





















AutoEmulate v1.0: 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](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: 

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Published: unpublished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](https://creativecommons.org/licenses/by/4.0/)).

Summary

Computational simulations lie at the heart of modern science and engineering, but they are often slow and computationally costly. This poses a significant bottleneck. A common solution is to use emulators: fast, cheap models trained to approximate the simulator. However, constructing these requires substantial expertise. AutoEmulate is a low-code Python package for emulation workflows, making it easy to replace simulations with fast, accurate emulators. In version 1.0, AutoEmulate has been fully refactored to use PyTorch as a backend, enabling GPU acceleration, automatic differentiation, and seamless integration with the broader PyTorch ecosystem. 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). Emulators are key to enabling any computationally expensive downstream tasks that require generating predictions for a large number of inputs. These tasks include sensitivity analysis to quantify the impact of each input parameter on the output as well as model calibration to identify input values most likely to have generated real-world observations.

Emulation requires significant expertise in machine learning as well as familiarity with a broad and evolving ecosystem of tools for model training and downstream tasks. 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, evaluation, model selection, and hyperparameter tuning). This makes emulation accessible to non-specialists while also offering a reference set of cutting-edge emulators, from classical approaches (e.g. Gaussian Processes) to modern deep learning methods, enabling benchmarking for experienced users.

AutoEmulate v1.0 introduces easy-to-use interfaces for common emulation tasks. By providing these tasks within a single package it enables users to construct sequential workflows. For instance, sensitivity analysis can be applied in order to narrow down the parameter space to key variables. This allows the user to calibrate the much smaller reduced set to match the output of the model to real-world observations. AutoEmulate 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.

AutoEmulate was originally built on scikit-learn, which is well suited for traditional machine learning but less flexible for complex workflows. Version 1.0 introduces 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 makes AutoEmulate easy to integrate with other PyTorch-based tools. For example, the PyTorch refactor enables fast Bayesian model calibration using gradient-based inference methods such as Hamiltonian Monte Carlo exposed through Pyro (Bingham et al., 2018).

Lastly, AutoEmulate v1.0 expands the set of implemented emulators, with a particular emphasis on predictive uncertainty quantification through ensemble methods. It also improves support for high-dimensional data through dimensionality reduction techniques such as principal component analysis (PCA) and variational autoencoders (VAEs). The software's modular design centred around a set of base classes for each component means that the toolkit can be easily extended by users with new emulators and transformations.

AutoEmulate fills a gap in the current landscape of emulation tools as it is both accessible to newcomers while offering flexibility and advanced features for experienced users. It also uniquely combines emulator training with support for a wide range of downstream tasks such as sensitivity analysis, model calibration and active learning.

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:

```
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 hyperparameter values and performance metrics 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. AutoEmulate implements principal component analysis (PCA) and variational autoencoders (VAEs) for handling high dimensional input or output data. Any Transform from PyTorch distributions can also be used. The

84 transforms are passed as a list to permit the user to define a sequence of transforms to apply
85 to the data. For example, the following code standardizes the input data and compares three
86 different output transformations: no transformation, PCA with 16 components, and PCA with
87 32 components in combination with the default set of emulators:

```
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)]
    ],
)
```

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

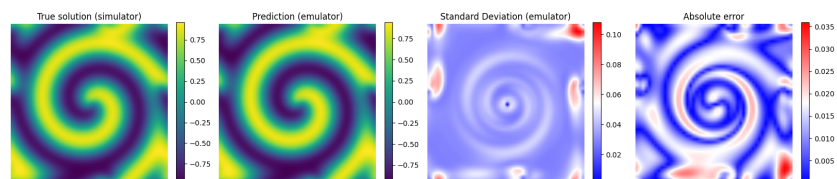


Figure 1: GP with PCA emulator prediction for a reaction diffusion simulation compared to the ground truth.

93 Once an emulator has been trained it can be used to generate fast predictions for new input
94 values or to perform [downstream tasks](#) such as sensitivity analysis or model calibration. For
95 example, to run Sobol sensitivity analysis one only needs to pass the trained emulator and
96 some information about the data. Below is a dummy example assuming a simulation with two
97 input parameters param1 and param2, each with a plausible range of values, and two outputs
98 output1 and output2:

```
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()
```

99 A more complete application of sensitivity analysis to a cardiovascular simulator is demonstrated
100 [here](#).

101 The PyTorch backend enables fast Bayesian model calibration using gradient-based inference
102 methods such as Hamiltonian Monte Carlo with Pyro. AutoEmulate provides a simple interface
103 for this given a trained PyTorch emulator, input parameter ranges (same as in the sensitivity
104 analysis example), and real-world observations:

```
from autoemulate.calibration.bayes import BayesianCalibration

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

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

105 A more complete application of Bayesian calibration to an epidemic simulation is demonstrated
106 [here](#).

107 Lastly, AutoEmulate makes it easy to integrate [custom simulators](#) through subclassing. Integrat-
108 ing custom simulators enables simulator-in-the-loop workflows like [active learning](#), which selects
109 the most informative simulations to improve emulator performance at minimal computational
110 cost.

111 References

- 112 Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh,
113 R., Szerlip, P. A., Horsfall, P., & Goodman, N. D. (2018). Pyro: Deep universal probabilistic
114 programming. *CoRR*, *abs/1810.09538*. <http://arxiv.org/abs/1810.09538>
- 115 Kennedy, M. C., & O'Hagan, A. (2000). Predicting the output from a complex computer code
116 when fast approximations are available. *Biometrika*, *87*(1), 1–13. [http://www.jstor.org/
117 stable/2673557](http://www.jstor.org/stable/2673557)
- 118 Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z.,
119 Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison,
120 M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch:
121 An imperative style, high-performance deep learning library. *CoRR*, *abs/1912.01703*.
122 <http://arxiv.org/abs/1912.01703>
- 123 Stoffel, M. A., Li, B. M., Westerling, K., Arana, S., Balmus, M., Daub, E., & Niederer, S.
124 (2025). AutoEmulate: A python package for semi-automated emulation. *Journal of Open
125 Source Software*, *10*(107), 7626. <https://doi.org/10.21105/joss.07626>