`bayesim`: a tool for fast model fitting with Bayesian inference

Rachel Kurchin

Department of Materials Science & Engineering, Massachusetts Institute of Technology,
77 Massachusetts Avenue, Cambridge, MA 02139, USA

Giuseppe Romano and Tonio Buonassisi*

Department of Mechanical Engineering, Massachusetts Institute of Technology,
77 Massachusetts Avenue, Cambridge, MA 02139, USA

abstract text here

INTRODUCTION

There are a plethora of examples across diverse scientific and engineering fields of mathematical models used to simulate the results of experimental observations. In many cases, there are input parameters to these models which are difficult to determine via direct measurement, and it is desirable to invert the numerical model – that is, use the experimental observations to determine values of the input parameters. Bayesian inference is a fruitful framework within which to do such fitting, since the resulting posterior probability distribution over the parameters of interest can give rich insights into not just the most likely values of the parameters, but also uncertainty about these values and the potentially complicated ways in which they can covary to give equally good fits to observations.

We have previously demonstrated the value of a Bayesian approach in using automated high-throughput temperature- and illumination-dependent current-voltage measurements (JVTi) to fit material/interface properties and defect recombination parameters in photovoltaic (PV) absorbers [1, 2]. In cases such as these, when the data model is not a simple analytical equation but rather a computationally intensive numerical model, efficient, nonredundant sampling of the parameter space when computing likelihoods becomes critical to making the fit feasible.

In this work, we introduce `bayesim`, a Python-based code that utilizes adaptive grid sampling to perform Bayesian parameter estimation. We discuss the structure of the code, its implementation, and provide several examples of its usage. While the authors' expertise is in the realm of semiconductor physics and thus the examples herein are drawn from that space, we also discuss the general characteristics of a problem amenable to this approach so that researchers from other fields might adopt it as well.

TECHNICAL BACKGROUND

Bayes' Theorem is a relationship between conditional probabilities. It states

$$P(H|E) = \frac{P(H)P(E|H)}{P(E)} \tag{1}$$

where the notation $P(A|B)$ indicates the probability of A being true given that B is true. H is a *hypothesis* and E the observed *evidence*. $P(H)$ is termed the *prior*, $P(E|H)$ the *likelihood*, $P(H|E)$ the *posterior*, and $P(E)$ is a normalizing constant. If there are n pieces of evidence, this can generalize to an iterative process where

$$P(H|\{E_1, E_2, ...E_n\}) = \frac{P(H|\{E_1, E_2...E_{n-1}\})P(E_n|H)}{P(E_n)} \tag{2}$$

In a multidimensional parameter estimation problem, each hypothesis H is a tuple of possible values for the fitting parameters, i.e. a point in the parameter space, while the evidence E is an observed output of a measurement as a function of various experimental conditions.

In order to compute a likelihood, a model capable of simulating that observed output as a function of both the fitting parameters and the experimental conditions is required. In `bayesim`, likelihoods are calculated for each point in parameter space using a Gaussian where the argument is the difference between observed and simulated output at that point, and the standard deviation is the sum of experimental uncertainty and model uncertainty. The experimental
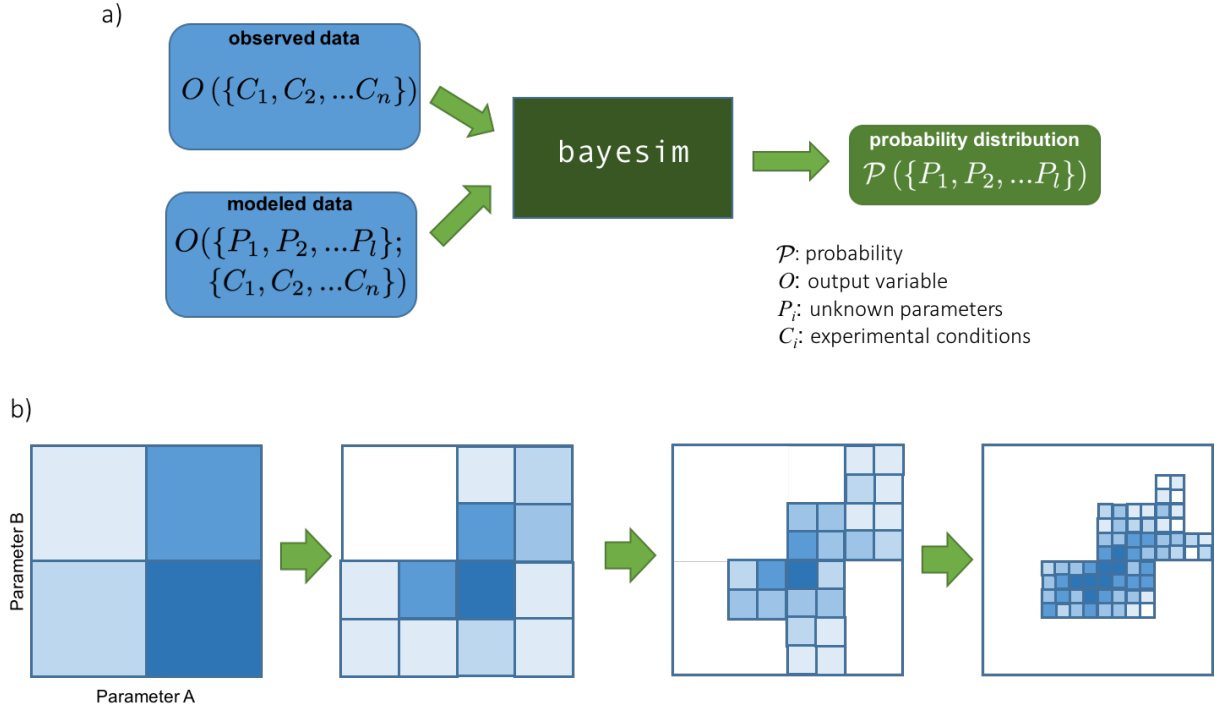
FIG. 1. a) High-level flowchart of inputs and outputs to `bayesim`, b) 2D schematic of adaptive grid sampling approach.

uncertainty is a number provided by the user that quantifies any noise/irreproducibility inherent to the measurement, while the model uncertainty is calculated by `bayesim` and reflects the sparseness of the parameter space grid, i.e. how much simulated output changes from one grid point to another.

A high-level summary of what `bayesim` does is shown in Figure 1. Part (a) is a flowchart showing that observed (as a function of experimental conditions $\{C\}$ and simulated (as a function of fitting parameters $\{P\}$ and experimental conditions $\{C\}$) outputs are compared to produce a probability distribution over $\{P\}$. Part (b) schematically indicates the adaptive grid sampling approach for a hypothetical two-dimensional parameter space, wherein grid boxes exceeding some threshold probability are subdivided and lower-probability regions discarded, allowing attainment of a high fitting precision without needing to sample the entire parameter space at the same high density.

SOFTWARE ARCHITECTURE AND INTERFACE

The basic structure of `bayesim` is shown in Figure 2a. Detailed and up-to-date documentation of classes, functions, and example applications is maintained online. [3] The top-level object with which users interact is implemented in the `Model` class. The `params` module defines classes to store information about the various types of parameters (fitting parameters, experimental conditions, and measured output) while the `Pmf` class stores the probability distribution and implements the manipulations required for Bayesian updates.

Figure 2b is a flowchart of using the code, with the main inference loop surrounded by a dotted line. One first needs to provide the observed and modeled data. Observed data is accepted in HDF5 format, while modeled data can be an HDF5 file (in which case `bayesimn` will determine the fitting parameter space grid from this file), or if the fitting parameter space has been predefined, one can also attach a Python function that computes the model data (e.g. an interface to a numerical model). Observed data should have associated experimental uncertainties, or the user can indicate a fixed value of uncertainty to use for all observations upon attaching the data.

Once modeled data is attached, model uncertainty can be computed. At each point in the space of experimental conditions, a matrix of the simulated output variable as a function of the fitting parameters is constructed and at each point in this matrix, the largest difference along any index is computed. This is set as the model uncertainty for this set of experimental conditions at this point in parameter space.

At this point, the Bayesian inference loop can proceed. We start with a bounded prior distribution over the entire
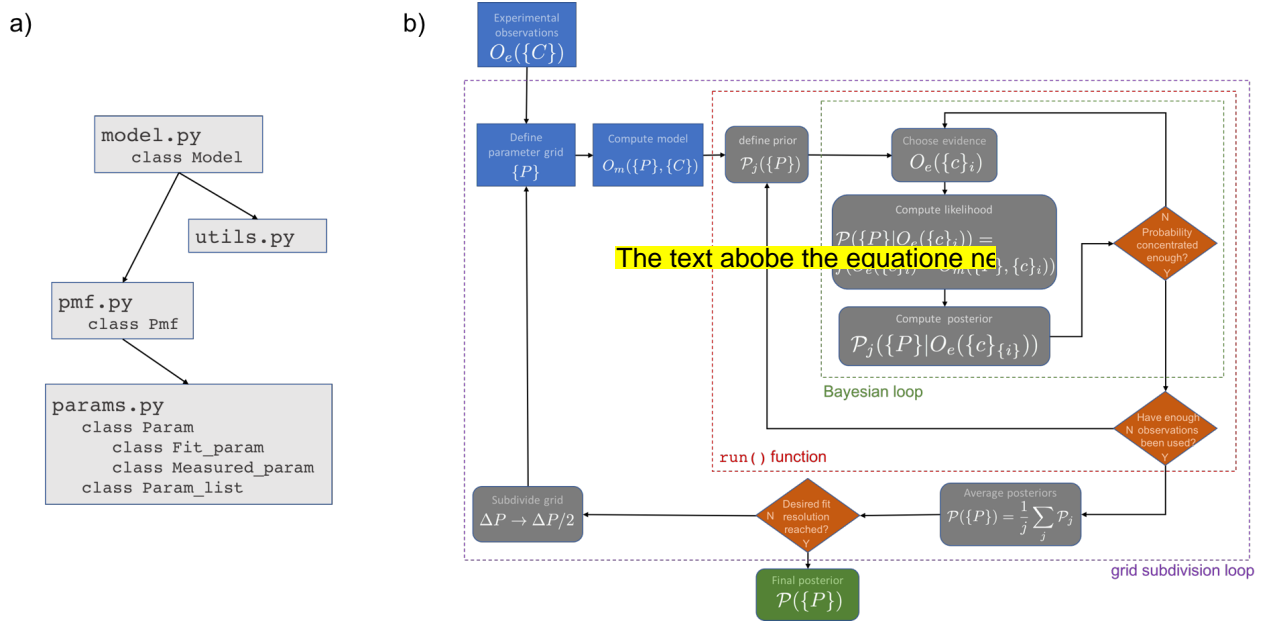
do we need a figure for this?

FIG. 2. a) `bayesim` software structure. b) Detailed flowchart of use of `bayesim`, with calculation steps and various levels of iteration indicated.

parameter space grid and the observed data in a randomized order. We feed in one piece of evidence (i.e. one observed data point) and compute the likelihood using a Gaussian. The standard deviation for each point in the parameter space is the sum of the experimental uncertainy of that observation and the model uncertainty at that point. The posterior distribution is computed as a Bayesian update on the prior. Next, we check whether the threshold condition for concentration of probability (defined as at least some percentage (by default 90%) of the probability mass residing in at most some percentage (by default 5%) of the parameter space volume. If this threshold is satisfied, the current posterior is stored and the prior reset to a uniform distribution. If not, the next observation is fed in. This procedure is repeated until the requisite number (by default, 80%) of observed data points has been used, with the final posterior being the average of every stored posterior. The posterior is calculated in this way (rather than a Bayesian multiplication through every single observed point) to avoid numerical errors that can arise if very small numbers are multiplied together. This procedure of running the Bayesian loop multiple times until the desired number of pieces of evidence are used is encapsulated in the `Model.run()` function.

Now the grid can be subdivided. The user defines a threshold probability (by default 0.001) for a grid box to possess, and all boxes with greater than this probability, as well as all their immediate neighbors, are divided in two along every parameter. `bayesim` saves a list of the new model calculations that need to be done for inference to proceed again, and the user provides these, again either as a callable or an HDF5 file. For the next round of inference, the user can also define a "bias" parameter from 0 to 0.5 (default 0) that will mix in some fraction of the posterior from the previous step with the uniform distribution over this new grid.

This overall procedure can be repeated as many times as the user wishes to get a more precise fit – eventually, the concentration of probability becomes limited by experimental rather than model uncertainty. There is also an option to define a minimum width for a box along any parameter direction, and if a box is already less than this width, `bayesim` will not subdivide in that direction.

The most flexible way to interact with `bayesim` is via Python scripting or through literate programming in a Jupyter notebook. There is also a command line interface (CLI) for users less familiar with coding in Python.

`bayesim` relies on a variety of external open-source packages. These include `numpy` [4] and `scipy` [5] for a variety of mathematical functions and vectorized implementations, `joblib` [6] for simple parallelism, `deepdish` [7] for saving and loading HDF5 files, `pandas` [8] for data manipulation, and `matplotlib` [9] for visualization.

APPLICATION EXAMPLES

Ideal Diode Model

As a first example, we fit a simple two-parameter model of solar cell current density J (as a function of voltage V and temperature T) known as the ideal diode model:

$$J(V, T) = J_L + J_0 \left(\exp \frac{qV}{nkT} - 1 \right) \tag{3}$$

where $k = 8.61733 \times 10^{-5}$ eV/K is Boltzmann's constant, by convention J_L (the light current) is negative and J_0 (the saturation current) is positive but strongly dependent on temperature, a dependence we can approximate as:

$$J_0 \approx B' T^{3/n} \exp \left(\frac{-E_{g0}}{nkT} \right) \tag{4}$$

where E_{g0} is the zero-temperature bandgap of the material. For this example, we will set this to 1.2 eV and J_L to -0.03 mA/cm^2. There are thus two parameters to be fit, the coefficient B' (which can range over a wide variety of values) and the ideality factor n (which is by definition between 1 and 2).

As validation, we generate the "observations" using the same model we'll use for fitting, with parameter values of $n = 1.36$ and $B' = 258$, and four temperature values of 150, 200, 250, and 300 K. These "observations" are shown in Figure 3a. When bayesim imports observed data, it can optionally discard data points that are very close by to each other in order to reduce computational load on modeling less informative points - this capacity is demonsrated in Figure 3b.

The lower part of Figure 3 shows the posterior distributions after the first inference run and after each round of subdivision, demonstrating that bayesim gets progressively closer to the "true" values as the loop continues iterating. In addition, on the first inference run (prior to any subdivision), an average of 43% of grid points had larger model uncertainty than experimental uncertainty, while after the second subdivision, only 2% did, while the average absolute error of the highest-probability model decreased from 0.03 after the first run to 0.007 after the second subdivision.

SnS Solar Cell

This example aims to demonstrate a more realistic application of bayesim, i.e. when the model being used to compute likelihoods is more computationally expensive. In this case, we will repeat the analysis from Ref. [1], using SCAPS [10] as the model, which takes 10-15 core-seconds for a single J-V sweep.

We start on a significantly coarser grid than Ref. [1] and show only one level of subdivision to make the example feasible to run on a personal computer. Figure 4 shows the results from this fit done in bayesim compared to the higher-resolution grid analysis done previously. Note that the inference step in full-resolution grid took a few hours, while the adaptive grid fit takes a few minutes. While the fit precision is of course lower at only one level of subdivision, the extracted values are generally in agreement and more importantly, the relationship between the mobility and lifetime (namely, that the product, i.e. the diffusion length), is constrained, emerges just as clearly.
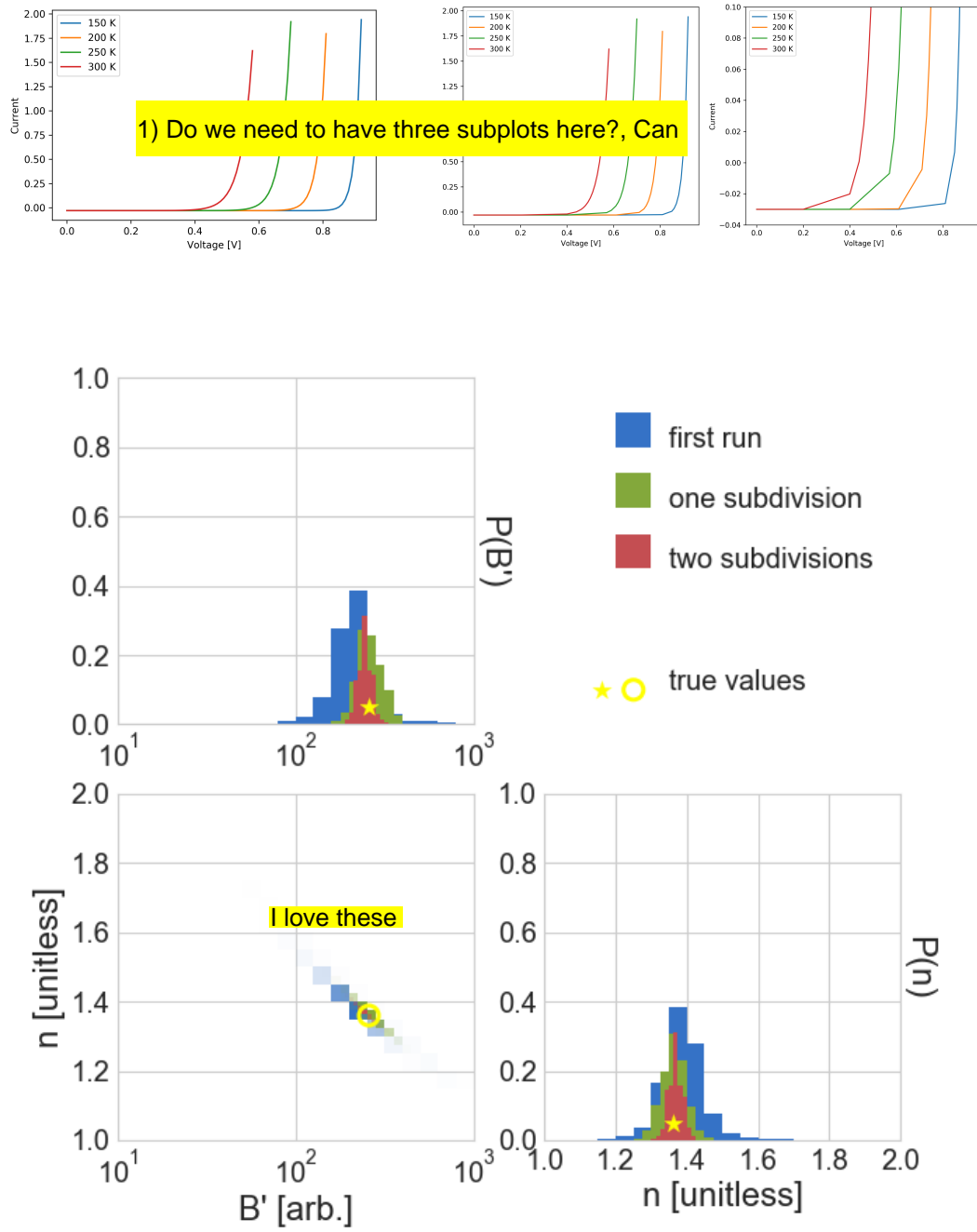
FIG. 3. a) "Observations" generated by ideal diode model, Equation 3, b) Observed data saved by `bayesim`, with zoom to show where neglected data points were, c) PMF's after the first inference run, second (one subdivision), and third (two subdivision), in blue, orange, and green, respectively. Values used to generate data indicated in red.
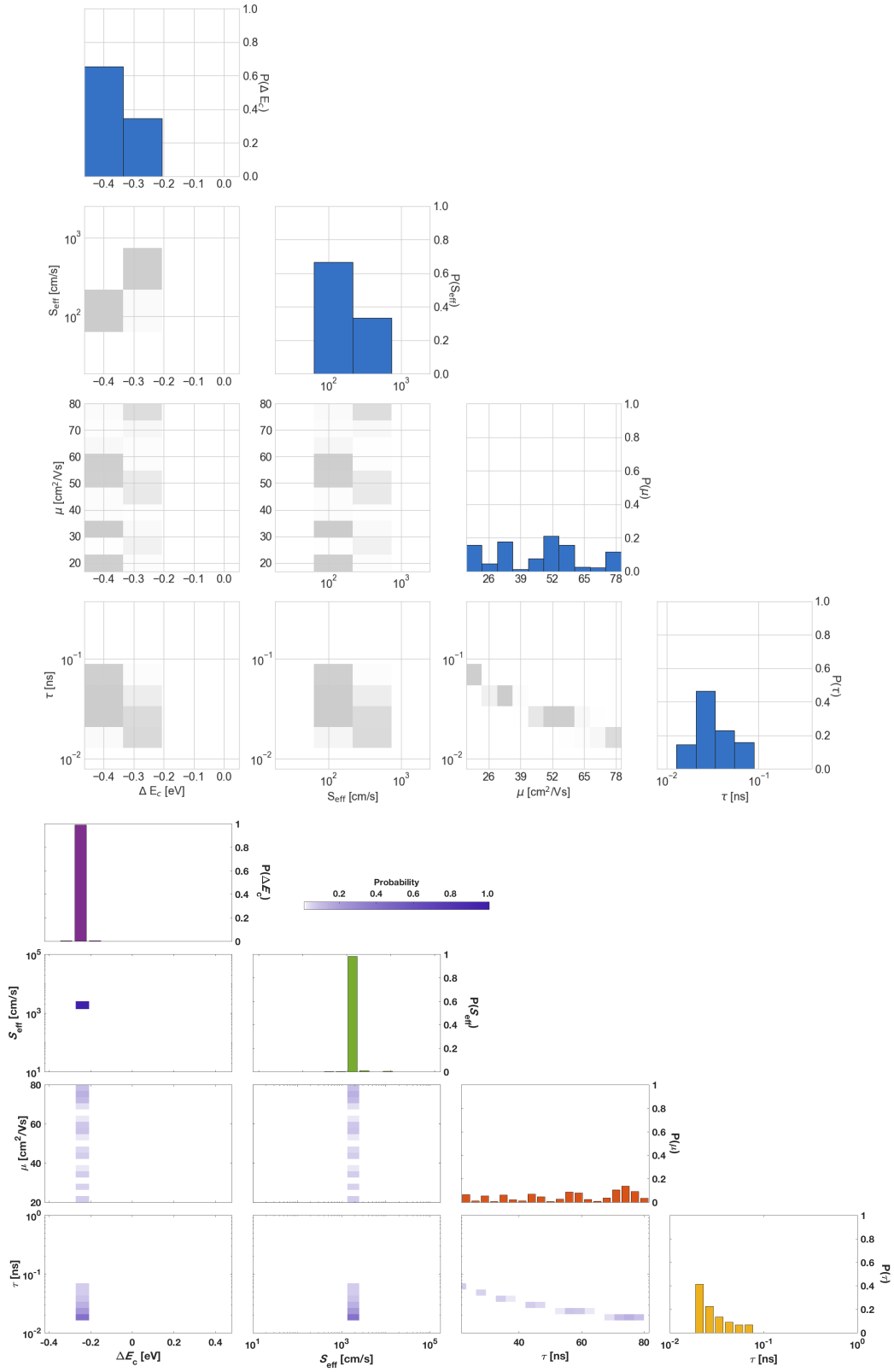
FIG. 4. Above: PMF produced by `bayesim`. Below: PMF from Reference [1]. TBD if we'll actually reproduce the plot from the Joule paper directly vs. replotting the data somehow

CONCLUSIONS

We have introduced `bayesim`, a Python-based adaptive grid sampled Bayesian inference code which is flexible to the model used to calculated likelihood as well as the parameter space over which to fit. We provided two examples of application of the code in the space of characterizing PV devices using JVT(i) measurements, as this is the authors' area of expertise. However, nothing about the code or the approach limits it to application in PV materials. Any situation where there is:

- a forward model that captures the relevant physics (particularly one that is relatively expensive to evaluate), and

- a set of measurements of the output of that model as a function of several experimental conditions

would be amenable to `bayesim`. The online documentation page [3] maintains a list of examples which will grow as the authors become aware of/implement more. This website also hosts other useful information, such as detailed documentation of every class and function in `bayesim`, a more extensive technical background section than is included here, and an updated manual for usage of the code.

ACKNOWLEDGEMENTS

APPENDIX

- include minimal code to run ideal diode example via CLI

- link to Github repo (which has installation instructions and documentation as well as list of planned future features)

---

* buonassi@mit.edu

[1] R. E. Brandt, R. C. Kurchin, V. Steinmann, D. Kitchaev, C. Roat, S. Levcenco, G. Ceder, T. Unold, T. Buonassisi, and H. Z. Berlin, "Rapid semiconductor device characterization through Bayesian parameter estimation," *Joule*, vol. 1, no. 4, pp. 843–856, 2017.

[2] R. C. Kurchin, J. R. Poindexter, D. Kitchaev, V. Vähänissi, C. del Cañizo, L. Zhe, H. S. Laine, C. Roat, S. Levcenco, G. Ceder, and T. Buonassisi, "Semiconductor parameter extraction via current-voltage characterization and bayesian inference methods," 6 2018.

[3] R. C. Kurchin and G. Romano, "bayesim." https://pv-lab.github.io/bayesim/_build/html/index.html, 2018.

[4] T. E. Oliphant, *A guide to NumPy.* Trelgol Publishing, 2006.

[5] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001.

[6] J. developers, "Joblib." https://joblib.readthedocs.io/en/latest/.

[7] G. Larsson and M. Stoehr, "deepdish." https://github.com/uchicago-cs/deepdish.

[8] W. McKinney, "Data structures for statistical computing in python," 2010.

[9] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[10] M. Burgelman, P. Nollet, and S. Degrave, "Modelling polycrystalline semiconductor solar cells," *Thin Solid Films*, vol. 361–362, pp. 527–532, 2000.