Simulation Framework: First approach at a Python-first General Simulation Package

Matt McAnear

December 15, 2025

1 Introduction

I have implemented a small simulation framework in Python that takes arbitrary user functions written with `NumPy`[5] or `JAX`[2] and manages simulation outputs. The package mainly deals with reproducibility through explicit randomization, data file storage and caching, method comparison and evaluation, and has limited visualization capabilities.

1.1 Motivation

Simulation studies often come with a lot of boilerplate code that can be shared across projects. R packages like `simChef` and `simulator` provide functionality to handle this, but Python lacks similar tools. This package aims to fill that gap in a Pythonic manner.

The main beneficiaries of the package are users who are conducting simulation studies in `JAX`. Most of the package's special implementations are built around this framework, though users may supply `NumPy` functions. However, these functions generally won't benefit from exceptional speedup under this version of the package, for reasons that will explored a bit later.

2 Description

The simluation framework (so far untitled formally) consists of components that roughly align with thir R counterparts [1], [4], [3]:

1. Data Generating Processes (DGPs)
2. Methods
3. Evaluators
4. Plotters

3 Architecture

These system components are tied together with the `run_simulations` function, which is invoked through a relatively simple mapping of functions with a dictionary of parameters. Users of `scikit-learn` should find the structure eminently familiar, as we use a list of tuples, though we've opted to attach a label to the object itself, rather than have the user pass it in.

An example usage of the entire pipeline is shown below, but can be found in greater detail in `./docs/examples/ridge_example.py`.

```
### example usage
data, methods, evaluations, plots = run_simulations(
    key,
    dgp_mapping=[
        (data_fn, {
            "n": [100, 200],
            "p": [5, 10, 20, 50],
            "dist": ["normal", "t"],
        },)
    ],
    method_mapping=[
        (method_fn, {"alpha": [0.01, 0.1, 0.5, 1.0, 2.0]}),
    ],
    evaluators=[rmse, bias, mae],
    plotters=[
        (rmse, create_plotter_fn(
            x="p",
            hue="method",
            col="dist",
            plot_class=sns.lineplot,
        )),
        # other plotters,
    ],
    targets=["beta"],
    n_sims=50,
    simulation_dir=tmpdir / "example",
    label="my_simulation",
    # this is actually just for labelling, as the "key" sets the
        randomization
    seed=42
)
```

To build this pipeline we have several helper functions the bridge between each step, with the main goal being to "automatically" utilize the outputs of one function as the input to another function through a decorator. In practice, this is a relatively simple affair that only requires users to annotate their functions with the names of their outputs and then utilize those same names for the inputs of methods.

For example, consider the following data generating process and model specification:

```
@dgp(outputs=["X", "y", "beta"], label="LinearRegressionDGP")
def linear_dgp(key, n, p, dist="normal"):
    """Generates data from a linear regression model."""
    key, subkey = jax.random.split(key)
    # logic
    return X, y, beta

@method(outputs="beta_hat", label="OLS")
def ols_regression(X, y):
    beta_hat = jnp.linalg.inv(X.T @ X) @ X.T @ y
    return beta_hat
```

These functions work exactly as before, but the `ols_regression` function has a input arguments `X, y` and the DGP `linear_dgp` has output arguments `X, y`, so these are matched together automatically by the package. Since the DGP also has the output `beta` and the method has the output `beta_hat`, these are matched by the overlapping string `beta`, and the evaluator computes metrics based on this pairing for specified evaluators.

3.1 Vectorization

`JAX` has a powerful `vmap` API that allows us to arbitrarily and efficiently vectorize functions so that users do not have to worry about proper vectorization to the same degree. This package simplifies replication by expanding outputs into higher dimensional tensor objects and then iterating over the leading index. This is done through simple assumptions:

1. The first argument of a data generating process is always a `jax.random.PRNGKey` object.
2. The output arguments of a DGP are batched in the first index.
3. All other arguments are copied across batches.

Therefore, we use a simple function to generate the appropriate `in_args` argument for `jax.vmap` that consists of a tuple of `0`'s and `None`'s that indicate which arguments should be vectorized over the leading index. This surprisingly powerful abstraction is unfortunately not available in `NumPy` to the same degree, as I haven't yet figured out a way to broadcast arbitrary functions in the same way. This limits the effectiveness of the `NumPy` imlementation.

For example, the data-generating process above could be easily written in `NumPy` to simply generate a single, 3D array of shape `(n_sims, n, p)`, but looping over the method fitting may still be unavoiable for arbitrary functions.

The reality of `JAX` is that, at some level, you can get higher performance code that is, if not sloppier, perhaps less elegant in an approach. You can write a straightforward loop that iterates over replications that is slow for smaller batches, but for larger ones, you face extremely gradual returns to scale.

3.2 Evaluation

For now, we have some simplifying API assumptions on the ‘Evaluator‘ class and ‘evaluator‘ decorator, namely that the output of each evaluator is a scalar. This is not ideal, nor is it a requirement of the package. Because of `JAX`'s automatic vectorization in fact, ‘vmap‘ can return outputs of arbitrary dimensions, even utilizing tuple collections in Python for better organization of results.

The evaluation results of the methods are stored in a `pandas.DataFrame` object, and are invoked through the `evaluate_methods` function. This function takes the outputs of the methods and data generating processes and computes evaluation metrics for the targets specified by the user.

3.3 Plotting

Rudimentary plotting capabilities are included via the `create_plotter_fn` function. Notably, this component is missing a decorator interface, but this is intentional; the plotting is arbitrary enough that given a particular plotting function (e.g., `seaborn.lineplot`), and target, users can effectively loop a plotter over different metrics. Arbitrary arguments are passed to `seaborn`'s `FacetGrid` class.

Any callable should work here, as long as it takes a `pandas.DataFrame` meets the signature requirements, which we may assert in the `evaluator` decorator.

3.4 Caching

The major beneefit the package gives is an ”invisible” caching layer that prevents data recalculation or method refitting. The `DataDict` class provides this functionality for arbitrary Python objects and uses the `dill` library for serialization. The `ImageDict` class does the same, but specializes in saving images.

4 Results

There is a clear tradeoff between the use of `NumPy` and `JAX` functions in this package. `JAX` is generally faster for larger data sizes, but the overhead of compilation and device management means that for smaller data sizes `NumPy` is often faster. Below are some timing results for fitting a ridge regression model with varying data sizes and replications.

To achieve these timings, I ran the package on my home server, with an AMD cpu and an NVidia Geforce RTX 2060 GPU. The GPU is showing its age, but still, the results demonstrate that after only 100 replications of fitting the ridge regression model on each of 40 configurations, `JAX` is able to outperform `NumPy`.

More surprisingly, these returns to scale are still realized on CPU alone, though the speedup is less pronounced. Using my local laptop, I found that `JAX` began to outperform `NumPy` after about 1000 replications of fitting the ridge regression models.

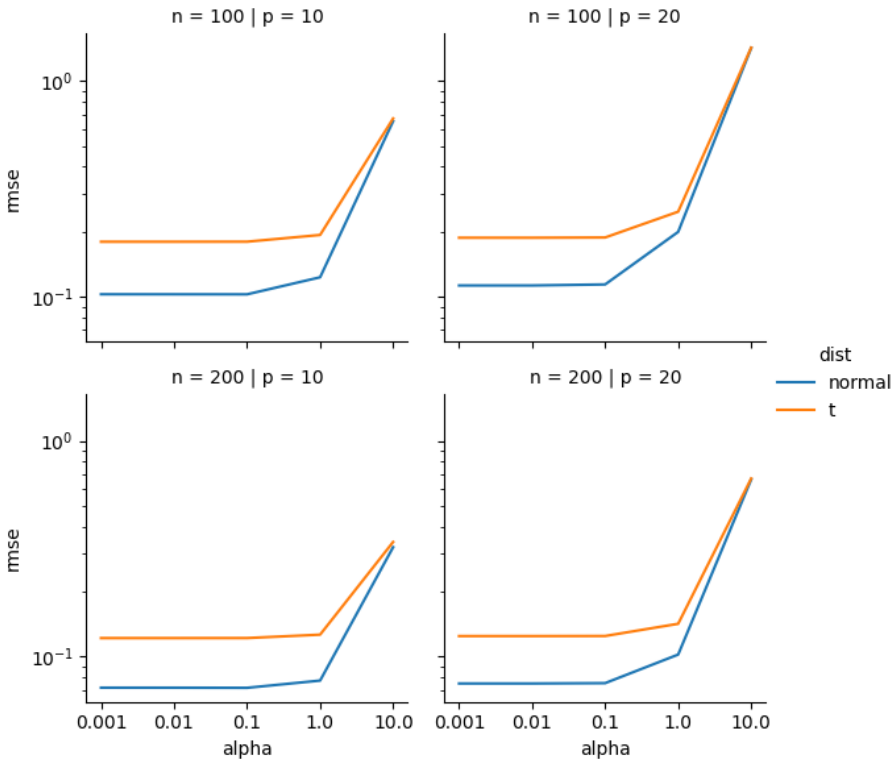

Figure 1: Example output plot. It's very basic currently.

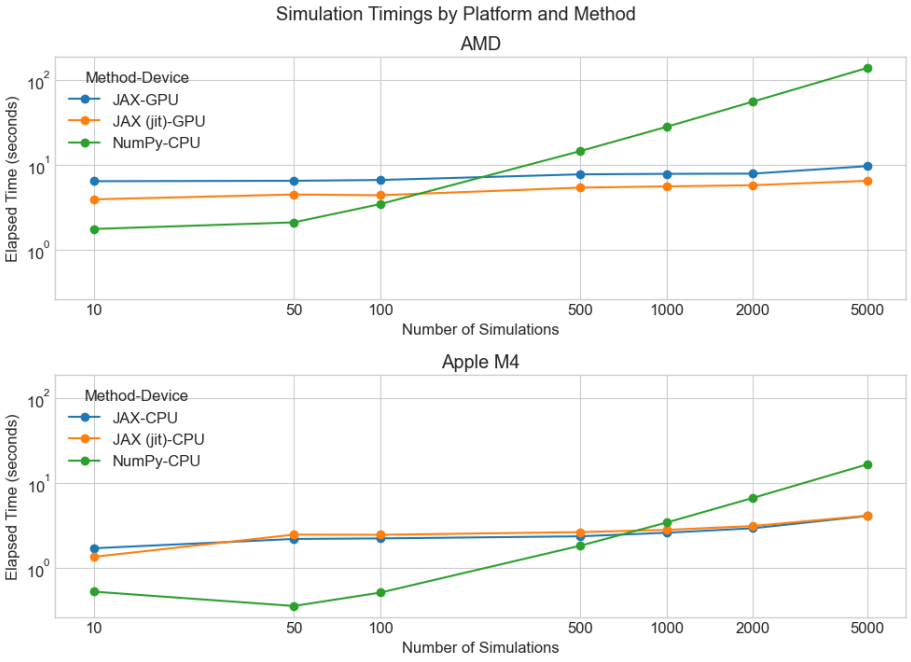

Figure 2: Timing results for fitting ridge regression models with varying data sizes and replications, both on and off a GPU. The GPU is not always better, and in fact, my local Mac was always faster.

These results are based around fairly subpar `NumPy` code for fitting ridge regression models, as the `NumPy` implementation loops over the first index dimension in a list comprehension. Still, for cases when users are writing arbitrary functions that may not be easily vectorizable, this package provides a gentle wrapper around `JAX` that allows faster code execution with fairly minimal setup and Pythonic style.

5 Lessons Learned

5.1 Challenges

1. Figuring out how to store metadata on functions without disrupting their signatures or usability
2. Keeping the API contract light. The more requirements we place on users, the less likely they are to use the package.
3. Logic for automatic caching
4. Efficient higher dimension vectorization
5. Useful plotting.
6. Type hinting and annotations across decorators and pipelines.

5.2 Course Impact

The biggest change from the course was my gradual emphasis on testing and reproducibility. Rigorous adherence to the unit tests prevented me from making breaking changes to the API and forced me to really encode the API from the start. Adding the `NumPy` functionality only took a couple of hours because of this emphasis.

Next was an emphasis on modularity. Generally in statistics courses I have let my commitment to software engineering principals wane to focus on the math. Here, I worked very hard to make sure each function was doing one and only one thing. The separation of responsibility made it easier to reason about the code and keep the pipeline intact.

Finally, the concept of a simulation framework was brought up early in the course. As I started the project I was sort of skeptical that it would be useful, but as I've worked on it, it really does allow me to stop focusing on things like saving outputs, caching inputs, and allows me to focus on the actual research. I'm excited to keep expanding this and get it out of MVP stage and maybe somewhere more useful.

6 Future Work

There are several improvements that could be made to the package in future work.

First, a post-processing step or a general "Pipeline" class. I didn't reimplement my previous project paper because it was three methods, two of which required the outputs of the first method. Here, the data generating process outputs data, the method fits the model, and then we evaluate. But if we wanted to utilize the outputs of one method in another method, that is impossible in the current setup.

Second, adding more evaluation metrics and default plots, specifically ones that don't require full scale aggregation. What about intervals and zipper plots, for example? Those are theoretically possible under the framework, but I haven't explored this fully.

Lastly, improving the `NumPy` functionality. Right now, it's a second class citizen, and performance improvements may be possible by utilizing something like a `Numba` pipeline path as a complement or alternative to `JAX`.

References

[1] Jacob Bien. *The Simulator: An Engine to Streamline Simulations*. arXiv:1607.00021 [stat]. July 2016. DOI: 10.48550/arXiv.1607.00021. URL: http://arxiv.org/abs/1607.00021 (visited on 11/12/2025).

[2] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. 2018. URL: http://github.com/jax-ml/jax.

[3] James Duncan et al. *simChef: High-quality data science simulations in R*. Issue: 95 Pages: 6156 Publication Title: Journal of Open Source Software Volume: 9. 2024.

[4] Peter Green and Catriona J. MacLeod. "SIMR: an R package for power analysis of generalized linear mixed models by simulation". en. In: *Methods in Ecology and Evolution* 7.4 (2016). _eprint: https://besjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/2041-210X.12504, pp. 493–498. ISSN: 2041-210X. DOI: 10.1111/2041-210X.12504. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/2041-210X.12504 (visited on 12/01/2025).

[5] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.