bayesim Documentation

Release 0.9.5

Rachel C Kurchin and Giuseppe Romano

Oct 31, 2018

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Prerequisites

`bayesim` has been most thoroughly tested in Python 3.6.

1.2 Download

To install bayesim simply type

```
pip install bayesim
```

1.3 Usage

For more details on both usage methods described below, see the *Manual*.

1.3.1 Python / Jupyter

The easiest way to learn how `bayesim` works is by stepping through one of the *Examples*, which have well-commented Jupyter notebooks associated with them and can be run in binder with no need to install Python, `bayesim`, or any of its dependencies locally.

If you are comfortable coding in Python, these examples will also make it clear how to script using `bayesim`.

1.3.2 Command line

A command line interface will be implemented in a future release!

CHAPTER

TWO

WHY BAYESIM?

There are plenty of tools already out there for Bayesian parameter estimation. What's special/useful about `bayesim`? What is it good for? What *isn't* it good for?

I won't reinvent the wheel of the Bayesian/frequentist debate here because many smarter people have written a lot about it in other places. Instead, I'll just emphasize a couple important points, the first of which is general to the approach, and the second of which is specific to how `bayesim` implements it.

2.1 Probability distributions are nice

The output of a `bayesim` analysis is not just a set of values and uncertainties, but rather a full multidimensional probability distribution. The power of this comes when the regions of parameter space that have highest probability are not simple ellipsoidal blobs with axes along the parameter axes, but instead take on more complicated shapes that can reflect underlying tradeoffs between fitting parameters. For example, in our research group's first published paper using Bayesian parameter estimation (which used a much earlier precursor of the code that was eventually developed into `bayesim`), we found that of the four parameters we fit, two were constrained quite well, while two others didn't seem to be on an individual basis:

However, when we look at the two-parameter marginalization of the final posterior distribution (second plot from the right on the bottom row), we see that while the values of μ and τ may not have been individually well-constrained, their *product* was pinned quite well. As it happens, this product is related to a quantity known as the *electron diffusion length*, and under the conditions we measured the solar cell, other effects that μ and τ have on performance were small enough that they couldn't be decoupled from each other.

This type of insight could not be gleaned from a more traditional parameter fitting approach such as a least-squares regression and requires the ability to see the probability distribution over parameters.

2.2 Simulations are expensive

A very common approach to multidimensional Bayesian parameter estimation involves Monte Carlo (MC) sampling rather than sampling on a grid as we do here. In general, such approaches scale very well to larger numbers of dimensions and hence may have great appeal ("larger numbers" is problem dependent but for "typical" problems if you have more than 10 parameters to estimate, an MC approach is preferable). Bayesim uses adaptive grid sampling, which has two major benefits over MC sampling for relatively low-dimensional problems:

1. It can be significantly less computationally expensive (from a more formal numerical perspective, it avoids the generally large prefactor in cost estimates for MC sampling, which can overwhelm the dimensional savings when the dimensionality is small) and

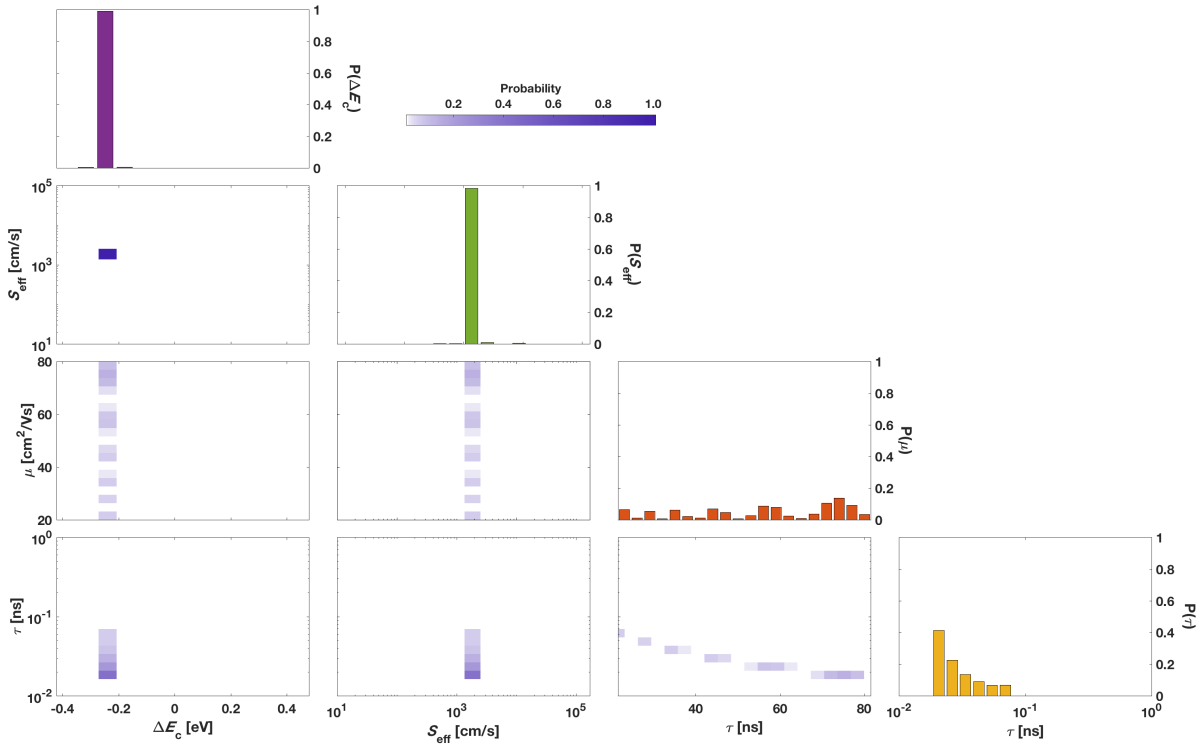


Fig. 1: Figure 3 from this paper. The ΔE_c and S_{eff} parameters are constrained well, while μ and τ exhibit a much larger spread in their individual distributions (red and yellow histograms).

2. There is significantly less uncertainty that all regions of non-negligible probability are quickly detected, since the coarsest iteration tends to identify all "hotspots" immediately, provided the *model uncertainty* associated with the sampling density is incorporated.

The take-home message here is that `bayesim`'s **approach shines in situations where the computational effort required to evaluate the likelihood is large, as when the data models are sophisticated numerical solvers, and the number of fitting parameters is (relatively) small.**

Many of the *Examples* we show here involve analytical models, however. This is done only to create examples that are tractable to run in a few seconds on a typical personal computer. In reality, while `bayesim` certainly works with these models, it is unlikely to be the most efficient approach to fitting their parameters. In addition, with an analytical model, tradeoffs such as the one between μ and τ described above are generally already apparent from the mathematical form of the model and hence the final probability distribution isn't necessarily required for such insights.

CHAPTER

THREE

TECHNICAL BACKGROUND

This page includes some references about Bayes' Theorem and Bayesian inference and discusses the particulars of the implementation of these ideas within `bayesim`.

3.1 Bayes' Theorem

There are a plethora of great explanations of Bayes' Theorem out there already, so I won't go through all the bayesics here but instead refer you to one of those linked above or any number of others you can find online or in a textbook.

Assuming you understand Bayes' Theorem to your own satisfaction at this point, let's remind ourselves of some **terminology**.

$$P(H|E) = \frac{P(H)P(E|H)}{P(E)} \tag{3.1}$$

The posterior probability of our hypothesis H given observed evidence E is the result of a Bayesian update to the prior estimate of the probability of H given the likelihood of observing E in a world where H is true and the probability of observing our evidence in the first place.

3.2 Bayesian Inference and Parameter Estimation

Note: I haven't found an online explanation of this material at a not-excessively-mathy level (I firmly believe that you don't need a lot of knowledge of mathematical terminology to understand this; it can really be done in a very visual way) so I wrote my own. If you know of another, please send it to me and I'd be happy to link to it here!

Update!! I found some nice explanations/examples in this repo! Check them out for some more material in addition to what I've included here.

Most of the examples used to explain Bayes' Theorem have two hypotheses to disginguish between (e.g. "is it raining?": yes or no). However, to use Bayes' Theorem for *parameter estimation*, which is the problem of interest here, we need to generalize to many more than two hypotheses, and those hypotheses may be about the values of multiple different parameters. In addition, we would like to incorporate many pieces of evidence, necessitating many iterations of the Bayesian calculation. These and other factors can make it confusing to conceptualize how to generalize the types of computations we do to estimate the probability of the answer to a yes-or-no question or a dice roll to a problem statement relevant to a more general scientific/modeling inquiry. I will walk through these factors here.

3.2.1 Many hypotheses, in multiple dimensions

The first step is let our hypotheses H range over more than two values. That is, rather than having H_1 = "yes, it is raining" and H_2 = "no, it is not raining", we would instead have something like H_1 = "the value of parameter A is a_1", H_2 = "the value of parameter A is a_2", etc. for as many a_n as we wanted to consider. While we could then in principle enumerate many different statements of Bayes' Theorem of the form

$$P(A = a_1|E) = \frac{P(A = a_1)P(E|A = a_1)}{P(E)} \tag{3.2}$$

$$P(A = a_2|E) = \frac{P(A = a_2)P(E|A = a_2)}{P(E)} \tag{3.3}$$

$$... \tag{3.4}$$

$$P(A = a_n|E) = \frac{P(A = a_n)P(E|A = A_n)}{P(E)} \tag{3.5}$$

this is quite cumbersome and so instead we will write

$$P(A|E) = \frac{P(A)P(E|A)}{P(E)} \tag{3.6}$$

with the understanding that this probability is not a single value but rather a function over all possible values of A. This also allows the number of equations we have to write not to explode when we want to add another fitting parameter, and instead the probability function just to be defined over an additional dimension:

$$P(A, B|E) = \frac{P(A, B)P(E|A, B)}{P(E)} \tag{3.7}$$

3.2.2 Iterative Bayesian updates

The next step is to reframe Bayes' Theorem as an explicitly iterative procedure. Imagine we've incorporated one piece of evidence E_1, resulting in a posterior probability $P(H|E_1)$. To update our posterior again given further observation E_2, we simply let this *posterior* become our new *prior*:

$$P(H|\{E_1, E_2\}) = \frac{P(H|E_1)P(E_2|H)}{P(E_2)} \tag{3.8}$$

Hopefully now it's easy to see that for *n* pieces of evidence, we can say that

$$P(H|\{E_1, E_2, ...E_n\}) = \frac{P(H|\{E_1, E_2...E_{n-1}\})P(E_n|H)}{P(E_n)} \tag{3.9}$$

3.2.3 Where does the likelihood come from?!? Data modeling!

At this point, it would be natural to say "Sure, that math all makes sense, but how do I actually *know* what that likelihood $P(E|H)$ *is?!?*"

This is where having a **model** of our experimental observations comes in. This model could take many forms - it might be a simple analytical equation, or it might be a sophisticated numerical solver. The key traits are that it can accurately predict the outcome of a measurement on your system as as function of all relevant experimental conditions as well as fitting parameters of interest.

More specifically, suppose your measurement yields some output variable O as a function of various experimental conditions $\{C\}$. Then your evidence looks like

$$O(C_1, C_2, ...C_n) \tag{3.10}$$

Suppose also that you have a set of model parameters $\{P\}$ that you wish to know the values of. That means that your posterior distribution after m observations will look something like

$$P(P_1, P_2, ...P_l|O_1, O_2...O_m) \tag{3.11}$$

where the hypotheses are sets of values of the parameters $\{P\}$, i.e., points in the fitting parameter space. Then your **model** must take the form

$$M(\{P_1, P_2, ...P_l\}, \{C_1, C_2, ...C_n\}) = O \tag{3.12}$$

Given an observation O_m at conditions $\{C_1^m, C_2^m, ...C_n^m\}$ (where the m superscript indicates specific values of the conditions rather than their full ranges), we can compute the likelihood over all parameters $\{P\}$ by evaluating our model for these conditions $\{C^m\}$ and comparing the simulated outputs $\{M(\{P\}, \{C^m\})\}$ to the measured output O_m. But then how do we know what probabilities to assign as a function of how much the measured and simulated outputs differ? Glad you asked...

3.2.4 Experimental Uncertainty

Our experimental measurement O_m will have some associated uncertainty ΔO, generally a known property of our equipment/measurement technique. Quantifying this uncertainty is key to understanding how to calculate likelihoods. Specifically, we need to introduce an **error model**. We'll use a Gaussian distribution, a very common pattern for experimental errors in all kinds of measurements:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.13}$$

where μ is the **mean**, σ is the **standard deviation**, and the term in front of the exponential is just a normalizing constant (to make sure that the probability distribution integrates to 1). The distribution looks like this:

What this means for our example is that our measurement of some value O_m^0 for our output parameter O is converted to a distribution of possible "true" values for O_m:

$$P(O_m) \propto \exp\left(-\frac{(O_m - O_m^0)^2}{2 * \Delta O^2}\right) \tag{3.14}$$

(I'm leaving off the normalization constant for convenience.)

3.2.5 Model Uncertainty

Of course, when our model function isn't a simple analytical equation but rather a numerical solver of some sort, we can't evaluate it on a continuous parameter space but we instead have to discretize the space into a grid and choose points on that grid at which to simulate. This introduces a so-called "model uncertainty" proportional to the magnitude of the variation in the model output as one moves around the fitting parameter space. This model uncertainty is calculated in `bayesim` at each experimental condition for each point in the parameter space as the largest change in model output from that point to any of the immediately adjacent points.

Then, when we compute likelihoods, we use the sum of these two uncertainties as the standard deviation of our Gaussian.

Especially if the parameter space grid is coarse, incorporating this model uncertainty is critical - if the variation in output variable from one grid point to another is significantly larger than the experimental uncertainty but this uncertainty is used as the standard deviation, it is possible that likelihood could be computed as zero everywhere in the parameter space, just because the measured output corresponded to parameters between several of the chosen sample points. And that wouldn't be very good.

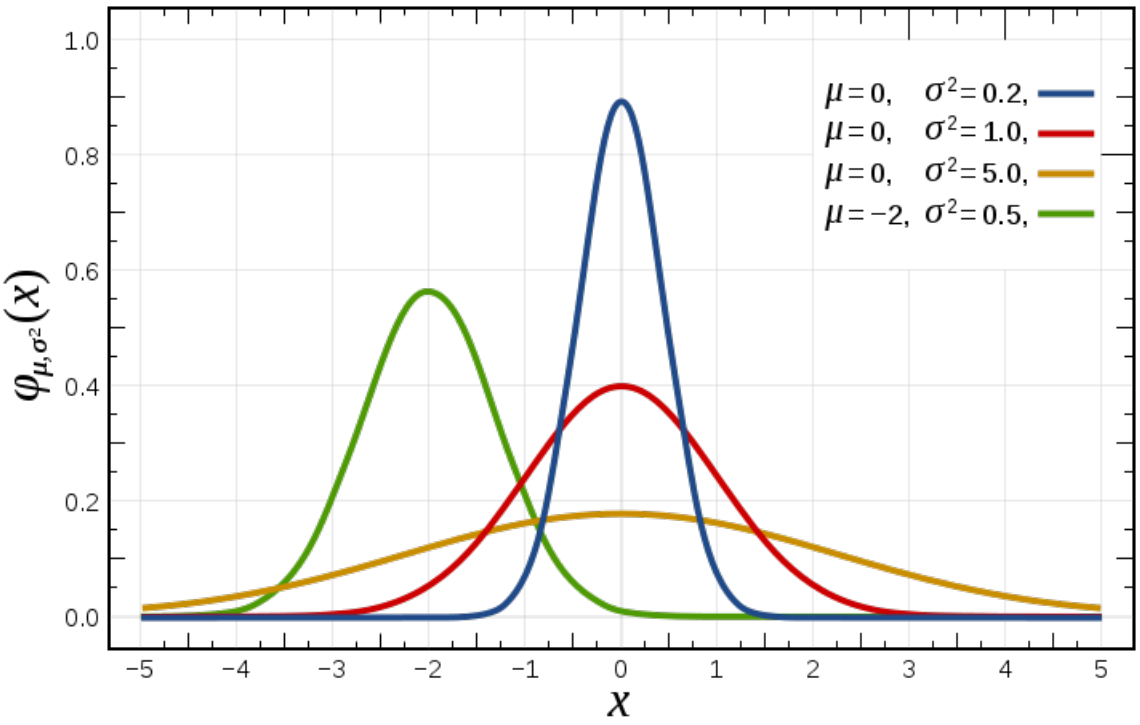


Fig. 1: You can see the impact of the two parameters - a larger σ value makes the distribution wider, while μ simply shifts the center. (Image from Wikipedia.)

3.3 An illustrative example: Kinematics

This probably all seems a bit abstract at this point, so illustrate how we do this in practice, let's use a simple example. Suppose we want to estimate the value of g, the acceleration due to gravity near Earth's surface, and v_0, the initial velocity of a vertically launched projectile (e.g. a ball tossed straight up), based on some measured data about the trajectory of the ball. We know from basic kinematics that the height of the ball as a function of time should obey (assuming that the projectile's initial height is defined as 0)

$$y(t) = v_0 t - \frac{1}{2} g t^2 \tag{3.15}$$

This function represents our **model** of the data we will measure and we can equivalently write

$$M(v_0, g; t) = v_0 t - \frac{1}{2} g t^2 \tag{3.16}$$

where we've now explicitly delineated our parameters g and v_0 and our measurement condition t.

Now let's suppose we make a measurement after 2 seconds of flight and find that $y(2) = 3$, with an uncertainty in the measurement of 0.2. Recalling our Gaussian error model from above, we can write

$$P(y(2)) \propto \exp\left(-\frac{(y(2) - 3)^2}{2 * 0.2^2}\right) \tag{3.17}$$

(Assume model uncertainty is negligible.) But what we *really* want is a probability distribution over our parameters, not over the measurement value itself. Fortunately, our model function lets us do just that! We can translate our

distribution over possible measured values into one over possible parameter values using the model function:

$$P(v_0, g|y(2) = 3 \pm 0.2) \propto \exp\left(-\frac{(M(v_0, g; 2) - 3)^2}{2 * 0.2^2}\right) \tag{3.18}$$

$$\propto \exp\left(-\frac{(2v_0 - 2g - 3)^2}{0.08}\right) \tag{3.19}$$

Now we can visualize what that distribution looks like in "v_0-g" space:

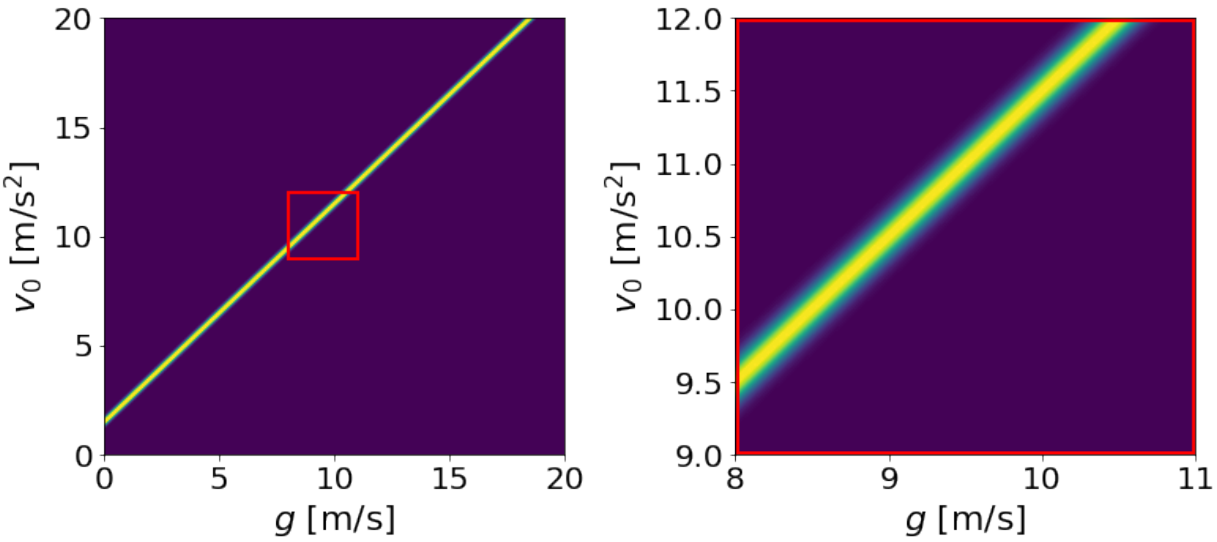


Fig. 2: On the left, the probability distribution over a wide range of possible values. On the right, zoomed in to near the true value of g to show Gaussian spread.

Another way we might want to visualize would be in the space of what the actual trajectories look like:

So we can see that what the inference step did was essentially "pin" the trajectories to go through (or close to) the measured point at (*t*,*y*)=(2.0,3.0).

Now let's suppose we take another measurement, a short time later: *y(2.3)=0.1*, but with a larger uncertainty, this time of 0.5. Now we return to Bayes' Theorem - our prior distribution will be the conditional distribution from Equation (**??**) above, and the likelihood will be a new conditional distribution generated in exactly the same way but for this new data point. What does the posterior look like?

As we would expect, we're starting to zero in on a smaller region. And how about the trajectories?

As expected, we've further winnowed down the possible trajectories. If we continued this process for more and more measurements, eventually zeroing in on the correct values with greater and greater precision.

To see this example as implemented in `bayesim`, check out the *Examples* page!

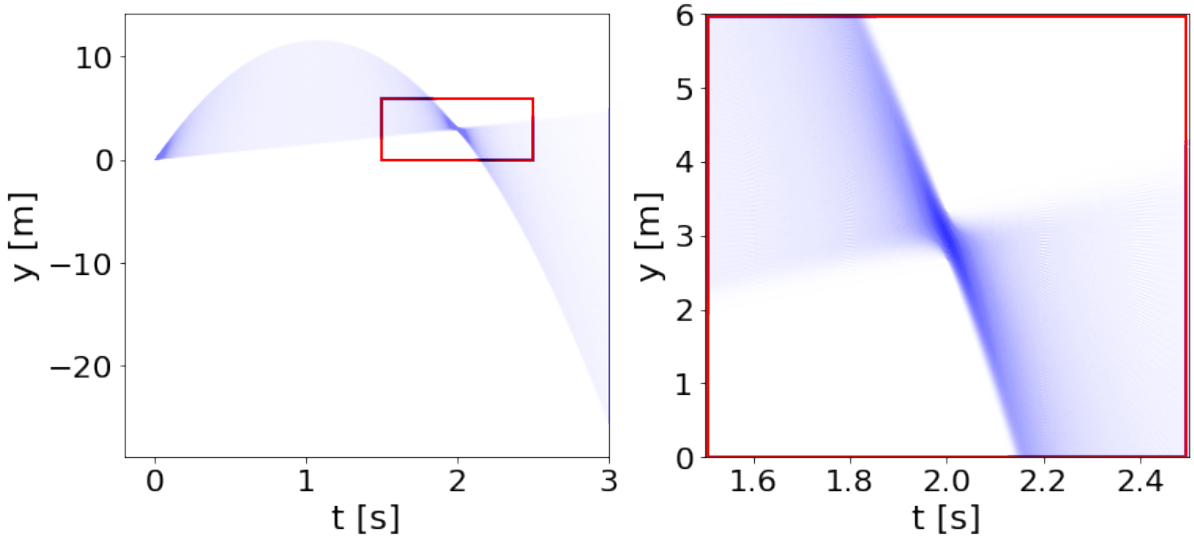


Fig. 3: On the left, $y(t)$ trajectories from $t = 0$ to $t = 3$. On the right, zooming in on the region indicated to see spread around *y(2)=3*.

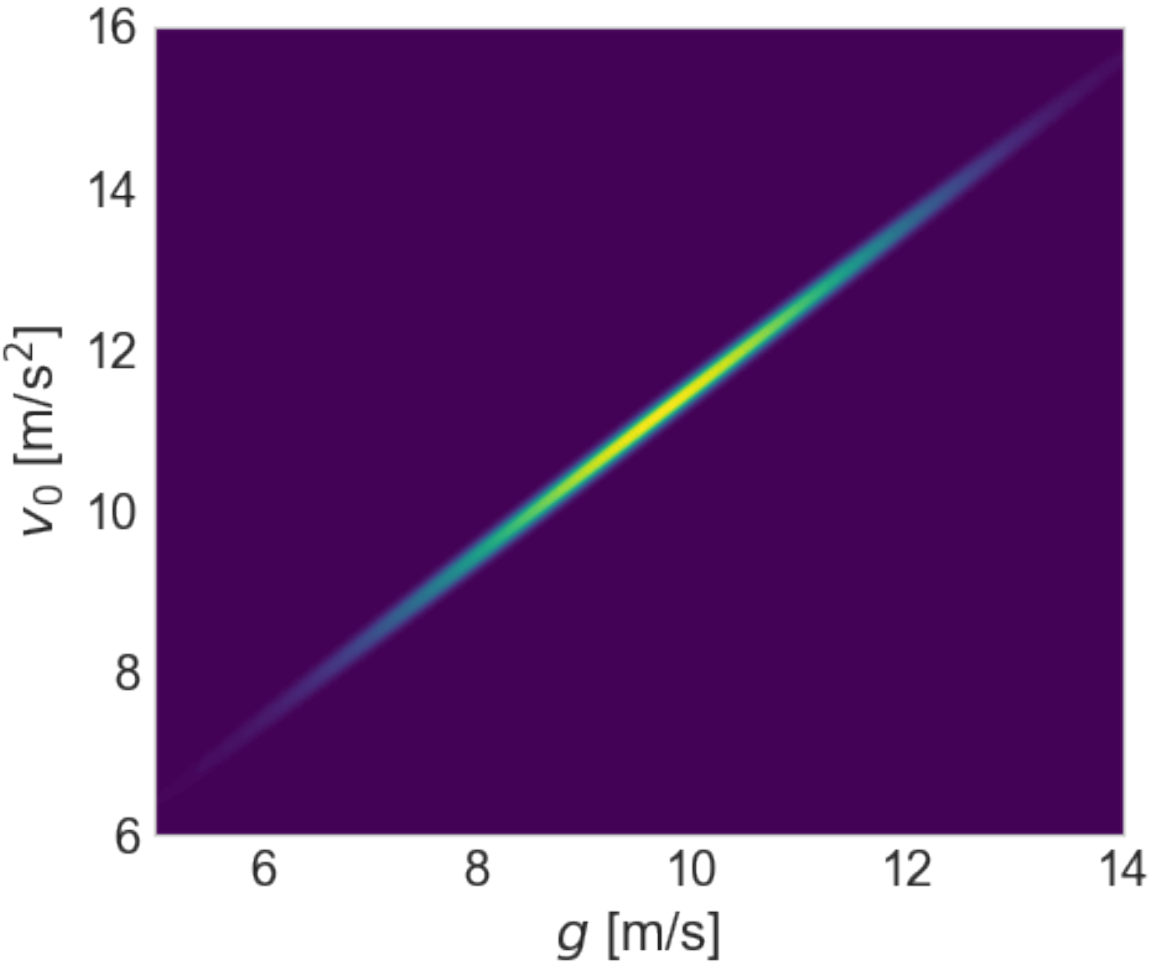


Fig. 4: (Note that the axis limits are smaller than above)

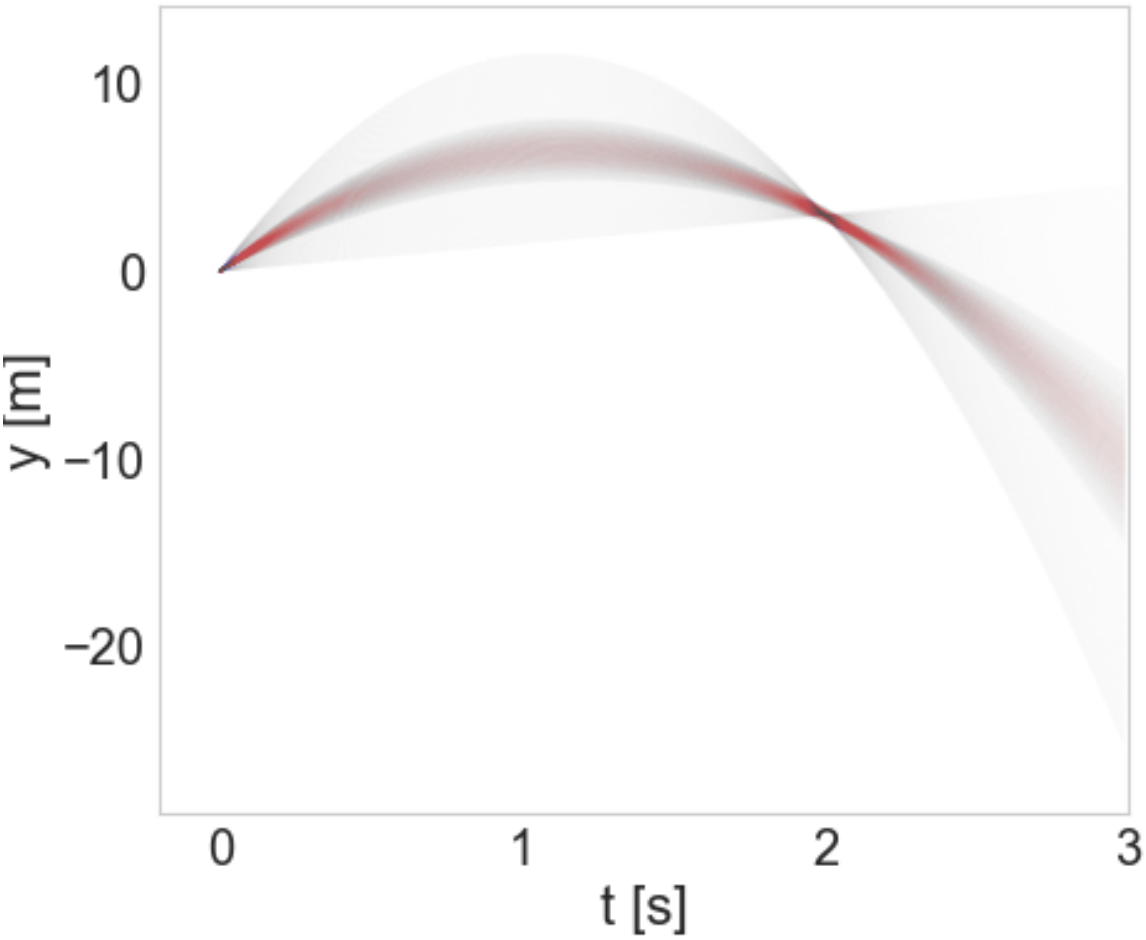


Fig. 5: Newly refined set of trajectories shown in red, overlaid on (paler) larger set from the previous step.

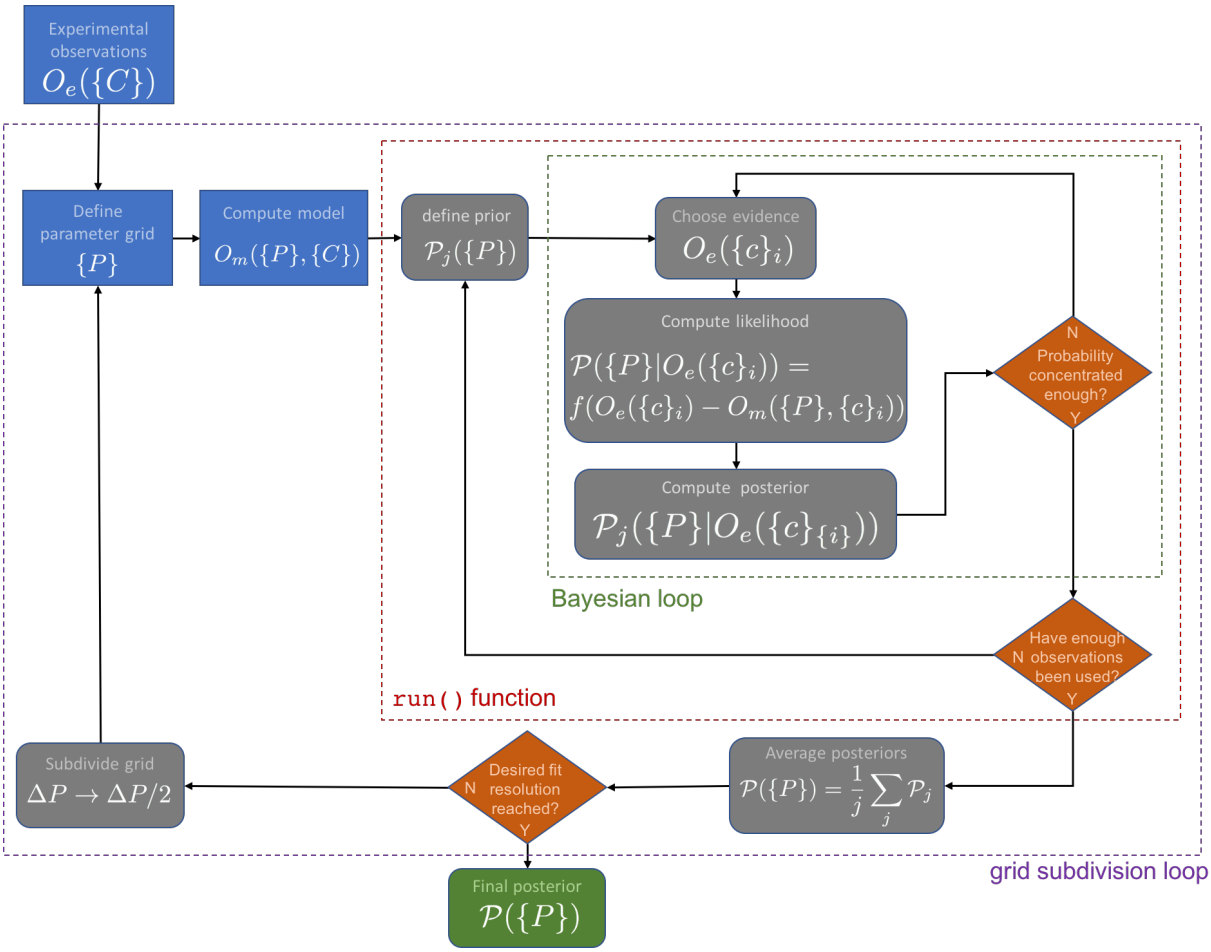
CHAPTER

FOUR

MANUAL

4.1 Overview

The basic procedure of performing a parameter fit with `bayesim` is as follows:



On this page, we will dive into each part of this flowchart in detail to understand what is happening and what options exist to tweak `bayesim`'s behavior. The code snippets will be assuming that we are modeling a solar cell's output current J with two experimental conditions: voltage V and temperature T.

The first step is to initialize a `Model` object. Several of the next steps can be rolled into this initialization using keywords, but the only required input is the name of the output variable. For example:

```
import bayesim.model as bym
m = bym.Model(output_var='J')
```

4.2 Attaching Observations

The experimental observations should reside in an HDF5 file with columns for each experimental condition and the output variable. Optionally, there may also be a column for experimental uncertainty. In addition, one should specify one of the experimental conditions (EC's) to be plotted on the x-axis when visualizing data later on using the `ec_x_var` keyword. The data are attached using:

```
m.attach_observations(obs_data_path='obs_data.h5', ec_x_var='V')
```

If this column isn't present, the `fixed_unc` keyword must be passed with a value to use as experimental uncertainty for every data point:

```
m.attach_observations(obs_data_path='obs_data.h5', ec_x_var='V', fixed_unc=0.02)
```

By default, to save computational time, `bayesim` will not import all data points, but rather attempt to maintain some minimum spacing between points such that each piece of evidence is more likely to contribute to constraining the posterior distribution. This behavior can be turned off by passing `keep_all=True` to the `attach_observations()` function, and can be tuned using the `max_ec_x_step` and `thresh_dif_frac` keywords. `thresh_dif_frac` is given as a fraction of the range of values of the output variable and defaults to 0.01. It defines the minimum difference in output variable along the x-axis EC (at fixed values of the other EC's) for which to save a point. `max_ec_x_step` is the largest step to take in the previously defined x-axis EC before saving a point anyway even if it *doesn't* differ by that threshold amount, and defaults to 5% of the range of values of the x-axis EC. Let's look at an example case where we use the default values for these parameters:

4.3 Defining Parameters

It is possible to predefine the parameter grid by directly constructing a `Param_list` object and passing it to the `Model` constructor, although this is not necessary, because `bayesim` is capable of determining these directly from the model data file (see next step). However, if one wants to use `bayesim` to generate the list of model points that need to be simulated, this approach is useful. Consider the *ideal diode* example, where the parameters to be fit were B' and n:

```
import bayesim.model as bym
import bayesim.params as byp
pl = byp.Param_list()
pl.add_fit_param(name='Bp', display_name="B'", val_range=[10,1000], spacing='log',␣
↪length=20, units='arb.')
pl.add_fit_param(name='n', val_range=[1,2], length=20, min_width=0.01)
m = bym.Model(params=pl, output_var='J')
```

The code block above sets the ranges of values, spacing (linear by default, logarithmic as an option), and number of divisions (10 by default) for the two parameters, and initializes the model object using this parameter grid. We can also set a display name for plot labeling (TeX input is accepted in this field), units (also used in plotting, defaults to 'unitless'), and minimum width beyond which a grid box won't be subdivided along this dimension (defaults to 1% of the value range in the appropriate spacing). Some of these parameters can also be set after data import using the `set_param_info()` function, as in:

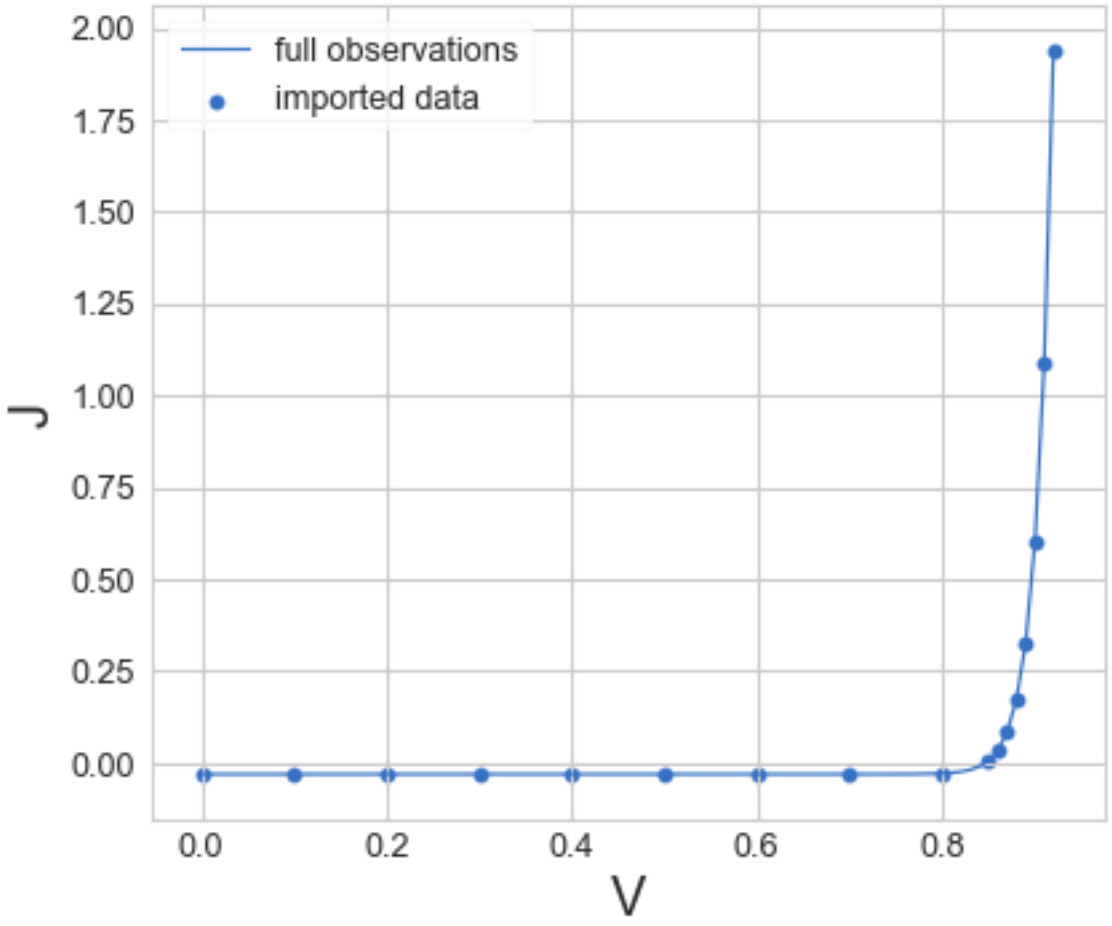


Fig. 1: Note how in the 'flatter' region of the curve, the spacing along the x-axis of imported points is wider than in the 'steeper' region.

```
m.set_param_info('J', units='mA/cm$^2$')
m.set_param_info('T', display_name='temperature', units='K')
```

4.4 Attaching Modeled Data

Next, we need to attach the modeled data. If you're using `bayesim` to tell your model what points to run, you can call `list_model_pts_to_run()` to write out an HDF5 file to pass to your forward model:

```
m.list_model_pts_to_run(fpath='model_inputs.h5')
```

You can also directly interface with your model by passing a Python callable, as in:

```
m.attach_model(mode='function', model_data_func=model_func)
```

where `model_func` is a callable in your namespace that accepts dictionaries of inputs, one with keys of the EC's and one of the fitting parameters (see *ideal diode* example).

However, typically, the model data will be precomputed and residing in an HDF5 file which will be attached to the model object:

```
m.attach_model(mode='file', model_data_path='model_data.h5')
```

The *model uncertainty* also needs to be computed. This can be done either with a separate function call to `calc_model_unc()`, or in a single step when attaching the model data:

```
m.attach_model(mode='file', model_data_path='model_data.h5', calc_model_unc=True)
```

The model uncertainty calculation can be somewhat expensive for large grids and will be parallelized by default on Unix-based systems.

4.5 Performing the Inference

Once experimental and model data and their associated uncertainties have been defined and the fitting parameters either explicitly specified or determined from the data, we can do Bayesian inference! The inference is performed by the `run()` function. Details about what calculations are actually done and what they mean can be found on the *Technical Background* page; here we will focus on the mechanics of the code.

First, a bounded uniform prior (equal probability in every grid box) is defined. Next, a piece of evidence (one experimental measurement, i.e. in this example a (V, T, J) tuple) is chosen at random. The likelihood at each point in parameter space is computed conditioned on this piece of evidence as a Gaussian in the difference between the measured output and the simulated output at that point, with a standard deviation equal to the sum of the experimental error for that observation and the model uncertainty at that point:

$$\mathcal{P}(\{P\}|O_e(\{c\}_i)) \propto \exp\left(\frac{-(O_e(\{c\}_i) - O_m(\{P\},\{c\}_i))^2}{2(\sigma_e(\{c\}_i) + \sigma_m(\{P\},\{c\}_i))^2}\right) \tag{4.1}$$

In the *ideal diode* example, this means more specifically that

$$\mathcal{P}(B', n|J_{\text{meas}}(V_i, T_i)) \propto \exp\left(\frac{-(J_{\text{meas}}(V_i, T_i) - J_{\text{mod}}(B', n, V_i, T_i))^2}{2(\sigma_{\text{meas}}(V_i, T_i) + \sigma_{\text{mod}}(B', n, V_i, T_i))^2}\right) \tag{4.2}$$

This likelihood is multiplied with the prior and normalized to compute the posterior.

Next, bayesim will check if the posterior distribution is "concentrated" enough. This concentration is defined by two optional parameters passed to run(), th_pm and th_pv, both on the interval (0,1) and defaulting to 0.9 and 0.05, respectively. The posterior is sufficiently concentrated is th_pm of the probability mass resides in th_pv of the parameter space. (Entropy-based thresholding is planned as an option for a future release)

If the posterior is not sufficiently concentrated, the posterior is set to the prior, another piece of evidence chosen, and another Bayesian update performed until it is. Once the concentration threshold is met, the posterior is saved and another check is done for whether enough pieces of evidence have been used. The requisite number to use is defined by the min_num_pts keyword in the run() function and defaults to 80% of the total number of observation data points imported. If not enough points have been used, the current posterior is saved, a new uniform prior is set and the process above repeated as many times as necessary for sufficient points to be used. The final posterior is then the average of all computed posteriors. bayesim will inform you how many posteriors were averaged and how many observed data points were used.

4.6 Visualizing the Output

bayesim has a variety of capacities for data visualization.

4.6.1 Plotting the PMF

Visualizing the posterior distribution (probability mass function, or PMF) is done using the visualize_probs() function. It takes an optional argument of a filepath to save the image, and can also highlight a specific point in the parameter space to compare to using the true_vals keyword.

4.6.2 Visualizing the Grid

To visualize the current state of the parameter space grid, use visualize_grid(), which can also accept a path to save the imge as well as the true_vals keyword to highlight a particular point.

4.6.3 Comparing the Data

The comparison_plot() function can directly compare modeled to simulated data. It will produce a number of plots with the output variable on the y-axis and the previously specified EC x-variable on the x-axis. It accepts a few optional keywords:

ec_vals dict or list of dict, specific experimental conditions at which to plot (values for x-axis EC are ignored)

num_ecs int, number of (randomly chosen) EC's at which to plot (ignored if previous option is provided)

num_param_pts int, number of parameter space points for which to plot modeled data (will choose the most probable)

In addition to a filepath to save the image as well as a flag to return average errors.

4.7 Subdividing the Grid

Once you have taken a look at your posterior PMF and compared observed and highest-probability modeled data, you'll have a sense for whether you'd like to go to a higher fit precision by subdividing the parameter space grid. This can be done via a call to the subdivide() function, which accepts one optional argument of threshold_prob, the minimum probability for a box to have in order to be subdivided. It defaults to 0.001. bayesim will then divide

every grid box meeting this threshold *as well as every box immediately neighboring it* into two along every parameter dimension, unless it is already narrower than the minimum width defined for that parameter.

`bayesim` will inform you how many boxes were subdivided and also automatically save a file of the new points that need to be simulated using the model in order to perform another inference run. This new model data can be attached exactly as described *above* using the `attach_observations()` function, and then inference can proceed again.

4.8 Miscellaneous

4.8.1 Saving a state file

At any point during an analysis, the state of your `Model` object can be saved using the `save_state()` function, and can be reloaded again using keywords in the `Model` constructor:

```
m.save_state(filename='statefile.h5')
m = bym.Model(load_state=True, state_file='statefile.h5')
```

This is useful, e.g., if you're working interactively and want to be able to pick up again later without rerunning all the same code, or if you'd like to continue work on a different machine.

4.8.2 Handling missing data

Sometimes things go run when running a large number of simulations. `bayesim` can handle cases where simulated data is missing (see the *SnS example*). When computing likelihoods, if the simulated data for the EC in question isn't present, it will just use what the uniform distribution probability would be for that point. The output of the `run()` function will inform you how many times this happened on average over each Bayesian loop.

CHAPTER

FIVE

EXAMPLES

This page is a (growing) list of example applications of `bayesim`, attempting to showcase the variety of applicability as well as ways to run the code.

5.1 Ideal diode solar cell

The most self-contained example. Available to run in two different ways:

5.1.1 Jupyter notebook

To run in Jupyter:

- Download the Github repository and run from your local machine using Jupyter

OR

- Run in the cloud using Binder! (MIT)/MIT/Bayes$project/bayesim/docs/_build/doctrees/images/https/mybinder.org/badg

5.1.2 Command line

To run from the command line:

1. Download the Github repository (e.g. using `git clone`) to your local machine.
2. Navigate to the folder `bayesim/examples/command_line/`.
3. . . . to be continued. . . (this example is not completely working yet)

5.2 Kinematics

Probably the simplest example, explained on the *Technical Background* page. It exists in the Github repo under `examples/kinematics/kinematics.py` and you can run it all in one go from a terminal (`python kinematics.py`) or step-by-step in an IDE like Spyder.

5.3 Tin Sulfide (SnS) solar cell

An example using an actual numerical model – in this case, reproducing the fit (from this paper) of four material and interface properties in a tin sulfide solar cell. Also in Jupyter notebook form:

- Download the Github repository and run from your local machine using Jupyter

OR

- Run in the cloud using Binder! (MIT)/MIT/Bayes$project/bayesim/docs/_build/doctrees/images/https/mybinder.org/badg

CHAPTER

SIX

FUTURE FEATURES

On this page, we maintain a list of planned future features of `bayesim`. They're listed under a variety of categories, in loosely descending order of priority within each.

6.1 Visualization

- visualizing model uncertainty on parameter grid
- visualizing measured vs. simulated data with experimental and model uncertainty
- option to generate a plot of which particular observed data was used in an inference run

6.2 Other New Capabilities

- option for entropy-based thresholding as well as *th_pm* and *th_pv* in `run()` function
- capability to pass a DataFrame objects to `attach_model()` and `attach_observations()` functions instead of just filepaths
- multiple output variables
- completely "closed-loop" version of `run()` function that does subdivision as well (need a way to save a handle to a model function for running new simulations)
- interpolation of simulated data
- alternative error models
- alternative initial sampling states other than grids

6.3 Efficiency/Speedup

- speed up `list_model_pts_to_run()` function
- do some more code profiling to identify other particularly slow functions/components
- more parallelism generally

6.4 Interfaces

- finish developing command-line interface
- build a GUI!

CHAPTER

SEVEN

CITING BAYESIM

If you use the code, please consider citing

[placeholder]