The Journal of Open Source Software

# AutoEmulate: A PyTorch tool for end-to-end emulation workflows

**Radka Jersakova**[1¶], **Sam F. Greenbury**[1*], **Ed Chalstrey**[1], **Edwin Brown**[2], **Marjan Famili**[1], **Chris Sprague**[1], **Paolo Conti**[1], **Camila Rangel Smith**[1], **Martin A. Stoffel**[1], **Bryan M. Li**[1,5], **Kalle Westerling**[1], **Sophie Arana**[1], **Max Balmus**[1,3], **Eric Daub**[1], **Steve Niederer**[1,3], **Andrew B. Duncan**[3], and **Jason D. McEwen**[1,4]

**1** The Alan Turing Institute, London, United Kingdom **2** University of Sheffield, Sheffield, United Kingdom **3** Imperial College London, London, United Kingdom **4** University College London, London, United Kingdom **5** University of Edinburgh, Edinburgh, United Kingdom ¶ Corresponding author * These authors contributed equally.

**DOI:** 10.xxxxxx/draft

**Software**

- Review
- Repository
- Archive

**Editor:** Evan Spotte-Smith

**Reviewers:**

- @stefanradev93
- @pmeier

**Submitted:** 25 September 2025
**Published:** unpublished

**License**
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

DRAFT

## Summary

Computational simulations lie at the heart of modern science and engineering, but they are often slow and computationally costly. A common solution is to use emulators: fast, cheap models trained to approximate the simulator. However, constructing these requires substantial expertise. AutoEmulate (Stoffel et al., 2025) is a low-code Python package for emulation workflows, making it easy to replace simulations with fast, accurate emulators. AutoEmulate has now been fully refactored to use PyTorch as a backend, enabling GPU acceleration, automatic differentiation, and seamless integration with the broader PyTorch ecosystem (released in v1.0). The toolkit has also been extended with easy-to-use interfaces for common emulation tasks, including model calibration (determining which input values are most likely to have generated real-world observations) and active learning (where simulations are chosen to improve emulator performance at minimal computational cost). Together these updates make AutoEmulate uniquely suited to running performant end-to-end emulation workflows.

## Statement of need

Physical systems are often modelled using computer simulations. Depending on the complexity of the system, these simulations can be computationally expensive and time-consuming. This bottleneck can be resolved by approximating simulations with emulators, which can be orders of magnitudes faster (Kennedy & O'Hagan, 2000).

Emulation requires significant expertise in machine learning as well as familiarity with a broad and evolving ecosystem of tools. This creates a barrier to entry for domain researchers whose focus is on the underlying scientific problem. AutoEmulate (Stoffel et al., 2025) lowers the barrier to entry by automating the entire emulator construction process (training, hyperparameter tuning and model selection). This makes emulation accessible to non-specialists while also offering a reference set of emulators for benchmarking to experienced users.

AutoEmulate was originally built on scikit-learn, which is well suited for traditional machine learning but less flexible for complex workflows. AutoEmulate v1.0 introduced a PyTorch (Paszke et al., 2019) backend that provides GPU acceleration for faster training and inference and automatic differentiation via PyTorch's autograd system. It also made AutoEmulate easy to integrate with other PyTorch-based tools. For example, the PyTorch refactor enabled fast

Bayesian model calibration (identifying input values most likely to have generated real-world observations) using gradient-based inference methods such as Hamiltonian Monte Carlo exposed through Pyro (Bingham et al., 2018).

The latest version of AutoEmulate now also supports direct integration of custom simulators and active learning, in which the tool adaptively selects informative simulations to run to improve emulator performance at minimal computational cost. Additionally, the AutoEmulate refactor improved support for high-dimensional data through dimensionality reduction techniques such as principal component analysis (PCA) and variational autoencoders (VAEs).

# State of the field

This paper describes an extensive contribution to an existing package. We felt that AutoEmulate (Stoffel et al., 2025) already filled a unique gap in the ecosystem by focusing on making emulation accessible to domain researchers unfamiliar with ML. However, in its reliance on scikit-learn as a backend we could not extend it to handle use cases that we were targetting. Refactoring the backend to be PyTorch-first allowed us to leverage the wider PyTorch ecosystem as well as the benefits of having end-to-end automatically differentiable emulators and GPU acceleration. This has resulted in a tool that uniquely brings together a wide range of emulation capabilities (e.g., sensitivity analysis, calibration, active learning) and translated to a significant growth of the user base and package contributors. We have also retained support for some of the non-PyTorch features following discussions with the community (e.g., the users can still opt in to fit classic ML models such as SVMs although this results in loss of compability with some of the more advanced features).

# Software Design

AutoEmulate design is centered around (i) low-code mode, (ii) modularity and (iii) integrating with the wider ecosystem wherever possible. The design has now been updated from being scikit-learn oriented to PyTorch-first.

AutoEmulate primarily targets users who are simulation but not ML experts, aiming to make it as easy as possible to fit emulators to their simulated data. We also offer flexibility to advanced users by exposing customizable parameters through our APIs (set to sensible defaults to abstract complexity away from novice users).

The software's modular design is built on base classes for each component, enabling users to easily add new emulators and functionality. Our documentation showcases how to do this, which has already encouraged community contributions to the software. We chose PyTorch as the backend because of its autodiff and GPU capabilities as well as the mature ecosystem that we could integrate with. For example, both GPyTorch (Gardner et al., 2021) and Pyro (Bingham et al., 2018) are extensively utilised within the package.

# Example usage

The AutoEmulate documentation provides a comprehensive set of tutorials showcasing all functionality. We are also collecting case studies demonstrating how to use AutoEmulate for real-world problems and complex workflows. Below we provide a brief overview of the main features.

The core use case for AutoEmulate is emulator construction. AutoEmulate takes as input variables x, y. The variable x is a 2D array with columns corresponding to simulation parameters and rows corresponding to parameter sets. The variable y is an array of one or more simulation outputs corresponding to each set of parameters. From this data, AutoEmulate constructs an emulator in just a few lines of code:

DRAFT

```python
from autoemulate import AutoEmulate

ae = AutoEmulate(x, y)

result = ae.best_result()

emulator = result.model
```

This simple script runs a search over a library of emulator models, performs hyperparameter tuning and compares models using cross validation. Each model is stored along with metadata in a `Results` object. The user can then easily extract the best performing emulator.

AutoEmulate can additionally search over different data preprocessing methods, such as normalization or dimensionality reduction techniques (PCA, VAEs). Any `Transform` from PyTorch distributions can also be used. The transforms are passed as a list to permit the user to define a sequence of transforms to apply to the data. For example, the following code standardizes the input data and compares three different output transformations: no transformation, PCA with 16 components, and PCA with 32 components in combination with the default set of emulators:

```python
from autoemulate.transforms import PCATransform, StandardizeTransform

ae = AutoEmulate(
    x,
    y,
    x_transforms_list=[[StandardizeTransform]]
    y_transforms_list=[
      [],
      [PCATransform(n_components=16)],
      [PCATransform(n_components=32)]
    ],
)
```

The result in this case will return the best combination of model and output transform. The returned emulator and transforms are wrapped together in a `TransformedEmulator` class, which outputs predictions in the original data space. The figure below shows an example result of fitting a Gaussian Process emulator in combination with PCA to a reaction-diffusion simulation (see the full tutorial for a detailed overview).

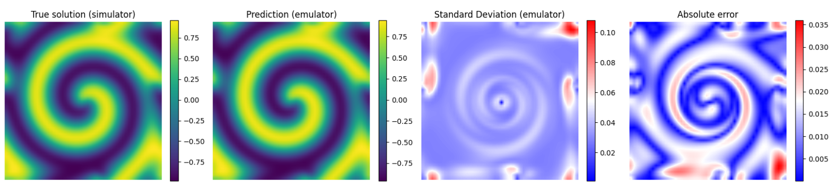


**Figure 1:** GP with PCA emulator prediction for a reaction diffusion simulation compared to the ground truth.

Once an emulator has been trained it can generate fast predictions for new input values, enabling downstream tasks such as sensitivity analysis or model calibration. For example, to run Sobol sensitivity analysis one only needs to pass the trained emulator and some information about the data. Below is a dummy example assuming a simulation with two input parameters `param1` and `param2`, each with a plausible range of values, and two outputs `output1` and `output2`:

```python
from autoemulate.core.sensitivity_analysis import SensitivityAnalysis

input_parameters_ranges = {
    'param1': (0, 1),
    'param2': (0, 10),
}

problem = {
    'num_vars': 2,
    'names': ["param1", "param2"],
    'bounds': input_parameters_ranges.values(),
    'output_names': ["output1", "output2"],
}

sa = SensitivityAnalysis(emulator, problem=problem)
sobol_df = sa.run()
```

AutoEmulate also provides a simple interface for calibration given a trained emulator, input parameter ranges (same as in the sensitivity analysis example), and real-world observations:

```python
from autoemulate.calibration.bayes import BayesianCalibration

observations = {'output1': 0.5, 'output2': 7.2}

bc = BayesianCalibration(
    emulator,
    input_parameters_ranges,
    observations,
)
mcmc = bc.run()
```

Lastly, AutoEmulate makes it easy to integrate custom simulators through subclassing. This enables simulator-in-the-loop workflows like active learning, which selects the most informative simulations to improve emulator performance at minimal computational cost.

# Research Impact Statement

In the last year, we have worked with around 10 collaborators across diverse domains including biomedicine and materials science. This has led to academic outputs such as a poster at OFEME2025. Additionally, our collaborations have driven software development through numerous feature requests and bug reports that we have addressed. For example, we have implemented a full end-to-end calibration workflow used by our collaborators in cardiac modelling and demonstrated how to use AutoEmulate in their pipelines in one of our case studies. We have also had contributions outside the core development team. This has included external contributors responding to existing issues as well as users adapting the tool for their own use cases (e.g., contributing new types of emulators).

# AI usage disclosure

Human authors have made all the core design decisions and authored much of the code and documentation. Generative AI tools have been used to assist with code and documentation writing. Specifically, the team uses GitHub Copilot in auto mode or selects one of the available versions of Claude, GPT and Gemini. We confirm that human authors have reviewed, edited and validated all AI-assisted outputs. We have also added a section on the use of generative AI tools in our contributing guidelines.

# References

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P. A., Horsfall, P., & Goodman, N. D. (2018). Pyro: Deep universal probabilistic programming. *CoRR*, *abs/1810.09538*. http://arxiv.org/abs/1810.09538

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., & Wilson, A. G. (2021). *GPyTorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration*. https://arxiv.org/abs/1809.11165

Kennedy, M. C., & O'Hagan, A. (2000). Predicting the output from a complex computer code when fast approximations are available. *Biometrika*, *87*(1), 1–13. http://www.jstor.org/stable/2673557

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *CoRR*, *abs/1912.01703*. http://arxiv.org/abs/1912.01703

Stoffel, M. A., Li, B. M., Westerling, K., Arana, S., Balmus, M., Daub, E., & Niederer, S. (2025). AutoEmulate: A python package for semi-automated emulation. *Journal of Open Source Software*, *10*(107), 7626. https://doi.org/10.21105/joss.07626

DRAFT