DNest4: Diffusive Nested Sampling in **C++** and **Python**

Brendon J. Brewer
The University of Auckland

Daniel Foreman-Mackey
University of Washington
Flatiron Institute

Abstract

In probabilistic (Bayesian) inferences, we typically want to compute properties of the posterior distribution, describing knowledge of unknown quantities in the context of a particular dataset and the assumed prior information. The marginal likelihood, also known as the "evidence", is a key quantity in Bayesian model selection. The diffusive nested sampling algorithm, a variant of nested sampling, is a powerful tool for generating posterior samples and estimating marginal likelihoods. It is effective at solving complex problems including many where the posterior distribution is multimodal or has strong dependencies between variables. **DNest4** is an open source (MIT licensed), multi-threaded implementation of this algorithm in C++11, along with associated utilities including: (i) 'RJObject', a class template for finite mixture models; and (ii) a Python package allowing basic use without C++ coding. In this paper we demonstrate **DNest4** usage through examples including simple Bayesian data analysis, finite mixture models, and approximate Bayesian computation.

1. Introduction

Bayesian inference, where probability theory describes degrees of logical implication or subjective certainty, provides a powerful general basis for data analysis (O'Hagan and Forster 2004; Sivia and Skilling 2006). The result of such an analysis is typically posterior probabilities of various hypotheses, or a joint posterior probability distribution for the values of unknown parameters. Throughout this paper we denote probability distributions using a lower case p and probabilities with an upper case P.

In a standard Bayesian inference situation, the posterior distribution for parameters θ given

data D, within the context of prior information M, is

$$p(\theta|D, M) = \frac{p(\theta|M)p(D|\theta, M)}{p(D|M)}$$

or

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{marginal likelihood}}.$$

If prior information I (dropped hereafter) implies a set of possible "models" $\{M_i\}$, rather than a single one M, the posterior model probabilities are given by

$$P(M_i|D) = \frac{P(M_i)p(D|M_i)}{\sum_j P(M_j)p(D|M_j)},$$

where

$$p(D|M_j) = \int p(\theta_j|M_j)p(D|\theta_j, M_j)\, d\theta_j$$

is the marginal likelihood of model j, equal to the expected value of the likelihood function with respect to the prior distribution. This kind of calculation is often called "model selection" or "model averaging", and the results are often presented as ratios of marginal likelihoods, known as Bayes factors. When discussing computational matters, the prior distribution for parameters is often written $\pi(\theta)$, the likelihood $L(\theta)$, and the marginal likelihood Z. A popular alternative name for the marginal likelihood, which emphasizes its role in Bayesian model averaging, is the "evidence".

Nested sampling (NS; Skilling 2006) is a Monte Carlo method whose main aim is to calculate Z. However, it can also be used to generate samples to represent the posterior distribution $\pi(\theta)L(\theta)/Z$, or any other distribution proportional to $\pi(\theta)\Phi\left[L(\theta)\right]$ where Φ is any monotonic function. This latter property makes nested sampling particularly useful for statistical mechanics calculations (Pártay, Bartók, and Csányi 2010; Baldock, Pártay, Bartók, Payne, and Csányi 2016), where the "canonical" family of distributions proportional to $\pi(\theta)L(\theta)^\beta$ is of interest. In such applications, $L(\theta)$ is usually equivalent to $\exp(-\text{energy})$. Nested sampling is particularly efficient for this, since only a single run (exploring regions of higher and higher L) is needed, and different canonical distributions can be obtained by re-weighting the output points.

A defining feature of nested sampling is that it works with a sequence of *constrained prior* distributions, proportional to π but restricted to regions of the parameter space where $L(\theta)$ is above some threshold ℓ:

$$p(\theta; \ell) = \frac{\pi(\theta)\mathbb{1}\left[L(\theta) > \ell\right]}{X(\ell)}, \tag{1}$$

where

$$X(\ell) = \int \pi(\theta)\mathbb{1}\left[L(\theta) > \ell\right] d\theta$$

is the amount of prior mass which has likelihood greater than ℓ, and $\mathbb{1}()$ is the indicator function which takes the value 1 if the argument is true and 0 otherwise. In the standard

nested sampling framework, the sequence of ℓ values is selected so that $X(\ell)$ shrinks by a factor $\approx e^{-1/N}$ per iteration, where N is the number of particles used. This geometric compression of the parameter space is a defining feature of nested sampling.

Sampling from the constrained priors (Equation 1) is required, and Markov chain Monte Carlo (MCMC) is a popular method of doing this, although alternatives exist (e.g., Feroz, Hobson, and Bridges 2009; Handley, Hobson, and Lasenby 2015). Diffusive nested sampling (DNS; Brewer, Pártay, and Csányi 2011) is an alternative to nested sampling for problems where MCMC is the only viable sampling method. DNS is based on the Metropolis algorithm, and evolves one or more particles in the parameter space, along with an integer index variable j for each particle, to explore the following joint distribution:

$$
\begin{aligned}
p(\theta, j) &= p(j)p(\theta|j) \\
&= w_j \times \frac{\pi(\theta)\mathbb{1}\left[L(\theta) > \ell_j\right]}{X(\ell_j)},
\end{aligned}
\tag{2}
$$

where the $\{\ell_j\}$ are a sequence of increasing likelihood thresholds or *levels*, $\ell_0 = 0$, and $\{w_j\}$ is the marginal distribution for j. The marginal distribution for θ is then a *mixture* of constrained priors:

$$
p(\theta) = \pi(\theta) \sum_{j=0}^{j_{\max}} \frac{w_j \mathbb{1}\left[L(\theta) > \ell_j\right]}{X(\ell_j)}.
\tag{3}
$$

The DNS algorithm consists of two stages. In the first stage, the particle(s) are initialized from the prior $\pi(\theta)$, equivalent to the mixture of constrained priors with a single level whose log-likelihood threshold is $-\infty$. The mixture of constrained priors evolves by adding new levels, each of which compresses the distribution by a factor of about $e \approx 2.71818$ (see Brewer *et al.* (2011) for details). In this stage, the mixture weights $\{w_j\}$ are set according to

$$
w_j \propto \exp(j/\lambda),
\tag{4}
$$

where λ is a scale length. This enhances the probability that particles stay close to the highest likelihood regions seen. The second stage sets the mixture weights to be approximately uniform ($w_j \propto 1$), with some tweaks described by Brewer *et al.* (2011).

The mixture of constrained priors tends to be easier to sample than the posterior, as the prior is always a mixture component, allowing the MCMC chain to mix between different modes in some circumstances. The marginal likelihood estimate can also be more accurate than standard MCMC-based nested sampling (Brewer *et al.* 2011), as less information is discarded. DNS has been applied several times in astrophysics (e.g., Pancoast, Brewer, and Treu 2014; Huppenkothen *et al.* 2015; Brewer and Donovan 2015) and was recently used in a biological application (Dybowski, Restif, Goupy, Maskell, Mastroeni, and Grant 2015).

There are several ways of using **DNest4**. After installing the software (Section 4), you can implement a model as a C++ class (Section 8) and compile it to create an executable file to run **DNest4** on that problem. This method offers full control over the design of your class, and allows the opportunity of optimizing performance by preventing a full re-computation of the log-likelihood when only a subset of the parameters has been changed.

Alternatively, the Python (Van Rossum *et al.* 2011) bindings (Section 11) allow you to specify a model class in Python, and run **DNest4** entirely in the Python interpreter without having to invoke the C++ compiler.

2. Relation to other algorithms

The diffusive nested sampling algorithm, and its implementation in **DNest4**, has advantages and disadvantages compared to other Bayesian computation algorithms and software.

In the nested sampling sphere, the simple C implementation given by Skilling (2006) is useful for understanding the classic nested sampling algorithm. Similarly to **DNest4**, it is the user's responsibility to implement the MCMC moves used. The DNS algorithm produces more accurate estimates of Z and ought to outperform classic nested sampling on multimodal problems.

MultiNest (Feroz *et al.* 2009) is a Fortran nested sampling implementation that dispenses with MCMC for generating new particles from constrained prior distributions. Instead, the new particles are generated by making an approximation to the constrained prior using ellipsoids. This tends to work well in low to moderate dimensional parameter spaces (up to ~ 40). In higher dimensions, **DNest4** is more useful than **MultiNest** since it is based on MCMC. We expect **MultiNest** to be more useful than **DNest4** only on low to moderate dimensional problems with slow likelihood evaluations, where **DNest4**'s reliance on MCMC becomes costly. The more recent **POLYCHORD** (Handley *et al.* 2015) combines **MultiNest**-like methods with MCMC.

Outside of nested sampling, the popular **JAGS** (Plummer 2003) and **Stan** (Carpenter *et al.* 2017) packages are more convenient than **DNest4** for specifying models, by virtue of their model-specification languages. For users who are interested only in the posterior distribution and do not need the marginal likelihood, these are very useful. However, being based on Gibbs sampling and Hamiltonian MCMC respectively, they can run into difficulty on multimodal posterior distributions.

emcee (Foreman-Mackey, Hogg, Lang, and Goodman 2013) is a popular Python package for MCMC based on affine-invariant ensemble sampling. The user needs only to specify a Python function evaluating the log of the posterior density (up to a normalizing constant). This user-friendliness is a key advantage of **emcee**, and its algorithm performs well on low to moderate dimensional (up to ~ 50) parameter spaces with a unimodel target distribution (which can be highly dependent). However, it can give misleading results in higher dimensions (Huijser, Goodman, and Brewer 2015).

Algorithms based on "annealing" or "tempering", such as parallel tempering (Hansmann 1997) and annealed importance sampling (Neal 2001) are related to nested sampling and are useful on multimodal posterior distributions. However, they require much more tuning than

Package	Easy to implement models?	High dimensions?	Multimodal distributions?	Dependent distributions?	Phase changes?	Computes Z?
DNest4		✓	✓	✓	✓	✓
emcee	✓			✓		
JAGS	✓	✓				
MultiNest	✓		✓	✓	✓	✓
Stan	✓	✓		✓		

Table 1: A simplified summary of the advantages and disadvantages of some Bayesian computation software packages.

nested sampling (in the form of an *annealing schedule*) and do not work on phase change problems, unlike nested sampling (Skilling 2006).

The advantages and disadvantages of a subset of these packages are summarized in Table 1.

3. Markov chain Monte Carlo

DNS is build upon the Metropolis-Hastings algorithm. In this algorithm, the acceptance probability α is given by

$$\alpha = \min\left(1, \frac{q(\theta'|\theta)}{q(\theta|\theta')}\frac{\pi(\theta')}{\pi(\theta)}\frac{L(\theta')}{L(\theta)}\right),$$

where $q(\theta'|\theta)$ is the proposal distribution used to generate a new position θ' from the current position θ. Often, q is *symmetric* so that the q terms cancel in the acceptance probability.

In DNS, the target distribution is not the posterior but rather the joint distribution in Equation 2. Moves of θ are done keeping j fixed, so we only need to consider the Metropolis acceptance probability for fixed j, i.e., with respect to a single constrained prior like Equation 1. Hence, the appropriate acceptance probability for a proposed move from θ to θ' is

$$\alpha = \min\left[1, \frac{q(\theta'|\theta)}{q(\theta|\theta')}\frac{\pi(\theta')}{\pi(\theta)}\mathbb{1}\left(L(\theta') > \ell_j\right)\right], \tag{5}$$

where ℓ_j is the likelihood threshold for the current level j. There are also moves that propose a change to j while keeping θ fixed, but the details are less relevant to the user.

For convenience later on, we separate the prior and proposal-related terms from the likelihood-related term, and write the former as

$$H = \frac{q(\theta'|\theta)}{q(\theta|\theta')} \times \frac{\pi(\theta')}{\pi(\theta)}.$$

The logarithm of H,

$$\ln(H) = \ln\left[\frac{q(\theta'|\theta)}{q(\theta|\theta')} \times \frac{\pi(\theta')}{\pi(\theta)}\right],$$

is the user's responsibility when implementing models, and will become relevant in Sections 8.2 and 8.3. In terms of $\ln(H)$, the acceptance probability becomes

$$\alpha = \min\left[1, e^{\ln(H)} \times \mathbb{1}\left(L(\theta') > \ell_j\right)\right]. \tag{6}$$

4. Dependencies and installation

The following instructions apply to Unix-like operating systems such as GNU/Linux, Mac OS X, and FreeBSD. Currently we have not tested **DNest4** on Microsoft Windows.

Development of **DNest4** takes place in the git repository located at `https://github.com/eggplantbren/DNest4/`. The software is licensed under the permissive open source MIT

license. To compile and run **DNest4**, you require a recent version of the GNU C++ compiler, **g++** (GCC 2016). **DNest4** uses features from the C++11 standard (Stroustrup 2013).

The Python packages **NumPy** (Van der Walt, Colbert, and Varoquaux 2011), **matplotlib** (Hunter 2007), **Numba** (Lam, Pitrou, and Seibert 2015), and **Cython** (Behnel, Bradshaw, Citro, Dalcin, Seljebotn, and Smith 2011) are also needed. To download and compile **DNest4** the following steps are sufficient:

```
$ wget https://github.com/eggplantbren/DNest4/archive/0.2.3.tar.gz
$ tar xvzf 0.2.3.tar.gz
$ mv DNest4-0.2.3 DNest4
$ cd DNest4/code
$ make
$ cd ../python
$ python setup.py install
```

In Mac OS X, the final line (which installs the Python parts of **DNest4**) needs to provide information about your OS version. For example, if your computer runs Mac OS X 10.9, the installation command for the Python package is:

```
$ MACOSX_DEPLOYMENT_TARGET=10.9 python setup.py install
```

We recommend to create an environment variable called `DNEST4_PATH` and set it to the directory *above* the **DNest4** directory. Then, if the model templates from the **DNest4** repository are copied to any other location on your system and used as the basis for new work, their Makefiles will continue to function.

5. Running DNest4

To demonstrate **DNest4**, we will use a simple linear regression example where the sampling distribution is

$$y_i | m, b, \sigma \sim \text{Normal}(mx_i + b, \sigma^2)$$

and the priors are

$$m \sim \text{Normal}(0, 1000^2),$$
$$b \sim \text{Normal}(0, 1000^2),$$
$$\ln \sigma \sim \text{Uniform}(-10, 10).$$

These are naïve diffuse priors and do not have any special status. The dataset is shown in Figure 2 on the right and the code is included in the `code/Examples/StraightLine` subdirectory, and a slightly simplified version of the code is explained in Section 8. To execute **DNest4** on this problem, go to this directory and execute `main`. The output to the screen should contain information about levels and "saving particles to disk". After 10,000 particles have been saved to the disk, the run will terminate.

6. Output files

The executable `main` is responsible for the exploration part of the DNS algorithm (i.e., running the MCMC chain, building levels, and then exploring all the levels). It creates three text output files, `sample.txt`, `sample_info.txt`, and `levels.txt`.

The first output file, `sample.txt`, contains a sampling of parameter values that represents the *mixture of constrained priors* (the target distribution used in DNS), *not* the posterior distribution. Each line of `sample.txt` represents a point in parameter space. In the linear example, there are three parameters (m, b, and σ); so there are three columns in `sample.txt`. Each time a point is saved to `sample.txt`, **DNest4** prints the message "Saving a particle to disk. N = ...".

The second output file, `sample_info.txt`, should have the same number of rows as the file `sample.txt`, because it contains metadata about the samples in `sample.txt`. The first column is the index j, which tells us which "level" the particle was in when it was saved. Level 0 represents the prior, and higher levels represent more constrained versions of the prior. The second column is the log-likelihood value, and the third column is the likelihood "tiebreaker", which allows nested sampling to work when there is a region in parameter space with nonzero prior probability where the likelihood is constant. The final column tells us which thread the particle belonged to: when you use **DNest4** in multithreaded mode (see Appendix A), each thread is responsible for evolving one or more particles.

The third output file, `levels.txt`, contains information about the levels that were built during the run. The first column has estimates of the $\log(X)$ values of the levels, i.e., how compressed they are relative to the prior, in units of nats. For example, if a level has $\log(X) = -1.02$, its likelihood value encloses $\exp(-1.02) \approx 36.1\%$ of the prior mass.

The second column contains the log-likelihoods of the levels. The first level, with a $\log(X)$ value of 0 and a log-likelihood of -10^{308} (basically "minus infinity"), is simply the prior. The third column has the "tiebreaker" values for the levels, which again are not particularly useful unless your problem has likelihood plateaus. The fourth and fifth columns are the number of accepted proposals and the total number of proposals that have occurred within each level, which are useful for monitoring the Metropolis acceptance ratio as a function of level. The final two columns, called "exceeds", and "visits", are used to refine the estimates of the level compressions (and hence the $\log(X)$ values of the levels in column 1), as discussed in Section 3 of Brewer *et al.* (2011). The visits column counts the number of times a level (level j, say) has been visited, but only starts counting after the next level ($j + 1$) has been created. The exceeds column counts the number of times a particle that was in level j had a likelihood that exceeded that of level $j + 1$.

7. Post-processing

The output files themselves are typically not immediately useful. The goal of running nested sampling is usually to obtain posterior samples and the marginal likelihood Z, whereas `sample.txt` only contains samples from the mixture of constrained priors. Additional post-processing is required. This can be achieved by running the following Python function[1]:

---

[1]Alternatively, the file `showresults.py` in the example directory runs this function, and then calls the code in `display.py` to create a further plot shown in Figure 2 on the right.
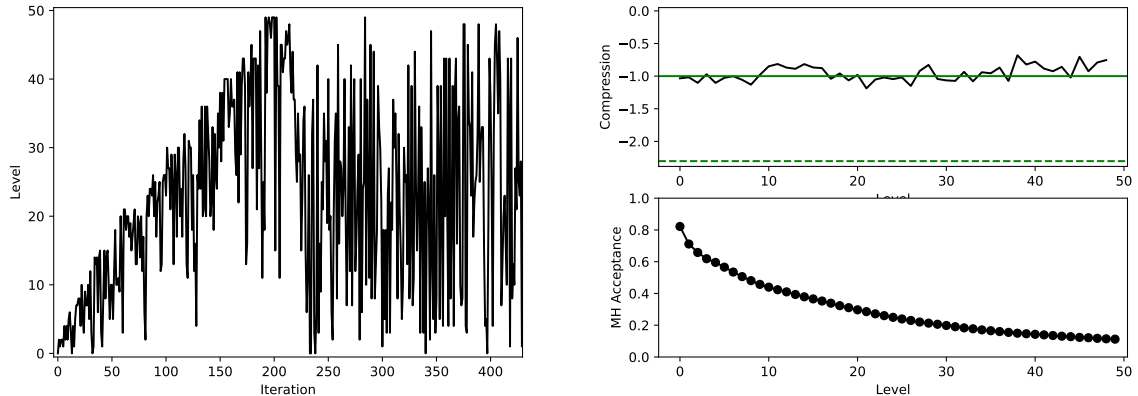
Figure 1: Left: The level j of the saved particles over time. Typically, this will trend upwards until all the levels have been created, and then diffuse evenly throughout all the levels. Right, top panel: The estimated compression factor between subsequent levels, expressed as $\ln(X_{i+1}/X_i)$. Right, bottom panel: The Metropolis acceptance fraction as a function of level.

```
>>> import dnest4
>>> dnest4.postprocess()
```

This produces the three diagnostic plots in Figures 1 and 2 on the left, along with the following output:

```
log(Z) = -175.47872682863076
Information = 15.166340875399044 nats.
Effective sample size = 1251.3576752091165
```

These are the natural log of the marginal likelihood, the information

$$\mathcal{H} = \int p(\theta|D, M) \ln \left[\frac{p(\theta|D, M)}{p(\theta|M)} \right] \, d\theta,$$

which quantifies the degree to which D restricted the range of possible θ values, and the effective sample size, or number of saved particles with significant posterior weight. The `postprocess` function also saves a file, `posterior_sample.txt`, containing posterior samples (one per row). Unfortunately, it is harder to compute justified error bars on $\ln(Z)$ in DNS than it is in standard nested sampling.

The `postprocess` function can be called while **DNest4** is running. This is helpful for monitoring the progress of a run, by inspecting the output plots.

7.1. Options

A plain-text file called `OPTIONS` resides in the directory from which you execute a run. This file contains numerical parameters controlling the DNS algorithm. Here are the contents of `OPTIONS` for the linear regression example:
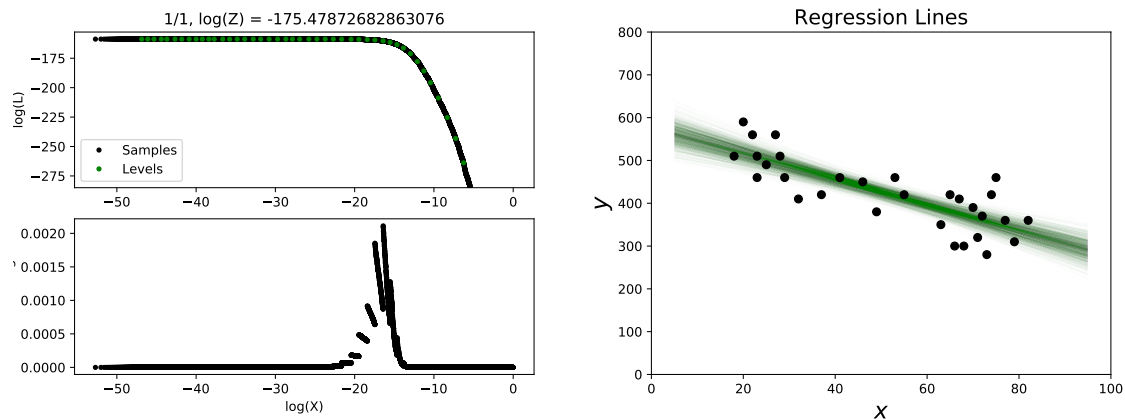
Figure 2: Left, top panel: The log-likelihood curve, showing the relationship between log-likelihood and the enclosed prior mass. Left, bottom panel: Posterior weights of the saved particles. For a successful run, there should be a clear peak, and saved particles to the left of this plot should have insignificant posterior weight compared to those in the peak. Right: Regression lines drawn from the posterior.

```
# File containing parameters for DNest4.
# Put comments at the top, or at the end of the line.
5        # Number of particles
10000    # New level interval
10000    # Save interval
100      # Thread steps - pooling interval
0        # Maximum number of levels (0 ==> automatic)
10       # Backtracking scale length lambda in the paper.
100      # Equal weight enforcement. Beta in the paper.
10000    # Maximum number of saves (0 ==> run forever)
```

Additional options are available on the command line. These are described in Appendix A.

7.2. Number of particles

The first option is the number of particles, here set to five. If you use more particles, the same amount of CPU time will be spent evolving more particles, so each one will not be evolved as far. On most problems, five is a sensible default value. On complex problems where the likelihood function has a challenging structure, more particles are useful, but it is usually better to run in multi-threaded mode (see Appendix A).

7.3. New level interval

The new level interval controls how quickly **DNest4** creates new levels. In this example, this is set to 10,000, so a new level will be created once 10,000 MCMC steps have resulted in 10,000 likelihood values above the current top level. It is difficult to give a sensible default for this quantity because it depends on the complexity of the problem (basically, how good the Metropolis proposals are at exploring the target distribution). However, 10,000 will work

for many problems, so we suggest it as a sensible default. Higher values are slower, but more fail-safe.

7.4. Save interval

The save interval controls how often **DNest4** writes a model to the output files; what is usually called "thinning". Saving more frequently (i.e., a smaller save interval) is usually better. However, this can result in big output files if your model prints a lot of parameters to `sample.txt` causing the post-processing to take a long time and/or a lot of RAM. A default suggestion, used in the example, is to set the save interval to the same value as the new level interval.

7.5. Thread steps

The "thread steps" parameter controls how frequently separate threads pool their information about the levels (when running in multi-threaded mode, see Appendix A). It should be set to a moderate value, but should also be a small fraction of the new level interval and the save interval. 100 is a suggested default value that should work without problems in the vast majority of cases.

7.6. Maximum number of levels

As the name suggests, this tells **DNest4** how many levels to create, and therefore controls the factor by which the parameter space is ultimately compressed. An appropriate value for this quantity depends on the specific model and dataset at hand – typically, a larger numbers of parameters and larger (more informative) datasets will lead to a larger value of $\mathcal{H}$, and therefore need more levels.

In the initial phase of DNS when levels are being created, the particles move to the left in Figure 2, increasing likelihood L and decreasing prior mass X. The posterior distribution is concentrated where the rate of increase of $\ln(L)$ and the rate of decrease of $\ln(X)$ are approximately equal. Figure 2 is the most useful diagnostic plot for setting the correct maximum number of levels. A clear peak should be visible in the posterior weights plot in the lower panel, such that moving further to the left would not add any more particles with comparable weight to those in the peak.

As described by Skilling (2006), there is no guarantee in principle that another peak might have appeared had a run continued for longer. These *phase changes* are more common in statistical mechanics problems than in data analysis problems, but can appear in the latter (e.g., Brewer 2014; Brewer and Donovan 2015).

Alternatively, **DNest4** can try to determine the maximum number of levels automatically, if you set the maximum number of levels to 0. This works well on problems where the MCMC exploration is efficient. When it fails, this is detectable as the posterior weights plot (lower panel of Figure 2) will peak at the left end of its domain. However, such a failed run may still be useful as it can be used to suggest an order of magnitude for the required number of levels. For example, if the automatic setting results in 150 levels and is later seen to fail, it might be worth a try to set the maximum number of levels to, say, $1.3 \times 150 = 195$.

7.7. Backtracking scale length

The backtracking scale length, denoted by λ, appeared in Equation 4, and controls the degree to which particles are allowed to "backtrack" down in level during the first stage of DNS when levels are being built. Higher values are more fail-safe, but make it take longer to create the levels. The value 10 used in the linear regression example is a suitable default value that should work in almost all cases. In simple problems where MCMC exploration is easy, lower values from 1 to 5 work sufficiently well.

7.8. Equal weight enforcement

This value is the parameter β described in Brewer *et al.* (2011), and compensates for imprecision in the spacing of the levels, so that the desired mixture weights $w_j \propto 1$ are achieved during the second stage of the DNS algorithm. The value 100 is recommended.

7.9. Maximum number of saves

This controls the length of a **DNest4** run, in units of saved particles in `sample.txt`, which represent the mixture of constrained priors. The number of posterior samples is always less than this. In most applications 5,000 (as in the linear regression example) provides enough posterior samples (typically a few hundred) for sufficiently accurate posterior summaries. However, if you want to plot smooth-looking posterior histograms, you'll need to increase this value.

If you set the maximum number of saves to zero, **DNest4** will run until you terminate it manually.

8. Implementing models

The "classic" method of implementing models in **DNest4** is by writing a C++ class, an object of which represents a point in your parameter space.

To run **DNest4** on any particular problem, such as this linear regression example, the user needs to define a C++ class to specify the model. Specifically, an object of the class represents a point in the model's parameter space. Member functions are defined which generate the object's parameters from the prior, make proposal steps, evaluate the likelihood, and so on. The sampler calls these member functions while executing a run.

For the simple linear regression example, we will call the class 'StraightLine'. The member variables representing the unknown parameters are defined in the header file `StraightLine.h`:

```
class StraightLine
{
    private:
        double m, b, sigma;
};
```

The class must also define and implement the following member functions:

(i) `void from_prior(DNest4::RNG& rng)`, which generates parameter values from the prior.

(ii) `double perturb(DNest4::RNG& rng)`, which proposes a change to the parameter values, and returns $\ln(H)$ as defined in Equation 6.

(iii) `double log_likelihood() const`, which evaluates the log of the likelihood function.

(iv) `void print(std::ostream& out) const`, which prints parameters of interest to the given output stream.

(v) `std::string description() const`, which returns a C++ string naming the parameters printed by `print(std::ostream&) const`. This string is printed (after a comment character #) at the top of the output file.

These are all described (using the linear regression example) in the following sections.

8.1. Generating from the prior

The member function used to generate straight line parameters from the prior is:

```
void StraightLine::from_prior(DNest4::RNG& rng)
{
    // Naive diffuse prior
    m = 1E3 * rng.randn();
    b = 1E3 * rng.randn();

    // Log-uniform prior
    sigma = exp(-10.0 + 20.0 * rng.rand());
}
```

This generates m, b, and σ from their joint prior. In this case, the priors are all independent, so this reduces to generating the parameters each from their own prior distribution. For convenience, **DNest4** provides an 'RNG' class to represent random number generators. The 'RNG' class is just a convenience wrapper for the random number generators built into C++11. As you might expect, there are `rand()` and `randn()` member functions to generate double precision values from a Uniform$(0, 1)$ and Normal$(0, 1)$ distribution respectively.

8.2. Proposal moves

The `perturb` member function for the straight line model is given below. This takes a random number generator as input, makes changes to a subset of the parameters, and returns a `double` corresponding to $\ln(H)$ as defined in Equation 6. The combination of the choice of proposal and the $\ln(H)$ value returned must be consistent with the prior distribution.

```
double StraightLine::perturb(DNest4::RNG& rng)
{
    // log_H value to be returned
    double log_H = 0.0;

    // Proposals must be consistent with the prior
    // Choose which of the three parameters to move
```

```
    int which = rng.rand_int(3);

    if(which == 0)
    {
        // log_H takes care of the prior ratio
        // i.e., log_H = log(pi(theta')/pi(theta))
        log_H -= -0.5 * pow(m / 1E3, 2);

        // Take a step
        m += 1E3 * rng.randh();

        // log_H takes care of the prior ratio
        log_H += -0.5 * pow(m / 1E3, 2);
    }
    else if(which == 1)
    {
        log_H -= -0.5*pow(b / 1E3, 2);
        b += 1E3 * rng.randh();
        log_H += -0.5*pow(b / 1E3, 2);
    }
    else
    {
        // Proposal for a parameter with a log-uniform
        // prior takes the log of the parameter,
        // takes a step with respect to a uniform prior,
        // then takes the exp of the parameter
        sigma = log(sigma);
        sigma += 20.0 * rng.randh();

        // Wrap proposed value back into the
        // interval allowed by the prior
        DNest4::wrap(sigma, -10.0, 10.0);
        sigma = exp(sigma);
    }

    return log_H;
}
```

This function first chooses a random integer from $\{0, 1, 2\}$ using the `rand_int(int)` member function of the 'DNest4::RNG' class. This determines which of the three parameters (m, b, σ) is modified. In this example, there are no proposals that modify more than one of the parameters at a time, and all proposals are "random walk" proposals that add a perturbation (drawn from a symmetric distribution) to the current value.

8.3. Proposals for single parameters

The proposal for m involves adding a perturbation to the current value using the line

```
m += 1E3 * rng.randh();
```

A challenge using MCMC for standard nested sampling is that the target distribution is not static – it gets compressed over time. Similarly, in the first stage of DNS the target distribution gets compressed (as levels are added), and in the second stage the target distribution is a mixture of distributions that have been compressed to varying degrees. This makes it difficult to tune step-sizes as you would when using the standard Metropolis algorithm to sample the posterior distribution.

Rather than trying to adapt proposal distributions as a function of level, it is much simpler to just use heavy-tailed proposals which have some probability of making a jump of appropriate size. This is slightly wasteful of CPU time, but it saves a lot of human time and is more fail-safe than tuned step sizes. In simple experiments, we have found that heavy-tailed proposals are about as efficient as slice sampling (Neal 2003), but much easier to implement. The following procedure generates x from a heavy-tailed distribution:

1. Generate $a \sim \text{Normal}(0, 1)$.

2. Generate $b \sim \text{Uniform}(0, 1)$.

3. Define $t := a/\sqrt{-\ln(b)}$.

4. Generate $n \sim \text{Normal}(0, 1)$.

5. Set $x := 10^{1.5 - 3|t|} n$.

The variable t has a student-t distribution with 2 degrees of freedom. Overall, this procedure generates values x with a maximum scale of tens to hundreds, down to a minimum scale of about 10^{-30} with 99% probability, by virtue of the t-distribution's heavy tails. For convenience, the 'RNG' class contains a member function `randh()` to generate values from this distribution. The factor of `1E3` is included because it is a measure of the prior width. Therefore, this proposal will attempt moves whose maximum order of magnitude is a bit wider than the prior (since it would be very surprising if any bigger moves were needed), and will propose moves a few orders of magnitude smaller with moderate probability. We recommend using this strategy (a measure of prior width, multiplied by `randh()`) as a default proposal that works well in almost all problems.

Recall that for nested sampling, the Metropolis acceptance probability, excluding the term for the likelihood, is

$$H = \frac{q(\theta'|\theta)}{q(\theta|\theta')} \times \frac{\pi(\theta')}{\pi(\theta)}.$$

When implementing a model class, the `perturb(DNest4::RNG&)` member function must return the logarithm of this value. Since the prior for m is a normal distribution with mean zero and standard deviation 1000, H is

$$H = \frac{\exp\left[-\frac{1}{2}(m'/1000)^2\right]}{\exp\left[-\frac{1}{2}(m/1000)^2\right]}.$$

This explains the use of the `log_H` variable.

In the example, the proposal for σ is implemented by taking advantage of the uniform prior for $\ln(\sigma)$. So σ is transformed by taking a logarithm, a proposal move is made (that satisfies detailed balance with respect to a uniform distribution), and then σ is exponentiated again. The step for the uniform prior between -10 and $+10$ uses the following code:

```
sigma += 20.0 * rng.randh();
DNest4::wrap(sigma, -10.0, 10.0);
```

The factor of 20 accounts for the prior width, and the `wrap(double&, double, double)` function uses the modulo operator to construct periodic boundaries. For example, if the perturbation results in a value of 10.2, which is outside the prior range, the value is modified to -9.8. The `wrap` function has no return value and works by modifying its first argument, which is passed by reference. Alternatively, the log-uniform prior, which has density proportional to $1/\sigma$, could have been used directly by adding $\ln[(1/\sigma')/(1/\sigma)]$ to the return value instead of using the log/exp trick. However, this is not recommended for a prior distribution like this which covers several orders of magnitude, because the appropriate scale size for the proposal is less clear. This is likely to cause inefficient sampling.

8.4. Consistency of prior and proposal

It is imperative that `from_prior` and `perturb` be consistent with each other, and that each implements the prior distributions that you want to use. One technique for testing this is to sample the prior for a long time (by setting the maximum number of levels to 1) and inspect `sample.txt` to ensure that each parameter is exploring the prior correctly.

8.5. Log-likelihood

The log-likelihood for the model is evaluated and returned by the `log_likelihood` member function. For the straight line fitting example, the log-likelihood is based on the normal density, and the code is given below.

```
double StraightLine::log_likelihood() const
{
    // Grab the dataset
    const std::vector<double>& x = Data::get_instance().get_x();
    const std::vector<double>& y = Data::get_instance().get_y();

    // Variance
    double var = sigma * sigma;

    // Conventional Gaussian sampling distribution
    double log_L = 0.0;
    double mu;
    for(size_t i = 0; i < y.size(); ++i)
    {
        mu = m * x[i] + b;
        log_L += -0.5 * log(2 * M_PI * var) - 0.5 * pow(y[i] - mu, 2) / var;
    }
```

```
    return log_L;
}
```

The dataset is assumed to be accessible inside this function. In the regression example, this is achieved by having a 'Data' class to represent datasets. Since there will usually only be one dataset, the *singleton pattern* (a class of which there is one instance accessible from anywhere) is recommended. The 'Data' class has one static member which is itself an object of class 'Data', and is accessible using Data::get_instance() – this is the singleton pattern, essentially a way of defining quasi-"global" variables:

```
class Data
{
    private:
        // 'static' means this exists at the level of the class
        // instead of being truly global
        static Data instance;

    public:
        // Getter
        static Data& get_instance();
};
```

Alternatively, the data could be defined using static members of your model class.

8.6. Parameter output

The print and description functions are very simple:

```
void StraightLine::print(std::ostream& out) const
{
    out << m << ' ' << b << ' ' << sigma;
}

std::string StraightLine::description() const
{
    return std::string("m, b, sigma");
}
```

The print function specifies that the parameters m, b, and σ are printed in a single line (of sample.txt) and are separated by spaces (the postprocess function assumes the delimiter is a space).

8.7. Running the sampler

The file main.cpp contains the main() function which is executed after compiling and linking. The contents of main.cpp for the linear regression example are:

```
#include <iostream>
#include "Data.h"
#include "DNest4/code/DNest4.h"
#include "StraightLine.h"

using namespace std;

int main(int argc, char** argv)
{
    Data::get_instance().load("road.txt");
    DNest4::start<StraightLine>(argc, argv);
    return 0;
}
```

The first line of `main()` loads the data into its global instance (so it can be accessed from within the log-likelihood function) and the second line uses a `start` template function to construct and run the sampler.

9. Finite mixture models with the 'RJObject' class

Mixture models are a useful way of representing realistic prior information in Bayesian data analysis. To reduce the amount of effort needed to implement mixture models in DNS, Brewer (2014) implemented a template class called 'RJObject' to handle the MCMC moves required. The RJ in 'RJObject' stands for reversible jump (Green 1995), as 'RJObject' implements birth and death moves for mixture models with an unknown number of components. An updated version of 'RJObject' is included in **DNest4**.

If N is the number of components, x_i denotes the vector of parameters of the ith component, and α is the vector of hyperparameters, the prior can be factorized via the product rule, giving

$$p\left(N, \alpha, \{x_i\}_{i=1}^N\right) = p(N)p(\alpha|N)p\left(\{x_i\}_{i=1}^N \,|\alpha, N\right).$$

The specific assumptions of 'RJObject' are that this simplifies to

$$p\left(N, \alpha, \{x_i\}_{i=1}^N\right) = p(N)p(\alpha)\prod_{i=1}^N p\left(x_i|\alpha\right). \tag{7}$$

That is, the prior for the hyperparameters is independent of the number of components, and each component is independent and identically distributed given the hyperparameters. The sampling distribution $p(D|x, \alpha)$ (not shown in the probabilistic graphical model) cannot depend on the ordering of the components. A probabilistic graphical model (PGM) showing this dependence structure is shown in Figure 3. There are no observed data nodes here, but if such a structure forms part of a Bayesian model, an 'RJObject' object within your model class can encapsulate this part of the model.

Brewer (2014) described the motivation for 'RJObject' and some details about the Metropolis proposals underlying it. Here, we demonstrate how to implement a finite mixture model using 'RJObject'. This example can be found in the `code/Examples/RJObject_1DMixture`
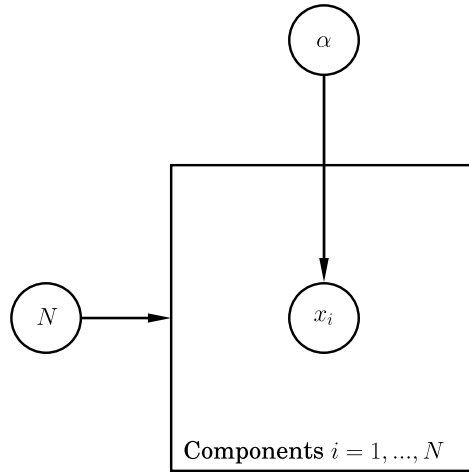
Figure 3: A PGM showing the kind of prior information the 'RJObject' template class expresses. Figure created using Daft (http://daft-pgm.org/).

directory. The "SineWave" and "GalaxyField" models from Brewer (2014) are also included in the **DNest4** repository.

9.1. An example mixture model

Consider a sampling distribution for data $D = \{D_1, D_2, \ldots, D_n\}$ which is a mixture of N Gaussians with means $\{\mu_j\}$, standard deviations $\{\sigma_j\}$, and mixture weights $\{w_j\}$. The likelihood for the ith data point is

$$p\left(D_i | N, \{\mu_j\}, \{\sigma_j\}, \{w_j\}\right) = \sum_{j=1}^{N} \frac{w_j}{\sigma_j \sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma_j^2}\left(D_i - \mu_j\right)^2\right].$$

Colloquially, one might say the data were "drawn from" a mixture of N normal distributions, and we want to infer N along with the properties (positions, widths, and weights) of those normal distributions. The mixture weights $\{w_j\}$ must obey a normalization condition $\sum_j w_j = 1$. The easiest way of implementing this is to use un-normalized weights $\{W_j\}$ which do not obey such a condition and then normalize them by dividing by their sum.

Making the connection with Equation 7 and Figure 3, the "components" are the Gaussians, the Gaussian parameters are $\{x_j\} = \{(\mu_j, \sigma_j, W_j)\}$, and hyperparameters α may be used to help specify a sensible joint prior for the Gaussian parameters.

We now describe the specific prior distributions we used in this example. The prior for N was

$$p(N) \propto \frac{1}{N+1}$$

for $N \in \{1, 2, \ldots, 100\}$. For the conditional prior of the Gaussian parameters, we used Laplace (biexponential) distributions[2]. Normal distributions would be more conventional,

---

[2]A Laplace distribution with location parameter a and scale parameter b has density $p(x|a, b) = \frac{1}{2}\exp\left(-\frac{1}{b}|x - a|\right)$.

but the analytic cumulative distribution function (CDF) of the Laplace distribution makes it easier to implement. These were:

$$\mu_j \sim \text{Laplace}(a_\mu, b_\mu) \tag{8}$$

$$\ln \sigma_j \sim \text{Laplace}(a_{\ln \sigma}, b_{\ln \sigma}) \tag{9}$$

$$\ln W_j \sim \text{Laplace}(0, b_{\ln W}). \tag{10}$$

The location parameters are denoted with a and scale parameters with b. These priors express the idea that the centers, widths, and relative weights of the Gaussian mixture components are probably clustered around some typical value.

We used the following priors for the hyperparameters:

$$a_\mu \sim \text{Uniform}(-1000, 1000)$$
$$\ln b_\mu \sim \text{Uniform}(-10, 10)$$
$$a_{\ln \sigma} \sim \text{Uniform}(-10, 10)$$
$$b_{\ln \sigma} \sim \text{Uniform}(0, 5)$$
$$b_{\ln W} \sim \text{Uniform}(0, 5).$$

The normalized weights are $w_k = W_k / \sum w_j$.

9.2. Using the 'RJObject' class

To implement this model for **DNest4**, the model class needs to contain an instance of an 'RJObject', which contains the parameters $\{(\mu_j, \sigma_j, w_j)\}$:

```
class MyModel
{
    private:
        DNest4::RJObject<MyConditionalPrior> gaussians;
....
```

where the template argument `<MyConditionalPrior>` is a class implementing the prior for the hyperparameters α and the form of the conditional prior $p(x_i|\alpha)$. The main advantage of the 'RJObject' class is that we do not need to implement any proposals for $\{(\mu_j, \sigma_j, W_j)\}$. Rather, these can be done trivially as follows:

```
    // Generate the components from the prior
    gaussians.from_prior(rng);

    // Do a Metropolis proposal
    double logH = gaussians.perturb(rng);

    // Print to output stream 'out'
    gaussians.print(out);
```

The 'RJObject' constructor definition is:

```
RJObject(int num_dimensions, int max_num_components, bool fixed,
        const ConditionalPrior& conditional_prior,
        PriorType prior_type=PriorType::uniform);
```

where `num_dimensions` is the number of parameters needed to specify a single component (three in this example), `max_num_components` is the maximum value of N allowed, `fixed` determines whether N is fixed at $N_{\max}$ or allowed to vary, `conditional_prior` is an instance of a conditional prior, and `prior_type` controls the prior for N (the default is uniform from $\{0, 1, 2, \ldots, N_{\max}\}$). The `RJObject_1DMixture` MyModel initializes its 'RJObject' as follows:

```
MyModel::MyModel()
:gaussians(3, 100, false, MyConditionalPrior(), PriorType::log_uniform)
{
}
```

Passing `PriorType::log_uniform` for the final argument specifies the $1/(N + 1)$ prior for N. One complication for this model is that 'RJObject', by default, allows N to be zero, which makes no sense for this particular problem. Therefore, we prevent $N = 0$ from being generated in `MyModel::from_prior`, and assert that it should always be rejected if proposed in `MyModel::perturb`:

```
void MyModel::from_prior(RNG& rng)
{
    do
    {
        gaussians.from_prior(rng);
    }while(gaussians.get_components().size() == 0);
}


double MyModel::perturb(RNG& rng)
{
    double logH = 0.0;

    logH += gaussians.perturb(rng);
    if(gaussians.get_components().size() == 0)
        return -std::numeric_limits<double>::max();

    return logH;
}
```

To access the component parameters, the 'RJObject' member function `get_components` is used. This was used in the above functions, but more typically it is needed in the log-likelihood. The member function `get_components` returns (by const reference) a `std::vector` of `std::vector`s of `double`s. For example, parameter two of component zero is:

```
gaussians.get_components()[0][2]
```

The order of $\{(\mu_j, \sigma_j, w_j)\}$ (i.e., which one is parameter 0, 1, and 2) is determined by the conditional prior class, and is explained in the following section.

The `print` function for 'RJObject' objects prints the dimensionality of each component and the maximum value of N, followed by the hyperparameters, then parameter zero of each component (zero padded when $N < N_{\max}$), parameter one for each component, and so on.

9.3. Conditional priors

The 'RJObject' class is used to manage the components. An additional class is needed to define and manage the hyperparameters α (i.e., define how they are generated from the prior, how they are proposed, and so on). The i.i.d. conditional prior $p(x_i|\alpha)$ is also defined by this extra class. In the example, the class is called 'MyConditionalPrior' and is inherited from an abstract base class 'DNest4::ConditionalPrior' which insists that certain member functions be specified.

The member functions `from_prior`, `perturb_hyperparameters`, and `print` do the same things as the similar functions in a standard model class, but for the hyperparameters α. In addition, three functions `from_uniform`, `to_uniform`, and `log_pdf` together define the form of the conditional prior for the component parameters, $p(x_i|\alpha)$. Each of these takes a `std::vector` of `doubles` by reference as input.

The `log_pdf` function just evaluates $\ln p(x_i|\alpha)$ at the current value of α. The position x_i where this is evaluated is passed in via the input vector. In the example, these were defined using Equations 8–10. To simplify this, we have defined a class for Laplace distributions, rather than explicitly writing out the densities. We decided (arbitrarily) that parameters 0, 1, and 2 are μ, $\ln(\sigma)$, and W respectively.

```
// vec = {mu, log_sigma, log_weight}
double MyConditionalPrior::log_pdf(const std::vector<double>& vec) const
{
    // Three Laplace distributions
    Laplace l1(location_mu, scale_mu);
    Laplace l2(location_log_sigma, scale_log_sigma);
    Laplace l3(0.0, scale_log_weight);
    return l1.log_pdf(vec[0]) + l2.log_pdf(vec[1]) + l3.log_pdf(vec[2]);
}
```

The functions `to_uniform` and `from_uniform` must implement the cumulative distribution function (CDF) of the conditional prior and its inverse, i.e., `from_uniform`, if the input vector contains i.i.d. draws from Uniform(0, 1), these should be modified to become draws from $p(x|\alpha)$, and `to_uniform` should be the inverse of `from_uniform`. Both of these functions modify the vector argument in-place. For the example, we again used an external class to define these functions for the Laplace distribution:

```
// vec (input) = {u1, u2, u3} ~ Uniform(0, 1) in the prior
// gets modified to {mu, log_sigma, log_weight}
void MyConditionalPrior::from_uniform(std::vector<double>& vec) const
{
    // Three Laplace distributions
    Laplace l1(location_mu, scale_mu);
    Laplace l2(location_log_sigma, scale_log_sigma);
```
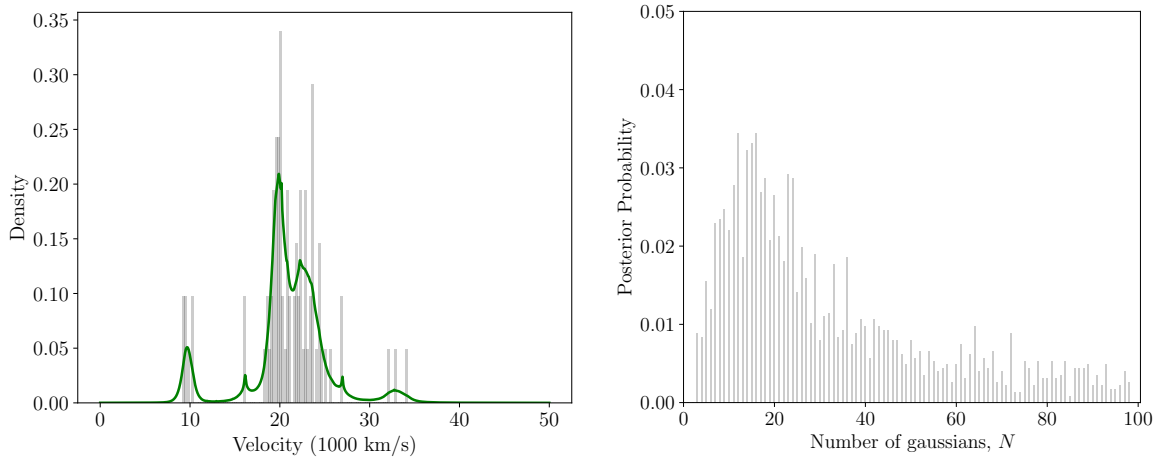
Figure 4: Left: The galaxy data, with the posterior mean fit (equivalent to the predictive distribution for the "next" data point). Right: The posterior distribution for N given the galaxy data.

```
    Laplace l3(0.0, scale_log_weight);

    vec[0] = l1.cdf_inverse(vec[0]);
    vec[1] = l2.cdf_inverse(vec[1]);
    vec[2] = l3.cdf_inverse(vec[2]);
}


// vec (input) = {mu, log_sigma, log_weight}
// gets modified to {u1, u2, u3} using the CDF of the conditional prior
// This is the inverse of from_uniform
void MyConditionalPrior::to_uniform(std::vector<double>& vec) const
{
    // Three Laplace distributions
    Laplace l1(location_mu, scale_mu);
    Laplace l2(location_log_sigma, scale_log_sigma);
    Laplace l3(0.0, scale_log_weight);

    vec[0] = l1.cdf(vec[0]);
    vec[1] = l2.cdf(vec[1]);
    vec[2] = l3.cdf(vec[2]);
}
```

9.4. Mixture model results

The data and the posterior mean fit are shown in Figure 4 on the left and the posterior distribution for N is shown in Figure 4 on the right. The marginal likelihood was $\ln(Z) = -232.1$, and the information was $\mathcal{H} = 29.5$ nats.
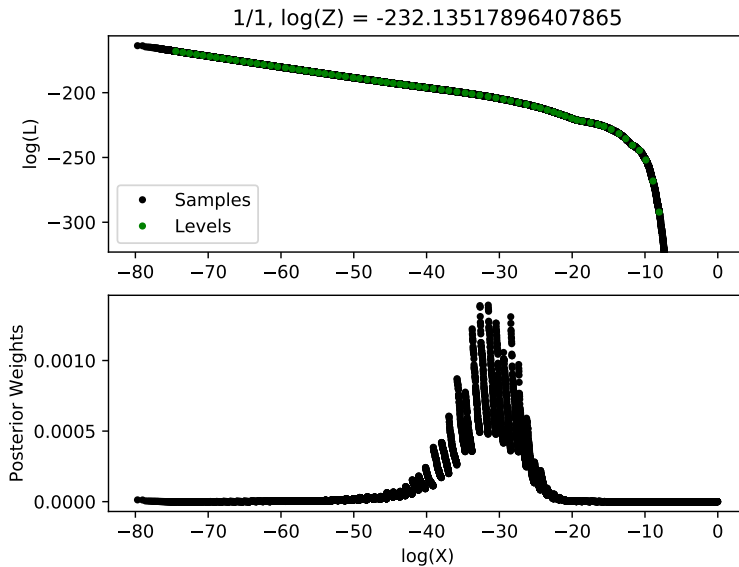
Figure 5: The log-likelihood and posterior weights as a function of compression X for the galaxy data. There are phase changes (Skilling 2006) for which nested sampling is the best known solution. There are some points with high posterior weight at the left of the plot, but there are so few of them that they make up a trivial fraction of the posterior mass. In practice, however, it is worth re-running with more levels to verify that a second peak is not forming.

10. Approximate Bayesian computation

10.1. Background

Nested sampling can be used to solve approximate Bayesian computation (ABC) problems elegantly and efficiently. Also known (misleadingly) as likelihood-free inference, ABC is a set of Monte Carlo techniques for approximating the posterior distribution without having to evaluate the likelihood function $L(\theta)$. Instead, the user must be able to cheaply generate simulated datasets from the sampling distribution $p(D|\theta, M)$. Since a sampling distribution must be specified, it is not that there is no likelihood function (indeed, a sampling distribution and a dataset imply a particular likelihood function), rather that we cannot evaluate it cheaply and therefore cannot use MCMC.

All Bayesian updating conditions on the truth of a proposition. We sometimes speak and use notation as if we are conditioning on the value of a variable, for example by writing the posterior distribution as $p(\theta|D, M)$. However, this is shorthand for $p(\theta|D = D_{\text{observed}}, M)$. In the prior state of knowledge, the statement $D = D_{\text{observed}}$ could have been either true or false, but it is known to be true in the posterior state of knowledge. In the case of "continuous data" (really a continuous *space of possibilities for the data before we learned it*) we condition on a proposition like $(D \in \mathcal{R})$ where $\mathcal{R}$ is a region, and then implicitly take a limit as the size of $\mathcal{R}$ goes to zero.

A simple "rejection sampling" version of ABC works by sampling the joint prior distribution

for the parameters and data

$$p(\theta, D|M) = p(\theta|M)p(D|\theta, M)$$

and rejecting samples for which $D \neq D_{\text{observed}}$, so that the samples represent

$$p(\theta, D|D = D_{\text{observed}}, M) \propto p(\theta|M)p(D|\theta, M)\mathbb{1}\left[D = D_{\text{observed}}\right]. \tag{11}$$

The marginal distribution for θ is also the conditional distribution $p(\theta|D = D_{\text{observed}}, M)$, that is, the posterior[3].

This approach is rarely usable in practice because the probability of generating a dataset that matches the observed one is extremely low. To work around this, we replace the proposition $D = D_{\text{observed}}$ with a logically weaker one (i.e., one that is implied by $D = D_{\text{observed}}$ but does not imply it). The weaker proposition is defined using a discrepancy function ρ, which measures how different a simulated dataset D is from the real one D_{observed}:

$$\rho\left(D; D_{\text{observed}}\right).$$

The discrepancy function should take a minimum value of zero when $D = D_{\text{observed}}$. The analysis then proceeds by Monte Carlo sampling the joint posterior for θ and D conditional on the logically weaker proposition

$$\rho\left(D; D_{\text{observed}}\right) < \epsilon,$$

where ϵ is some small number. With this proposition, the rejection rate will still typically be very high, but lower than with $D = D_{\text{observed}}$. Typically, ρ is defined by introducing a few *summary statistics* $s_1(D), s_2(D), \ldots, s_n(D)$, which hopefully capture most of the relevant information in the data, and then computing the Euclidean distance between the summary statistics of D and D_{observed}. That is,

$$\rho(D; D_{\text{observed}}) = \sqrt{\sum_i \left[s_i(D) - s_i(D_{\text{observed}})\right]^2}.$$

The main challenges associated with ABC are:

(i) The choice of discrepancy function $\rho(D; D_{\text{observed}})$, which may involve a choice of summary statistics.

(ii) How to choose the value of ϵ, or how to change it as a run progresses.

(iii) How to make algorithms more efficient than rejection sampling.

Challenge (i) is Bayesian in nature, i.e., it relates to the very definition of the posterior distribution itself. On the other hand, challenges (ii) and (iii) are about the computational implementation. Most ABC analyses are done using sequential Monte Carlo (SMC; Del Moral, Doucet, and Jasra 2012), another family of algorithms closely related to nested sampling and annealing in the sense that they work with sequences of probability distributions.

---

[3]Incidentally, this usage of the *joint* posterior distribution for the parameters *and the data* provides a bridge connecting Bayesian updating to maximum entropy updating (Caticha and Giffin 2006; Giffin and Caticha 2007).

10.2. ABC with nested sampling

Nested sampling can be used for ABC by noting that Equation 11 is of the same form as a constrained prior, only instead of being the prior for the parameters $p(\theta|M)$ defined on the parameter space, it is the *joint prior* for the parameters and the data, $p(\theta, D|M) = p(\theta|M)p(D|\theta, M)$, defined on the product space of possible parameters and datasets. Therefore, to implement ABC in **DNest4**, you implement a model class whose `from_prior` member function generates parameters *and simulated data* from the joint prior, and whose `perturb` member function proposes changes to the parameters and/or the simulated data, in a manner consistent with the joint prior. The `log_likelihood` function of the class, instead of actually evaluating the log-likelihood, should evaluate $-\rho(D; D_{\text{observed}})$. Then, running the sampler will create levels which systematically decrease ρ. The samples from a particular level then represent the ABC posterior defined in Equation 11 with ϵ corresponding to minus the "log-likelihood" of the level. A single **DNest4** run allows you to test sensitivity to the value of ϵ using a single run. The marginal likelihood reported will be the probability that $\rho(D; D_{\text{observed}}) < \epsilon$ given the model, although caution should be applied if using this for model selection (Robert, Cornuet, Marin, and Pillai 2011).

10.3. ABC example

A simple ABC example is included in the directory `code/Examples/ABC`. The example tries to infer the mean μ and standard deviation σ of a normal distribution from a set of samples. The sampling distribution is

$$x_i \sim \text{Normal}(\mu, \sigma^2).$$

and the prior is

$$\mu \sim \text{Uniform}(-10, 10),$$
$$\ln \sigma \sim \text{Uniform}(-10, 10).$$

However, instead of conditioning on the full dataset $\{x_i\}$, the example uses the minimum and maximum values in the data, $\min(\{x_i\})$ and $\max(\{x_i\})$, as summary statistics[4].

In ABC applications the model class needs to describe a point in the joint (parameters, data) space, and the `from_prior` and `perturb` functions need to respect the joint prior distribution. In Bayesian inference the joint prior tends to make the parameters and data highly dependent – otherwise the data would never be informative about the parameters. To improve efficiency, it is helpful to use a change of variables such that the parameters and data-controlling variables are independent. In this example we use

$$\mu \sim \text{Uniform}(-10, 10),$$
$$\ln \sigma \sim \text{Uniform}(-10, 10),$$
$$n_i \sim \text{Normal}(0, 1),$$

where each of the n_i enables a data point to be calculated using

$$x_i := \mu + \sigma n_i.$$

---

[4]This is not a serious application of ABC, since the likelihood function can easily be evaluated. It is included to demonstrate the principle of using nested sampling as an ABC method.

A general procedure for achieving this is to imagine generating a simulated dataset from the parameters. In this process, you would need to call a random number generator many times. The results of the random number calls are independent of the parameters and can be used to parameterize the dataset. In the example, the $\{n_i\}$ variables play this role. Therefore, the variables in the model class are:

```
class MyModel
{
    private:
        double mu, log_sigma;
        std::vector<double> n;
```

The proposal involves changing either `mu`, `log_sigma`, or one of the `ns`:

```
double MyModel::perturb(RNG& rng)
{
    int which = rng.rand_int(3);

    if(which == 0)
    {
        mu += 20 * rng.randh();
        wrap(mu, -10.0, 10.0);
    }
    if(which == 1)
    {
        log_sigma += 20 * rng.randh();
        wrap(log_sigma, -10.0, 10.0);
    }
    if(which == 2)
    {
        int i = rng.rand_int(n.size());
        n[i] = rng.randn();
    }

    return 0.0;
}
```

The "log-likelihood" is really minus the discrepancy function. Since we parameterized the joint (parameter, data) space using the n variables instead of the data $\{x_i\}$ itself, we must "assemble" the simulated dataset in order to evaluate the discrepancy function:

```
double MyModel::log_likelihood() const
{
    double x_min = Data::get_instance().get_x_min();
    double x_max = Data::get_instance().get_x_max();

    double sigma = exp(log_sigma);
```

```
    // Assemble fake dataset
    vector<double> x_fake = n;
    for(size_t i = 0; i < x_fake.size(); i++)
        x_fake[i] = mu + sigma * x_fake[i];

    // Goodness
    double logL = 0.;
    logL -= pow(*min_element(x_fake.begin(), x_fake.end()) - x_min, 2);
    logL -= pow(*max_element(x_fake.begin(), x_fake.end()) - x_max, 2);

    return logL;
}
```

10.4. ABC example results

For ABC applications, there is an alternate `postprocess` function called `postprocess_abc`. This generates posterior samples using the ϵ value corresponding to level $\approx 0.8\times$ (maximum number of levels).

```
dnest4.postprocess_abc()
```

To try different values of ϵ, you can set the argument `threshold_fraction` to a value other than 0.8. For example

```
dnest4.postprocess_abc(threshold_fraction = 0.6)
```

In the example directory there is a dataset generated from a standard normal distribution. The inference should result in a joint posterior for μ and σ with significant density near $(\mu = 0, \sigma = 1)$. We show posterior samples in Figure 6.
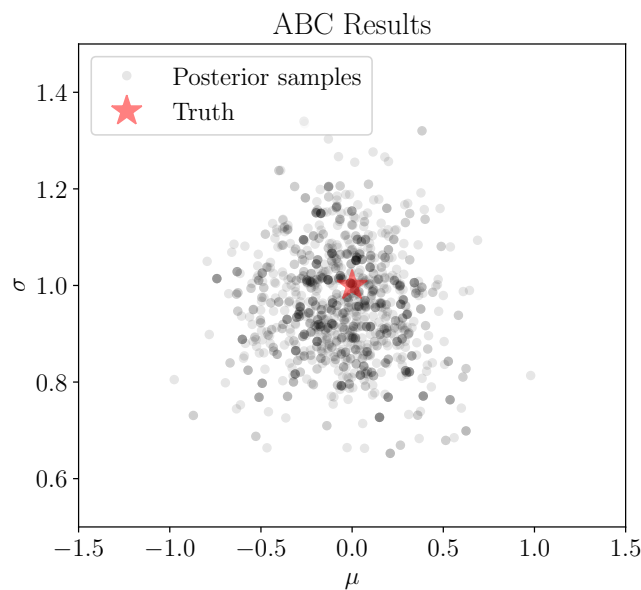

Figure 6: Posterior samples for μ and σ for the ABC demonstration.

11. **Python** bindings

In **DNest4**, it is also possible to specify and run models in Python. Currently, the only models supported are ones where the parameters are a **NumPy** array of double precision quantities. Only a single thread is supported, although parallelization at the level of the Python likelihood function should work.

A model class implementing the straight line example is given below and included in the script python/examples/straightline/straightline.py. The member functions of this class have similar names to those used in C++, but work slightly differently. Most notably, an instance of the class does not represent a point in parameter space, but represents the problem itself. The functions take **NumPy** arrays of parameters as input, or produce them as output, rather than having parameters as data members as in C++.

```python
class Model(object):
    """
    Specify the model in Python.
    """
    def __init__(self):
        """
        Parameter values *are not* stored inside the class
        """
        pass

    def from_prior(self):
        """
        Unlike in C++, this must *return* a numpy array of parameters.
        """
        m = 1E3 * rng.randn()
        b = 1E3 * rng.randn()
        sigma = np.exp(-10.0 + 20.0 * rng.rand())
        return np.array([m, b, sigma])

    def perturb(self, params):
        """
        Unlike in C++, this takes a numpy array of parameters as input,
        and modifies it in-place. The return value is still logH.
        """
        logH = 0.0
        which = rng.randint(3)

        if which == 0 or which == 1:
            logH -= -0.5 * (params[which] / 1E3) **2
            params[which] += 1E3 * dnest4.randh()
            logH += -0.5 * (params[which]/1E3) **2
        else:
            log_sigma = np.log(params[2])
            log_sigma += 20 * dnest4.randh()
```

```python
        # Note the difference between dnest4.wrap in Python and
        # DNest4::wrap in C++. The former *returns* the wrapped value.
        log_sigma = dnest4.wrap(log_sigma, -10.0, 10.0)
        params[2] = np.exp(log_sigma)

    return logH

def log_likelihood(self, params):
    """
    Gaussian sampling distribution.
    """
    m, b, sigma = params
    var = sigma ** 2
    return -0.5 * data.shape[0] * np.log(2 * np.pi * var)\
            - 0.5 * np.sum((data[:,1] - (m * data[:,0] + b)) **2) / var
```

A sampler can then be created and run using the following code. The Python sampler does not load an OPTIONS file. Instead, the options are provided as function arguments when setting up the sampler.

```python
# Create a model object and a sampler
model = Model()
sampler = dnest4.DNest4Sampler(model,
                        backend=dnest4.backends.CSVBackend(".",
                                                    sep=" "))


# Set up the sampler. The first argument is max_num_levels
gen = sampler.sample(max_num_levels=30, num_steps=1000,\
                new_level_interval=10000,\
                num_per_step=10000, thread_steps=100,\
                num_particles=5, lam=10, beta=100, seed=1234)


# Do the sampling (one iteration here = one particle save)
for i, sample in enumerate(gen):
    print("# Saved {k} particles.".format(k=(i+1)))
```

The text file output can be analyzed using `dnest4.postprocess()` as usual.

Acknowledgments

References

Baldock RJN, Pártay LB, Bartók AP, Payne MC, Csányi G (2016). "Determining Pressure-Temperature Phase Diagrams of Materials." *Physical Review B*, **93**(17), 174108. doi:10.1103/physrevb.93.174108.

Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K (2011). "**Cython**: The Best of Both Worlds." *Computing in Science & Engineering*, **13**(2), 31–39. doi:10.1109/mcse.2010.118.

Brewer BJ (2014). "Inference for Trans-Dimensional Bayesian Models with Diffusive Nested Sampling." arXiv:1411.3921 [stat.CO], URL http://arxiv.org/abs/1411.3921.

Brewer BJ, Donovan CP (2015). "Fast Bayesian Inference for Exoplanet Discovery in Radial Velocity Data." *Monthly Notices of the Royal Astronomical Society*, **448**(4), 3206–3214. doi:10.1093/mnras/stv199.

Brewer BJ, Pártay LB, Csányi G (2011). "Diffusive Nested Sampling." *Statistics and Computing*, **21**(4), 649–656. doi:10.1007/s11222-010-9198-8.

Carpenter B, Gelman A, Hoffman M, Lee D, Goodrich B, Betancourt M, Brubaker MA, Guo J, Li P, Riddell A (2017). "**Stan**: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. doi:10.18637/jss.v076.i01.

Caticha A, Giffin A (2006). "Updating Probabilities." *AIP Conference Proceedings*, **872**(1), 31–42. doi:10.1063/1.2423258.

Del Moral P, Doucet A, Jasra A (2012). "An Adaptive Sequential Monte Carlo Method for Approximate Bayesian Computation." *Statistics and Computing*, **22**(5), 1009–1020. doi:10.1007/s11222-011-9271-y.

Dybowski R, Restif O, Goupy A, Maskell DJ, Mastroeni P, Grant AJ (2015). "Single Passage in Mouse Organs Enhances the Survival and Spread of Salmonella Enterica." *Journal of the Royal Society Interface*, **12**(113), 20150702. doi:10.1098/rsif.2015.0702.

Feroz F, Hobson M, Bridges M (2009). "**MultiNest**: An Efficient and Robust Bayesian Inference Tool for Cosmology and Particle Physics." *Monthly Notices of the Royal Astronomical Society*, **398**(4), 1601–1614. doi:10.1111/j.1365-2966.2009.14548.x.

Foreman-Mackey D, Hogg DW, Lang D, Goodman J (2013). "**emcee**: The MCMC Hammer." *Publications of the Astronomical Society of the Pacific*, **125**(925), 306. doi:10.1086/670067.

GCC (2016). *Using the GNU Compiler Collection.* Free Software Foundation. URL https://gcc.gnu.org/onlinedocs/gcc/.

Giffin A, Caticha A (2007). "Updating Probabilities with Data and Moments." In *Bayesian Inference and Maximum Entropy Methods in Science and Engineering*, volume 954, pp. 74–84.

Green PJ (1995). "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination." *Biometrika*, **82**(4), 711–732. `doi:10.1093/biomet/82.4.711`.

Handley WJ, Hobson MP, Lasenby AN (2015). "**POLYCHORD**: Next-Generation Nested Sampling." *Monthly Notices of the Royal Astronomical Society*, **453**(4), 4384–4398. `doi:10.1093/mnras/stv1911`.

Hansmann UHE (1997). "Parallel Tempering Algorithm for Conformational Studies of Biological Molecules." *Chemical Physics Letters*, **281**(1), 140–150. `doi:10.1016/s0009-2614(97)01198-6`.

Huijser D, Goodman J, Brewer BJ (2015). "Properties of the Affine Invariant Ensemble Sampler in High Dimensions." arXiv:1509.02230 [stat.CO], URL `http://arxiv.org/abs/1509.02230`.

Hunter JD (2007). "**Matplotlib**: A 2D Graphics Environment." *Computing in Science & Engineering*, **9**(3), 90–95. `doi:10.1109/mcse.2007.55`.

Huppenkothen D, Brewer BJ, Hogg DW, Murray I, Frean M, Elenbaas C, Watts AL, Levin Y, Van Der Horst AJ, Kouveliotou C (2015). "Dissecting Magnetar Variability with Bayesian Hierarchical Models." *The Astrophysical Journal*, **810**(1), 66. `doi:10.1088/0004-637x/810/1/66`.

Lam SK, Pitrou A, Seibert S (2015). "**Numba**: A LLVM-based Python JIT compiler." In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–7. ACM.

Neal RM (2001). "Annealed Importance Sampling." *Statistics and Computing*, **11**(2), 125–139. `doi:10.1023/a:1008923215028`.

Neal RM (2003). "Slice Sampling." *The Annals of Statistics*, pp. 705–741. `doi:10.1214/aos/1056562461`.

O'Hagan A, Forster JJ (2004). *Kendall's Advanced Theory of Statistics, Volume 2B: Bayesian Inference*, volume 2. Arnold.

Pancoast A, Brewer BJ, Treu T (2014). "Modelling Reverberation Mapping Data – I. Improved Geometric and Dynamical Models and Comparison with Cross-Correlation Results." *Monthly Notices of the Royal Astronomical Society*, **445**(3), 3055–3072. `doi:10.1093/mnras/stu1809`.

Pártay LB, Bartók AP, Csányi G (2010). "Efficient Sampling of Atomic Configurational Spaces." *The Journal of Physical Chemistry B*, **114**(32), 10502–10512. `doi:10.1021/jp1012973`.

Plummer M (2003). "**JAGS**: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Technische Universität Wien, Vienna, Austria. URL `https://www.R-project.org/conferences/DSC-2003/Proceedings/Plummer.pdf`.

Robert CP, Cornuet JM, Marin JM, Pillai NS (2011). "Lack of Confidence in Approximate Bayesian Computation Model Choice." *Proceedings of the National Academy of Sciences*, **108**(37), 15112–15117. doi:10.1073/pnas.1102900108.

Sivia D, Skilling J (2006). *Data Analysis: A Bayesian Tutorial.* OUP Oxford.

Skilling J (2006). "Nested Sampling for General Bayesian Computation." *Bayesian Analysis*, **1**(4), 833–859. doi:10.1214/06-ba127.

Stroustrup B (2013). *The C++ Programming Language, Fourth Edition.* Pearson Education.

Van der Walt S, Colbert SC, Varoquaux G (2011). "The **NumPy** Array: A Structure for Efficient Numerical Computation." *Computing in Science & Engineering*, **13**(2), 22–30. doi:10.1109/mcse.2011.37.

Van Rossum G, *et al.* (2011). *Python Programming Language.* URL https://www.python.org/.

A. Command line options

The following command line options are available.

`-o <filename>`
This option loads the **DNest4** options from the specified file, allowing an alternative to the default OPTIONS.

`-s <seed>`
This seeds the random number generator with the specified value. If unspecified, the system time is used.

`-d <filename>`
Load data from the specified file, if required.

`-c <value>`
Standard DNS creates levels with a volume ratio of approximately $e \approx 2.71828$. To use a different value, such as 10, use this option. In our experience, `-c 10` tends to work better than the default value on problems with a large information $\mathcal{H}$. The output units remain in nats. This option is incompatible with setting the maximum number of levels to 0 (automatic) in the OPTIONS file.

`-t <num_threads>`
Run on the specified number of threads. The default is 1. Be aware that the number of particles specified in the `OPTIONS` file is actually the number of particles *per thread*. Therefore, if running on 8 or more cores, we recommend setting the number of particles per thread to 1, so the number of particles equals the number of threads.

Affiliation:

Brendon J. Brewer
Department of Statistics
The University of Auckland
Private Bag 92019
Auckland, 1142, New Zealand
E-mail: bj.brewer@auckland.ac.nz
URL: https://www.stat.auckland.ac.nz/~brewer/