Computer Physics Communications 239 (2019) 161–165

Contents lists available at ScienceDirect

Computer Physics Communications

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

Bayesim: A tool for adaptive grid model fitting with Bayesian inference☆

Rachel Kurchin *, Giuseppe Romano, Tonio Buonassisi *

Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA

ARTICLE INFO

Article history:
Received 12 October 2018
Received in revised form 23 January 2019
Accepted 26 January 2019
Available online 18 February 2019

Keywords:
Bayesian inference
Photovoltaics

ABSTRACT

Bayesian inference is a widely used and powerful analytical technique in fields such as astronomy and particle physics but has historically been underutilized in some other disciplines including semiconductor devices. In this work, we introduce `bayesim`, a Python package that utilizes adaptive grid sampling to efficiently generate a probability distribution over multiple input parameters to a forward model using a collection of experimental measurements. We discuss the implementation choices made in the code, showcase two examples in photovoltaics, and discuss general prerequisites for the approach to apply to other systems.

Program summary

Program Title: Bayesim
Program Files doi: http://dx.doi.org/10.17632/fch5m6p3nn.1
Licensing provisions: GNU General Public License 2
Programming language: Python 3
Supplementary material: uncertainty_figure.png
Nature of problem: Many parameters in forward numerical models, e.g. for photovoltaic (PV) device behavior, are difficult to measure via direct experiment. In addition, in early-stage PV materials (and many other systems of interest), there are many unknown parameters and a desire to know their values in a small amount of time to enable high-throughput materials screening, making the time investment for direct measurement prohibitive. However, measurement of electrical properties of devices is comparatively easy, cheap, and automatable.
Solution method: We employ Bayesian inference to "invert" the device models using the high-throughput experimental data, running the model forward with many combinations of parameters and generating a probability distribution over the inputs. This enables fitting of a number of parameters limited only by the quantity of experimental data and computational power available. The code is available open-source as a Python package and includes features such as adaptive grid fitting, model uncertainty calculation, and a variety of visualization options.
Additional comments: The code is freely available on Github (https://github.com/pv-lab/bayesim) and thoroughly documented online (https://pv-lab.github.io/bayesim/_build/html/index.html).

1. 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, or Bayesian parameter estimation (BPE) 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. It has been widely used for decades in fields such as cosmology [1], particle physics [2,3], and biology [4,5], with well-developed and widely used code packages specialized to these purposes. [6,7] However, the use of Bayesian inference in materials science and related contexts is much more recent, having only emerged in the past few years [8–10].

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding authors.
E-mail addresses: rkurchin@alum.mit.edu (R. Kurchin), buonassisi@mit.edu (T. Buonassisi).

https://doi.org/10.1016/j.cpc.2019.01.022
0010-4655/

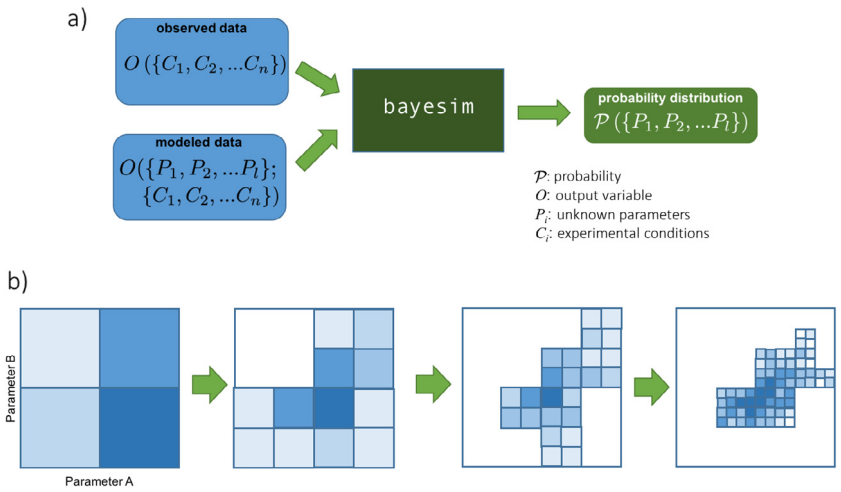


Fig. 1. (a) High-level flowchart of inputs and outputs to `bayesim`. Observed data as a function of experimental conditions and modeled data over those same conditions as well as fitting parameters are fed in, and the code produces a probability distribution over the fitting parameter values. (b) 2D schematic of adaptive grid sampling approach, illustrating progressive subdivision in higher-probability regions of parameter space.

We have previously demonstrated the value of a BPE 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 [10,11], i.e. as a replacement for direct experimental characterization via probabilistic inversion of a forward model. 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 BPE. It is designed to be easy to use by anyone with basic familiarity with Python. 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.

2. 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

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

i.e. as each new piece of evidence is fed in, the posterior probability distribution is updated to accommodate it and should in general become more concentrated into one region of parameter space. 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 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. This model uncertainty calculation is summarized in Figure S1.

A high-level summary of what `bayesim` does is shown in Fig. 1a. The flowchart shows 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 $\{P\}$. Fig. 1b 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.

3. Software architecture and interface

The basic structure of `bayesim` is shown in Fig. 2a. Detailed and up-to-date documentation of classes, functions, and example applications is maintained online. [12] 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.

Fig. 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 `bayesim` 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

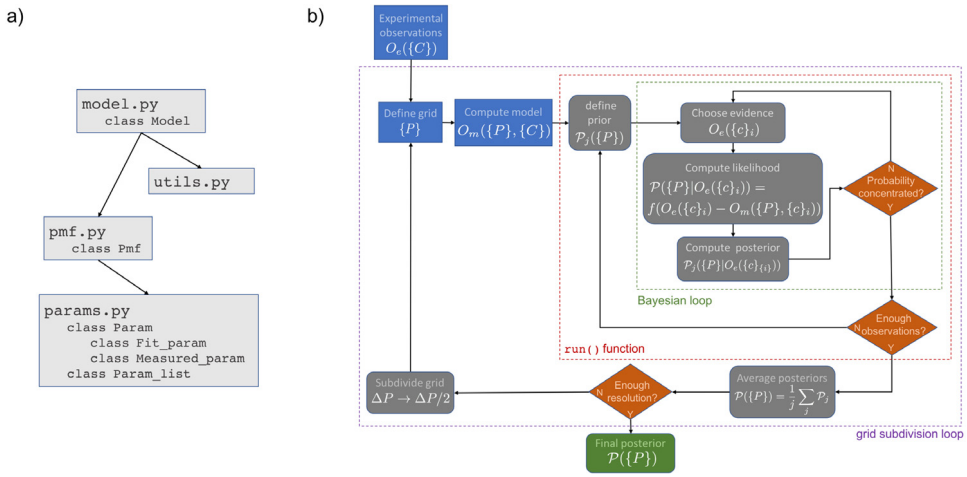


Fig. 2. (a) `bayesim` software structure. (b) Detailed flowchart of use of `bayesim`, with calculation steps and various levels of iteration indicated. Boxes in blue indicate steps that can or must be done explicitly by the user, gray steps that are done by the code with behavior tunable by the user, orange diamonds control points for staying in or leaving a loop, and green the final output . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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 uniform prior distribution over the entire 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 uncertainty 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 observations 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 into 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 "relaxation" parameter from 0 to 1.0 (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 planned for future release.

`bayesim` relies on a variety of external open-source packages. These include `numpy` [13] and `scipy` [14] for a variety of mathematical functions and vectorized implementations, `joblib` [15] for simple parallelism, `deepdish` [16] for saving and loading HDF5 files, `pandas` [17] for data manipulation, and `matplotlib` [18] for visualization.

4. Application examples

4.1. 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 will use for fitting, with parameter values of $n = 1.36$

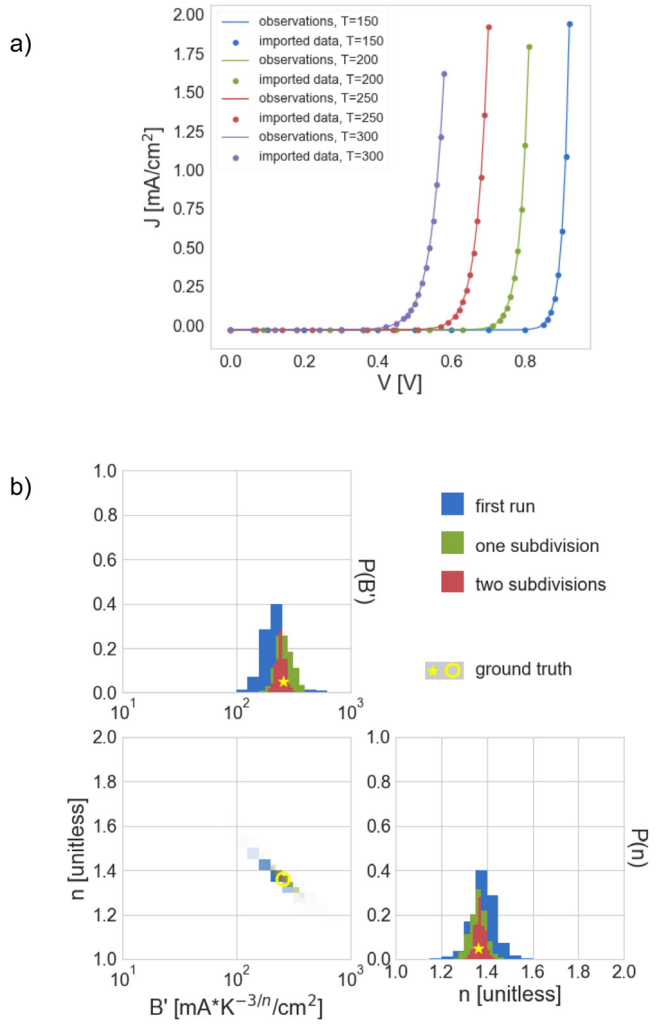


Fig. 3. (a) "Observations" generated by ideal diode model, Eq. (3), with overlaid scatter to show which points were actually imported and used by `bayesim`, (b) PMFs after the first inference run, second (one subdivision), and third (two subdivision), in blue, orange, and green, respectively. Values used to generate data indicated in red . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and $B' = 258$, and four temperature values of 150, 200, 250, and 300 K. These "observations" are shown in Fig. 3a. When `bayesim` imports observed data, it can optionally discard data points that are very close to each other in order to reduce computational load on modeling less informative points — the observed points that were actually imported (59 out of 305 total "observations") are shown as overlaid scatter dots.

Fig. 3b 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 45% of grid points had larger model uncertainty than experimental uncertainty, while after the second subdivision, only 1.9% 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.

In addition, the initial grid had 400 boxes, the singly subdivided grid 164, and the final grid (4x initial resolution) had 144, meaning the overall additional computational effort to quadruple the fit precision was less than producing the initial simulation set. In comparison, using a "brute-force" grid approach, reaching 4x precision would take 4^n times the computational load in n dimensions, i.e. 16x for this case. The exact factor of improvement is problem-dependent, but in all cases the load is significantly less than the exponential scaling of the prior approach.

4.2. SnS solar cell

This example demonstrates 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. [10], using SCAPS [19] (a coupled drift–diffusion/Poisson solver) as the model, which takes 10–15 core-seconds for a single J-V sweep.

We start on a significantly coarser grid than Ref. [10] and show only one level of subdivision to make the example feasible to run on a personal computer. Fig. 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. More importantly, the relationship between mobility and lifetime (namely that their product and hence the diffusion length is constrained) emerges just as clearly. The differences in fitted values stem primarily from the fact that the prior published analysis did not account for model uncertainty, while this feature is included in `bayesim`. This additional uncertainty causes less concentration of probability mass. We also emphasize that the grid used for demonstration here is far coarser than would be recommended for achieving a quantitatively reliable result, and was adopted for this example for computational feasibility (i.e. no HPC required) only.

5. 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 [12] maintains a list of examples which will grow as the authors become aware of/implement more. The latest version of the code can always be found in the Github repository [20], and the most recent stable and thoroughly-tested version is installable via PyPI [21]. The online documentation 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, a list of planned future features, and an updated manual for usage of the code.

Acknowledgments

R.C.K. acknowledges the funding of a Blue Waters Graduate Fellowship, USA. The experimental data acquisition shown was supported by a TOTAL SA research, USA grant funded through MITei, the MIT Energy Initiative and the computational work was supported as part of the Center for Next-Generation Materials by Design, USA, an Energy Frontier Research Center, USA funded by the U.S. Department of Energy, Office of Science. The authors also acknowledge I.M. Peters for helpful conversations.

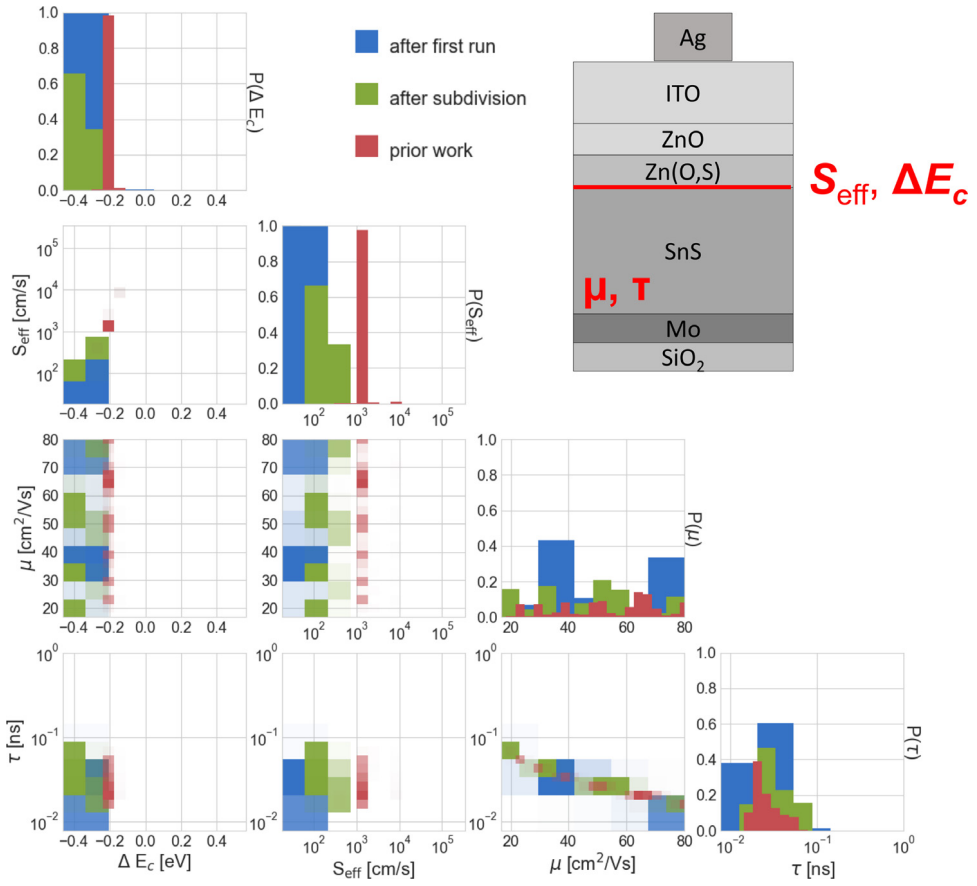


Fig. 4. PMFs produced by `bayesim`, after initial run and after one subdivision (blue and green, respectively). In red, PMF from Ref. [10]. Inset: device structure (illuminated from above) with bulk SnS (μ, τ fit) and SnS/Zn(O,S) interface (ΔE_c, S_{eff} fit) indicated . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

References

[1] R. Trotta, Contemp. Phys. 49 (2) (2008) 71–104, http://dx.doi.org/10.1080/00107510802066753.

[2] R. Trotta, F. Feroz, M. Hobson, L. Roszkowski, R.R. de Austri, J. High Energy Phys. 2008 (12) (2008) 024, http://stacks.iop.org/1126-6708/2008/i=12/a=024.

[3] J. de Blas, M. Ciuchini, E. Franco, D. Ghosh, S. Mishima, M. Pierini, L. Reina, L. Silvestrini, 37th International Conference on High Energy Physics (ICHEP), Nucl. Part. Phys. Proc. 273–275 (2016) 834–840, http://dx.doi.org/10.1016/j.nuclphysbps.2015.09.128.

[4] J.P. Huelsenbeck, F. Ronquist, R. Nielsen, J.P. Bollback, Science 294 (5550) (2001) 2310–2314, http://dx.doi.org/10.1126/science.1065889, arXiv:http://science.sciencemag.org/content/294/5550/2310.full.pdf.

[5] D.J. Wilkinson, Brief. Bioinform. 8 (2) (2007) 109–116, http://dx.doi.org/10.1093/bib/bbm007, arXiv:/oup/backfile/content_public/journal/bib/8/2/10.1093/bib/bbm007/2/bbm007.pdf.

[6] F. Feroz, M.P. Hobson, M. Bridges, Mon. Not. R. Astron. Soc. 398 (4) (2009) 1601–1614, http://dx.doi.org/10.1111/j.1365-2966.2009.14548.x.

[7] S.H. Scheres, J. Struct. Biol. 180 (3) (2012) 519–530, http://dx.doi.org/10.1016/j.jsb.2012.09.006.

[8] D. Xue, P.V. Balachandran, R. Yuan, T. Hu, X. Qian, E.R. Dougherty, T. Lookman, Proc. Natl. Acad. Sci. 113 (47) (2016) 13301–13306, http://dx.doi.org/10.1073/pnas.1607412113, arXiv:http://www.pnas.org/content/113/47/13301.full.pdf.

[9] T. Ueno, T.D. Rhone, Z. Hou, T. Mizoguchi, K. Tsuda, Mater. Discov. 4 (2016) 18–21, http://dx.doi.org/10.1016/j.md.2016.04.001.

[10] R.E. Brandt, R.C. Kurchin, V. Steinmann, D. Kitchaev, C. Roat, S. Levcenco, G. Ceder, T. Unold, T. Buonassisi, H.Z. Berlin, Joule 1 (4) (2017) 843–856, http://dx.doi.org/10.1016/j.joule.2017.10.001.

[11] R.C. Kurchin, J.R. Poindexter, D. Kitchaev, V. Vähänissi, C. del Cañizo, L. Zhe, H.S. Laine, C. Roat, S. Levcenco, G. Ceder, T. Buonassisi, Proceedings of the 2018 7th World Conference on Photovoltaic Energy Conversion, IEEE, 2018.

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

[13] T.E. Oliphant, A Guide to NumPy, Trelgol Publishing, 2006.

[14] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, 2001, http://www.scipy.org/.

[15] Joblib developers, Joblib, https://joblib.readthedocs.io/en/latest/.

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

[17] W. McKinney, Proceedings of the 9th Python in Science Conference, 2010, pp. 51–56.

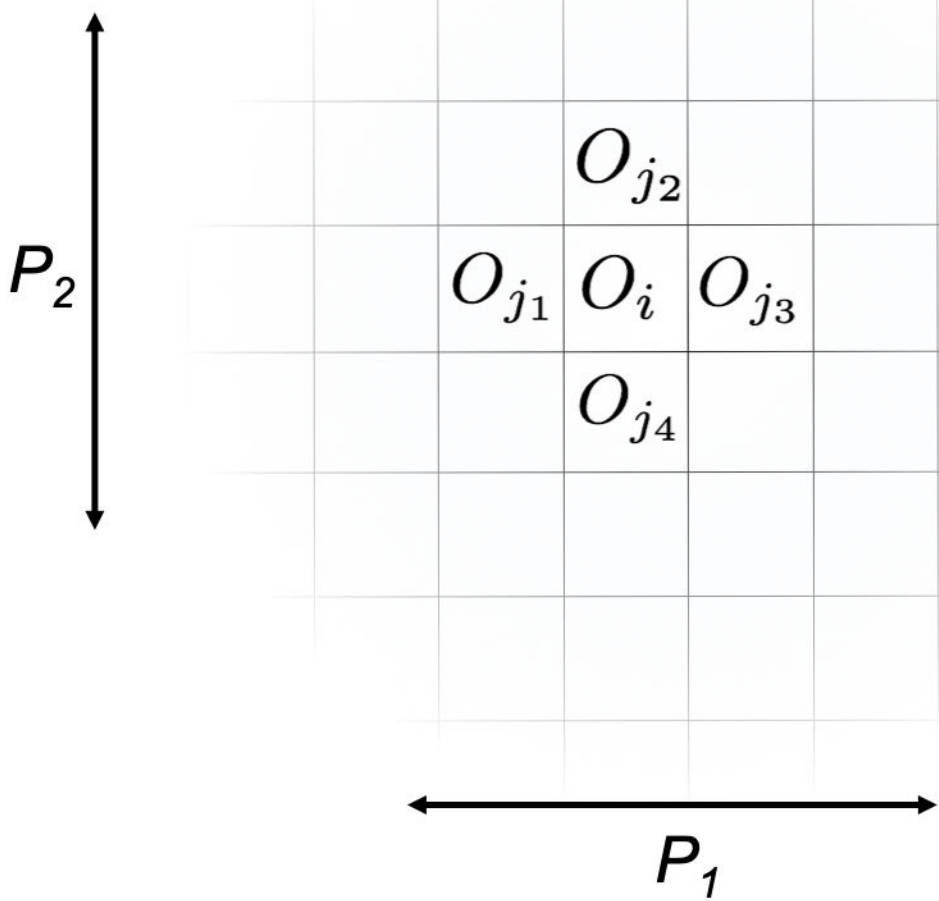
[18] J.D. Hunter, Comput. Sci. Eng. 9 (3) (2007) 90–95, http://dx.doi.org/10.1109/MCSE.2007.55.

[19] M. Burgelman, P. Nollet, S. Degrave, Thin Solid Films 361–362 (2000) 527–532, http://dx.doi.org/10.1016/S0040-6090(99)00825-1.

[20] R.C. Kurchin, G. Romano, Bayesim, https://github.com/pv-lab/bayesim.

[21] R.C. Kurchin, G. Romano, Bayesim, 2018, https://pypi.org/project/bayesim.

At a given set of condition $\{C\}_i$ and parameters $\{P\}_i$, we simulate an output O_i...



$$\text{Then, } \sigma_{i,mod} = \max_j(|O_i - O_j|)$$