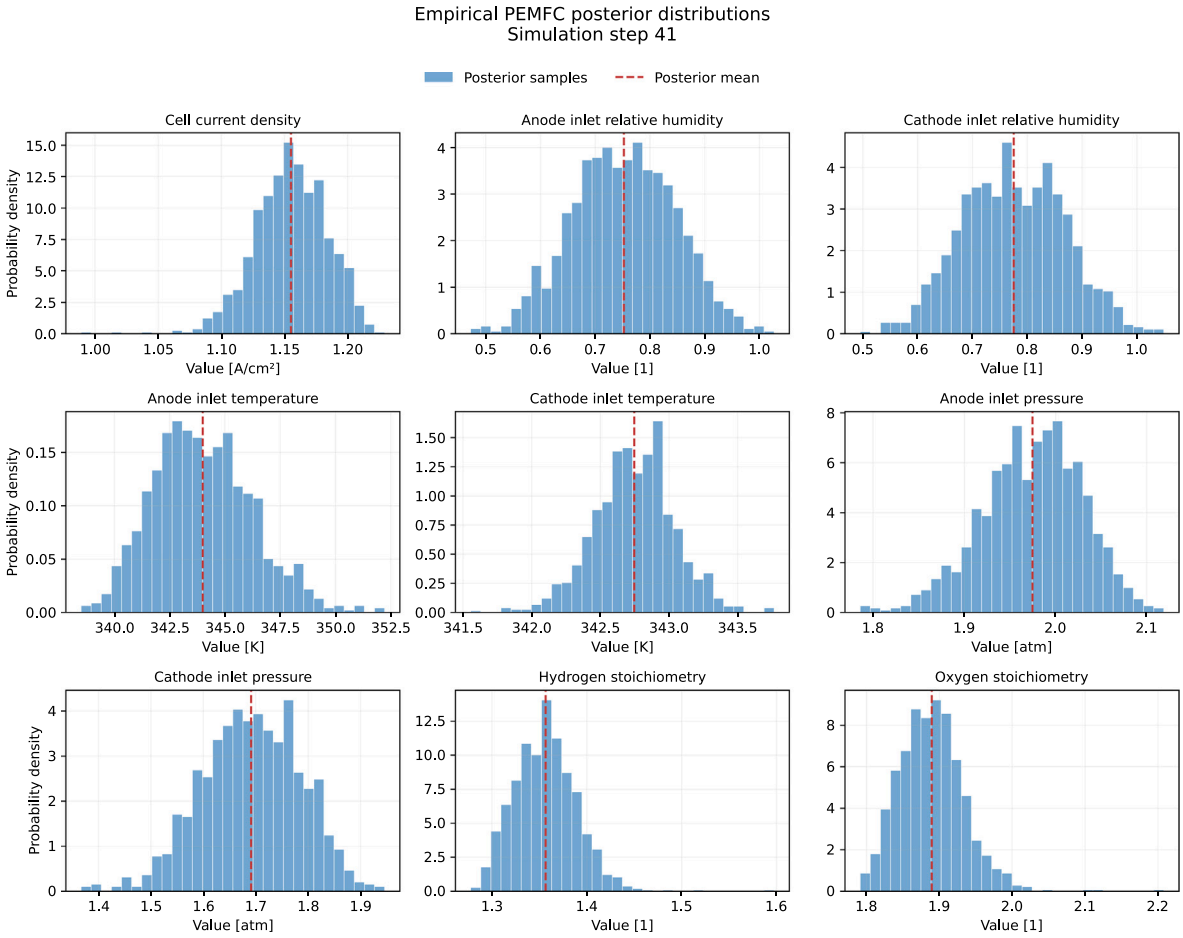


Fig. E.1. Histograms of 1000 posterior samples for the nine reconstructed PEMFC operating parameters at simulation step 41. Dashed lines indicate the posterior mean values calculated from these samples.

Data availability

A demonstrator that reproduces the results shown here is on GitHub.[1] The demonstrator repository includes the upstream repositories of VIPR's components (`vipr-core`,[2] `vipr-api`,[3] `vipr-framework`,[4] `vipr-frontend`[5]) and the reflectometry plugin `vipr-reflectometry`,[6] which are hosted publicly on the infrastructure of the Helmholtz Association, as Git submodules. Detailed steps to reproduce the presented results are included in the repository's `README.md`,[7] including steps to retrieve the NSF model weights [36].

References

[1] Taylor G. The phase problem. Acta Crystallogr. D 2003;59:1881–90. https://doi.org/10.1107/S0907444903017815

[2] Bayes T. LII. an essay towards solving a problem in the doctrine of chances. By the late Rev. MR. Bayes, FRS communicated by MR. price, in a letter to John Canton, AMFR S. Philos Trans R Soc Lond 1763;370–418. https://doi.org/10.1098/rstl.1763.0053

[3] Glavic A, Björck M. GenX 3: the latest generation of an established tool. J Appl Crystallogr 2022;55:1063–71. https://doi.org/10.1107/S1600576722006653

[4] Maranville BB, Kienzle P, Doucet M, Nelson A, Hoogerheide DP, Book A, Ghose R. reflectometry/refl1d: v0. 8.13, zenodo; 2020. https://doi.org/10.5281/zenodo.4329338

[5] Maranville BB, Kienzle P, Prescott S, Hoogerheide DP, Caruana AJ, Doucet M, Wu S, Detiste A, McCluskey A, Nelson A, Glass JB, Backman M, Ahamed AF, Krzywon J, Miller B. bumps/bumps: v1. 0.0 RC3, zenodo; 2025. https://doi.org/10.5281/zenodo.17203566

[6] Doucet M, Leal RF, Hobson TC. Web interface for reflectivity fitting. SoftwareX 2018;7:287–93. https://doi.org/10.1016/j.softx.2018.09.001

[7] Nelson AR, Prescott SW. refnx: neutron and X-ray reflectometry analysis in Python. J Appl Crystallogr 2019;52:193–200. https://doi.org/10.1107/S1600576718017296

[8] Robertson H, Gresham IJ, Prescott SW, Webber GB, Wanless EJ, Nelson A. refellips: a Python package for the analysis of variable angle spectroscopic ellipsometry data. SoftwareX 2022;20:101225. https://doi.org/10.1016/j.softx.2022.101225

[9] Nejati A, Svechnikov M, Wuttke J. BornAgain, software for GISAS and reflectometry: releases 1.17 to 20. In: EPJ web of conferences, vol. 286. EDP Sciences; 2023. https://doi.org/10.1051/epjconf/202328606004, p. 06004.

[10] Koutsioubas A. anaklasis: a compact software package for model-based analysis of specular neutron and X-ray reflectometry data sets. J Appl Crystallogr 2021;54:1857–66. https://doi.org/10.1107/S1600576721009262

[11] Fan M, Zhang Z, Lu D, Zhang G. GenAI4UQ: a software for forward and inverse uncertainty quantification using conditional generative AI. SoftwareX 2025;31:102232. https://doi.org/10.1016/j.softx.2025.102232

[12] Doucet M, Archibald RK, Heller WT. Machine learning for neutron reflectometry data analysis of two-layer thin films. Mach Learn Sci Technol 2021;2:035001. https://doi.org/10.1088/2632-2153/abf257

[13] Hinderhofer A, Greco A, Starostin V, Munteanu V, Pithan L, Gerlach A, Schreiber F. Machine learning for scattering data: strategies, perspectives and applications to surface scattering. J Appl Crystallogr 2023;56:3–11. https://doi.org/10.1107/S1600576722011566

[14] Greco A, Starostin V, Edel E, Munteanu V, Rußegger N, Dax I, Shen C, Bertram F, Hinderhofer A, Gerlach A, Schreiber F. Neural network analysis of neutron and X-ray reflectivity data: automated analysis using mlreflect, experimental errors and feature engineering. J Appl Crystallogr 2022;55:362–9. https://doi.org/10.1107/S1600576722002230

[1] https://github.com/Photon-AI-Research/vipr_demonstrator.

[2] https://codebase.helmholtz.cloud/vipr/vipr-core.git.

[3] https://codebase.helmholtz.cloud/vipr/vipr-api.git.

[4] https://codebase.helmholtz.cloud/vipr/vipr-framework.git.

[5] https://codebase.helmholtz.cloud/vipr/vipr-frontend.git.

[6] https://codebase.helmholtz.cloud/vipr/vipr-reflectometry-plugin.git.

[7] https://github.com/Photon-AI-Research/vipr_demonstrator/tree/main#steps-to-reproduce-the-results-presented-in-the-paper.

[15] Greco A, Starostin V, Hinderhofer A, Gerlach A, Skoda MW, Kowarik S, Schreiber F. Neural network analysis of neutron and x-ray reflectivity data: pathological cases, performance and perspectives. Mach Learn Sci Technol 2021;2:045003. https://doi.org/10.1088/2632-2153/abf9b1

[16] Greco A, Starostin V, Karapanagiotis C, Hinderhofer A, Gerlach A, Pithan L, Liehr S, Schreiber F, Kowarik S. Fast fitting of reflectivity data of growing thin films using neural networks. J Appl Crystallogr 2019;52:1342–7. https://doi.org/10.1107/S1600576719013311

[17] Xu Y, Tiwari S, Drago M. PycWB: a user-friendly, modular, and Python-based framework for gravitational wave unmodelled search. SoftwareX 2024;26:101639. https://doi.org/10.1016/j.softx.2024.101639

[18] Couto M, Parapar J, Losada DE. Labchain: enabling reproducible and modular scientific experiments in Python. SoftwareX 2026;33:102543. https://doi.org/10.1016/j.softx.2026.102543

[19] Munteanu V, Starostin V, Greco A, Pithan L, Gerlach A, Hinderhofer A, Kowarik S, Schreiber F. Neural network analysis of neutron and x-ray reflectivity data incorporating prior knowledge. J Appl Crystallogr 2024;57:456–69. https://doi.org/10.1107/S1600576724002115

[20] DataFolk Labs. Cement: CLI application framework for Python; 2026. https://github.com/datafolklabs/cement/.

[21] Photon AI Research. HZDR's AI consultant team for MATTER research, Helm & walter IT solutions, normalising flows for inverse problems (NF4IP); 2026. https://github.com/Photon-AI-Research/NF4IP.

[22] Open Reflectometry Standards Organization. Open reflectometry standards organization: an international, open effort to improve the scientific techniques of neutron and X-ray reflectometry; 2026. https://www.reflectometry.org.

[23] The HDF Group. Hierarchical data format, version 5; 2026. https://zenodo.org/doi/10.5281/zenodo.17808558.

[24] Munteanu V, Starostin V, Hinderhofer A, Gerlach A, Lapkin D, Schreiber F. reflectorch: a deep learning package for x-ray and neutron reflectometry. J Open Source Softw 2025;10:8169. https://doi.org/10.21105/joss.08169

[25] Pithan L, Starostin V, Mareček D, Petersdorf L, Völter C, Munteanu V, Jankowski M, Konovalov O, Gerlach A, Hinderhofer A, Murphy B, Kowarik S, Schreiber F. Closing the loop: autonomous experiments enabled by machine-learning-based online data analysis in synchrotron beamline environments. J Synchrotron Radiat 2023;30:1064–75. https://doi.org/10.1107/S160057752300749X

[26] Papamakarios G, Nalisnick E, Rezende DJ, Mohamed S, Lakshminarayanan B. Normalizing flows for probabilistic modeling and inference. J Mach Learn Res 2021;22:1–64. https://dl.acm.org/doi/10.5555/3546258.3546315.

[27] Cranmer K, Brehmer J, Louppe G. The frontier of simulation-based inference. In: Proceedings of the national academy of sciences, vol. 117; 2020. p. 30055–62. https://doi.org/10.1073/pnas.1912789117

[28] Rezende DJ, Mohamed S. Variational inference with normalizing flows. In: Proceedings of the 32nd international conference on machine learning, vol. 37. ICML'15, JMLR.org; 2015. p. 1530–8. https://dl.acm.org/doi/10.5555/3045118.3045281.

[29] Durkan C, Bekasov A, Murray I, Papamakarios G. Neural spline flows; 2019. https://dl.acm.org/doi/10.5555/3454287.3454962.

[30] Starostin V, Dax M, Gerlach A, Hinderhofer A, Tejero-Cantero Á, Schreiber F. Fast and reliable probabilistic reflectometry inversion with prior-amortized neural posterior estimation. Sci Adv 2025;11:eadr9668. https://doi.org/10.1126/sciadv.adr9668

[31] Ardizzone L, Lüth C, Kruse J, Rother C, Köthe U. Guided image generation with conditional invertible neural networks. arXiv:1907.02392. 2019. https://doi.org/10.48550/arXiv.1907.02392

[32] Löser R, Creutzburg S, Mothes N, Dix M. Conditional invertible neural network for online-capable monitoring of polymer electrolyte membrane fuel cells. Int J Hydrog Energy 2026;250:155929. https://doi.org/10.1016/j.ijhydene.2026.155929

[33] Creutzburg S, Löser R, Mothes N. VIPR fuel cell plugin; 2026. https://doi.org/10.5281/zenodo.22078852

[34] Mattauch S, Koutsioubas A, Rücker U, Korolkov D, Fracassi V, Daemen J, Schmitz R, Bussmann K, Suxdorf F, Wagener M, Kämmerling P, Kleines H, Fleischhauer-Fuß L, Bednareck M, Ossoviy V, Nebel A, Stronciwilk P, Staringer S, Gödel M, Richter A, Kusche H, Kohnke T, Ioffe A, Babcock E, Salhi Z, Bruckel T. The high-intensity reflectometer of the Jülich centre for neutron science: maria. J Appl Crystallogr 2018;51:646–54. https://doi.org/10.1107/S1600576718006994

[35] Abeles F. La théorie générale des couches minces. J. Phys. Radium 1950;11:307–9. https://doi.org/10.1051/jphysrad:01950001107030700

[36] ML for Photons and Complex Systems at HZDR. Neutron reflectometry NSF model weights; 2026. https://doi.org/10.57967/hf/8498.

[37] Kiranyaz S, Avci O, Abdeljaber O, Ince T, Gabbouj M, Inman DJ. 1D convolutional neural networks and applications: a survey. Mech Syst Signal Process 2021;151:107398. https://doi.org/10.1016/j.ymssp.2020.107398

[38] Foreman-Mackey D, Hogg DW, Lang D, Goodman J. <tt>emcee</tt>: the MCMC hammer. Publ Astron Soc Pac 2013;125:306–12. http://dx.doi.org/10.1086/670067.