

---

# **dynesty Documentation**

***Release 1.0.0***

**Josh Speagle**

**Apr 04, 2022**



---

## Contents

---

|          |   |            |
|----------|---|------------|
| <b>1</b> | <b>Installation</b>                                   | <b>3</b>   |
| <b>2</b> | <b>Citations</b>                                      | <b>5</b>   |
| <b>3</b> | <b>Changelog</b>                                      | <b>7</b>   |
| 3.1      | 1.0.0 (2019-09-22) . . . . .                          | 7          |
| 3.2      | 0.9.7 (2019-06-13) . . . . .                          | 7          |
| 3.3      | 0.9.5.3 (2019-03-29) . . . . .                        | 7          |
| 3.4      | 0.9.5 (2019-03-14) . . . . .                          | 8          |
| 3.5      | 0.9.4 (2019-03-07) . . . . .                          | 8          |
| 3.6      | 0.9.3 (2019-02-10) . . . . .                          | 8          |
| 3.7      | 0.9.2 (2018-03-17) . . . . .                          | 8          |
| 3.8      | 0.9.1 (2018-03-01) . . . . .                          | 8          |
| 3.9      | 0.9.0 (2018-02-25) . . . . .                          | 9          |
| 3.10     | 0.8.4 (2018-02-24) . . . . .                          | 9          |
| 3.11     | 0.8.3 (2017-12-13) . . . . .                          | 9          |
| 3.12     | 0.8.2 (2017-09-15) . . . . .                          | 9          |
| 3.13     | 0.8.1 (2017-09-12) . . . . .                          | 9          |
| 3.14     | 0.8.0 (2017-09-08) . . . . .                          | 9          |
|          | 3.14.1 Crash Course . . . . .                         | 10         |
|          | 3.14.2 Background . . . . .                           | 11         |
|          | 3.14.3 Getting Started . . . . .                      | 15         |
|          | 3.14.4 Dynamic Nested Sampling with dynesty . . . . . | 34         |
|          | 3.14.5 Nested Sampling Errors . . . . .               | 48         |
|          | 3.14.6 Examples . . . . .                             | 68         |
|          | 3.14.7 FAQ . . . . .                                  | 96         |
|          | 3.14.8 References and Acknowledgements . . . . .      | 101        |
|          | 3.14.9 API . . . . .                                  | 102        |
|          | <b>Python Module Index</b>                            | <b>149</b> |
|          | <b>Index</b>  | <b>151</b> |



`dynesty` is a Pure Python, MIT-licensed [Dynamic Nested Sampling](#) package for estimating Bayesian posteriors and evidences. See [Crash Course](#) and [Getting Started](#) for more information. The latest development version can be found [here](#).

**The release paper describing the code can be found [here](#).**



# CHAPTER 1

---

## Installation

---

`dynesty` is compatible with both Python 2.7 and Python 3.6. It requires `numpy` (for arithmetic), `scipy` (for special functions), `matplotlib` (for plotting), and `six` (to enforce Python 2/3 compliance). While not required, `tqdm` also allows for a nice progress bar.

Installing the most recent stable version of the package is as easy as:

```
pip install dynesty
```

Alternately, for users who might want newer development versions, it can also be installed directly from a local copy of the repository by running:

```
python setup.py install
```





If you find `dynesty` useful in your research, please cite [Speagle \(2019\)](#). You are also encouraged to cite:

- Nested Sampling: [Skilling \(2004\)](#) and [Skilling \(2006\)](#).
- Dynamic Nested Sampling: [Higson et al. \(2017b\)](#).

You are also encouraged to cite the following papers as relevant:

- Single ellipsoid bound: [Mukherjee, Parkinson & Liddle \(2006\)](#).
- Multiple ellipsoid bounds: [Feroz, Hobson & Bridges \(2009\)](#).
- Overlapping balls/cubes: [Buchner \(2016\)](#) and [Buchner \(2017\)](#).
- Random walks/staggers: [Skilling \(2006\)](#).
- Multivariate/Random slice sampling: [Neal \(2003\)](#), [Handley, Hobson & Lasenby \(2015a\)](#), and [Handley, Hobson & Lasenby \(2015b\)](#).
- Hamiltonian/Reflective slice sampling: [Neal \(2003\)](#), [Skilling \(2012\)](#), and [Feroz & Skilling \(2013\)](#).
- Nested Sampling error analysis: [Chopin & Robert \(2010\)](#) and [Higson et al. \(2017a\)](#).

See *References and Acknowledgements* for additional details.



### 3.1 1.0.0 (2019-09-22)

- Added support for period and reflective boundaries (with [Gregory Ashton](#)).
- Added support for interactive progress bar (with [Daniel Foreman-Mackey](#)).
- Added support for stopping criterion based on ESS (with [Colm Talbot](#)).
- Small bugfixes to code and documentation.
- Small quality-of-life improvements.

### 3.2 0.9.7 (2019-06-13)

- Ensemble bounds can now adapt to elongated distributions (with [Johannes Buchner](#)).
- Random walks now behave differently near boundaries (with [Gregory Ashton](#)).
- Pickling sampler states should now work better in Python 3 (with [Dustin Lang](#)).
- Doubled output errors in default approximation in line with theoretical expectations.
- Small bugfixes and docfixes (with [Patricio Cubillos](#)).

### 3.3 0.9.5.3 (2019-03-29)

- Various small bugfixes, with contributions by [Gregory Ashton](#) and [Johannes Buchner](#).

### **3.4 0.9.5 (2019-03-14)**

- Added support for periodic boundary conditions.
- Set up basic tests for continuous integration.

### **3.5 0.9.4 (2019-03-07)**

- Added a logo!
- Updated and reorganized documentation and demos.
- Added proper support for gradients.
- Changed defaults and added several “quality of life” improvements.

### **3.6 0.9.3 (2019-02-10)**

- Updated documentation.
- Modified re-scaling behavior to better deal with inefficient proposals.
- Improved stability of the current ellipsoid decomposition algorithm.
- Added new 'auto' options and changed a number of defaults to make things easier for general users.
- Plotting now defaults to 95% credible intervals instead of 68%.

### **3.7 0.9.2 (2018-03-17)**

- Added in a fast approximation option for `jitter_run` and `simulate_run`.
- Modified the default stopping heuristic. It now evaluates significantly faster but is a less accurate probe of the “true” KL divergence.
- Modified 'rwalk' behavior to better deal with edge cases.
- Changed defaults so performance should now be more stable (albiet slower) for the average user.
- Improved the stability of bounding ellipsoids.
- Fixed performance issues with 'rslice' and 'hslice'.
- Small plotting improvements.

### **3.8 0.9.1 (2018-03-01)**

- Fixed a minor bootstrapping bug that affected performance for some users.
- Fixed a serious bug associated with the new singular decomposition algorithm and changed its behavior so it no longer auto-kills user runs when it fails.

### 3.9 0.9.0 (2018-02-25)

- `dynesty` is now on PyPI!

### 3.10 0.8.4 (2018-02-24)

- Added two new slice sampling options ('`rslice`' and '`hslice`').
- Changed internals to allow user to access quantities during dynamic batch allocation. **WARNING: Breaks some aspects of backwards compatibility for advanced users utilizing generators.**
- Simplified parallelism options.
- Fixed a singular decomposition bug that occasionally appeared during runtime.
- Small plotting/utility improvements.

### 3.11 0.8.3 (2017-12-13)

- Fixed additional Python 2/3 compatibility bugs.
- Added the ability to pass user-specified custom print functions.
- Added importance reweighting.
- Small improvements to plotting utilities.
- Small changes to improve user outputs and basic functionality.

### 3.12 0.8.2 (2017-09-15)

- Fixed `map` bugs that broke compatibility between Python 2 and 3.
- Fixed a bug where the sampler could break during the first update from the unit cube when using a `pool`.

### 3.13 0.8.1 (2017-09-12)

- Introduced a function wrapper for `prior_transform` and `loglikelihood` functions to allow users to pass `args` and `kwargs`.
- Fixed a small bug that could cause bounding ellipsoids to fail.
- Introduced a stability fix to the default `weight_function` when computing evidence-based weights.

### 3.14 0.8.0 (2017-09-08)

Initial beta release.

### 3.14.1 Crash Course

`dynesty` requires three basic ingredients to sample from a given distribution:

- the likelihood (via a `loglikelihood()` function),
- the prior (via a `prior_transform()` function that transforms samples from the unit cube to the target prior), and
- the dimensionality of the parameter space.

As an example, let's define our likelihood to be a 3-D correlated multivariate Normal (Gaussian) distribution and our prior to be uniform in each dimension from  $[-10, 10]$ :

```
import numpy as np

# Define the dimensionality of our problem.
ndim = 3

# Define our 3-D correlated multivariate normal likelihood.
C = np.identity(ndim) # set covariance to identity matrix
C[C==0] = 0.95 # set off-diagonal terms
Cinv = np.linalg.inv(C) # define the inverse (i.e. the precision matrix)
lnorm = -0.5 * (np.log(2 * np.pi) * ndim +
                np.log(np.linalg.det(C))) # ln(normalization)

def loglike(x):
    """The log-likelihood function."""

    return -0.5 * np.dot(x, np.dot(Cinv, x)) + lnorm

# Define our uniform prior.
def ptform(u):
    """Transforms samples `u` drawn from the unit cube to samples to those
    from our uniform prior within  $[-10., 10.]$  for each variable."""

    return 10. * (2. * u - 1.)
```

Estimating the evidence and posterior is as simple as:

```
import dynesty

# "Static" nested sampling.
sampler = dynesty.NestedSampler(loglike, ptform, ndim)
sampler.run_nested()
sresults = sampler.results

# "Dynamic" nested sampling.
dsampler = dynesty.DynamicNestedSampler(loglike, ptform, ndim)
dsampler.run_nested()
dresults = dsampler.results
```

Combining the results from multiple (independent) runs is easy:

```
from dynesty import utils as dyfunc

# Combine results from "Static" and "Dynamic" runs.
results = dyfunc.merge_runs([sresults, dresults])
```

We can visualize our results using several of the built-in plotting utilities. For instance:

```

from dynesty import plotting as dyplot

# Plot a summary of the run.
rfig, raxes = dyplot.runplot(results)

# Plot traces and 1-D marginalized posteriors.
tfig, taxes = dyplot.traceplot(results)

# Plot the 2-D marginalized posteriors.
cfig, caxes = dyplot.cornerplot(results)

```

We can post-process these results using some built-in utilities. For instance:

```

from dynesty import utils as dyfunc

# Extract sampling results.
samples = results.samples # samples
weights = np.exp(results.logwt - results.logz[-1]) # normalized weights

# Compute 10%-90% quantiles.
quantiles = [dyfunc.quantile(samps, [0.1, 0.9], weights=weights)
              for samps in samples.T]

# Compute weighted mean and covariance.
mean, cov = dyfunc.mean_and_cov(samples, weights)

# Resample weighted samples.
samples_equal = dyfunc.resample_equal(samples, weights)

# Generate a new set of results with statistical+sampling uncertainties.
results_sim = dyfunc.simulate_run(results)

```

### 3.14.2 Background

#### Bayesian Inference

In the context of [Bayesian inference](#), we are often interested in estimating the **posterior**  $P(\Theta|\mathbf{D}, M)$  of a set of **parameters**  $\Theta$  for a given **model**  $M$  given some **data**  $\mathbf{D}$ . This can be factored into a form commonly known as **Bayes' Rule** to give

$$P(\Theta|\mathbf{D}, M) = \frac{P(\mathbf{D}|\Theta, M)P(\Theta|M)}{P(\mathbf{D}|M)} \equiv \frac{\mathcal{L}(\Theta)\pi(\Theta)}{\mathcal{Z}}$$

where

$$P(\mathbf{D}|\Theta, M) \equiv \mathcal{L}(\Theta)$$

is the **likelihood**,

$$P(\Theta|M) \equiv \pi(\Theta)$$

is the **prior**, and

$$P(\mathbf{D}|M) \equiv \mathcal{Z} = \int_{\Omega_{\Theta}} \mathcal{L}(\Theta)\pi(\Theta) d\Theta$$

is the **evidence**, with the integral taken over the entire domain  $\Omega_{\Theta}$  of  $\Theta$  (i.e. over all possible  $\Theta$ ).

For complicated data and models, the posterior is often intractable and must be estimated using numerical methods (see, e.g., [here](#)).

## Nested Sampling

### Overview

**Nested sampling** is a method for estimating the Bayesian evidence  $\mathcal{Z}$  first proposed and developed by [John Skilling](#). The basic idea is to approximate the by integrating the prior in nested “shells” of constant likelihood. Unlike [Markov Chain Monte Carlo \(MCMC\)](#) methods which can only generate samples *proportional to* the posterior, Nested Sampling simultaneously estimates both the evidence and the posterior. It also has a variety of appealing statistical properties, which include:

- well-defined stopping criteria for terminating sampling,
- generating a sequence of independent samples,
- flexibility to sample from complex, multi-modal distributions,
- the ability to derive how statistical and sampling uncertainties impact results *from a single run*, and
- being trivially parallelizable.

**Dynamic Nested Sampling** goes even further by allowing samples to be allocated adaptively during the course of a run to better sample areas of parameter space to maximize a chosen objective function. This allows a particular Nested Sampling algorithm to adapt to the shape of the posterior in real time, improving both accuracy and efficiency.

These points will be discussed elsewhere in the documentation when relevant.

### How It Works

Nested Sampling attempts to estimate  $\mathcal{Z}$  by treating the integral of the posterior over all  $\Theta$  as instead an integral over the **prior volume**

$$X(\lambda) \equiv \int_{\Theta: \mathcal{L}(\Theta) > \lambda} \pi(\Theta) d\Theta$$

contained within an **iso-likelihood contour** set by  $\mathcal{L}(\Theta) = \lambda$  via:

$$\mathcal{Z} = \int_0^{+\infty} X(\lambda) d\lambda = \int_0^1 \mathcal{L}(X) dX$$

assuming  $\mathcal{L}(X(\lambda)) = \lambda$  exists. In other words, if we can evaluate the iso-likelihood contour  $\mathcal{L}_i \equiv \mathcal{L}(X_i)$  associated with a bunch of samples from the prior volume  $1 > X_0 > X_1 > \dots > X_N > 0$ , we can compute the evidence using standard numerical integration techniques (e.g., the [trapezoid rule](#)). Computing the evidence using these “nested shells” is what gives Nested Sampling its eponymous name.

### Basic Algorithm

Draw  $K$  “**live**” points (i.e. particles) from the prior  $\pi(\Theta)$ . At each iteration  $i$ , remove the live point with the lowest likelihood  $\mathcal{L}_i$  and replace it with a new live point *sampled from the prior* subject to the constraint  $\mathcal{L}_{i+1} \geq \mathcal{L}_i$ . It can



be shown through some neat statistical arguments (see *Nested Sampling Errors*) that this sampling procedure actually allows us to estimate the prior volume of the *previous* live point (a “dead” point) as:

$$\ln X_i \approx -\frac{i \pm \sqrt{i}}{K}$$

Once some stopping criteria are reached and sampling terminates, the remaining set of live points are distributed uniformly within the final prior volume. These can then be “recycled” and added to the list of samples.

## Evidence Estimation

The evidence integral can be numerically approximated using a set of  $N$  dead points via

$$\mathcal{Z} = \int_0^1 \mathcal{L}(X) dX \approx \hat{\mathcal{Z}} = \sum_{i=1}^N f(\mathcal{L}_i) f(\Delta X_i) \equiv \sum_{i=1}^N \hat{w}_i$$

where  $\hat{w}_i$  is each point’s estimated weight. For a simple linear integration scheme using rectangles, we can take  $f(\mathcal{L}_i) = \mathcal{L}_i$  and  $f(\Delta X_i) = X_{i-1} - X_i$ . For a quadratic integration scheme using trapezoids (as used in *dynesty*), we instead can take  $f(\mathcal{L}_i) = (\mathcal{L}_{i-1} + \mathcal{L}_i)/2$ .

## Posterior Estimation

We can subsequently estimate posteriors “for free” from the same  $N$  dead points by assigning each sample its associated **importance weight**

$$P(\Theta_i) = P(X_i) \equiv p_i \approx \hat{p}_i = \frac{\hat{w}_i}{\sum_{i=1}^N \hat{w}_i} = \frac{\hat{w}_i}{\hat{\mathcal{Z}}}$$

## Stopping Criteria

The remaining evidence  $\Delta \hat{\mathcal{Z}}_i$  at iteration  $i$  can roughly be bounded by

$$\Delta \hat{\mathcal{Z}}_i \approx \mathcal{L}_{\max} X_i$$

where  $\mathcal{L}_{\max}$  is the the maximum likelihood point contained within the remaining set of  $K$  live points. This essentially assumes that the remaining prior volume interior to the last dead point is a uniform slab with likelihood  $\mathcal{L}_{\max}$ .

This can be turned into a relative stopping criterion by using the (log-)ratio between the current estimated evidence  $\hat{\mathcal{Z}}_i$  and the remaining evidence  $\Delta \hat{\mathcal{Z}}_i$ :

$$\Delta \ln \hat{\mathcal{Z}}_i \equiv \ln \left( \hat{\mathcal{Z}}_i + \Delta \hat{\mathcal{Z}}_i \right) - \ln \hat{\mathcal{Z}}_i$$

Stopping at a given  $\Delta \ln \hat{\mathcal{Z}}_i$  value (`dlogz`) then means sampling until only a *fraction* of the evidence remains unaccounted for.

In general, this error estimate serves as a (rough) upper bound (since  $X_i$  is not exactly known) that can be used for deciding when to stop sampling from an arbitrary distribution while estimating the evidence. Other stopping criteria are discussed in *Dynamic Nested Sampling*.

## Challenges

Nested Sampling has two main theoretical requirements:

1. Samples must be evaluated *sequentially* subject to the likelihood constraint  $\mathcal{L}_{i+1} \geq \mathcal{L}_i$ , and
2. All samples used to compute/replace live points must be **independent and identically distributed (i.i.d.)** random variables *drawn from the prior*.

The first requirement is entirely algorithmic and straightforward to satisfy (even when sampling in parallel). The second requirement, however, is much more challenging if we hope to sample efficiently: while it is straightforward to generate samples from the prior, by design Nested Sampling makes this simple scheme increasingly more inefficient since the remaining prior volume shrinks *exponentially* over time.

Solutions to this problem often involve some combination of:

1. Proposing new live points by “evolving” a copy of one (or more) current live points to new (independent) positions subject to the likelihood constraint, and
2. Bounding the iso-likelihood contours using simple but flexible functions in order to exclude regions with lower likelihoods.

In both cases, it is much easier to deal with uniform (rather than arbitrary) priors. As a result, most nested sampling algorithms/packages (including *dynesty*) are designed to sample within the  $D$ -dimensional unit cube. Samples are transformed back to the original parameter space “on the fly” only when needed to evaluate the likelihood. Accomplishing this requires an appropriate **prior transform**, described in more detail under *Prior Transforms*.

## Typical Sets

One of the elegant features of Nested Sampling is it directly incorporates the ideas behind a *typical set* into the estimation. Since this concept is **crucial** in most Bayesian inference applications but rarely discussed explicitly in applied methods such as MCMC, it is important to take some time to discuss it in more detail.

## Quick Overview

In general, the contribution to the posterior at a given value (position)  $\Theta$  has two components. The first arises from the particular *value* of the posterior itself,  $P(\Theta)$ . The second arises from the total (differential) *volume*  $dV(\Theta)$  encompassed by all  $\Theta$ ’s with the particular  $P(\Theta)$ . We can understand this intuitively: the contributions from a small region with large posterior values can be overwhelmed by contributions from much larger regions with small posterior values.

This “tug of war” between the two elements means that the regions which contribute the most to the overall posterior are those that maximize the joint quantity

$$w(\Theta) \propto P(\Theta) dV(\Theta)$$

This region typically forms a “shell” surrounding the mode (i.e. the *maximum a posteriori (MAP)* value) and is what is usually referred to as the **typical set**. This behavior becomes more accentuated as the dimensionality increases: since volume scales as  $r^D$ , increasing the dimensionality of the problem creates exponentially more volume further away from the posterior mode.

## Typical Sets in Nested Sampling

Under the framework of Nested Sampling, this concept naturally emerges from the concept of integrating the evidence in shells of “prior volume”:

$$\mathcal{Z} = \int_0^1 \mathcal{L}(X) dX \approx \sum_{i=1}^N f(\mathcal{L}_i) f(\Delta X_i)$$

We can see directly that the contribution of a particular iso-likelihood contour  $\mathcal{L}(X)$  to the integral depends both on its “amplitude”  $\mathcal{L}(X)$  along with the (differential) prior volume  $dX$  it occupies. This is maximized when both these quantities are jointly maximized, which occurs over points that represent the typical set. Because of the contribution from the “density” and “volume” terms are clearly seen here, this is sometimes also referred to as the **posterior mass**. Since the posterior importance weights

$$\hat{p}_i \propto \hat{w}_i = f(\mathcal{L}_i) f(\Delta X_i)$$

are also directly proportional to these quantities, Nested Sampling also naturally weights samples by their contribution to the typical set.

## Priors in Nested Sampling

Unlike MCMC or similar methods, Nested Sampling starts by randomly sampling from the entire parameter space specified by the prior. This is not possible unless the priors are “proper” (i.e. that they integrate to 1). So while Normal priors spanning  $(-\infty, +\infty)$  are fine, Uniform priors spanning the same range are not and must be bounded.

**It cannot be stressed enough that the evidence is entirely dependent on the “size” of the prior.** For instance, a wider Uniform prior will decrease the contribution of high-likelihood regions to the evidence estimate, leading to a lower overall value. Priors should thus be carefully chosen to ensure models can be properly compared using the evidences computed from Nested Sampling.

In addition to affecting the evidence estimate, the prior also directly affects the overall expected runtime. Since, in general, the posterior  $P(\Theta)$  is (much) more localized than the prior  $\pi(\Theta)$ , the “information” we gain from updating from the prior to the posterior can be characterized by the **Kullback-Leibler (KL) divergence** (see [here](#) for more information):

$$H \equiv \int_{\Omega_{\Theta}} P(\Theta) \ln \frac{P(\Theta)}{\pi(\Theta)} d\Theta$$

It can be shown/argued that the total number of steps  $N$  needed to integrate over the posterior roughly scales as:

$$N \propto HK$$

In other words, increasing the size of the prior *directly* impacts the amount of time needed to integrate over the posterior. This highlights one of the main drawbacks of nested sampling: **using less “informative” priors will increase the expected number of nested sampling iterations.**

### 3.14.3 Getting Started

#### Prior Transforms

The **prior transform** function is used to implicitly specify the Bayesian prior  $\pi(\Theta)$  for Nested Sampling. It functions as a transformation from a space where variables are i.i.d. within the  $D$ -dimensional unit cube (i.e. uniformly distributed from 0 to 1) to the parameter space of interest. For independent parameters, this would be the product of

the [inverse cumulative distribution function \(CDF\)](#) (also known as the “percent point function” or “quantile function”) associated with each parameter.

It is crucial to note that increasing the size of the prior *directly* impacts the amount of time needed to integrate over the posterior. We highlight some examples of prior transforms below.

### Example: Uniform Priors

Suppose we want our prior to be Uniform from [-10, 10) for all variables:

$$p(x) \propto \begin{cases} 1 & -10 \leq x < 10 \\ 0 & \text{otherwise} \end{cases}$$

The prior transform for this distribution would be:

```
def prior_transform(u):
    """Transforms the uniform random variable `u ~ Unif[0., 1.)`
    to the parameter of interest `x ~ Unif[-10., 10.)`."""

    x = 2. * u - 1. # scale and shift to [-1., 1.)
    x *= 10. # scale to [-10., 10.)

    return x
```

### Example: Non-uniform priors

Suppose we instead have a more complicated prior in 5 variables. The first 2 are drawn from a [bivariate Normal](#) distribution, the third is drawn from a [Beta](#) distribution, the fourth from a [Gamma](#) distribution, and the fifth from a truncated normal distribution. To handle more complicated functions like these, we can use the built-in [functions](#) in `scipy.stats`, which include a **percent point function (ppf)** that is analogous to our prior transform. Using those, our above examples would look like:

```
def prior_transform(u):
    """Transforms the uniform random variables `u ~ Unif[0., 1.)`
    to the parameters of interest."""

    x = np.array(u) # copy u

    # Bivariate Normal
    t = scipy.stats.norm.ppf(u[0:2]) # convert to standard normal
    Csqrt = np.array([[2., 1.],
                      [1., 2.]]) # C^1/2 for C=((5, 4), (4, 5))
    x[0:2] = np.dot(Csqrt, t) # correlate with appropriate covariance
    mu = np.array([5., 2.]) # mean
    x[0:2] += mu # add mean

    # Beta
    a, b = 2.31, 0.627 # shape parameters
    x[2] = scipy.stats.beta.ppf(u[2], a, b)

    # Gamma
    alpha = 5. # shape parameter
    x[3] = scipy.stats.gamma.ppf(u[3], alpha)

    # Truncated Normal
```

(continues on next page)

(continued from previous page)

```

m, s = 5, 2 # mean and standard deviation
low, high = 2., 10. # lower and upper bounds
low_n, high_n = (low - m) / s, (high - m) / s # standardize
x[4] = scipy.stats.truncnorm.ppf(u[4], low_n, high_n, loc=m, scale=s)

return x

```

### Example: Conditional priors

This procedure can be generalized to construct priors that only can be expressed in conditional form. As an example, let's assume we have a three-parameter model where the prior for the third parameter depends on the values for the first two. This might be the case in, e.g., a [hierarchical](#) model where the prior over  $c$  is a Normal distribution whose mean  $m$  and standard deviation  $s$  are determined by a corresponding “hyper-prior”. We can easily set up a prior transform for this model by just going through the variables in order. This would look like:

```

def prior_transform(u):
    """Transforms the uniform random variables `u ~ Unif[0., 1.)`
    to the parameters of interest."""

    x = np.array(u) # copy u

    # Mean hyper-prior
    mu, sigma = 5., 1. # mean, standard deviation
    x[0] = scipy.stats.norm.ppf(u[0], loc=mu, scale=sigma)

    # Standard deviation hyper-prior
    x[1] = 10. ** (u[1] * 2. - 1.) # log10(std) ~ Uniform[-1, 1]

    # Prior
    x[2] = scipy.stats.norm.ppf(u[2], loc=x[0], scale=x[1])

    return x

```

More complicated dependencies can be constructed using similar approaches.

### Nested Sampling with dynesty

To give a concrete example of running dynesty on a real problem, let's return to the simple 3-D multivariate normal likelihood and uniform prior from [-10, 10) used in [Crash Course](#) to define the `loglikelihood()` and `prior_transform()` functions:

```

import numpy as np

# Define the dimensionality of our problem.
ndim = 3

# Define our 3-D correlated multivariate normal log-likelihood.
C = np.identity(ndim)
C[C==0] = 0.95
Cinv = linalg.inv(C)
lnorm = -0.5 * (np.log(2 * np.pi) * ndim +
                np.log(np.linalg.det(C)))

```

(continues on next page)

(continued from previous page)

```
def loglike(x):
    return -0.5 * np.dot(x, np.dot(Cinv, x)) + lnorm

# Define our uniform prior via the prior transform.
def ptform(u):
    return 20. * u - 10.
```

## Initialization

Nested Sampling in `dynesty` is done via a particular `sampler` object that is initialized from the *Top-Level Interface*. To start, let's use `NestedSampler()` to initialize a particular sampler from `nestedsamplers`. There are only 3 required arguments: a log-likelihood function (`loglike`), a prior transform function (`ptform`), and the number of dimensions taken by the loglikelihood (`ndim`).

Using the functions above, we can initialize our sampler using:

```
from dynesty import NestedSampler

# initialize our nested sampler
sampler = NestedSampler(loglike, ptform, ndim)
```

See *Top-Level Interface* for more details on the API, *Examples* for more examples of usage, and *FAQ* for some additional advice. Here we'll go over just the basics.

## Live Points

Similar to ensemble sampling methods such as `emcee`, the behavior of Nested Sampling can also be sensitive to the number of live points used. Increasing the number of live points leads to smaller changes in the prior volume  $\ln X$  over time. This improves the effective resolution while simultaneously increasing the runtime.

In addition, the number of live points can also affect the stability of our *Bounding Options*. By default, `dynesty` inflates the size of the chosen bounds by an enlargement factor to ensure they effectively bound the iso-likelihood contours. These bounds become more robust the more live points are used, leading to more efficient proposals.

It is important to note that running with too few live points can lead to mode “die off”. When there are multiple modes with live points distributed between them, live points can randomly “jump” between them at any given iteration. If there are only a handful of live points at a particular mode, it is possible that, entirely by chance, all of them could transfer completely to the other mode even as both remain equally likely, leading it to “die off” and likely never be located again. As a rule-of-thumb, you should allocate around 50 live points per possible mode to guard against this.

The number of live points can be specified upon initialization via the `nlive` argument. For example, if we want to run with 1000 live points rather than the default 250, we would use:

```
NestedSampler(loglike, ptform, ndim, nlive=1500)
```

## Bounding Options

`dynesty` supports a number of options for bounding the target distribution:

- **no bound** ('none'), i.e. sampling from the entire unit cube,
- **a single bounding ellipsoid** ('single'),
- **multiple** (possibly overlapping) **bounding ellipsoids** ('multi'),

- **overlapping balls** centered on each live point ('balls'), and
- **overlapping cubes** centered on each live point ('cubes').

By default, `dynesty` uses multi-ellipsoidal decomposition ('multi'), which often is flexible enough to capture the complexity of many likelihood distributions while simple enough to quickly and efficiently generate new samples. For more complex distributions, overlapping balls ('balls') or cubes ('cubes') can generate more flexible bounding distributions but come with significantly more overhead that can be less efficient at generating samples. For simpler distributions, a single ellipsoid ('single') is often sufficient. Sampling directly from the unit cube ('none') is extremely inefficient but is a useful option to verify your results and look for possible biases. It otherwise should only be used if the log-likelihood is trivial to compute.

Specifying the particular bounding distribution can be done upon initialization via the `bound` argument. If we wanted to sample using overlapping balls rather than multiple bounding ellipsoids, for instance, we would use:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls')
```

As mentioned in [Live Points](#), bounding distributions in `dynesty` are enlarged in an attempt to conservatively encompass the iso-likelihood contour associated with each dead point. The default behavior increases the volume by 25%, although this can also be done in real-time using bootstrapping methods (this procedure can lead to some instability in the size of the bounds if fewer than the optimal number of live points are being used; see the [FAQ](#) for additional details). The volume enlargement factor and/or the number of bootstrap realizations used can be specified using the `enlarge` and `bootstrap` arguments.

For instance, if we want to use 50 bootstraps to determine expansion factors with an additional fixed volume enlargement factor of 10%, we would specify:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls',
              bootstrap=50, enlarge=1.10)
```

Additional information on the bounding objects can be found under [Bounding](#) and in [Examples](#).

To avoid excessive overhead spent constructing bounding distributions, `dynesty` only updates bounding distributions after a fixed number of likelihood calls specified by the `update_interval` argument. Larger values generally decrease the sampling efficiency but can improve overall performance. This value by default is set to different values for different sampling methods (see the [API](#) for additional details), but if we wanted to instead use a particular value we could just specify that via:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls',
              bootstrap=50, enlarge=1.10, update_interval=1.2)
```

Passing a float like 1.2 sets the update interval to be after `round(1.2 * nlive)` functional calls so that it scales based on the number of live points (and thus the speed at which we expect to traverse the prior volume). If we'd like to specify the number of function calls directly, however, we can instead pass an integer:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls',
              bootstrap=50, enlarge=1.10, update_interval=600)
```

This now specifies that we will update our bounds after 600 function calls.

`dynesty` tries to avoid constructing bounding distributions early in the run to avoid issues where the bounds can significantly exceed the unit cube. For instance, in most cases the bounding distribution of the initial set of points by *construction* will exceed the bounds of the unit cube when `enlarge > 1`. This can lead to a variety of problems associated with each method, especially in higher dimensions (since volume scales as  $\propto r^D$ ).

To avoid this behavior, `dynesty` deliberately delays the first bounding update until at least `2 * nlive` function calls have been made *and* the efficiency has fallen to 10%. This generally assumes that the overall efficiency will be below 10%, which is the case for almost all sampling methods (see below). If we wanted to adjust this behavior so

that we construct our first bounding distributions much earlier, we could do so by passing some parameters using the `first_update` argument:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls',
              bootstrap=50, enlarge=1.10, update_interval=600,
              first_update={'min_ncall': 100, 'min_eff': 50.})
```

This will now trigger an update when 100 log-likelihood function calls have been made and the efficiency drops below 50%.

For specific problems, `dynesty` also enables the use of **periodic boundary conditions**. This allows points to wrap around the unit cube (once), which can help with sampling parameters with periodic boundary conditions whose solutions end up near the bounds (e.g., 0 or  $2\pi$  for phases). These can be enabled by just specifying the indices of the relevant periodic parameters, as shown below:

```
NestedSampler(loglike, ptform, ndim, nlive=1500, bound='balls',
              periodic=[0, 2], bootstrap=50, enlarge=1.10,
              update_interval=600, first_update={'min_eff': 25.})
```

See [Top-Level Interface](#) for more information.

## Sampling Options

`dynesty` also supports several different sampling methods *conditioned on* the provided bounds which can be passed via the `sample` argument:

- **uniform** sampling ('unif'),
- **random walks** away from a current live point ('rwalk'),
- **random “staggering”** away from a current live point ('rstagger'),
- **multivariate slice sampling** away from a current live point ('slice'),
- **random slice sampling** away from a current live point ('rslice'), and
- **“Hamiltonian” slice sampling** away from a current live point ('hslice').

By default, `dynesty` automatically picks a sampling method based on the dimensionality of the problem via the 'auto' argument, which uses the following logic:

- If  $D < 10$ , 'unif' is chosen since uniform proposals can be quite efficient in low dimensions.
- If  $10 \leq D \leq 20$ , 'rwalk' is chosen since random walks are more robust to underestimated bounding distributions in higher dimensions,
- If  $D > 20$  and a gradient is not provided, 'slice' is chosen since non-rejection sampling methods scale in polynomial (rather than exponential) time as the dimensionality increases.
- If  $D > 20$  and a gradient *is* provided, 'hslice' is chosen to take advantage of Hamiltonian dynamics, which scale better than 'slice' as the dimensionality increases.

'rslice' and 'rstagger' can be quite effective for particular problems but currently are not considered as “robust” as the approaches above. **Use them at your own risk.**

One benefit to using random walks or slice sampling is that they require many fewer live points to adapt to structure in higher dimensions (since they only sample *conditioned* on the bounds, rather than **within** them). They also do not require any sort of bootstrap-style corrections since they contain built-in methods to tune their step sizes. This, however, does not mean that they are immune to issues that arise when running with fewer live points such as mode “die-off” (see [Live Points](#)).



Following the example above, let’s say we wanted to combine the flexibility of multiple bounding ellipsoids and slice sampling. This might look something like:

```
NestedSampler(loglike, ptform, ndim, bound='multi', sample='slice')
```

See [Top-Level Interface](#) for additional information.

## Parallel Support

If you want to run computations in parallel, `dynesty` can use a user-defined `pool` to execute a variety of internal operations in “parallel” rather than in serial. This can be done by passing the `pool` object to the sampler upon initialization:

```
# initialize sampler with pool
sampler = NestedSampler(loglike, ptform, ndim, pool=pool)
```

By default, `dynesty` tries to grab the size of the pool from the `pool.size` attribute of the `pool`. If this is not defined, the number of function evaluations to execute in parallel can be set manually using the `queue_size` argument:

```
# initialize sampler with pool with pre-defined queue
sampler = NestedSampler(loglike, ptform, ndim, pool=pool, queue_size=8)
```

Parallel operations in `dynesty` are done by simply swapping in the `pool.map` function over the default `map` function when making likelihood calls. Note that this is a *synchronous* function call, which requires that all members of the pool have completed their respective tasks before receiving the pool’s output. The call time for functions is therefore limited by the slowest-performing member of the pool.

The reason why “parallel” is written in quotes above is that while function evaluations can be made in parallel, live point proposals must be done serially in order to avoid breaking the statistical properties of Nested Sampling. Assuming we are using  $M$  processes with  $K$  live points, this leads to sub-linear scaling  $S$  of the form (Handley et al. 2015):

$$S(M, K) = K \ln \left( 1 + \frac{M}{K} \right)$$

This scales pretty linearly as long as the number of processes is much smaller than the number of live points, but falls off as the pool becomes relatively larger.

Depending on where the bottleneck of the computation lies, the provided `pool` can be disabled during certain function evaluations (e.g., when initializing points) using the `use_pool` argument:

```
# initialize sampler with pool with pre-defined queue
sampler = NestedSampler(loglike, ptform, ndim,
                        nlive=2000, bound='single', sample='rwalk',
                        pool=pool, queue_size=16,
                        use_pool={'prior_transform': False})
```

See [Pool Questions](#) on the [FAQ](#) page for additional troubleshooting tips.

Note that, as discussed in [Combining Runs](#), it is actually possible to combine multiple independent Nested Sampling runs into a single run, giving users an option as to whether they want to parallelize `dynesty` *during* runtime (using a user-provided `pool`) or *after* runtime (by merging the runs together).

## Running Internally

Sampling from our target distribution can be done using the `run_nested()` function in the provided `sampler`:

```
sampler.run_nested()
```

Sampling will continue until specified stopping criteria are reached, and the current state of the sampler is by default output to `stderr` in real time. The stopping criteria can be any combination of:

- a fixed number of iterations (`maxiter`),
- a fixed number of likelihood calls (`maxcall`),
- a maximum log-likelihood (`logl_max`),
- a specified  $\Delta \ln \hat{Z}_i$  tolerance (`dlogz`), and
- a specified Effective Sample Size (**ESS**).

For instance, running one of the examples above would produce output like:

Out:

```
iter: 12521 | +1500 | bound: 7 | nc: 1 | ncall: 66884 | eff(%): 20.963 |
loglstar: -inf < -0.301 < inf | logz: -8.960 +/- 0.082 |
dlogz: 0.001 > 1.509
```

From left to right, this records: the current iteration (plus the number of live points added after stopping), the current bound being used, the number of log-likelihood calls made before accepting the last sample, the total number of log-likelihood calls, the overall sampling efficiency, the current log-likelihood and log-likelihood bounds (`-inf` and `inf` because we began sampling from the prior and didn't declare a `logl_max`), the current estimated evidence, and the remaining `dlogz` relative to the stopping criterion.

By default, the stopping criteria are optimized for evidence estimation, with posteriors treated as a nice byproduct. We can modify this by passing in something like:

```
sampler.run_nested(dlogz=0.5, maxiter=10000, maxcall=50000)
```

Since sampling is done through the `sampler` objects, users can also continue to add new samples based on where they left off. This is as easy as:

```
# initialize our sampler
sampler = NestedSampler(loglike, ptform, ndim, nlive=1000)

# start our run
sampler.run_nested(dlogz=0.5)
res1 = sampler.results

# (possibly) add more samples
sampler.run_nested(maxcall=10000)
res2 = sampler.results

# (possibly) add more samples again
sampler.run_nested(dlogz=0.01)
res3 = sampler.results
```

## Running Externally

Similar to `emcee`, `sampler` objects in `dynesty` can also be run externally as a **generator** via the `sample()` function. This might look something like:

```
# The main nested sampling loop.
for it, res in enumerate(sampler.sample(dlogz=0.5)):
    pass

# Adding the final set of live points.
for it_final, res in enumerate(sampler.add_live_points()):
    pass
```

as opposed to:

```
# The main nested sampling loop.
sampler.run_nested(dlogz=0.5, add_live=False)

# Adding the final set of live points.
sampler.add_final_live()
```

This can be extremely useful if you would like to manipulate the results in real-time, generate plots, save intermediate outputs, etc.

## Combining Runs

Nested sampling is “trivially parallelizable”, which makes it really straightforward to combine the results from multiple independent runs. `dynesty` contains built-in utilities for combining results from separate runs into a single run with improved posterior/evidence estimates. This can be extremely useful if, for instance, you have performed multiple independent analyses over the course of a project that you would like to combine, or if you want to add additional samples to a preliminary analysis (but don’t have the `sampler` currently loaded in memory).

`dynesty` makes this process relatively straightforward. An example is shown below:

```
from dynesty import utils as dyfunc

# Create several independent nested sampling runs.
sampler = NestedSampler(loglike, ptform, ndim)
rlist = []
for i in range(10):
    sampler.run_nested()
    rlist.append(sampler.results)
    sampler.reset()

# Merge into a single run.
results = dyfunc.merge_runs(rlist)
```

This process works with *Dynamic Nested Sampling* as well. See *Unraveling/Merging Runs* for additional details.

## Sampling with Gradients

As mentioned in *Sampling Options*, `dynesty` can utilize log-likelihood gradients  $\nabla \ln \mathcal{L}$  by proposing new samples using Hamiltonian dynamics (often referred to as **reflective slice sampling**). However, because sampling in `dynesty` occurs on the *unit cube* ( $\mathbf{u}$ ) rather than in the target space of our original variables ( $\mathbf{x}$ ), these gradients have to be defined with respect to  $\mathbf{u}$  rather than  $\mathbf{x}$  even though they are evaluated at  $\mathbf{x}$ . This requires computing the Jacobian matrix  $\mathbf{J}$  from  $\mathbf{x}$  to  $\mathbf{u}$ .

While this Jacobian might seem difficult to derive, it can be shown that given independent priors on each parameter

$$\pi(\mathbf{x}) = \prod_i \pi_i(x_i)$$

where  $\pi_i(x_i)$  is the prior for the  $i$ -th parameter  $x_i$  that the Jacobian is diagonal where each diagonal element is simply

$$J_i i = 1/\pi_i(x_i)$$

By default, `dynesty` assumes that any gradient you pass in **already has the appropriate Jacobian applied**. If not, you can tell `dynesty` to numerically estimate the Jacobian by setting `compute_jac=True`.

For the simple 3-D multivariate normal likelihood and uniform prior from [-10, 10) used in *Crash Course*, sampling with gradients would look something like:

```
import numpy as np
import dynesty

# Define the dimensionality of our problem.
ndim = 3

# Define our 3-D correlated multivariate normal log-likelihood.
C = np.identity(ndim)
C[C==0] = 0.95
Cinv = linalg.inv(C)
lnorm = -0.5 * (np.log(2 * np.pi) * ndim +
                np.log(np.linalg.det(C)))

def loglike(x):
    return -0.5 * np.dot(x, np.dot(Cinv, x)) + lnorm

# Define our uniform prior via the prior transform.
def ptform(u):
    return 20. * u - 10.

# Define our gradient with and without the Jacobian applied.
def grad_x(x):
    return -np.dot(Cinv, x) # without Jacobian

def grad_u(x):
    return -np.dot(Cinv, x) * 20. # with Jacobian for uniform [-10, 10)

# Sample with `grad_u` (including Jacobian).
sampler = dynesty.NestedSampler(loglike, ptform, ndim, sample='hslice',
                               gradient=grad_u)
sampler.run_nested()
results_with_jac = sampler.results

# Sample with `grad_x` (compute Jacobian numerically).
sampler = dynesty.NestedSampler(loglike, ptform, ndim, sample='hslice',
                               gradient=grad_x, compute_jac=True)
sampler.run_nested()
results_without_jac = sampler.results
```

For other independent priors discussed in *Prior Transforms*, we can use the built-in functions in `scipy.stats`, which include a **probability density function (pdf)** that is exactly our desired  $\pi_i(v_i)$  function. These then enable us to compute and apply the (diagonal) Jacobian matrix directly. In more complex cases such as the simple hierarchical model in *Example: Conditional priors*, however, we may need to resort to estimating the Jacobian numerically to deal with the expected off-diagonal terms.

## Results

Sampling results can be accessed through the `results` property and are returned as a (modified) dictionary:

```
results = sampler.results
```

We can print a quick summary of the run using `summary()`, which provides basic information about the evidence estimates and overall sampling efficiency:

```
# Print out a summary of the results.
res1.summary()
res2.summary()
```

Out:

```
Summary
=====
nlive: 1000
niter: 6718
ncall: 39582
eff(%): 19.499
logz: -8.832 +/- 0.132

Summary
=====
nlive: 1000
niter: 13139
ncall: 49499
eff(%): 28.564
logz: -8.818 +/- 0.084
```

## Quick Rundown

While a number of quantities are contained in the `Results` instance, the relevant quantities for most users will be the collection of samples from the run (`samples`), their corresponding (unnormalized) log-weights (`logwt`), the cumulative log-evidence (`logz`), and the *approximate* error on the evidence (`logzerr`). The remaining quantities are used to help visualize the output (see [Visualizing Results](#)) and might also be useful for more advanced users who want additional information about the nested sampling run.

## Full Summary

As a dictionary, the full set of quantities provided in `Results` can be accessed using `keys()`. A description of the full set of quantities included in `Results` are listed below:

- `nlive`: the number of live points used in the run,
- `niter`: the number of iterations (samples),
- `ncall`: the total number of function calls,
- `eff`: the overall sampling efficiency,
- `samples`: the set of samples in the *native parameter space*,
- `samples_u`: the set of samples in the *unit cube*,
- `samples_id`: the unique particle index associated with each sample,
- `samples_it`: the iteration the sample was *originally* proposed,
- `logwt`: the log-weight (unnormalized) associated with each sample,

- `logl`: the log-likelihood associated with each sample,
- `logvol`: the (expected)  $\ln(\text{prior volume})$  associated with each sample,
- `logz`: the cumulative evidence at each iteration (sample),
- `logzerr`: the estimated error (standard deviation) on `logz`, and
- `information`: the estimated “information” (see Role of Priors in Nested Sampling) at each iteration (sample).

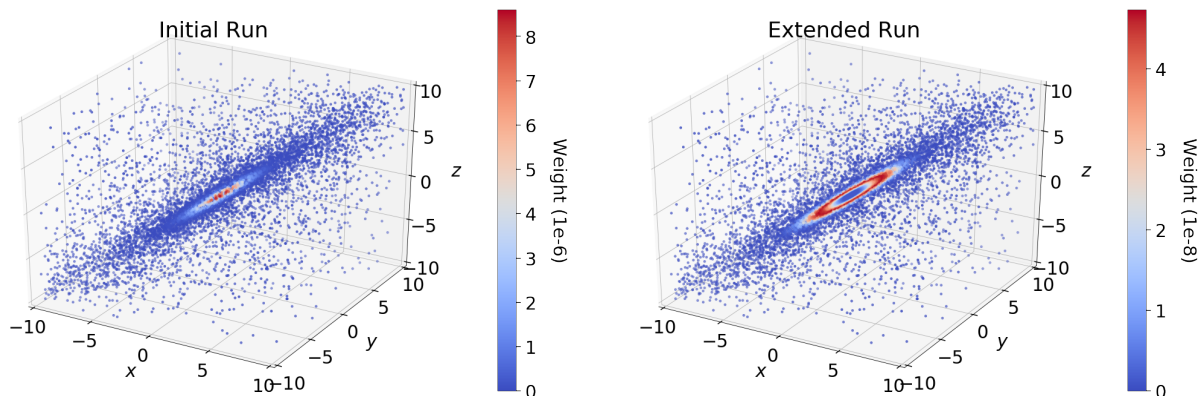
If the bounding distributions are also saved (the default behavior), then the following quantities are also provided:

- `bound`: a (deep) copy of the set of bounding objects,
- `bound_iter`: the index of the bounding object active at a given iteration,
- `samples_bound`: the index of the bounding object the sample was *originally proposed from*, and
- `scale`: the scale-factor used at a given iteration (used to scale the bounds for non-uniform proposals).

Note that some of these quantities change when using *Dynamic Nested Sampling*.

## Visualizing Results

Assuming we’ve completed a run and stored the resulting `res1` and `res2` *Results* dictionaries as defined above, we can compare what their relative weights by comparing them directly, as shown below.



In the initial run (`res1`), we see that the majority of the importance weight  $\hat{p}_i$  is concentrated near the mode; in the extended run, however, it is instead concentrated in a ring around the mode. This behavior represents the fundamental compromise between the likelihood  $\mathcal{L}_i$  and the change in prior volume  $\Delta X_i$ . The stark difference in the distribution of weights between the two samples is driven entirely by differences in  $\Delta X_i$ . In the extended run (`res2`), the distribution of weights directly follows the shape expected from the “typical set” (see *Typical Sets* for additional discussion).

By contrast, since the final set of live points after  $N$  samples are uniformly sampled within  $X_{i=N}$ , the expected change in the prior volume is *constant*. This leads to *linear* (rather than exponential) compression of the remaining prior volume, where the weight assigned to the live point with the  $k$ -th lowest likelihood is then  $\propto f(\mathcal{L}_{N+k}) X_N$ . In the case where there is a significant portion of prior volume remaining (as with `res1`), this leads to extremely rapid traversal of the remaining prior volume and hence large importance weights.

## dyplot

To avoid introducing an excessive burden on typical users, `dynesty` comes with a variety of built-in plotting utilities in the `plotting` module. These include a variety of generic summary plots as well as ways of visualizing bounding distributions throughout the course of a run. We can import them using:

```
from dynesty import plotting as dyplot
```

The `dyplot` alias will be used for convenient shorthand throughout the remainder of the documentation. While some basic usage will be demonstrated below, please see the [API](#) for additional details.

One important note is that **the default credible intervals in all plotting utilities are defined to be 95% (2-sigma) rather than 68% (1-sigma)**. This is a deliberate choice meant to highlight more realistic uncertainties (1-in-3 vs 1-in-20 chances) and better capture possible secondary solutions at the 2.5% level rather than the roughly 16% level.

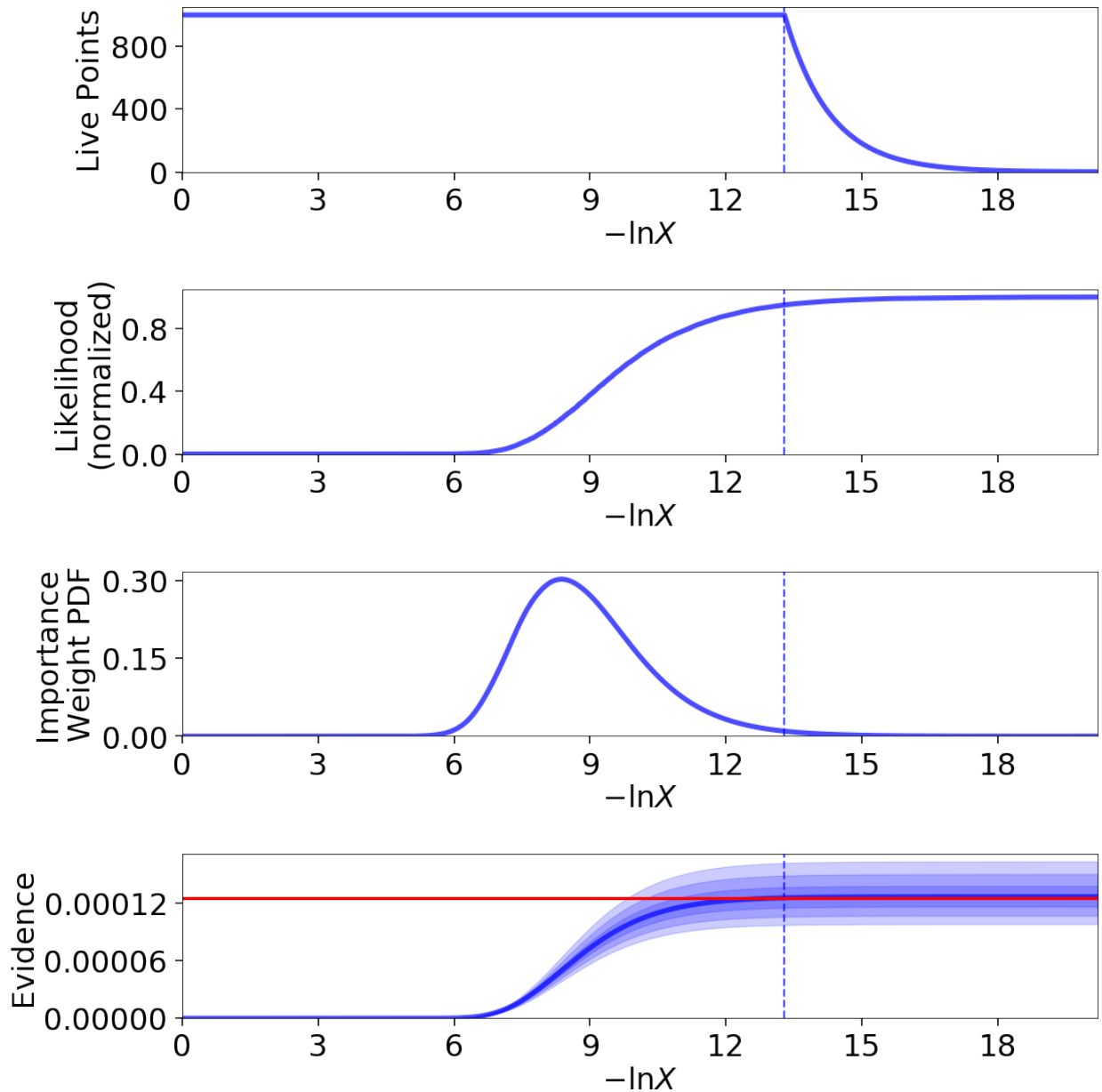
## Summary Plots

One of the most direct ways of visualizing how Nested Sampling computes the *evidence* is by examining the relationship between the prior volume  $\ln X_i$  and:

1. the (effective) iteration  $i$ , which illustrates how quickly/slowly our samples are compressing the prior volume,
2. the likelihood  $\mathcal{L}_i$ , to see how smoothly we sample “up” the likelihood distribution to the [maximum likelihood \(ML\) estimate](#),
3. the importance weight  $\hat{p}_i$ , showcasing where the bulk of the **posterior mass** is located (i.e. the typical set), and
4. the evidence  $\hat{Z}_i$ , to see where most of the contribution to the evidence (and its respective errors) are coming from.

A **summary (run) plot** showcasing these features can be generated using `runplot()`. As an example, a summary plot for `res2` comparing it to the actual analytic  $\ln \mathcal{Z}$  evidence solution can be generated using:

```
lnz_truth = ndim * -np.log(2 * 10.) # analytic evidence solution
fig, axes = dyplot.runplot(res2, lnz_truth=lnz_truth) # summary (run) plot
```



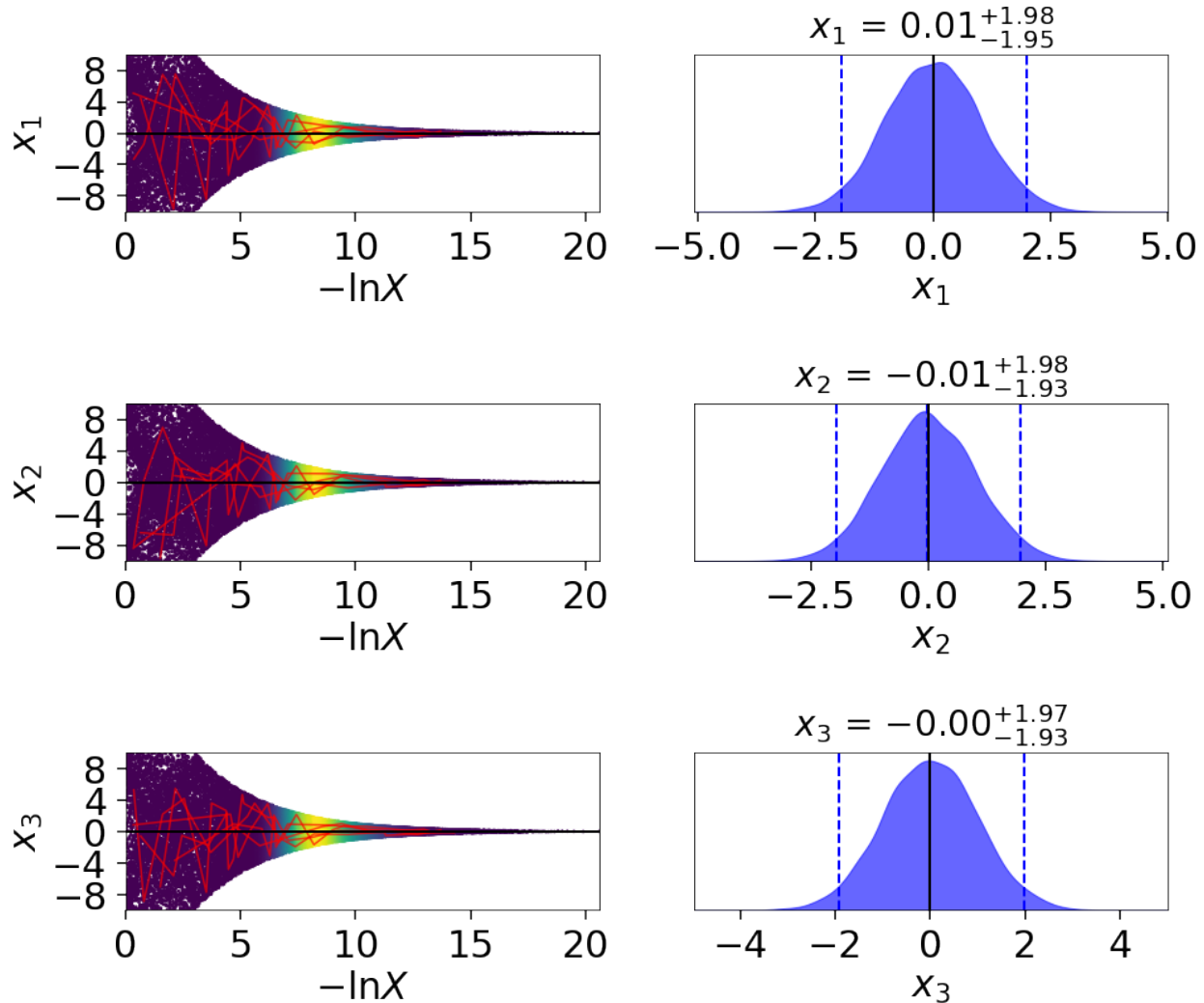
Up until we recycle our final set of live points (see [Basic Algorithm](#)), as indicated by the dashed lines, the relationship between  $\ln X_i$  and  $i$  is linear (i.e. prior volume compression is exponential). Afterwards, however, it stretches out, rapidly traversing the remaining prior volume in linear fashion. Comparing the general shape of the likelihood and importance weights subplots also highlight how the typical set is as much a function of  $\Delta X_i$  as  $\mathcal{L}_i$ : although contributions initially increase as the likelihood increases, they quickly fall as the ML region encompasses increasingly smaller effective volumes.

### Trace Plots

Another common way to visualize the results of many sampling algorithms is to generate a **trace plot** showing the evolution of particles (and their marginal posterior distributions) in 1-D projections. This can be done using the `traceplot()` function, which plots a combination of particle positions as a function of  $\ln X$  (colored by importance weight) and the corresponding 1-D marginalized posterior:



```
fig, axes = dyplot.traceplot(res2, truths=np.zeros(ndim),
                             truth_color='black', show_titles=True,
                             trace_cmap='viridis', connect=True,
                             connect_highlight=range(5))
```



By default, `traceplot()` returns the samples color-coded by their relative posterior mass and the 1-D marginalized posteriors smoothed by a Normal (Gaussian) kernel with a standard deviation set to  $\sim 2\%$  of the provided range (which defaults to the 5-sigma bounds computed from the set of weighted samples). It also can overplot input truth vectors as well as highlight specific particle paths (shown above) to inspect the behavior of individual particles. These can be useful to qualitatively identify problematic behavior such as strongly correlated samples.

### Corner Plots

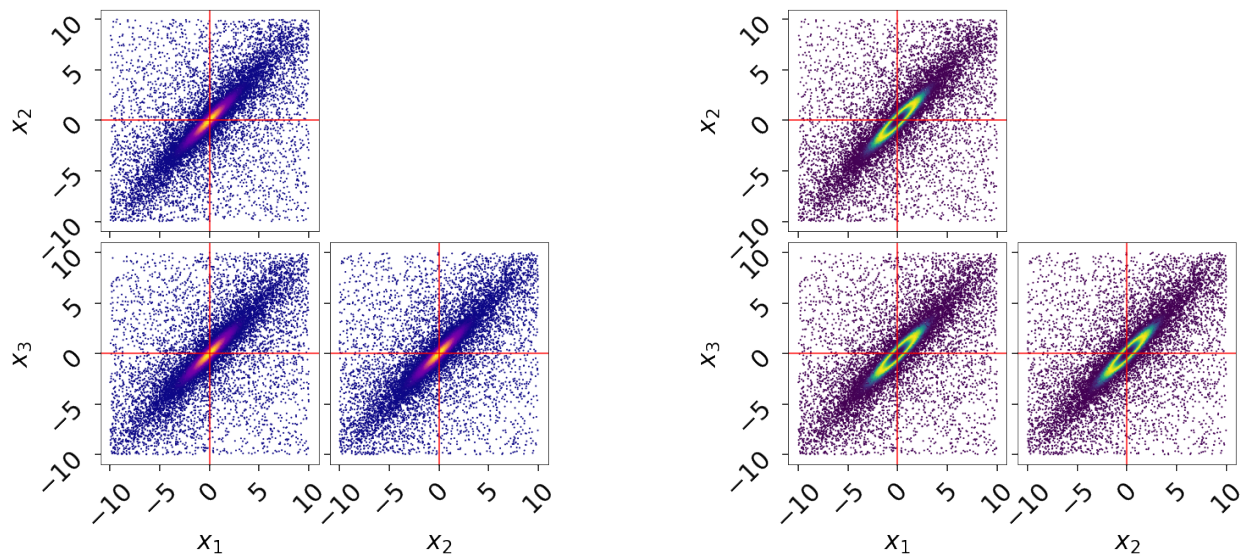
In addition to trace plots, another common way to visualize (weighted) samples is using **corner plots** (also called “triangle plots”), which show a combination of 1-D and 2-D marginalized posteriors. `dynesty` supports several corner plotting functions. The most straightforward is `cornerpoints()`, which simply plots the sample positions colored according to their estimated posterior mass if `kde=True` and raw importance weights if `kde=False`. An example highlighting the difference between the two runs is shown below:

```
# initialize figure
fig, axes = plt.subplots(2, 5, figsize=(25, 10))
axes = axes.reshape((2, 5)) # reshape axes

# add white space
[a.set_frame_on(False) for a in axes[:, 2]]
[a.set_xticks([]) for a in axes[:, 2]]
[a.set_yticks([]) for a in axes[:, 2]]

# plot initial run (res1; left)
fg, ax = dyplot.cornerpoints(res1, cmap='plasma', truths=np.zeros(ndim),
                             kde=False, fig=(fig, axes[:, :2]))

# plot extended run (res2; right)
fg, ax = dyplot.cornerpoints(res2, cmap='viridis', truths=np.zeros(ndim),
                             kde=False, fig=(fig, axes[:, 3:]))
```



Just by looking at our projected samples, it is apparent that the results from the extended run `res2` does a much better job of localizing the overall distribution compared to `res1`. We can get a better qualitative and quantitative handle on this by plotting the marginal 1-D and 2-D posterior density estimates using `cornerplot()` as:

```
# initialize figure
fig, axes = plt.subplots(3, 7, figsize=(35, 15))
axes = axes.reshape((3, 7)) # reshape axes

# add white space
[a.set_frame_on(False) for a in axes[:, 3]]
[a.set_xticks([]) for a in axes[:, 3]]
[a.set_yticks([]) for a in axes[:, 3]]

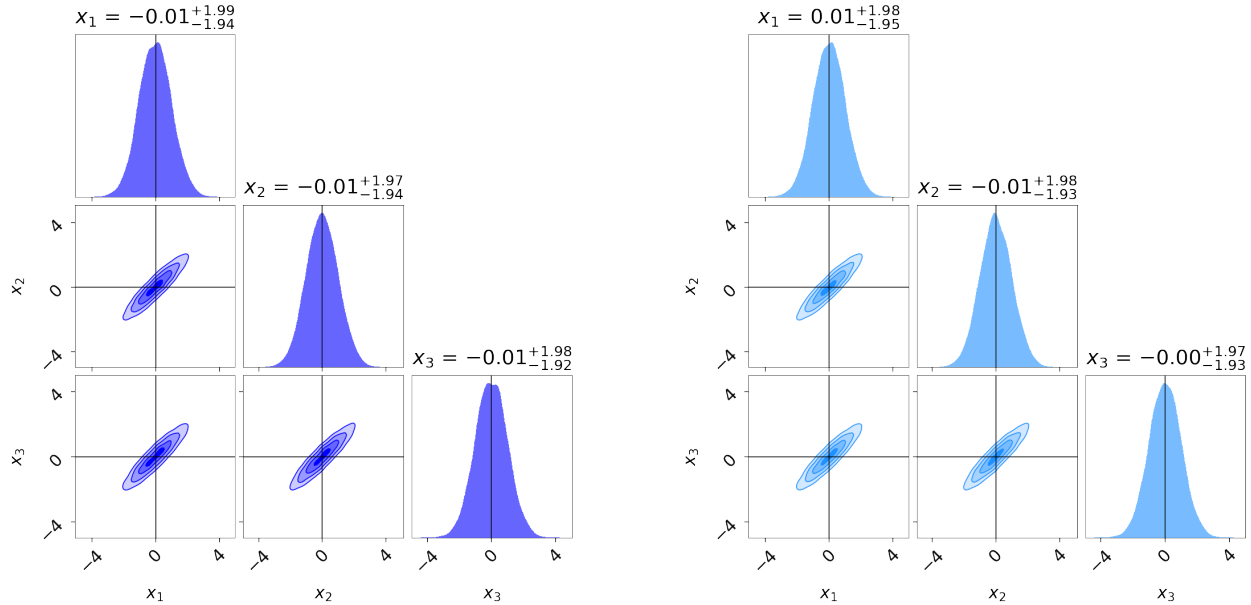
# plot initial run (res1; left)
fg, ax = dyplot.cornerplot(res1, color='blue', truths=np.zeros(ndim),
                           truth_color='black', show_titles=True,
                           max_n_ticks=3, quantiles=None,
                           fig=(fig, axes[:, :3]))

# plot extended run (res2; right)
```

(continues on next page)

(continued from previous page)

```
fig, ax = dyplot.cornerplot(res2, color='dodgerblue', truths=np.zeros(ndim),
                           truth_color='black', show_titles=True,
                           quantiles=None, max_n_ticks=3,
                           fig=(fig, axes[:, 4:]))
```



Similar to `runplot()`, the marginal distributions shown are by default smoothed by 2% in the specified range using a Normal (Gaussian) kernel. Notice that even though our original run `res1` gave similar evidence estimates to the extended run `res2`, it gives somewhat “noisier” estimates of the posterior.

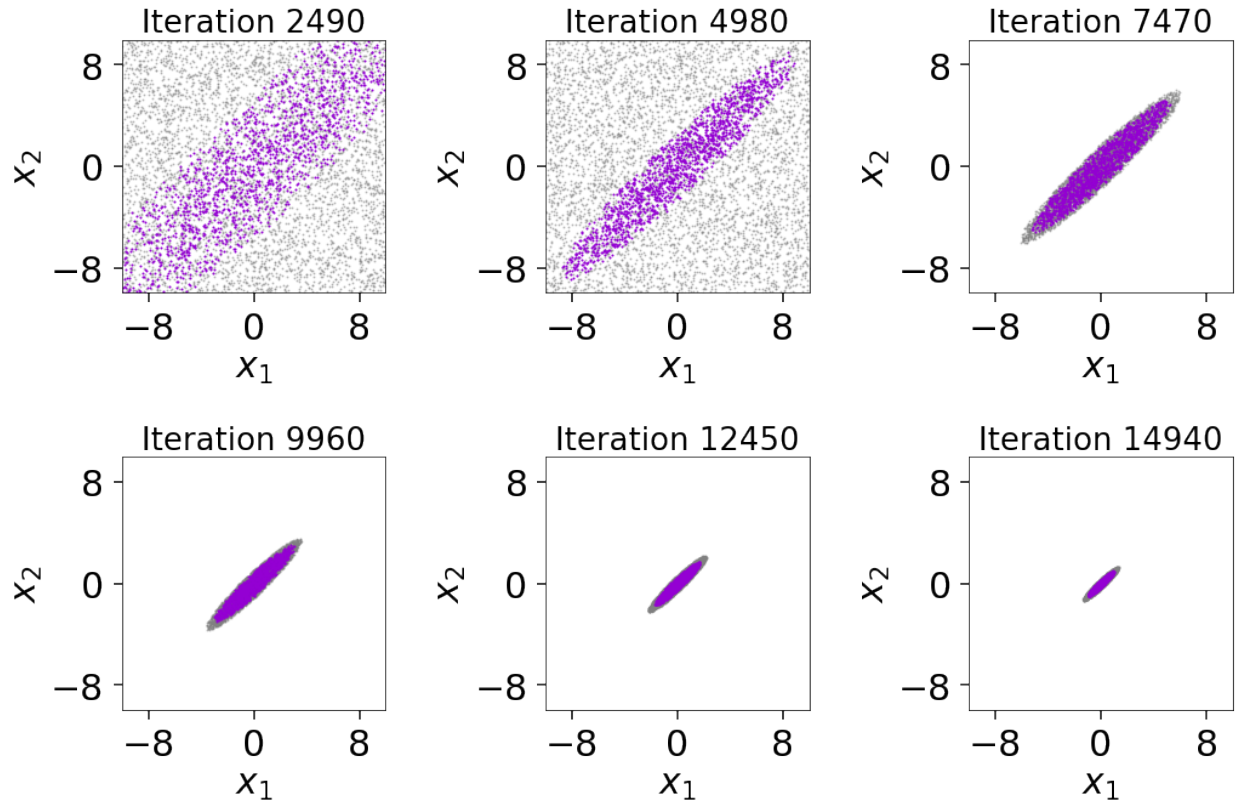
## Bounding Distribution Plots

To visualize how we’re sampling in nested “shells”, we can look at the evolution of our bounding distributions in a given 2-D projection over the course of a run. The `boundplot()` function allows us to look at the bounding distributions from two different perspectives: the bounding distribution used when proposing new live points at a specific iteration (specified using `it`), or the bounding distribution that a given dead point originated from (specified using `idx`). While `boundplot()` natively plots in the space of the unit cube, if a specified `prior_transform()` is passed all samples are instead converted to the original (native) model space.

Using `boundplot()`, we can examine the evolution of the bounding distributions over a given run via:

```
# initialize figure
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

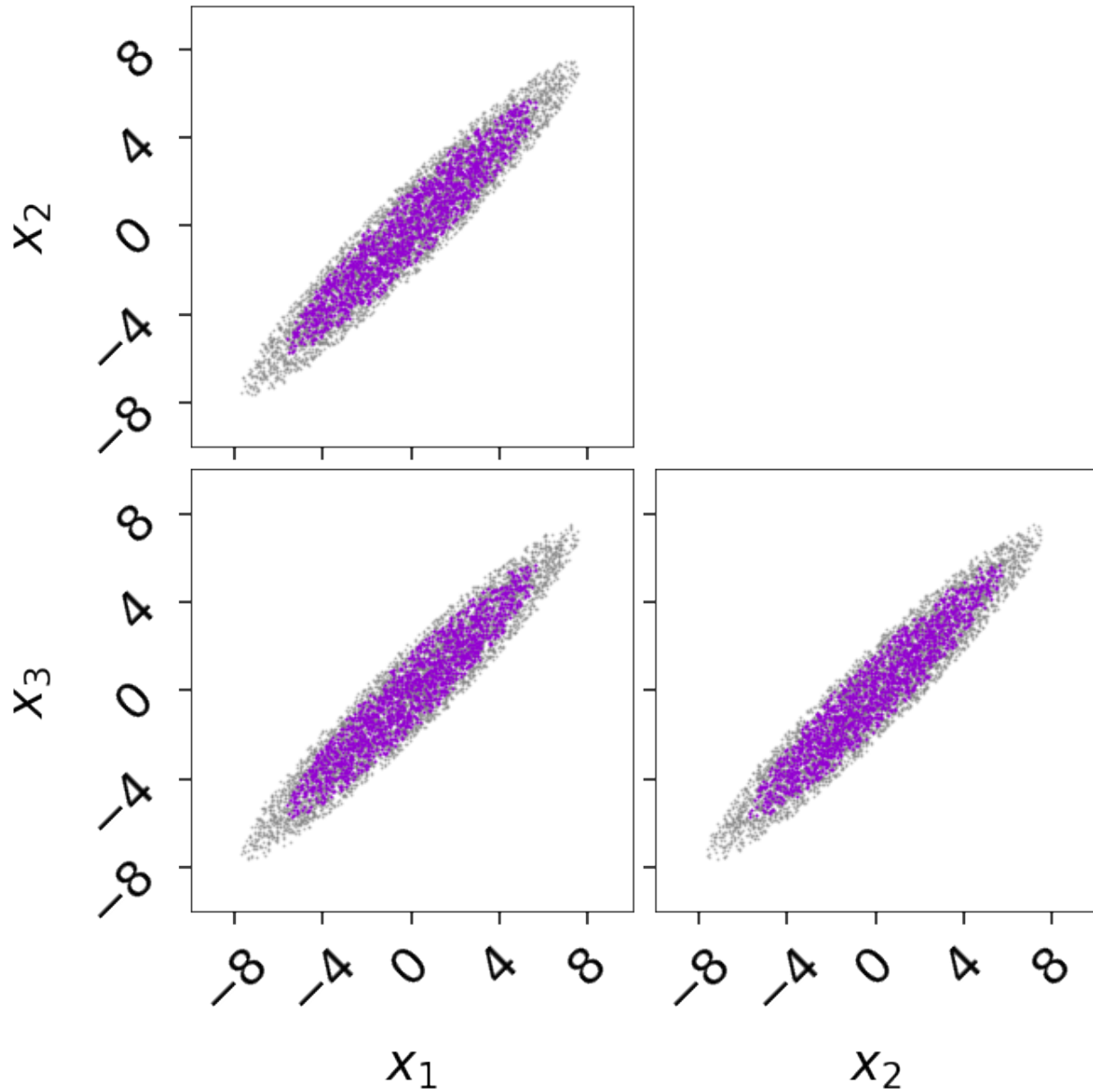
# plot 6 snapshots over the course of the run
for i, a in enumerate(axes.flatten()):
    it = int((i+1)*res2.niter/8.)
    # overplot the result onto each subplot
    temp = dyplot.boundplot(res2, dims=(0, 1), it=it,
                           prior_transform=prior_transform,
                           max_n_ticks=3, show_live=True,
                           span=[(-10, 10), (-10, 10)],
                           fig=(fig, a))
    a.set_title('Iteration {0}'.format(it), fontsize=26)
fig.tight_layout()
```



The figure illustrates that we first begin sampling directly from the unit cube. After the conditions in `first_update` are satisfied, we then switch over to the default multi-ellipsoidal bounding distributions. We see that these are able to adapt well to the target distribution over time, ensuring we continue to sample efficiently. We can also see the impact of bootstrapping on the bounding ellipsoids since they always remain slightly larger than the set of live points. While it slightly decreases the overall sampling efficiency, this shows how the procedure helps to ensure no likelihood is “left out” during the course of the Nested Sampling run.

Alternately, we can generate a corner plot of the bounding distribution using `cornerbound()` via:

```
fig, axes = dyplot.cornerprop(res2, it=5000,
                             prior_transform=prior_transform,
                             show_live=True,
                             span=[(-10, 10), (-10, 10)])
```



### Basic Post-Processing

In addition to plotting, `dynesty` also contains some post-processing utilities in the `utils` module. In many cases, a rough approximation of the posterior using the first two moments (mean and covariance) can be useful. These can be computed from the set of (weighted) samples using the `mean_and_cov()` function:

```
from dynesty import utils as dyfunc

samples, weights = res2.samples, np.exp(res2.logwt - res2.logz[-1])
mean, cov = dyfunc.mean_and_cov(samples, weights)
```

Runs can also be resampled to give a new set of points with equal weights, similar to MCMC methods, using the `resample_equal()` function:

```
new_samples = dyfunc.resample_equal(samples, weights)
```

See *Nested Sampling Errors* for some additional discussion and demonstration of more functions.

### 3.14.4 Dynamic Nested Sampling with dynesty

#### Static Nested Sampling

In most applications, scientists are often as interested (if not significantly more interested) in estimating the **posterior** rather than the evidence. From a posterior-oriented perspective, Nested Sampling’s ability to robustly sample from complex, multi-modal distributions often makes it an attractive alternative to methods such as Markov Chain Monte Carlo (MCMC) which struggle under those conditions.

The main drawback of Nested Sampling, however, is that **it is designed to estimate the evidence, not the posterior**. In particular, in a given Nested Sampling run with  $K$  live points, the prior volume  $X$  evolves as:

$$\Delta \ln X_i \approx \frac{1}{K}$$

This behavior holds true everywhere, regardless of where the bulk of the posterior mass is. So while increasing the number of live points increases our resolution while integrating over the typical set, it simultaneously increases our resolution everywhere else, leading to longer runtimes. In other words, the *proportion* of “wasted” samples remains approximately constant.

We can illustrate this directly using the same example from *Crash Course*:

```
import numpy as np
import dynesty
from dynesty import plotting as dyplot

# Define the dimensionality of our problem.
ndim = 3

# Define our 3-D correlated multivariate normal log-likelihood.
C = np.identity(ndim)
C[C==0] = 0.95
Cinv = linalg.inv(C)
lnorm = -0.5 * (np.log(2 * np.pi) * ndim +
                np.log(np.linalg.det(C)))

def loglike(x):
    return -0.5 * np.dot(x, np.dot(Cinv, x)) + lnorm

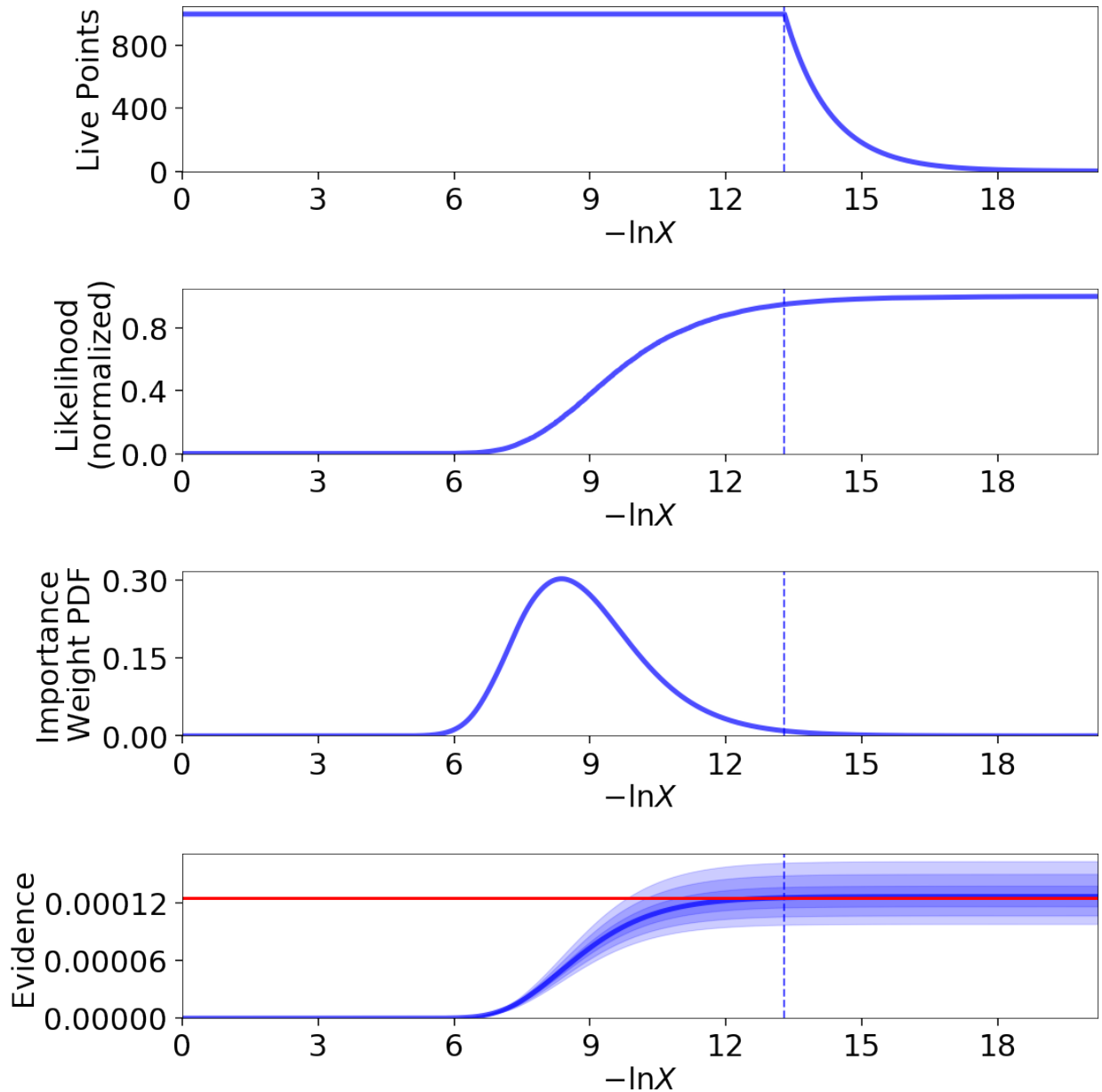
# Define our uniform prior via the prior transform.
def ptform(u):
    return 20. * u - 10.

# Sample from our distribution.
sampler = dynesty.NestedSampler(loglikelihood, prior_transform, ndim,
                                bound='single', nlive=1000)
sampler.run_nested(dlogz=0.01)
res = sampler.results

# Plot results.
lnz_truth = ndim * -np.log(2 * 10.) # analytic evidence solution
fig, axes = dyplot.runplot(res, lnz_truth=lnz_truth)
```

Out:

```
iter: 13301 | +1000 | bound: 14 | nc: 1 | ncall: 56724 | eff(%): 25.212 |
loglstar: -inf < -0.294 < inf | logz: -8.978 +/- 0.085 |
dlogz: 0.000 > 0.010
```



In this particular example, approximately a third of the samples give negligible contributions to the posterior. While these samples are crucial for evidence estimation (since they provide information on the current prior volume  $\ln X_i$ ), they are essentially useless when constructing posterior density estimates.

### Dynamic Nested Sampling

Instead of using a constant number of live points  $K$  throughout the entire run, it is possible to allocate live points *dynamically* such that at a given iteration  $i$  we can have a variable number  $K_i$  of effective live points. Since the change

in prior volume at a given iteration goes as

$$\Delta \ln X_i \approx \frac{1}{K_i}$$

allowing  $K_i$  to vary gives us the ability to control the effective resolution as a function of prior volume. For posterior-oriented applications, this means we could sample preferentially in and/or near the typical set around the bulk of the posterior mass. This would improve our posterior density estimate at the cost of increasing the relative error on our evidence estimate.

## Basic Implementation

Although in theory dynamic sampling can be done by adding one live point at a time, in practice this approach is difficult to implement because the number of points that are “live” can change rapidly as we traverse the prior volume. We instead insert additional live points in “**batches**” based on results from an initial “**baseline**” run. The basic algorithm is:

1. Compute a set of “baseline” samples with  $K_0$  live points.
2. Decide whether to stop sampling.
3. If we want to continue sampling, decide the bounds  $\left[ \mathcal{L}_{\text{low}}^{(b)}, \mathcal{L}_{\text{high}}^{(b)} \right)$  where additional samples should be allocated.
4. Compute a new set of samples for batch  $b$  within  $\left[ \mathcal{L}_{\text{low}}^{(b)}, \mathcal{L}_{\text{high}}^{(b)} \right)$  using  $K_b$  live points.
5. Add the final set of  $K_b$  live points sampled beyond  $\mathcal{L}_{\text{high}}^{(b)}$  to the new batch of samples.
6. “Combine” the new batch of samples with the set of previous set of samples and return to step (2).

## Weight Function

While dynamic sampling is powerful, the additional flexibility it provides requires additional (hyper-)parameters. The first set is associated with a **weight function**, which takes the current set of dead points (samples) and decides where we should allocate additional samples.

The default `weight_function()` used in dynesty is:

$$I_i(f_p) = f_p I_i^p + (1 - f_p) I_i^{\mathcal{Z}}$$

where  $i$  is the iteration associated with prior volume  $X_i$  and position  $\Theta_i$ ,  $f_p$  is the relative fractional importance we place on posterior estimation,

$$I_i^p = \hat{p}_i$$

is the posterior importance weight,

$$I_i^{\mathcal{Z}} = \frac{1}{N} \left( 1 - \frac{\hat{\mathcal{Z}}_i}{\hat{\mathcal{Z}}_{\text{upper}}} \right)$$

is the (normalized) evidence weight,  $\hat{\mathcal{Z}}_{\text{upper}} = \hat{\mathcal{Z}} + \Delta \hat{\mathcal{Z}}$  is the estimated upper limit on the total evidence, and  $K_i$  is the number of live points at  $X_i$ . In other words, the importance of a given point for estimating the posterior is just proportional to the amount that a given sample contributes to our estimate of the posterior at the current iteration, while the importance of a given point for estimating the evidence is proportional to the amount of the posterior interior to the log-volume probed by that point.



The likelihood ranges  $\left[\mathcal{L}_{\text{low}}^{(b)}, \mathcal{L}_{\text{high}}^{(b)}\right)$  where new samples will be allocated is then specified by taking the minimum and maximum (effective) iterations  $i_{\text{min}}$  and  $i_{\text{max}}$  that satisfy

$$I_i(f_p) \geq f_{\text{max}} \times \max(\{\dots, I_i(f_p), \dots\})$$

with some additional left/right padding of  $\pm n_{\text{pad}}$ . The default values are  $f_p = 0.8$  (80% posterior/20% evidence),  $f_{\text{max}}=0.8$ , and  $n_{\text{pad}} = 1$ .

## Stopping Function

The second set of hyper-parameters is associated with a **stopping function**, which takes the current set of dead points and decides when we should stop sampling. The default `stopping_function()` used in `dynesty` is:

$$S(f_p, s_p, s_Z, n) \equiv f_p \times \frac{S_p(n)}{s_p} + (1 - f_p) \times \frac{S_Z(n)}{s_Z} < 1$$

where  $f_p$  is the fractional importance we place on posterior estimation,  $S_p$  is the posterior stopping function,  $S_Z$  is the evidence stopping function,  $s_p$  is the posterior “error threshold”,  $s_Z$  is the evidence error threshold, and  $n$  is the total number of Monte Carlo realizations used to generate the posterior/evidence stopping values.

The default values of these are  $f_p = 1$  (100% posterior/0% evidence),  $s_p = 0.02$ ,  $s_Z = 0.1$ , and  $n = 128$ . More details on  $S_p(n)$  and  $S_Z(n)$  are outlined below.

## How Many Samples are Enough?

In any sampling-based approach to estimating the posterior density, it is difficult to determine how many samples are sufficient to estimate the posterior “well”. Part of this is because the question itself is often ill-defined: what, exactly, does “well” *mean*?

The typical response to this question is that it depends on what the samples will be used for. For instance, let’s assume we are specifically interested in the mean vector  $\mu$  and the covariance matrix  $\mathbf{C}$  characterizing the first and second moments of our posterior distribution, respectively. Using Normal and/or Student-t approximations can give us estimates as to how many samples are needed to achieve some desired error. Alternately, other methods such as subsampling or bootstrapping could be employed to estimate the errors as more samples are added. This answer, however, would be different if we were trying instead trying to estimate the 95% **credible interval**.

For evidence estimation, the default metric used to determine when to stop adding new samples is the error on the evidence as characterized by the standard deviation:

$$S_Z(n) = \sigma(\{\ln \hat{Z}'_1, \dots, \ln \hat{Z}'_n\})$$

where  $\ln \hat{Z}' \sim P(\ln \hat{Z})$  are *realizations* of the evidence computed from the current set of samples. More details on this procedure are described under [Nested Sampling Errors](#).

For posterior estimation, however, many researchers do not have such well-posed goals that they can use to determine the necessary sample size. As such, the default choice in `dynesty` is to assume that “well” means that the “difference” between the posterior density estimate  $\hat{P}(\Theta)$  we construct from our set of samples  $\{\Theta_1, \dots, \Theta_N\}$  and the true posterior density  $P(\Theta)$  is below some threshold.

We determine the “difference” between the two distributions using the **Kullback–Leibler (KL) divergence**:

$$H(\hat{P}|P) \equiv \int_{\Omega_{\Theta}} \hat{P}(\Theta) \ln \frac{\hat{P}(\Theta)}{P(\Theta)} d\Theta$$

Since we do not actually have access to  $P(\Theta)$ , we instead attempt to approximate this quantity based on realizations of  $\hat{P}(\Theta)$ :

$$H(\hat{P}'|\hat{P}) = \int_{\Omega_{\Theta}} \hat{P}'(\Theta) \ln \frac{\hat{P}'(\Theta)}{\hat{P}(\Theta)} d\Theta = \sum_i \hat{p}'_i (\ln \hat{p}'_i - \ln \hat{p}_i)$$

Since  $\hat{P}'$  is based on a realization of the posterior weights  $\hat{\mathbf{p}}' \sim P(\hat{\mathbf{p}})$ , our computed distance  $H(\hat{P}'|\hat{P}) \sim P(H(\hat{P}'|\hat{P}))$  is also a realization of the distance.

The expected value  $\mathbb{E}[P(H(\hat{P}'|\hat{P}))]$  of the distance will generally be non-zero, with the exact value dependent on the distribution in question. The fractional width of this distribution then characterizes the overall *uncertainty* in  $H(\hat{P}'|\hat{P})$  based on the current set of samples, giving us a probe of the underlying distance  $H(\hat{P}|P)$  between  $\hat{P}(\Theta)$  and the true posterior density  $P(\Theta)$ .

For posterior estimation, the default metric used to determine when to stop adding new samples is the fractional sample standard deviation in  $H(\hat{P}'|\hat{P})$ :

$$S_p(n) = \frac{\sigma(\{H(\hat{P}'_1|\hat{P}), \dots, H(\hat{P}'_n|\hat{P})\})}{\mathbb{E}(\{H(\hat{P}'_1|\hat{P}), \dots, H(\hat{P}'_n|\hat{P})\})}$$

More discussion can be found in *Nested Sampling Errors*.

## Usage in dynesty

### Initializing the DynamicSampler

Dynamic Nested Sampling in `dynesty` can be accessed from the *Top-Level Interface*'s `DynamicNestedSampler()` function and is done using the `DynamicSampler` class. Like the previous sampler showcased in *Getting Started*, the `DynamicSampler` uses a fixed set of bounding and sampling methods and can be initialized using a very similar API. One key difference, however, is that we don't need to declare the number of live points upon initialization:

```
from dynesty import DynamicNestedSampler

dsampler = DynamicNestedSampler(loglike, ptform, ndim, bound='single')
```

### Sampling Dynamically

Like `sampler`, our Dynamic Nested Sampler `dsampler` can be run internally using the `run_nested()` function:

```
dsampler.run_nested()
```

or externally as a generator:

```
from dynesty.dynamicsampler import stopping_function, weight_function

# Baseline run.
for results in dsampler.sample_initial():
    pass

# Add batches until we hit the stopping criterion.
while True:
    stop = stopping_function(dsampler.results) # evaluate stop
```

(continues on next page)

(continued from previous page)

```

if not stop:
    logl_bounds = weight_function(dsampler.results) # derive bounds
    for results in dsampler.sample_batch(logl_bounds=logl_bounds):
        pass
    dsampler.combine_runs() # add new samples to previous results
else:
    break

```

Since the number of live points that will be used during a run are not declared upon initialization, they must instead be declared during runtime via `run_nested()` using the `nlive_init` and `nlive_batch` keywords. Similarly, the `dlogz` tolerance used when terminating the initial baseline run can be declared using `dlogz_init`. For instance, if we wanted to use  $K_0 = 500$  live points for our baseline run, sample until  $\Delta \ln \hat{Z} < 0.05$ , and then add points in batches of  $K_b = 100$ , we would do:

```
dsampler.run_nested(dlogz_init=0.05, nlive_init=500, nlive_batch=100)
```

Like `sampler.run_nested()`, `dsampler.run_nested()` also allows users to specify a range of hard stopping criteria based on:

- the maximum number of iterations and log-likelihood calls made during the course of the entire run (`maxiter`, `maxcall`),
- the maximum number of iterations, log-likelihood calls, or log-likelihood value made during the course of the initial run (`maxiter_init`, `maxcall_init`, `logl_max_init`),
- the maximum number of iterations and log-likelihood calls made while adding batches (`maxiter_batch`, `maxcall_batch`), and
- the maximum number of allowed batches (`maxbatch`).

As an example, if we wanted to limit the total number of batches to 10, our initial run to only 10000 samples and each batch to only 1000 samples, we would do:

```
dsampler.run_nested(dlogz_init=0.05, nlive_init=500, nlive_batch=100,
                    maxiter_init=10000, maxiter_batch=1000, maxbatch=10)
```

In addition, users can specify their own `wt_function()` and `stop_function()` using the associated keywords if they would like to change the way live point are allocated during a run. The only restrictions on these functions are that they take in a `Results` instance and a dictionary of arguments (`args`) and return results in the same format as the default `weight_function()` and `stopping_function()`. That might look something like:

```
dsampler.run_nested(dlogz_init=0.05, nlive_init=500, nlive_batch=100,
                    maxiter_init=10000, maxiter_batch=1000, maxbatch=10,
                    wt_function=weight_function,
                    stop_function=stopping_function)
```

Alternately, `dsampler` can avoid evaluating the stopping criteria altogether if the `use_stop` option is disabled:

```
dsampler.run_nested(dlogz_init=0.05, maxiter=30000, use_stop=False)
```

This can be useful if other stopping criteria will be used instead since the default `stopping_function()` can take a while to evaluate for larger samples.

Like the Static Nested Sampling case, users can also continue sampling where they left off if they would like to add more samples. For instance, if we would like to add a few more batches of points to our pre-existing set of samples, we could use:

```
dsampler.run_nested(maxbatch=10) # initial run
dsampler.run_nested(maxiter=50000) # (possibly) adding more samples
dsampler.run_nested(maxbatch=50) # (possibly) adding more samples
```

A new batch of points can also be added explicitly using the `add_batch()` function. As an example, a new batch with  $K_b = 250$  live points and at most 1000 samples could be added to the previous set of samples using:

```
dsampler.add_batch(nlive=250, maxiter=1000)
```

## Dynamic vs Static

To get a good sense of how Dynamic and Static Nested Sampling compare, let's examine the relative behavior of both samplers using the same number of samples (iterations).

Let's first start using the default behavior, which allocates samples favoring a 80%/20% posterior/evidence split:

```
# 80/20 posterior/evidence split, maxiter limit
dsampler.reset()
dsampler.run_nested(maxiter=res.niter+res.nlive, use_stop=False)
dres = dsampler.results
```

Out:

```
iter: 14301 | batch: 62 | bound: 392 | nc: 1 | ncall: 37803 |
eff(%): 37.830 | loglstar: -6.195 < -0.351 < -1.108 |
logz: -8.877 +/- 0.137 | stop: nan
```

Since `dsampler` is by default optimized for posterior estimation over evidence estimation (via the default values assigned in `weight_function`), the errors on our evidence estimates are significantly larger than the results from `sampler`.

Note that while the outputs are largely similar to the `sampler` case, they include three additional quantities: `batch`, which shows the current batch, `loglstar`, which lists the log-likelihood bounds used to define the current batch as well as the current log-likelihood value, and `stop`, which records the current stopping value (not computed here).

In addition to having slightly different output formats, the `Results` objects also contain slightly different information:

```
print('Static Nested Sampling:', res.keys())
print('Dynamic Nested Sampling:', dres.keys())
```

Out:

```
Static Nested Sampling: ['niter', 'logvol', 'information', 'samples_id',
                        'logz', 'bound', 'ncall', 'samples_bound',
                        'scale', 'nlive', 'samples', 'bound_iter',
                        'samples_u', 'samples_it', 'logl', 'logzerr',
                        'eff', 'logwt']

Dynamic Nested Sampling: ['niter', 'samples_n', 'batch_bounds',
                        'information', 'samples_id', 'batch_nlive',
                        'bound_iter', 'logz', 'bound', 'ncall',
                        'samples_bound', 'logvol', 'logwt', 'samples',
                        'samples_batch', 'samples_u', 'samples_it',
                        'logl', 'logzerr', 'eff', 'scale']
```

The differences between these are:

- `samples_n` (replaces `nlive`): records the number of live points at a given iteration.
- `samples_batch`: index of the batch the points were sampled from.
- `batch_nlive`: tracks the number of live points added in a given batch.
- `batch_bounds`: the log-likelihood bounds used to allocate samples in a given batch.

Let's now examine two edge cases by changing the arguments passed to the weight function via `wt_kwargs`. In the first case, we will allocate samples with 100% of the weight placed on the posterior ( $f_p = 1$ ):

```
# 100/0 posterior/evidence split, maxiter limit
dsampler.reset()
dsampler.run_nested(maxiter=res.niter+res.nlive, use_stop=False,
                    wt_kwargs={'pfrac': 1.0})
dres_p = dsampler.results
```

Out:

```
iter: 14316 | batch: 71 | bound: 412 | nc: 3 | ncall: 30890 |
eff(%): 46.345 | loglstar: -8.855 < -0.817 < -1.129 |
logz: -9.267 +/- 0.374 | stop: nan
```

In the second case, we will allocate samples with 100% of the weight placed on the evidence ( $f_p = 0$ ):

```
# 0/100 posterior/evidence split, maxiter limit
dsampler.reset()
dsampler.run_nested(maxiter=res.niter+res.nlive, use_stop=False,
                    wt_kwargs={'pfrac': 0.0})
dres_z = dsampler.results
```

Out:

```
iter: 14301 | batch: 30 | bound: 0 | nc: 1 | ncall: 68940 |
eff(%): 20.744 | loglstar: -inf < -40.112 < -2.295 |
logz: -9.007 +/- 0.075 | stop: nan
```

Here we see that there are some significant differences in behavior.

To round things off, let's finally compare the above cases but using the default automated stopping criteria from `stopping_function`:

```
# weight: 80/20 posterior/evidence split
# stop: 100/0 posterior/evidence split
dsampler.reset()
dsampler.run_nested()
dres2 = dsampler.results

# weight: 100/0 posterior/evidence split
# stop: 100/0 posterior/evidence split
dsampler.reset()
dsampler.run_nested(wt_kwargs={'pfrac': 1.0})
dres2_p = dsampler.results

# weight: 0/100 posterior/evidence split
# stop: 0/100 posterior/evidence split
dsampler.reset()
dsampler.run_nested(wt_kwargs={'pfrac': 0.0}, stop_kwargs={'pfrac': 0.0})
dres2_z = dsampler.results
```

Out:

```

iter: 22165 | batch: 10 | bound: 56 | nc: 1 | ncall: 55509 |
eff(%): 39.930 | loglstar: -7.838 < -0.298 < -0.789 |
logz: -9.115 +/- 0.116 | stop: 0.970

iter: 21597 | batch: 10 | bound: 56 | nc: 1 | ncall: 55058 |
eff(%): 39.226 | loglstar: -6.004 < -0.299 < -0.854 |
logz: -8.995 +/- 0.116 | stop: 0.923

iter: 16031 | batch: 2 | bound: 29 | nc: 1 | ncall: 77598 |
eff(%): 20.659 | loglstar: -inf < -0.346 < -1.851 |
logz: -8.812 +/- 0.085 | stop: 0.990

```

These contain a similar number of samples and give similar answers to the previous cases shown above.

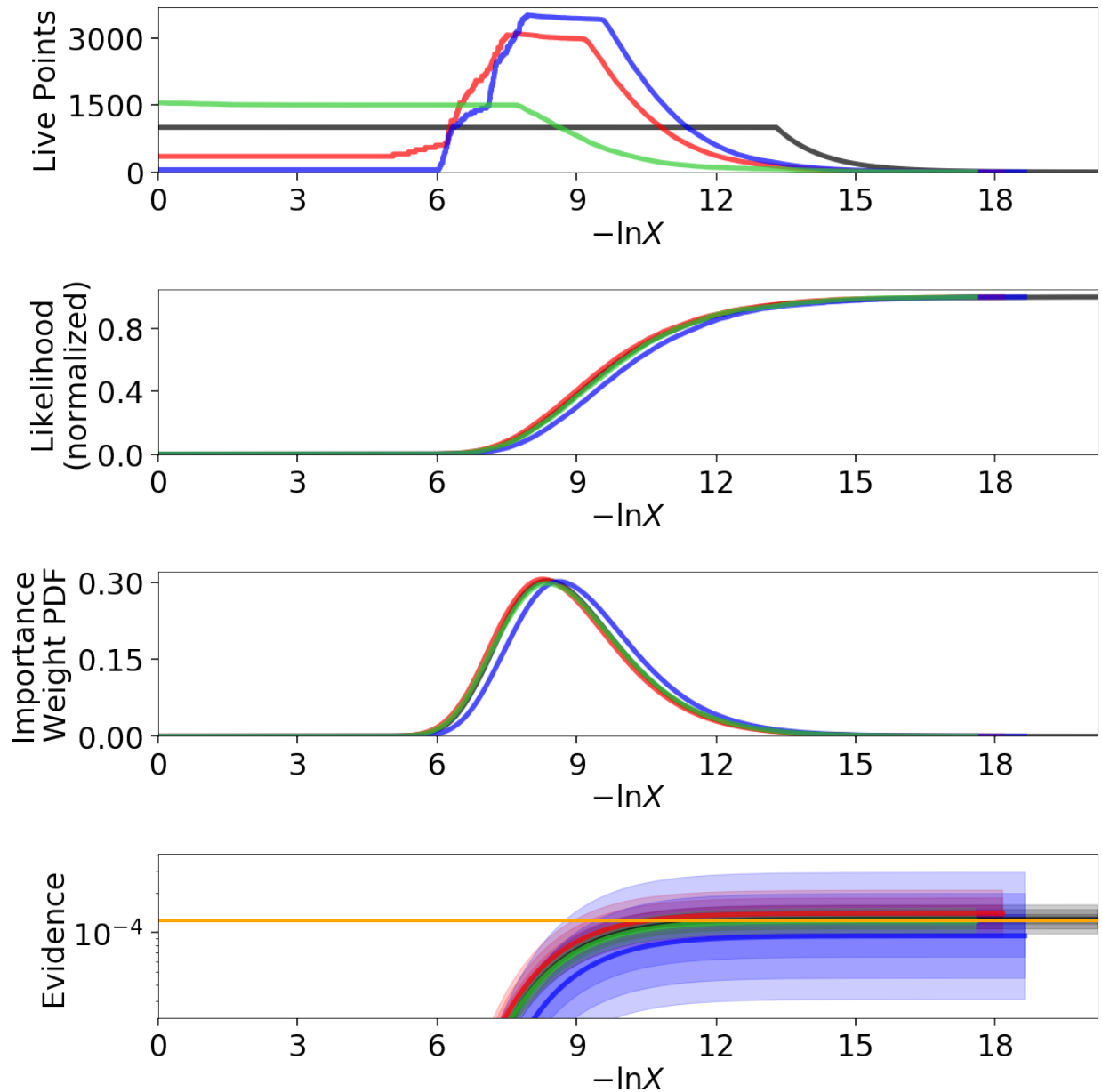
## Visualizing the Results

We can get a better sense of how these different strategies affect our results using the *Plotting Utilities* demonstrated previously. The first thing we can examine is the different behaviors shown on summary plots:

```

fig, axes = dyplot.runplot(res, color='black', mark_final_live=False,
                           logplot=True) # static run
fig, axes = dyplot.runplot(dres, color='red', logplot=True,
                           fig=(fig, axes)) # default dynamic run
fig, axes = dyplot.runplot(dres_p, color='blue', logplot=True,
                           fig=(fig, axes)) # posterior dynamic run
fig, axes = dyplot.runplot(dres_z, color='limegreen', logplot=True,
                           lnz_truth=lnz_truth, truth_color='orange',
                           fig=(fig, axes)) # evidence dynamic run
fig.tight_layout()

```

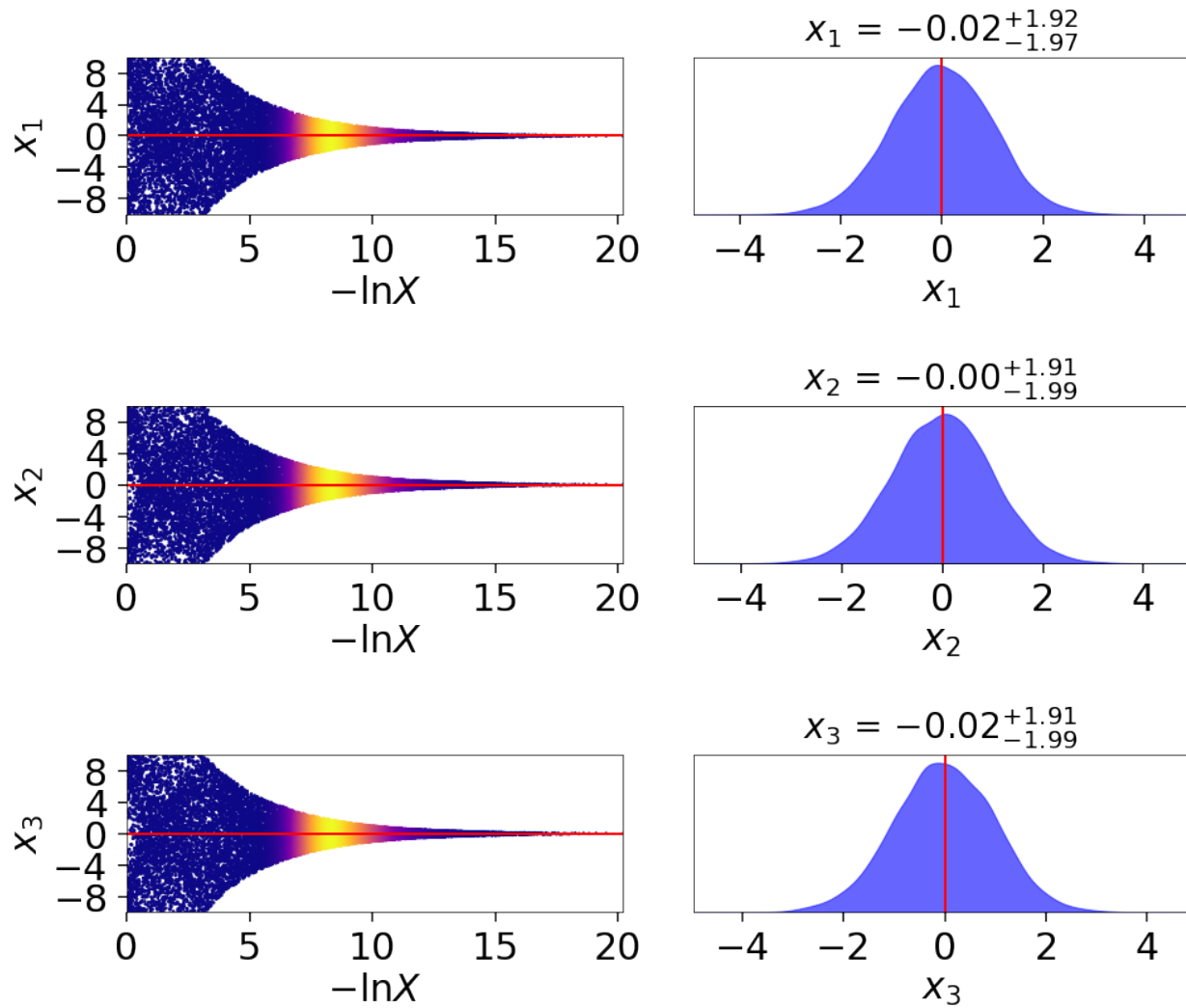


We can see that the general shape of the dynamic runs traces the overall shape of the weights: our posterior-based samples are concentrated around the bulk of the posterior mass (see [Typical Sets](#)) while the evidence-based samples are concentrated away from the typical set towards the prior. The general skewness to the distribution is primarily because we recycle live points sampled past the log-likelihood bounds set during each batch. This allows us to get more information “inward” of the bounds whenever we add a batch, so as a result new samples tend to be systematically allocated “outward”.

In other words, `dsampler` is doing exactly what we want: although each run has the same amount of samples, the places where they are located differs dramatically among our runs. For the posterior-oriented case, we spend (significantly) less time sampling regions with little posterior weight and samples are concentrated around the typical set. This gives us significantly greater resolution in that region compared to the resolution elsewhere. Conversely, in the evidence-oriented case we spend many fewer samples tracing out the typical set. Instead, the most samples are allocated in prior-dominated regions to help constrain the exact location  $\ln X_i$  where the typical set is located. As expected, the default case effectively compromises between these two behaviors.

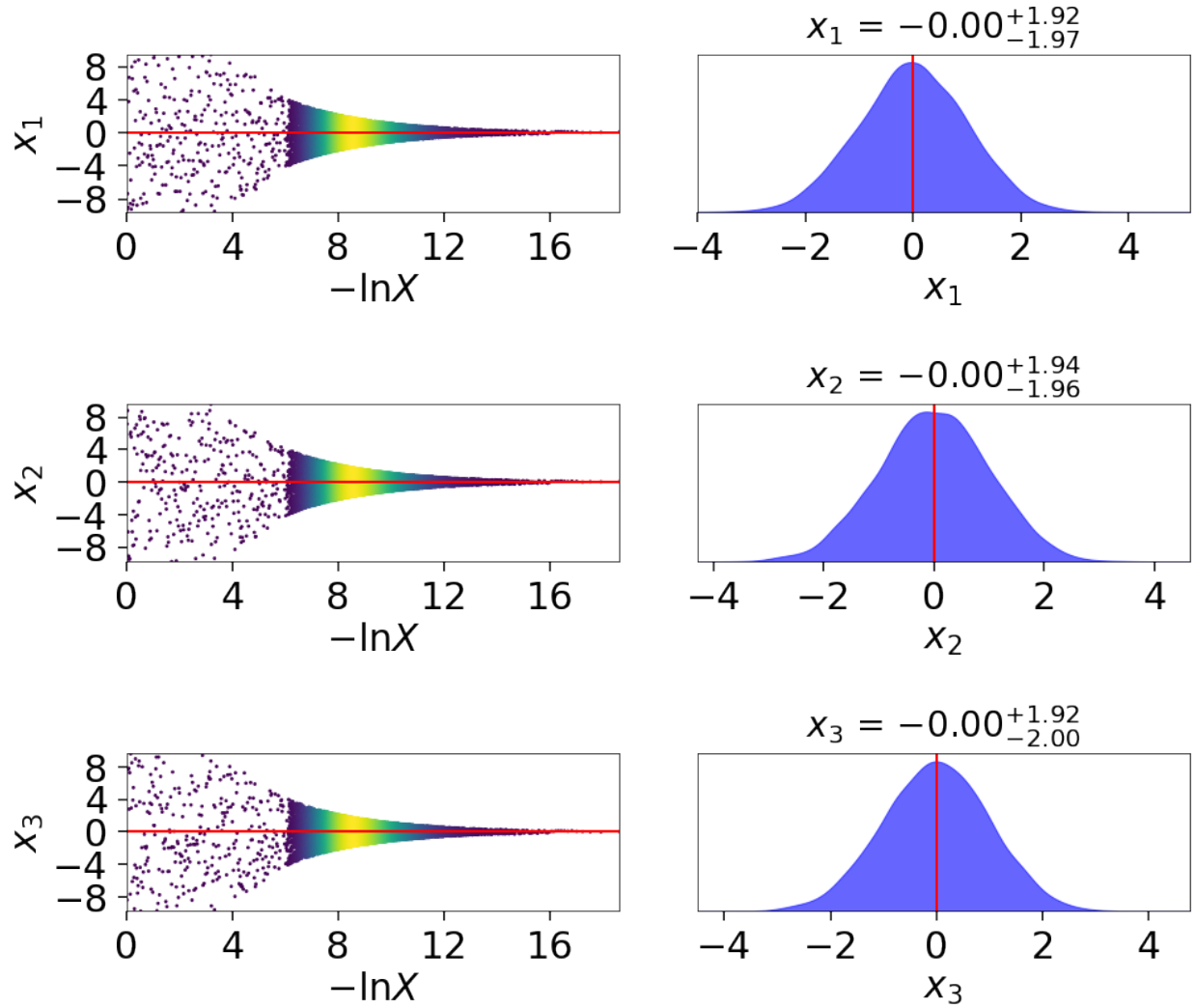
This behavior can be made even more apparent by examining where samples are allocated on trace plots:

```
# plotting the static run
fig, axes = dyplot.traceplot(res, truths=np.zeros(ndim),
                             show_titles=True, trace_cmap='plasma',
                             quantiles=None)
```

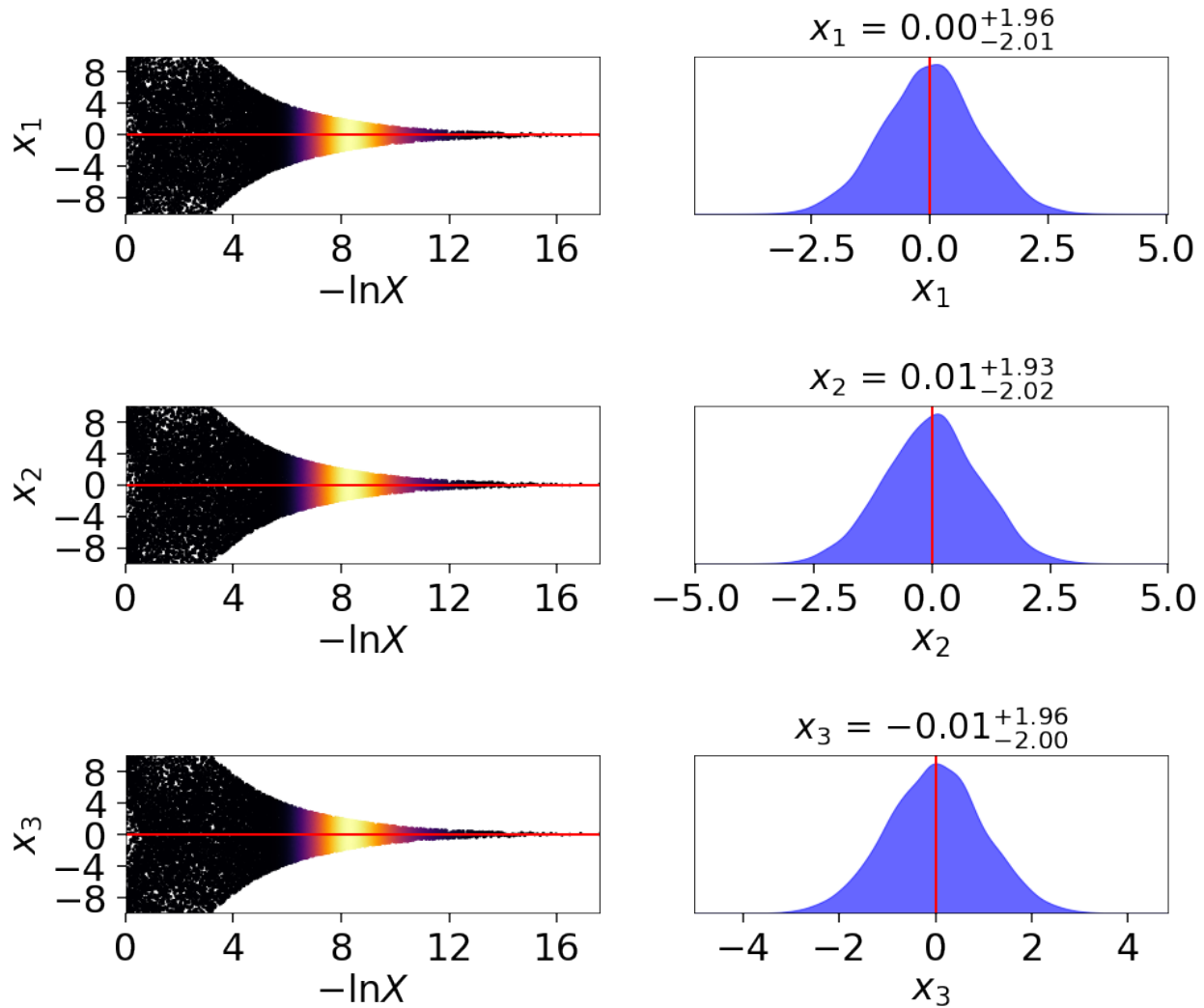


```
# plotting the posterior-oriented dynamic run
fig, axes = dyplot.traceplot(dres_p, truths=np.zeros(ndim),
                             show_titles=True, trace_cmap='viridis',
                             quantiles=None)
```





```
# plotting the evidence-oriented dynamic run
fig, axes = dyplot.traceplot(dres_z, truths=np.zeros(ndim),
                             show_titles=True, trace_cmap='inferno',
                             quantiles=None)
```



and on a (sub-)corner plot of the samples:

```
# initialize figure
fig, axes = plt.subplots(2, 8, figsize=(40, 10))
axes = axes.reshape((2, 8))
[a.set_frame_on(False) for a in axes[:, 2]]
[a.set_xticks([]) for a in axes[:, 2]]
[a.set_yticks([]) for a in axes[:, 2]]
[a.set_frame_on(False) for a in axes[:, 5]]
[a.set_xticks([]) for a in axes[:, 5]]
[a.set_yticks([]) for a in axes[:, 5]]

# plot static run (left)
fg, ax = dyplot.cornerpoints(res, cmap='plasma', truths=np.zeros(ndim),
                             kde=False, fig=(fig, axes[:, 0:2]))

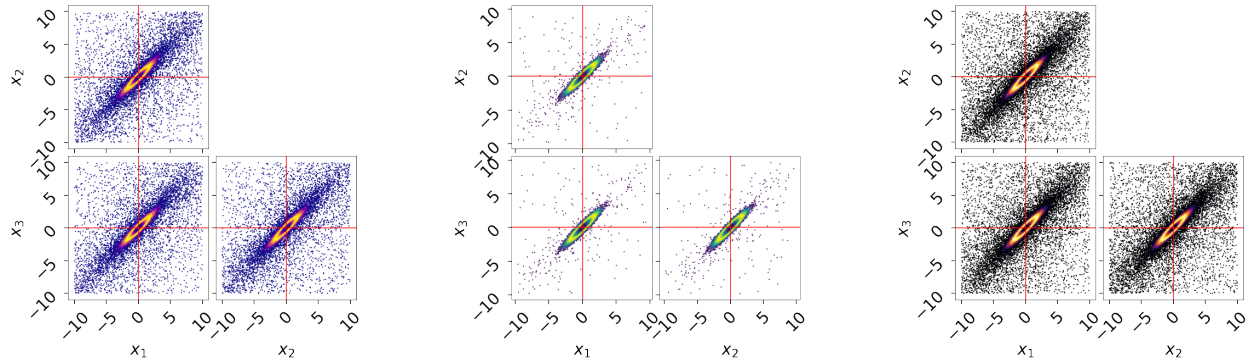
# plot posterior-oriented dynamic run (middle)
fg, ax = dyplot.cornerpoints(dres_p, cmap='viridis', truths=np.zeros(ndim),
                             kde=False, fig=(fig, axes[:, 3:5]))

# plot evidence-oriented dynamic run (right)
```

(continues on next page)

(continued from previous page)

```
fig, ax = dyplot.cornerpoints(dres_z, cmap='inferno', truths=np.zeros(ndim),
                             kde=False, fig=(fig, axes[:, 6:8]))
```

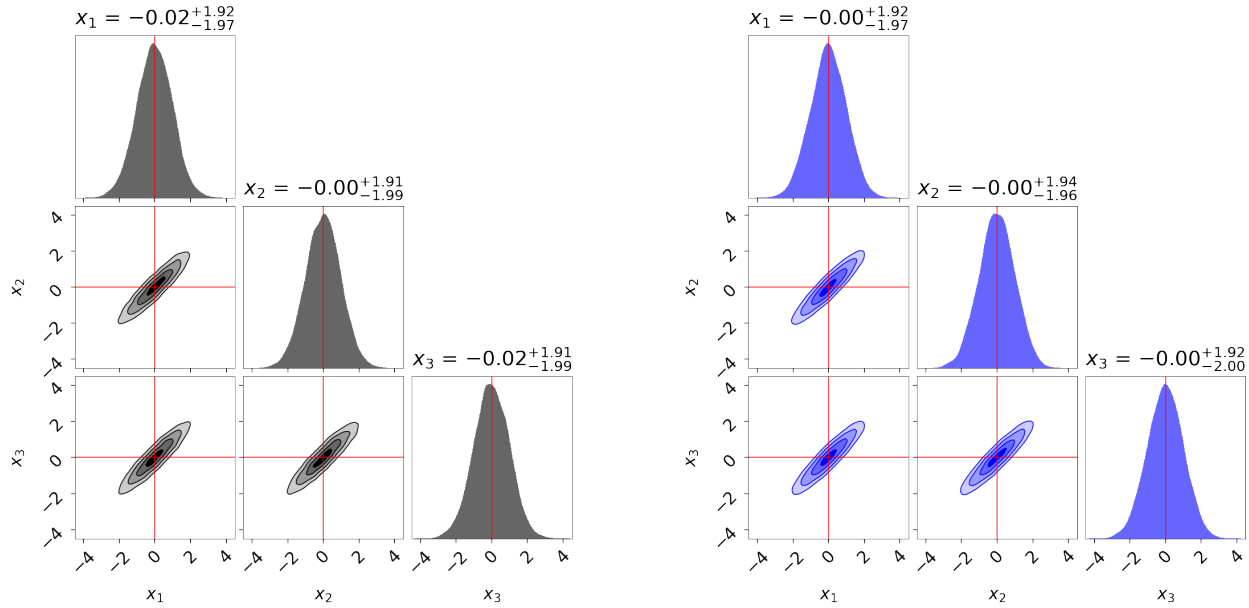


Finally, let's take a quick look at how this impacts the quality of our inferred posterior:

```
# initialize figure
fig, axes = plt.subplots(3, 7, figsize=(35, 15))
axes = axes.reshape((3, 7))
[a.set_frame_on(False) for a in axes[:, 3]]
[a.set_xticks([]) for a in axes[:, 3]]
[a.set_yticks([]) for a in axes[:, 3]]

# plot initial run (left)
fig, ax = dyplot.cornerplot(res, color='black', truths=np.zeros(ndim),
                             span=[(-4.5, 4.5) for i in range(ndim)],
                             show_titles=True, quantiles=None,
                             fig=(fig, axes[:, :3]))

# plot extended run (right)
fig, ax = dyplot.cornerplot(dres_p, color='blue', truths=np.zeros(ndim),
                             span=[(-4.5, 4.5) for i in range(ndim)],
                             show_titles=True, quantiles=None,
                             fig=(fig, axes[:, 4:])))
```



### 3.14.5 Nested Sampling Errors

Nested Sampling has two main sources of error. The first is the **statistical errors** associated with uncertainties on the prior volume  $X_i$  at a given iteration  $i$ . This leads to uncertainties on the estimated log-evidence  $\ln \hat{Z}$  and the associated posterior importance weights  $\hat{p}_i$ . The second is the **sampling errors** associated with replacing the integral over the parameters  $\Theta$  of interest with a single sample  $\Theta_i$  from the corresponding iso-likelihood contour defined by  $\mathcal{L}(\Theta) = \lambda_i$ .

One of the neat features of Nested Sampling is that we are able to probe these uncertainties **within the same run used to compute the results**. We exploit this fact within `dynesty` in two ways. The first is a set of functions within `utils` that can inject these errors into Nested Sampling `Results`. This allows users to compute realistic error budgets without going through the tedious task of computing many Nested Sampling runs. The second way is through the default Dynamic Nested Sampling `stopping_function()`, which uses this error budget when deciding whether to stop adding samples.

This page will go through some of the main results associated with deriving both exact and approximate error budgets for different aspects of Nested Sampling and show how to implement them in `dynesty`.

#### Approximate Evidence Errors

In a given Static Nested Sampling run with  $K$  live points, the prior volume evolves as:

$$\Delta \ln X \approx \frac{1}{K}$$

As mentioned in Role of Priors in Nested Sampling, the **“information”** gained from moving from the prior  $\pi(\Theta)$  to the posterior  $P(\Theta)$  can be quantified using the **KL Divergence** between the two distributions:

$$H(P|\pi) \equiv \int_{\Omega_{\Theta}} P(\Theta) \ln \frac{P(\Theta)}{\pi(\Theta)} d\Theta$$

This can be re-written in terms of an integral over the prior volume as:

$$H = \frac{1}{Z} \int_0^1 \mathcal{L}(X) \ln \mathcal{L}(X) dX - \ln Z$$

where  $\mathcal{Z}$  is the again the Bayesian evidence.

As such, the number of steps  $N$  needed to integrate over the majority of the posterior starting from the prior subject to some  $\Delta \ln \hat{\mathcal{Z}}$  (see [Stopping Criteria](#)) must scale with  $H$ . We also know that  $N$  should scale inversely with the typical  $\Delta \ln X$ . This gives us that the expected number of steps  $\mathbb{E}[N]$  goes as

$$\mathbb{E}[N] \sim \frac{H}{\Delta \ln X}$$

Assuming that the number of steps follows a Poisson distribution, we then expect the variance  $\mathbb{V}[N]$  should also scale as

$$\mathbb{V}[N] \sim \frac{H}{\Delta \ln X}$$

Since the the prior volumes compress exponentially, the uncertainty on  $N$  leads to exponential uncertainty in  $X$  (and hence  $\mathcal{Z}$ ) at a given iteration, so this uncertainty actually contributes in  $\ln \mathcal{Z}$  rather than  $\mathcal{Z}$ . The rough uncertainty in  $\ln \mathcal{Z}$  is then:

$$\sigma[\ln \hat{\mathcal{Z}}] \sim \sigma[\ln \hat{X}] \sim \sigma[\ln N] \Delta \ln X = \sqrt{H \Delta \ln X} = \sqrt{\frac{H}{K}}$$

This approximation can be extended to Dynamic Nested Sampling runs by exploiting the fact that

$$\mathbb{V}[\ln \hat{\mathcal{Z}}] = \mathbb{V}\left[\sum_{i=1}^N (\ln \hat{\mathcal{Z}}_i - \ln \hat{\mathcal{Z}}_{i-1})\right] \equiv \mathbb{V}\left[\sum_{i=1}^N \Delta \ln \hat{\mathcal{Z}}_i\right] \approx \sum_{i=1}^N \mathbb{V}[\Delta \ln \hat{\mathcal{Z}}_i]$$

where we take  $\ln \hat{\mathcal{Z}}_0 = 0$ . Approximating  $\mathbb{V}[\Delta \ln \hat{\mathcal{Z}}_i]$  as

$$\mathbb{V}[\Delta \ln \hat{\mathcal{Z}}_i] \approx \mathbb{V}[\ln \hat{\mathcal{Z}}_i] - \mathbb{V}[\ln \hat{\mathcal{Z}}_{i-1}] \sim H_i \Delta \ln X - H_{i-1} \Delta \ln X \equiv \Delta H_i \Delta \ln X$$

shows that the contribution to the error at each iteration is based on the differential change in information  $\Delta H_i$  multiplied by the differential change in log-prior volume  $\Delta \ln X$ , which we can substitute with  $\Delta \ln X_i$ . This gives us:

$$\mathbb{V}[\ln \hat{\mathcal{Z}}] \approx \sum_{i=1}^N \Delta H_i \Delta \ln X_i \approx \sum_{i=1}^N \frac{\Delta H_i}{K_i}$$

These are the errors that are returned by default in the output `stderr` statements and output `Results` instances and used in plotting functions.

As an example, here's a comparison among two different runs to showcase how the approximate errors take into account varying numbers of live points (and the associated changes in prior volume) throughout a given Nested Sampling run:

```
# static nested sampling
sampler = dynesty.NestedSampler(loglikelihood, prior_transform, ndim,
                                bound='single', nlive=1000)

sampler.run_nested()
res = sampler.results
sys.stderr.write('\n')

sampler.reset()
sampler.run_nested(dlogz=0.01)
res2 = sampler.results
sys.stderr.write('\n')

# dynamic nested sampling
dsampler = dynesty.DynamicNestedSampler(loglikelihood, prior_transform,
                                         ndim, bound='single')
dsampler.run_nested(maxiter=res2.niter+res2.nlive, use_stop=False)
dres = dsampler.results
```

Out:

```

iter: 8973 | +1000 | bound: 8 | nc: 1 | ncall: 47632 | eff(%): 20.938 |
loglstar: -inf < -0.300 < inf | logz: -9.169 +/- 0.097 |
dlogz: 0.001 > 1.009
iter: 13175 | +1000 | bound: 14 | nc: 1 | ncall: 54140 | eff(%): 26.182 |
loglstar: -inf < -0.294 < inf | logz: -8.852 +/- 0.084 |
dlogz: 0.000 > 0.010
iter: 14175 | batch: 7 | bound: 35 | nc: 1 | ncall: 39494 |
eff(%): 35.892 | loglstar: -5.792 < -0.329 < -0.645 |
logz: -8.930 +/- 0.116 | stop: nan

```

The differences among the results illustrate how the location where samples are allocated can significantly affect the error budget, as discussed in *Dynamic Nested Sampling*.

## Statistical Uncertainties

This section deals primarily with the **statistical uncertainties** associated with Nested Sampling. These arise from the probabilistic way a prior volume  $X_i$  is assigned to a particular sample  $\Theta_i$  and iso-likelihood contour  $\mathcal{L}_i$ .

## Order Statistics

Nested Sampling works thanks to the “magic” of **order statistics**. At the start of a Nested Sampling run, we sample  $K$  points from the prior  $\pi(\Theta)$  with likelihoods  $\{\mathcal{L}_1, \dots, \mathcal{L}_K\}$  and associated prior volumes  $\{X_1, \dots, X_K\}$ . We then want to pick the point with the *smallest* (worst) likelihood  $\mathcal{L}_{(1)}$  out of the **ordered set**  $\{\mathcal{L}_{(1)}, \dots, \mathcal{L}_{(K)}\}$  from smallest to largest. These likelihoods correspond to an ordered set of prior volumes  $\{X_{(1)}, \dots, X_{(K)}\}$ , where the likelihoods and prior volumes are inversely ordered such that  $\mathcal{L}_{(i)} \leftrightarrow X_{(K-i+1)}$ .

What is this prior volume? Since all the points were drawn from the prior, the **probability integral transform (PIT)** tells us that the corresponding prior volumes are uniformly distributed **random variables** such that

$$X_1, \dots, X_K \stackrel{i.i.d.}{\sim} \text{Unif}$$

where Unif is the standard Uniform distribution. It can be shown through the **Renyi representation** (and other methods) that the set of *ordered* uniform random variables (the prior volumes) can be *jointly* represented by  $K + 1$  standard Exponential random variables

$$X_{(j)} \sim \frac{Y_1 + \dots + Y_j}{Y_1 + \dots + Y_{K+1}}$$

$$Y_1, \dots, Y_{K+1} \stackrel{i.i.d.}{\sim} \text{Expo}$$

where Expo is the standard Exponential distribution.

## Prior Volumes and Order Statistics

### Constant Number of Live Points

The marginal distribution of the prior volume  $X_{(K)}$  associated with the live point with the lowest likelihood  $\mathcal{L}_{(1)}$  is

$$X_{(j=K)} \sim \text{Beta}(j, K - j + 1) = \text{Beta}(K, 1)$$

where  $\text{Beta}(\alpha, \beta)$  is the Beta distribution with concentration parameters  $(\alpha, \beta)$ .

Once we replace a live point with a new live point drawn from the prior with  $\mathcal{L}_i \geq \mathcal{L}_{(1)}$ , we now want to do the same procedure again. Using the same logic as above, we know that our prior volumes must be independently and identically (i.i.d.) uniformly distributed *within the previous volume* since we just replaced the worst point with a new independent draw. At a given iteration  $i$  the prior volume associated of the live point with the worst likelihood is then

$$X_i = t_i X_{i-1}$$

$$t_i \sim \text{Beta}(K, 1)$$

This means that we’re compressing by a factor of  $\mathbb{E}[t_i] = K/(K + 1)$  at each iteration. This result allows us to *simulate* the change in prior volume using numerical methods such as `numpy.random.beta`.

## Increasing Number of Live Points

In the Dynamic Nested Sampling case at a given iteration we can add in new live points so that the number of effective live points  $K_i > K_{i-1}$ . Since all the samples are i.i.d. by construction, we end up with the modified result

$$t_i \sim \text{Beta}(K_i, 1), \quad K_i \geq K_{i-1}$$

## Decreasing Number of Live Points

In the case where the number of live points are decreasing, we are now directly sampling “down” the set of order statistics  $\{X_{(1)}, \dots, X_{(K_j)}\}$  described above. If at iteration  $i > j$  we have  $K_i < K_{i+1} < \dots < K_j$  live points, then the prior volume is the  $X_{(K_i)}$ -th order statistic. Relative to  $X_j$ , these have an expectation value of:

$$\mathbb{E} \left[ \frac{\sum_{n=1}^{K_i} Y_n}{\sum_{n=1}^{K_j+1} Y_n} \right] = \frac{K_j - K_i + 1}{K_j + 1}$$

This leads to the prior volume changing in discrete “chunks” of  $X_j/(K_j + 1)$ . In the *Static Nested Sampling* case, this only occurs at the end when adding the final set of live points. In the *Dynamic Nested Sampling* case, however, the changes in prior volume from iteration to iteration can swap back and forth between **exponential** and **uniform** shrinkage.

We can simulate the joint distribution of these prior volumes by identifying contiguous sequences of strictly decreasing live points and then simulating random numbers using `numpy.random.exponential`.

## Jittering Runs

`dynesty` contains a variety of useful utilities in the `utils` module, some of which were demonstrated in *Getting Started*. In addition to those, it also contains several functions that operate over the output `Results` dictionary from a Nested Sampling run that implement the results discussed on this page.

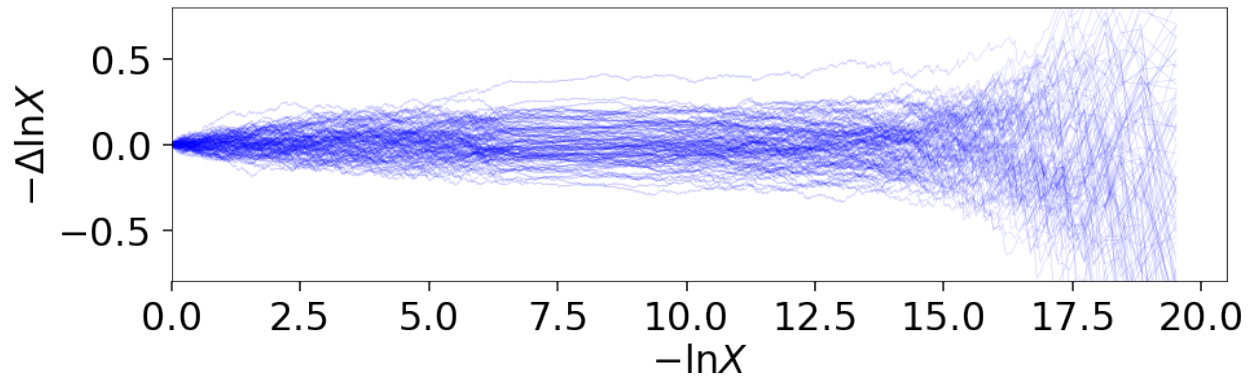
The `jitter_run()` function probes the statistical uncertainties in Nested Sampling by drawing a large number of random variables from the corresponding (joint) prior volume distributions described above in order to simulate the set of possible prior volumes associated with each dead point. It then returns a new `Results` dictionary with a new set of prior volumes, importance weights, and evidences (with new errors). This approach of adding “jitter” to the weights works for both Static and Dynamic Nested Sampling runs and can capture complex covariance structure.

Let’s go through an example using the results from *Approximate Evidence Errors*. First, let’s examine what the distribution of possible prior volumes looks like:

```

from dynesty import utils as dyfunc

# plot ln(prior volume) changes
for i in range(100):
    dres_j = dyfunc.jitter_run(dres)
    plt.plot(-dres.logvol, -dres.logvol + dres_j.logvol, color='blue',
             lw=0.5, alpha=0.2)
plt.ylim([-0.8, 0.8])
plt.xlabel(r'$-\ln X$')
plt.ylabel(r'$-\Delta \ln X$')
    
```



How do these realizations compare with our evidence approximation? We can compare them directly:

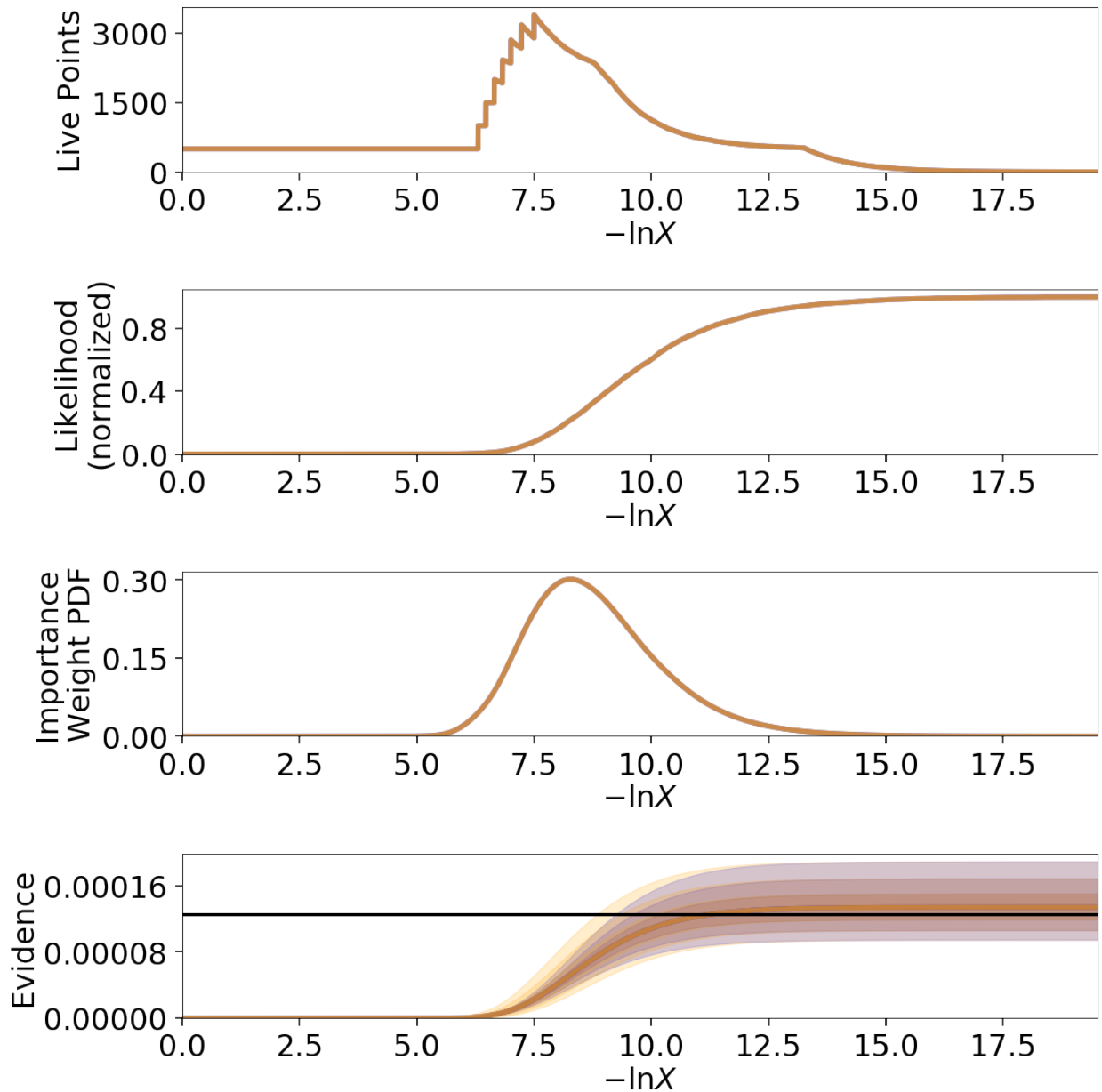
```

import copy

# compute ln(evidence) error
lnzs = np.zeros((100, len(dres.logvol)))
for i in range(100):
    dres_j = dyfunc.jitter_run(dres)
    lnzs[i] = np.interp(-dres.logvol, -dres_j.logvol, dres_j.logz)
lnzerr = np.std(lnzs, axis=0)

# plot comparison
dres_j = copy.deepcopy(dres)
dres_j['logzerr'] = lnzerr
fig, axes = dyplot.runplot(dres, color='blue')
fig, axes = dyplot.runplot(dres_j, color='orange',
                           lnz_truth=lnzerr, truth_color='black',
                           fig=(fig, axes))
fig.tight_layout()
    
```





While the analytic evidence approximations tend to underestimate the error while sampling within the typical set, the final errors are almost identical.

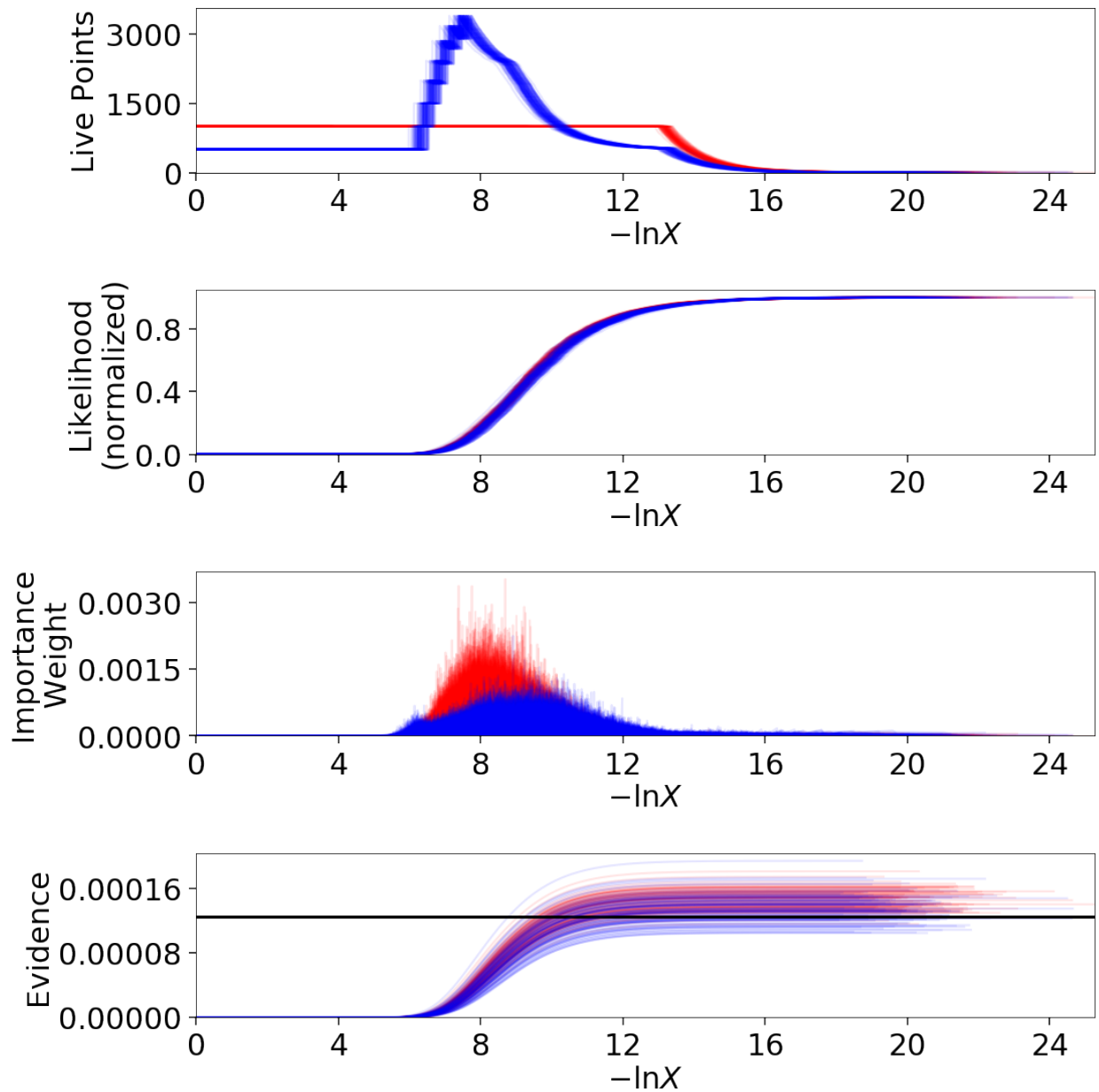
Finally, let's just plot a number of realizations directly to get a sense of how changes to the prior volumes propagate through to other quantities:

```
# overplot draws on summary plots
fig, axes = plt.subplots(4, 1, figsize=(16, 16))
for i in range(100):
    res2_j = dyfunc.jitter_run(res2)
    fig, axes = dyplot.runplot(res2_j, color='red',
                               plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                               mark_final_live=False, lnz_error=False,
                               fig=(fig, axes))
for i in range(100):
```

(continues on next page)

(continued from previous page)

```
dres_j = dyfunc.jitter_run(dres)
fig, axes = dyplot.runplot(dres_j, color='blue',
                           plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                           mark_final_live=False, lnz_error=False,
                           lnz_truth=lnz_truth, truth_color='black',
                           truth_kwargs={'alpha': 0.1},
                           fig=(fig, axes))
fig.tight_layout()
```



## Sampling Uncertainties

In addition to the statistical uncertainties associated with the unknown prior volumes, Nested Sampling is also subject to **sampling uncertainties** due to the “**path**” taken by a particular live point through the prior. This encompasses two different sources of error intrinsic to sampling itself. The first is **Monte Carlo noise** that arises from probing a continuous distribution using a finite set of samples. The second is **path-dependency**, where the results depend on the particular paths taken by the set of particles. This affects the results since the number of positions sampled along each path is subject to Poisson noise (see *Approximate Evidence Errors*); positions can be correlated in some way rather than fully independent draws from the target distribution, subtly violating the sampling assumptions in Nested Sampling.

In other words, although the prior volume  $X_i$  at a given iteration  $i$  might be known exactly, the particular *position*  $\Theta_i$  on the iso-likelihood contour  $\mathcal{L}_i$  is randomly distributed. This adds some additional noise to our posterior and evidence estimates. This can also complicate things if there are problems with the live point proposals that violate the assumptions described in *Nested Sampling*.

## Unraveling/Merging Runs

One way to interpret Nested Sampling is that it is a scheme that takes a set of ordered likelihoods  $0 < \mathcal{L}_1 < \dots < \mathcal{L}_N$  and associates them with a set of corresponding prior volumes  $1 > X_1 > \dots > X_N > 0$  by means of a number of live points.

One neat property of Nested Sampling is that if we have two Static Nested Sampling runs with  $K_1$  and  $K_2$  live points, respectively, composed of two sets of ordered likelihoods  $0 < \mathcal{L}_{(1)}^{(K_1)} < \dots < \mathcal{L}_{(N)}^{(K_1)}$  and  $0 < \mathcal{L}_{(1)}^{(K_2)} < \dots < \mathcal{L}_{(N)}^{(K_2)}$ , the combined set of ordered likelihoods has the same properties as the set of ordered likelihoods associated with a run using  $K_1 + K_2$  live points!

This property can be directly extended to **merge** any combination of  $M$  Static Nested Sampling runs. It can also be applied in reverse to **unravel** a run with  $K$  live points into  $K$  runs with a single live point. These “**strands**” form the base unit of a Nested Sampling run.

This “trivially parallelizable” property of Static Nested Sampling can also be directly extended to Dynamic Nested Sampling runs over where strands/batches are added over different likelihood ranges. For instance, combining two runs with  $K_1$  and  $K_2$  live points from  $\mathcal{L}_{\min(K_1)} < \mathcal{L}_{\min(K_2)} < \mathcal{L}_{\max(K_2)} < \mathcal{L}_{\max(K_1)}$  is equivalent to a Dynamic Nested Sampling run with  $K_1 + K_2$  live points between  $\mathcal{L}_{\min(K_2)} < \mathcal{L}_{\max(K_2)}$  and  $K_1$  elsewhere.

This process of unraveling/merging Nested Sampling runs can be done using the `unravel_run()` and `merge_runs()` functions. Both functions work with Static and Dynamic Nested Sampling results, although some of the provided ancillary quantities are not always valid. Their usage is straightforward:

```
res_list = dyfunc.unravel_run(res) # unravel run into strands
res_merge = dyfunc.merge_runs(res_list) # merge strands
```

**Note that these functions are mostly included for completeness and are not intended for heavy use in most practical applications.**

## Bootstrapping Runs

In theory, to properly incorporate sampling errors we have to marginalize over all possible paths particles can take through the distribution. In practice, however, we can approximate the set of all possible paths using the discrete set of paths taken from the set of  $K$  particles (live points) in a given run. By bootstrap resampling a new set of  $K$  strands (paths) from the current set of  $K$  live points, we are able to construct a new “**resampled**” run that probes these intrinsic sampling uncertainties. This both allows us to probe Poisson noise in the number of total steps  $N$  as well as the particular path-dependencies of the set of particles.

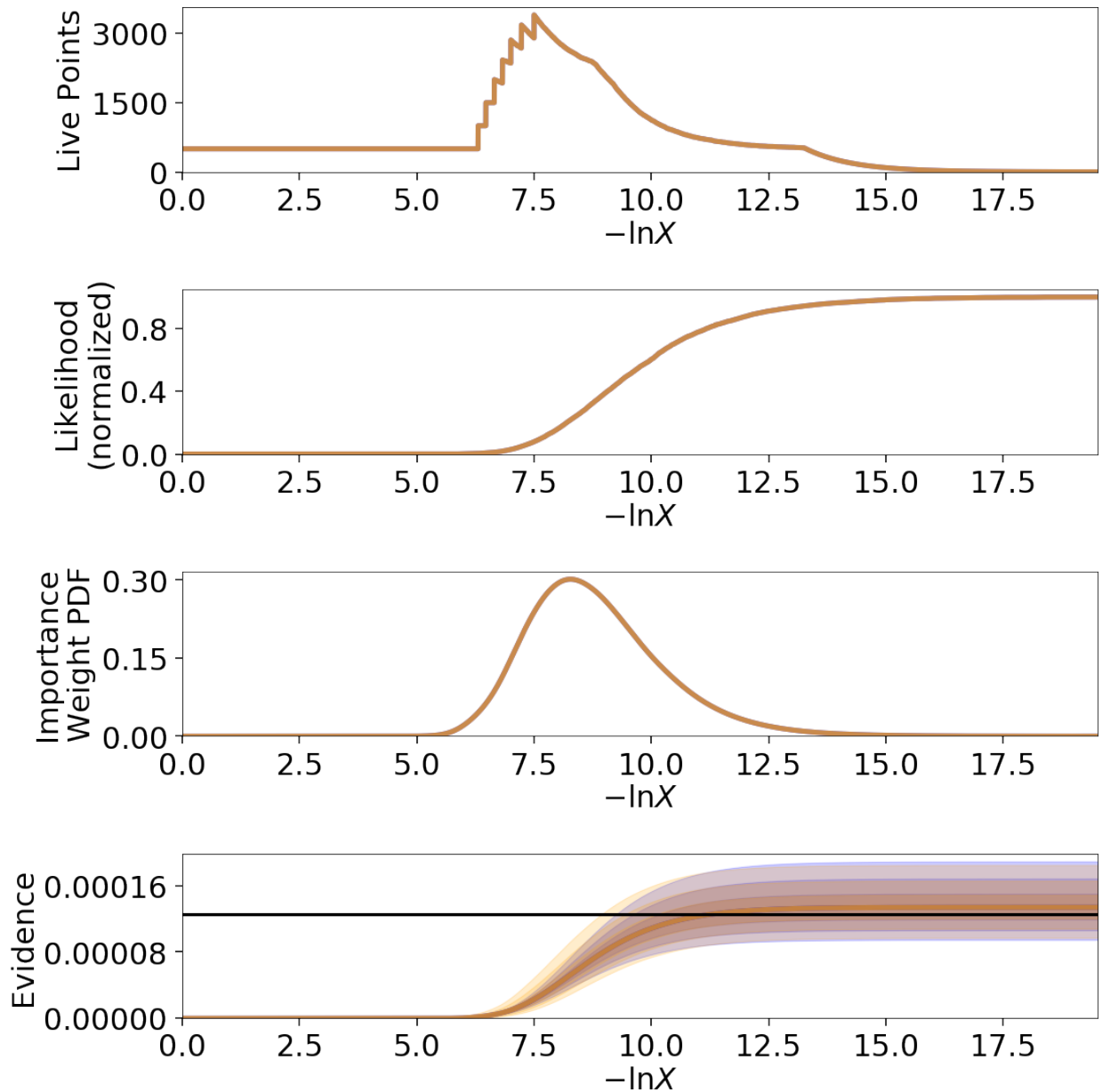
There is one small caveat to this result. When the number of live points remains constant, there is a symmetry in the information content provided by each strand: since all points are initialized from the prior  $\pi(\Theta)$ , they provide information on the prior volume  $X$  at a given iteration, allowing for both evidence estimation and posterior inference. Adding live points dynamically, however, can break this symmetry since not all strands are initialized starting from the prior: while these provide *relative* information useful for posterior inference, they are useless for evidence estimation. Since these two sets of “baseline” and “add-on” strands have qualitatively different properties, we use a stratified bootstrap to preserve their relative contributions to the final set of results.

The `resample_run()` function implements the bootstrap resampling approach. It then returns a new `Results` dictionary with a new set of samples and associated quantities.

Let’s use the same examples as *Jittering Runs* to demonstrate it’s usage. First, we will examine how these realizations compare with the original analytic evidence approximation:

```
# compute ln(evidence) error
lnzs = np.zeros((100, len(dres.logvol)))
for i in range(100):
    dres_r = dyfunc.resample_run(dres)
    lnzs[i] = np.interp(-dres.logvol, -dres_r.logvol, dres_r.logz)
lnzerr = np.std(lnzs, axis=0)

# plot comparison
dres_r = copy.deepcopy(dres)
dres_r['logzerr'] = lnzerr
fig, axes = dyplot.runplot(dres, color='blue')
fig, axes = dyplot.runplot(dres_r, color='orange',
                           lnz_truth=lnz_truth, truth_color='black',
                           fig=(fig, axes))
fig.tight_layout()
```



The final errors are again almost identical.

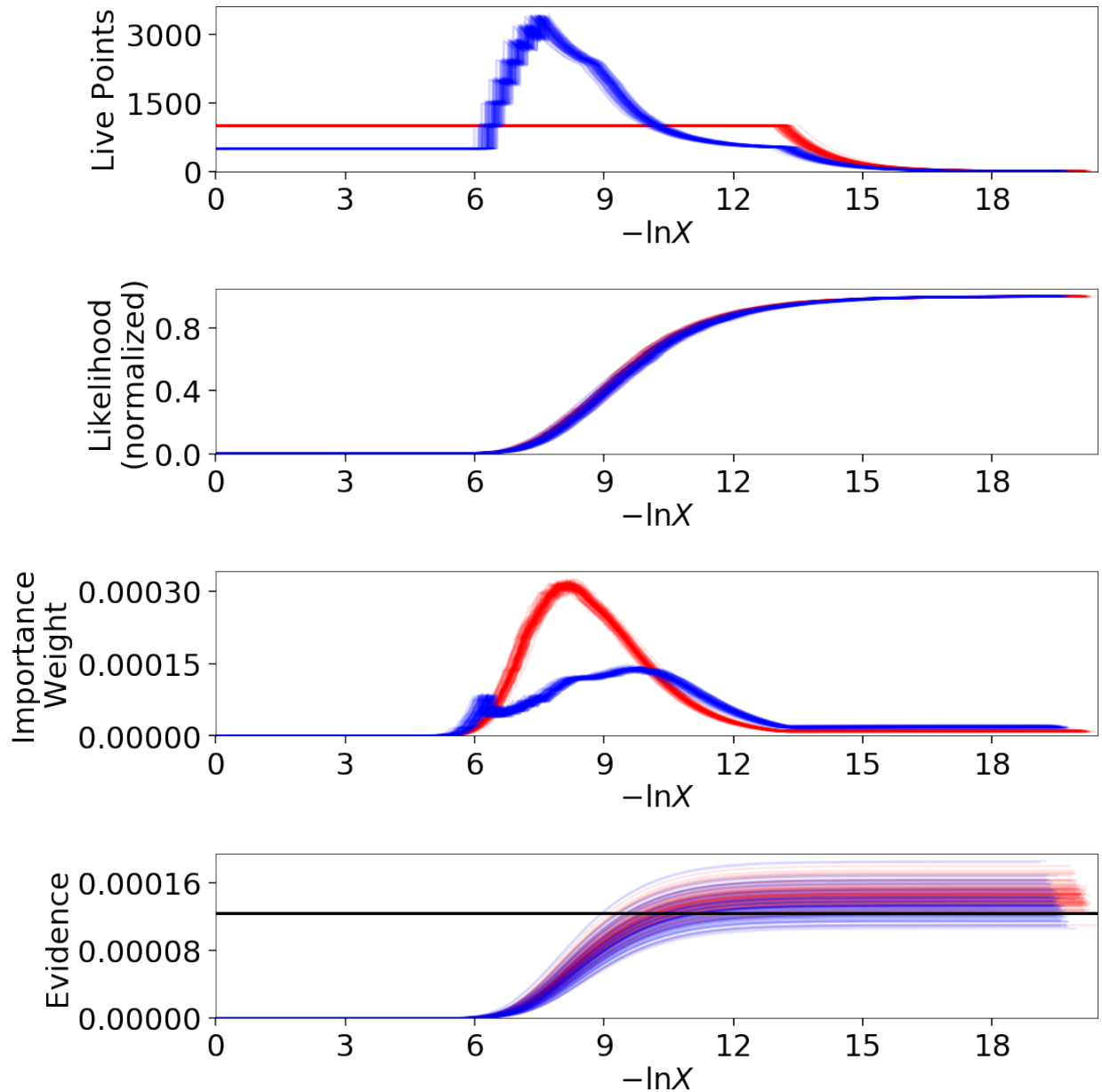
Now let's just plot a number of realizations directly to get a sense of how our (stratified) bootstrap affects other quantities:

```
# overplot draws on summary plots
fig, axes = plt.subplots(4, 1, figsize=(16, 16))
for i in range(100):
    res2_r = dyfunc.resample_run(res2)
    fig, axes = dyplot.runplot(res2_r, color='red',
                               plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                               mark_final_live=False, lnz_error=False,
                               fig=(fig, axes))
for i in range(100):
    dres_r = dyfunc.resample_run(dres)
```

(continues on next page)

(continued from previous page)

```
fig, axes = dyplot.runplot(dres_r, color='blue',
                           plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                           mark_final_live=False, lnz_error=False,
                           lnz_truth=lnz_truth, truth_color='black',
                           truth_kwargs={'alpha': 0.1},
                           fig=(fig, axes))
fig.tight_layout()
```



### Combined Uncertainties

Probing the combined statistical and sampling uncertainties just involves combining the results from *Bootstrapping Runs* and *Jittering Runs*. This is implemented via the `simulate_run()` function in `dynesty` or can be done

explicitly by the user:

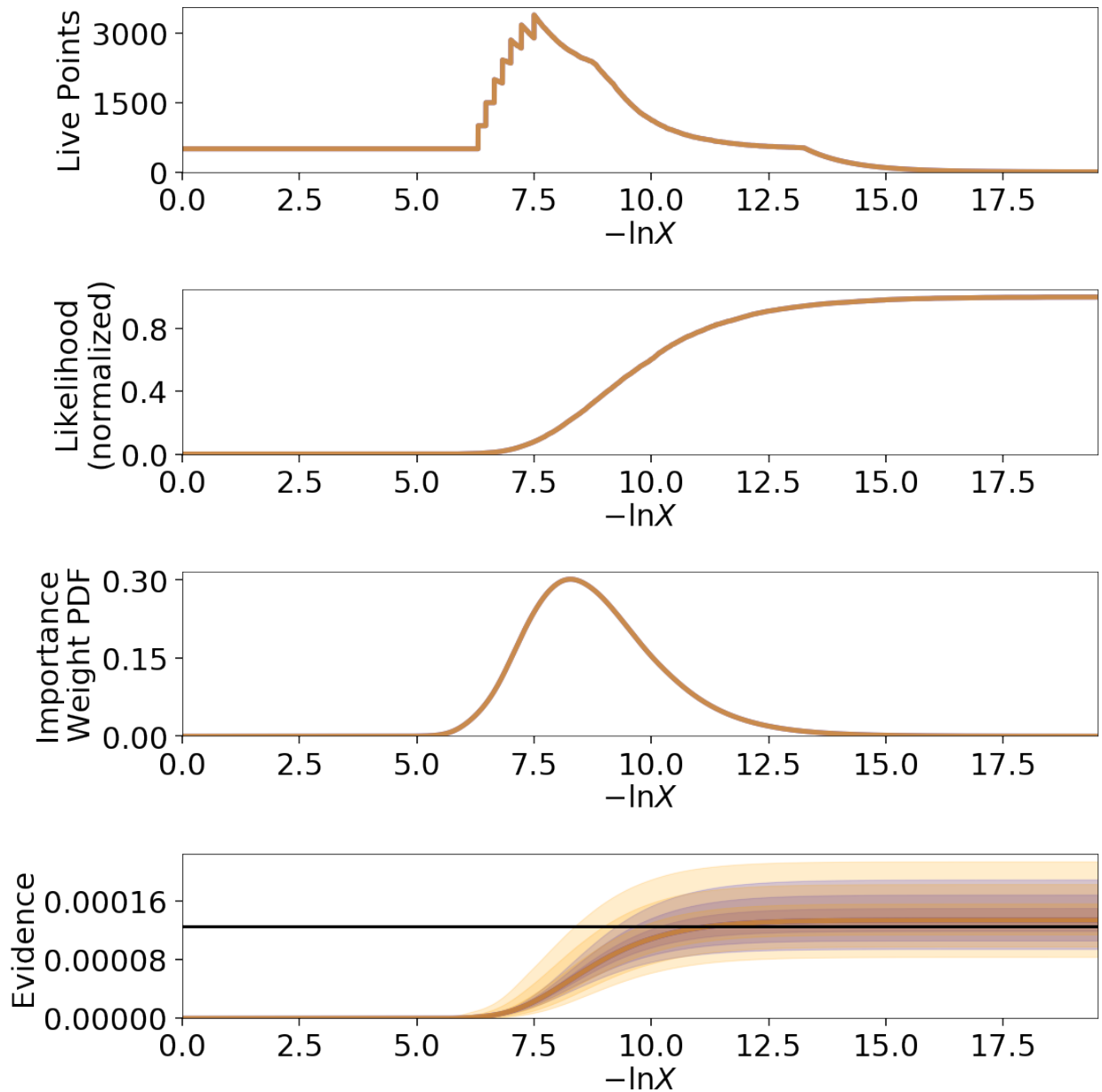
```
# simulating combined uncertainties (explicit)
new_res = dyfunc.jitter_run(dyfunc.resample_run(res))

# simulating combined uncertainties (implicit)
new_res2 = dyfunc.simulate_run(res)
```

Let's first examine the behavior using the same examples as shown in *Jittering Runs* and *Bootstrapping Runs*.

```
# compute ln(evidence) error
lnzs = np.zeros((100, len(dres.logvol)))
for i in range(100):
    dres_s = dyfunc.simulate_run(dres)
    lnzs[i] = np.interp(-dres.logvol, -dres_s.logvol, dres_s.logz)
lnzerr = np.std(lnzs, axis=0)

# plot comparison
dres_s = copy.deepcopy(dres)
dres_s['logzerr'] = lnzerr
fig, axes = dyplot.runplot(dres, color='blue')
fig, axes = dyplot.runplot(dres_s, color='orange',
                           lnz_truth=lnz_truth, truth_color='black',
                           fig=(fig, axes))
fig.tight_layout()
```



```
# overplot draws on summary plots
fig, axes = plt.subplots(4, 1, figsize=(16, 16))
for i in range(100):
    res2_s = dyfunc.simulate_run(res2)
    fig, axes = dyplot.runplot(res2_s, color='red',
                               plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                               mark_final_live=False, lnz_error=False,
                               fig=(fig, axes))
for i in range(100):
    dres_s = dyfunc.simulate_run(dres)
    fig, axes = dyplot.runplot(dres_s, color='blue',
                               plot_kwargs={'alpha': 0.1, 'linewidth': 2},
                               mark_final_live=False, lnz_error=False,
                               lnz_truth=lnz_truth, truth_color='black',
```

(continues on next page)

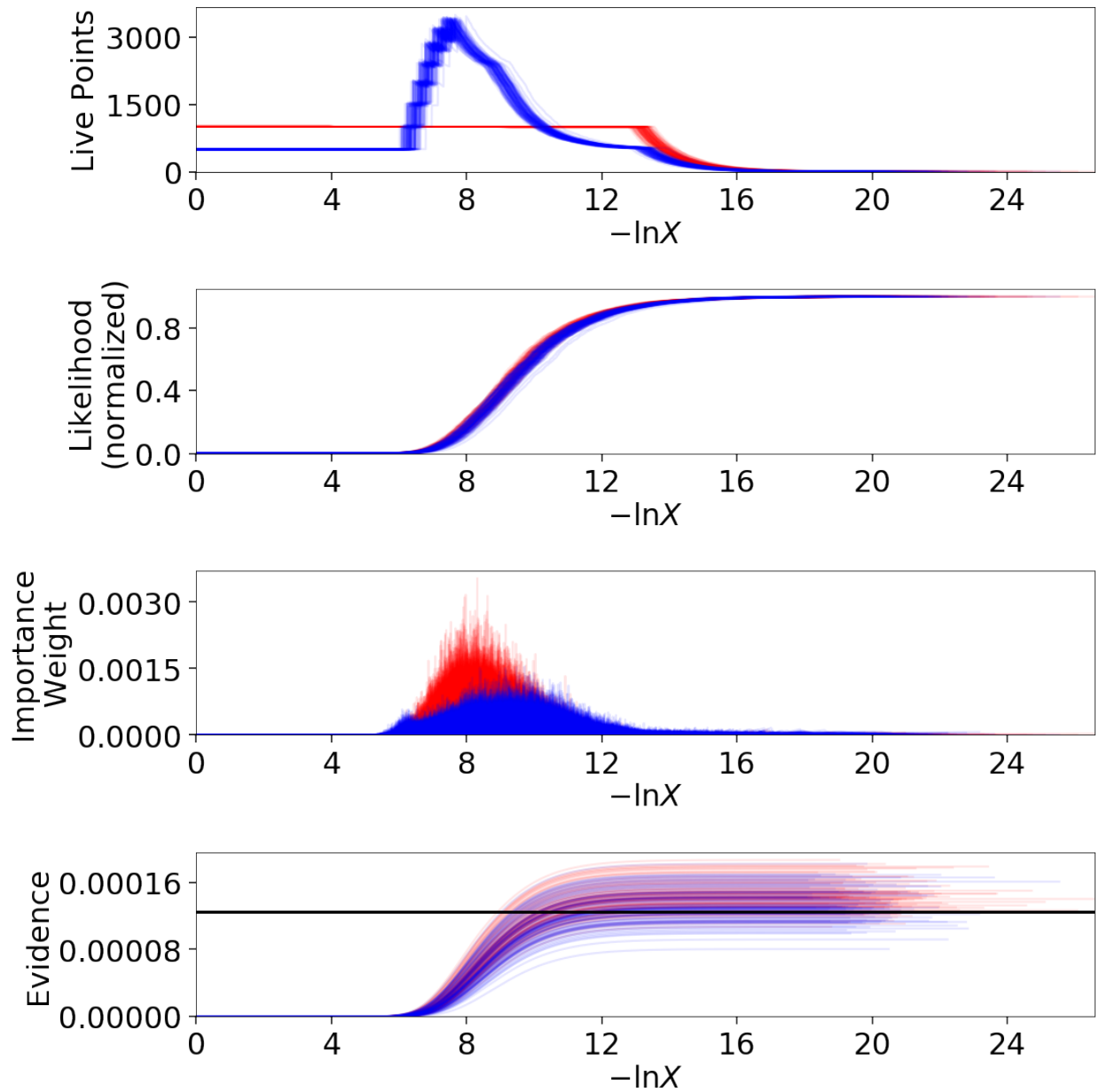


(continued from previous page)

```

truth_kwargs={'alpha': 0.1},
fig=(fig, axes))
fig.tight_layout()

```



We see that the final errors are about 50% larger than our approximation. This is quite typical, and reflects uncertainties that we ignored when deriving our approximation above.

### Validation Against Repeated Runs

As a quick demonstration of usage, we check the fidelity of these results against a set a repeated Nested Sampling runs:

```
# generate repeat nested sampling runs
Nrepeat = 500
repeat_res = []
dsampler = dynesty.DynamicNestedSampler(loglikelihood, prior_transform,
                                         ndim, bound='single')

for i in range(Nrepeat):
    dsampler.reset()
    dsampler.run_nested(print_progress=False, maxiter=5000, use_stop=False)
    repeat_res.append(dsampler.results)
```

```
# establish our comparison run
dsampler.reset()
dsampler.run_nested(print_progress=False, maxiter=5000, use_stop=False)
r = dsampler.results

# generate jittered runs
sim_res = []
for i in range(Nrepeat):
    sim_res.append(dyfunc.jitter_run(r))

# generate resampled runs
rsamp_res = []
for i in range(Nrepeat):
    rsamp_res.append(dyfunc.resample_run(r))

# generate simulated runs
samp_res = []
for i in range(Nrepeat):
    samp_res.append(dyfunc.simulate_run(r))
```

As an initial test, we can compare the estimated  $\ln \hat{Z}$  from each set of runs:

```
# compare evidence estimates

# analytic first-order approximation
lnz_mean, lnz_std = r.logz[-1], r.logzerr[-1]
print('Approx.:      {:.6.3f} +/- {:.6.3f}'.format(lnz_mean, lnz_std))

# jittered draws
lnz_arr = [results.logz[-1] for results in jitter_res]
lnz_mean, lnz_std = np.mean(lnz_arr), np.std(lnz_arr)
print('Sim.:        {:.6.3f} +/- {:.6.3f}'.format(lnz_mean, lnz_std))

# resampled draws
lnz_arr = [results.logz[-1] for results in rsamp_res]
lnz_mean, lnz_std = np.mean(lnz_arr), np.std(lnz_arr)
print('Resamp.:     {:.6.3f} +/- {:.6.3f}'.format(lnz_mean, lnz_std))

# repeated runs
lnz_arr = [results.logz[-1] for results in repeat_res]
lnz_mean, lnz_std = np.mean(lnz_arr), np.std(lnz_arr)
print('Rep. (mean): {:.6.3f} +/- {:.6.3f}'.format(lnz_mean, lnz_std))

# simulated draws
lnz_arr = [results.logz[-1] for results in sim_res]
lnz_mean, lnz_std = np.mean(lnz_arr), np.std(lnz_arr)
```

(continues on next page)

(continued from previous page)

```
print('Comb.:      {:6.3f} +/- {:6.3f}'.format(lnz_mean, lnz_std))

# jittered draws from repeated runs
lnz_arr = [dyfunc.jitter_run(results).logz[-1] for results in repeat_res]
lnz_mean, lnz_std = np.mean(lnz_arr), np.std(lnz_arr)
print('Rep. (sim.): {:6.3f} +/- {:6.3f}'.format(lnz_mean, lnz_std))
```

Out:

```
Approx.:      -8.670 +/-  0.207
Sim.:         -8.696 +/-  0.192
Resamp.:      -8.676 +/-  0.180
Rep. (mean):  -8.912 +/-  0.211
Comb.:        -8.699 +/-  0.262
Rep. (sim.):  -8.946 +/-  0.289
```

We can also compare the first and second moments of the posterior:

```
# compare posterior first moments

# jittered draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[0]
                  for results in jitter_res])
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Sim.:      {0} +/- {1}'.format(x_mean, x_std))

# resampled draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[0]
                  for results in rsamp_res])
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Resamp.:   {0} +/- {1}'.format(x_mean, x_std))

# repeated runs
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[0]
                  for results in repeat_res])
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Rep. (mean): {0} +/- {1}'.format(x_mean, x_std))

# simulated draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[0]
                  for results in sim_res])
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Comb.:     {0} +/- {1}'.format(x_mean, x_std))

# jittered draws from repeated runs
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(dyfunc.jitter_run(results).logwt))[0]
                  for results in repeat_res])
x_mean = np.round(np.mean(x_arr, axis=0), 3)
```

(continues on next page)

(continued from previous page)

```
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Rep. (sim.): {0} +/- {1}'.format(x_mean, x_std))
```

Out:

```
Sim.:      [-0.022 -0.022 -0.021] +/- [0.016 0.016 0.016]
Resamp.:   [-0.023 -0.023 -0.022] +/- [0.016 0.017 0.017]
Rep. (mean): [0.002 0.002 0.002] +/- [0.016 0.016 0.016]
Comb.:     [-0.022 -0.022 -0.021] +/- [0.021 0.021 0.022]
Rep. (sim.): [0.003 0.003 0.002] +/- [0.023 0.023 0.023]
```

```
# compare posterior second (diagonal) moments

# jittered draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[1]
                 for results in jitter_res])
x_arr = [np.diag(x) for x in x_arr]
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Sim.:      {0} +/- {1}'.format(x_mean, x_std))

# resampled draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[1]
                 for results in rsamp_res])
x_arr = [np.diag(x) for x in x_arr]
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Resamp.:   {0} +/- {1}'.format(x_mean, x_std))

# repeated runs
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[1]
                 for results in repeat_res])
x_arr = [np.diag(x) for x in x_arr]
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Rep. (mean): {0} +/- {1}'.format(x_mean, x_std))

# simulated draws
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(results.logwt))[1]
                 for results in sim_res])
x_arr = [np.diag(x) for x in x_arr]
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Comb.:     {0} +/- {1}'.format(x_mean, x_std))

# jittered draws from repeated runs
x_arr = np.array([dyfunc.mean_and_cov(results.samples,
                                     weights=np.exp(dyfunc.jitter_run(results).logwt))[1]
                 for results in repeat_res])
x_arr = [np.diag(x) for x in x_arr]
x_mean = np.round(np.mean(x_arr, axis=0), 3)
x_std = np.round(np.std(x_arr, axis=0), 3)
print('Rep. (sim.): {0} +/- {1}'.format(x_mean, x_std))
```

Out:

```
Sim.:      [1.041 1.038 1.046] +/- [0.026 0.026 0.026]
Resamp.:   [1.039 1.035 1.044] +/- [0.026 0.026 0.027]
Rep. (mean): [0.994 0.994 0.993] +/- [0.026 0.026 0.026]
Comb.:     [1.041 1.037 1.045] +/- [0.035 0.036 0.037]
Rep. (sim.): [0.993 0.993 0.992] +/- [0.035 0.034 0.035]
```

Our simulated uncertainties seem to do an excellent job of capturing the intrinsic combined statistical and sampling uncertainties.

## Posterior Uncertainties

As discussed in *How Many Samples are Enough?*, it can be difficult to determine how many samples are needed to guarantee the posterior density estimate  $\hat{P}(\Theta)$  constructed from the set of samples  $\{\Theta_1, \dots, \Theta_N\}$  is a “good” approximation to the true posterior density  $P(\Theta)$ . One way of getting a handle on this is to measure the “difference” between the two distributions using the KL divergence:

$$H(\hat{P}|P) \equiv \int_{\Omega_{\Theta}} \hat{P}(\Theta) \ln \frac{\hat{P}(\Theta)}{P(\Theta)} d\Theta$$

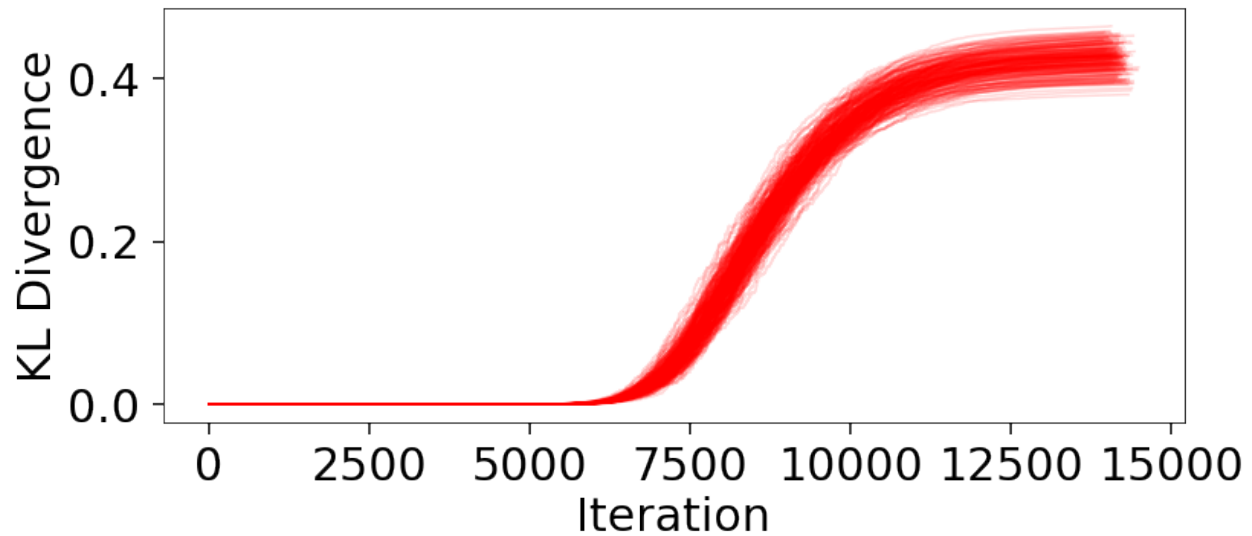
Since we do not know  $P(\Theta)$ , we can substitute  $\hat{P} \rightarrow \hat{P}'$  and  $P \rightarrow \hat{P}$  to construct an empirical estimate of this quantity based on realizations of  $\hat{P}(\Theta)$ :

$$H(\hat{P}'|\hat{P}) = \int_{\Omega_{\Theta}} \hat{P}'(\Theta) \ln \frac{\hat{P}'(\Theta)}{\hat{P}(\Theta)} d\Theta = \sum_i \hat{p}'_i (\ln \hat{p}'_i - \ln \hat{p}_i)$$

KL divergences between (realizations of) Nested Sampling runs can be computed in `dynesty` using the `kl_divergence()` and `kld_error()` functions. The former is slower but slightly more flexible while the latter generates comparisons directly over realizations of a single run. Let’s examine the results from the Static Nested Sampling run used above to get a sense of what these look like:

```
# compute KL divergences
klds = []
for i in range(Nrepeat):
    kld = dyfunc.kld_error(res2, error='simulate')
    klds.append(kld)

# plot (cumulative) KL divergences
plt.figure(figsize=(12, 5))
for kld in klds:
    plt.plot(kld, color='red', alpha=0.15)
plt.xlabel('Iteration')
plt.ylabel('KL Divergence');
```



The behavior appears qualitatively similar to our evidence results, since the majority of the KL divergence is coming from integrating over the bulk of the posterior mass in the typical set. The variation in these results are plotted below:

```
from scipy.stats import gaussian_kde

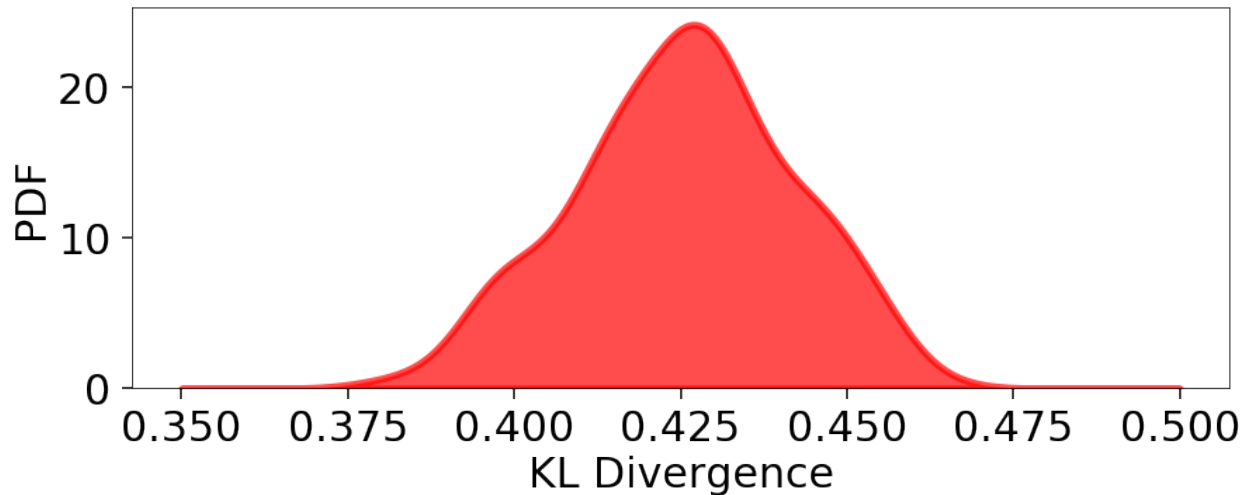
# compute KLD kernel density estimate
kl_div = [kld[-1] for kld in kl_ds]
kde = gaussian_kde(kl_div)

# plot results
plt.figure(figsize=(10, 4))
x = np.linspace(0.35, 0.5, 1000)
plt.fill_between(x, kde.pdf(x), color='red', alpha=0.7, lw=5)
plt.ylim([0., None])
plt.xlabel('KL Divergence')
plt.ylabel('PDF')

# summarize results
kl_div_mean, kl_div_std = np.mean(kl_div), np.std(kl_div)
print('Mean:    {:.3f}'.format(kl_div_mean))
print('Std:     {:.3f}'.format(kl_div_std))
print('Std(%):  {:.3f}'.format(kl_div_std / kl_div_mean * 100.))
```

Out:

```
Mean:    0.425
Std:     0.016
Std(%):  3.833
```



Our Dynamic Nested Sampling run contains the same number of samples but preferentially places them around the typical set to improve posterior estimation. The corresponding results are shown below for comparison:

```
klds2 = []
for i in range(Nrepeat):
    kld2 = dyfunc.kld_error(dres)
    klds2.append(kld2)

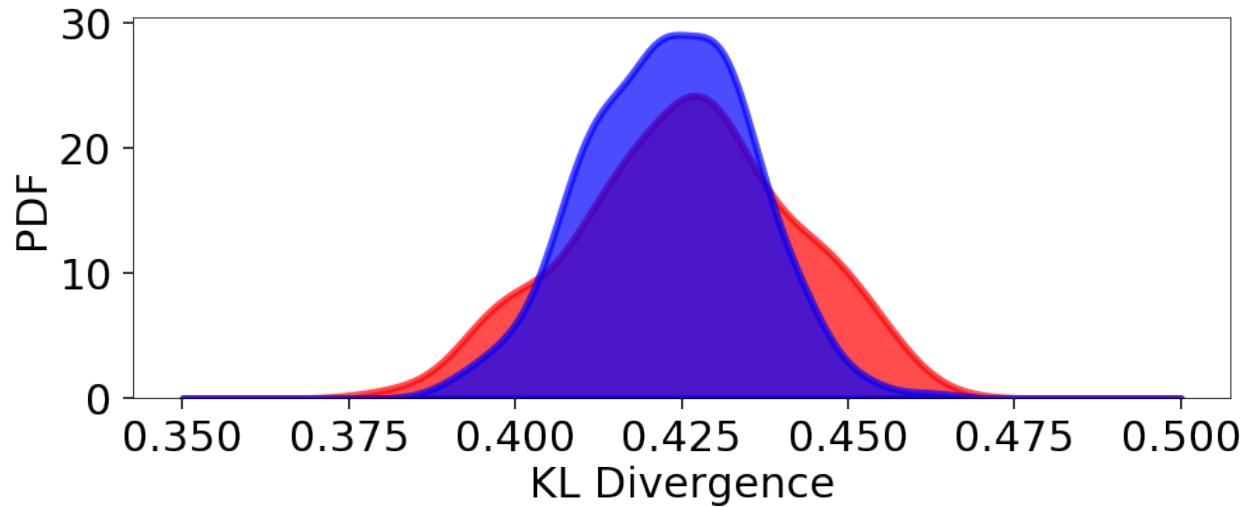
# compute KLD kernel density estimate
kl_div2 = [kld2[-1] for kld2 in klds2]
kde2 = gaussian_kde(kl_div2)

# plot results
plt.figure(figsize=(14, 5))
plt.fill_between(x, kde.pdf(x), color='red', alpha=0.7, lw=5)
plt.fill_between(x, kde2.pdf(x), color='blue', alpha=0.7, lw=5)
plt.ylim([0., None])
plt.xlabel('KL Divergence')
plt.ylabel('PDF')

# summarize results
kl_div2_mean, kl_div2_std = np.mean(kl_div2), np.std(kl_div2)
print('Mean:    {:.3f}'.format(kl_div2_mean))
print('Std:     {:.3f}'.format(kl_div2_std))
print('Std(%):  {:.3f}'.format(kl_div2_std / kl_div2_mean * 100.))
```

Out:

```
Mean:    0.423
Std:     0.012
Std(%):  2.909
```



We see that although the mean KL divergence is similar, the fractional variation around the mean is smaller.

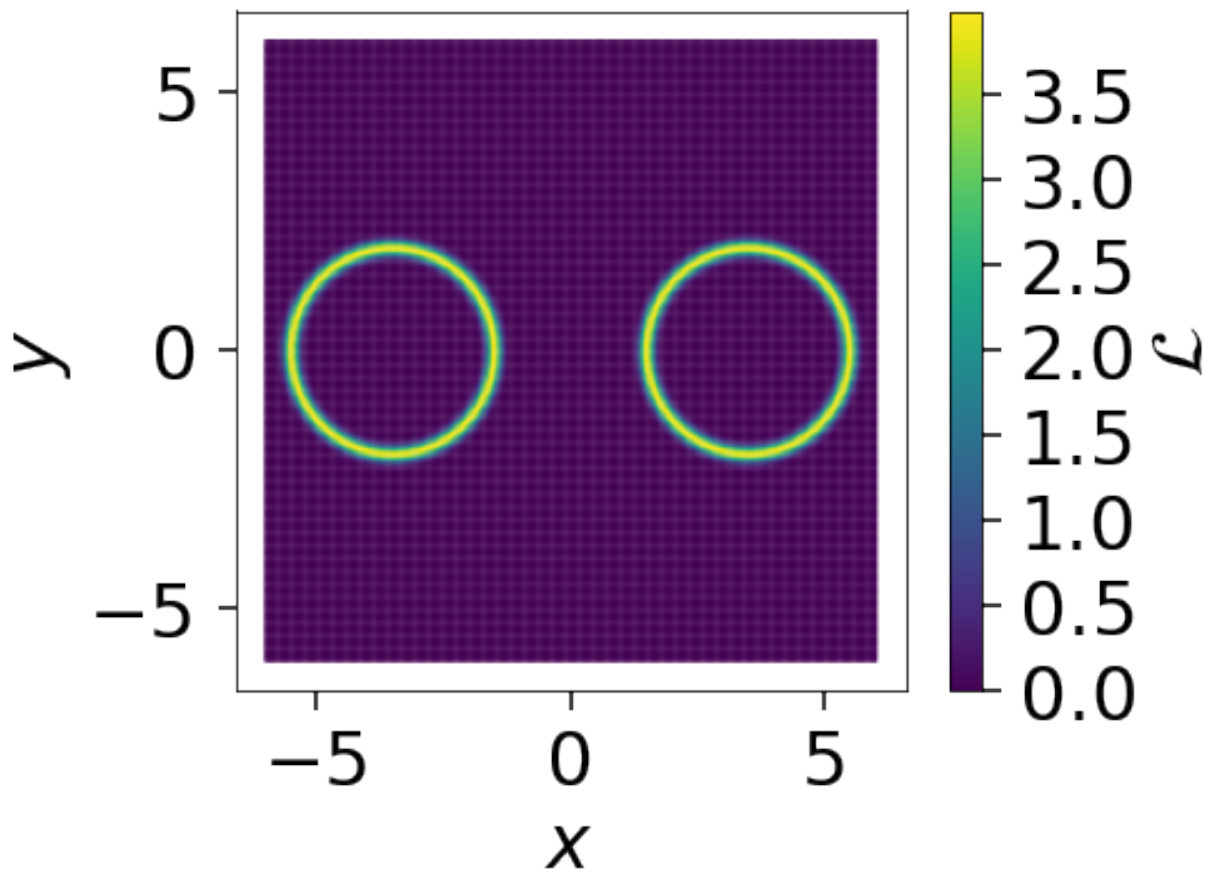
### 3.14.6 Examples

This page highlights several examples on how `dynesty` can be used in practice, illustrating both simple and more advanced aspects of the code. Jupyter notebooks containing more details are available [on Github](#).

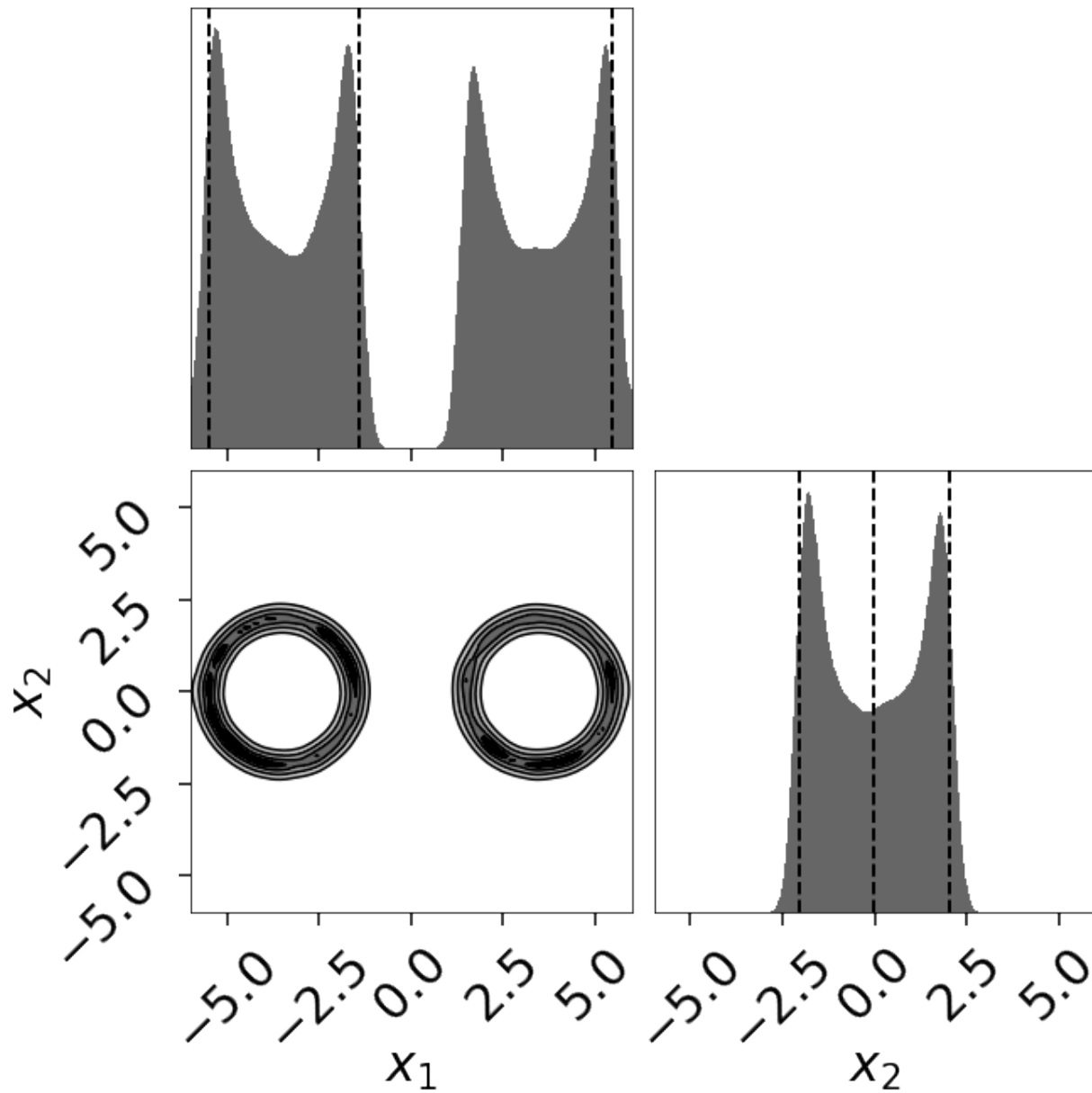
#### Gaussian Shells

The “Gaussian shells” likelihood is a useful test case for illustrating the ability of nested sampling to deal with oddly-shaped distributions that can be difficult to probe with simple random-walk MCMC methods.



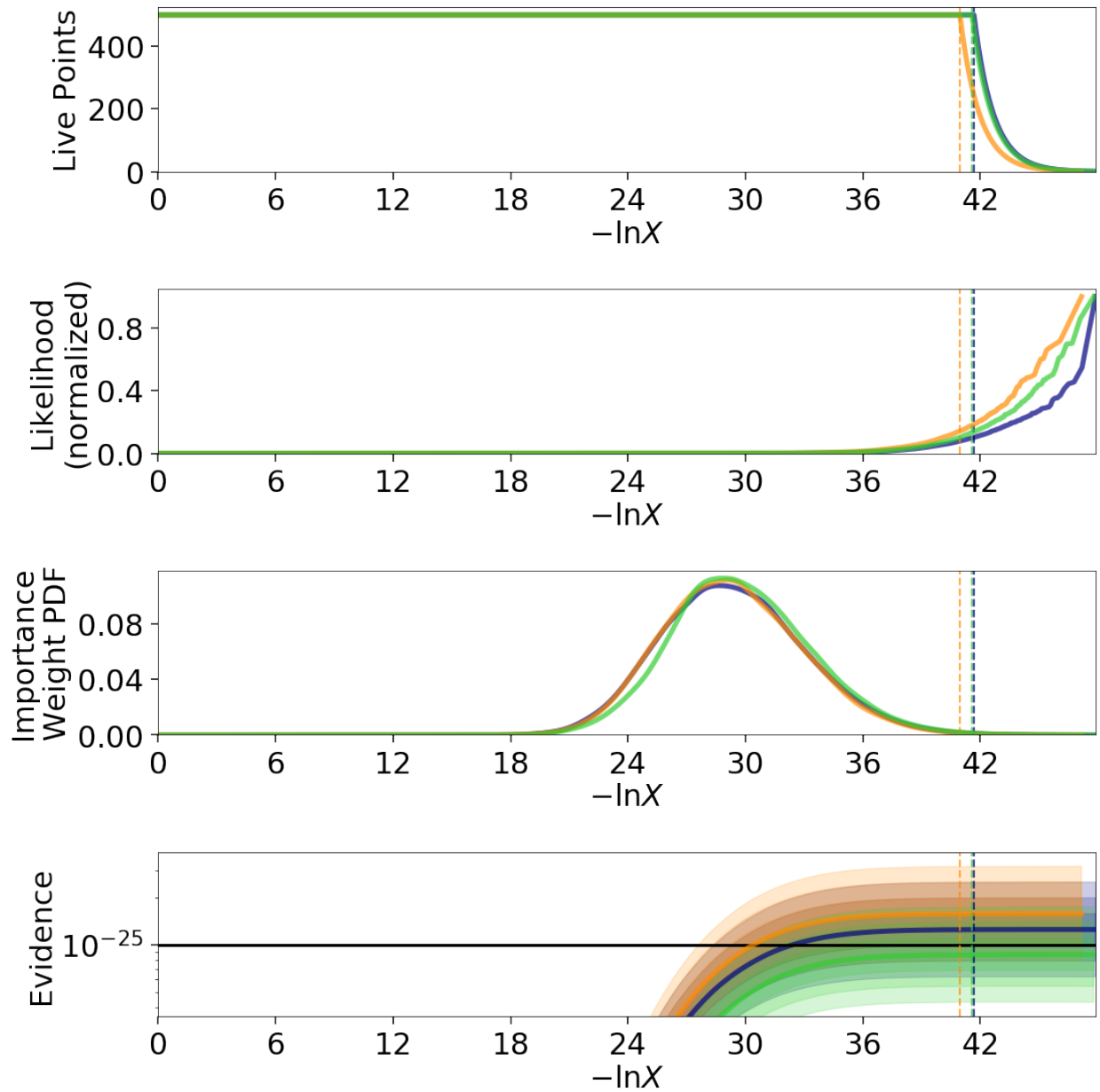


`dynesty` returns the following posterior estimate:



### 25-D Correlated Normal

`dynesty` supports three tiers of sampling techniques: uniform sampling for low dimensional problems, random walks for low-to-moderate dimensional problems, and slice sampling for high-dimensional problems. The performance of our three slice sampling algorithms is shown below with 'slice' in blue, 'rslice' in orange, and 'hslice' in green:



The recovery of the mean and variances also looks reasonable:

Out:

Mean:

```
slice: [ 0.00464139  0.01779515 -0.00059102  0.01545529  0.00196981
        -0.01102295  0.0224263   0.00128731  0.0165881   0.01004745
         0.02403447  0.02490094  0.02634624  0.0221378   0.01095415
         0.01451562  0.00810768  0.0186859   0.01441112  0.0142988
         0.00562206  0.01013293  0.03261655  0.01514425 -0.00096744]

rslice: [-0.01262438 -0.02215703 -0.02894398  0.00412   -0.01239199
         0.00537269 -0.00883646 -0.01124688 -0.00622417 -0.02345228
        -0.01226172 -0.01741414 -0.00340907 -0.02107367 -0.0440053]
```

(continues on next page)

(continued from previous page)

```
-0.00461723  0.00210266 -0.00553831 -0.0342508  -0.04259448
-0.03088255  0.00615101 -0.00708561 -0.01839912 -0.01779207]

hslice: [-0.00770402 -0.02471443 -0.042487   -0.01095513 -0.03419283
-0.01587577 -0.0073069  -0.01633131 -0.01914578 -0.02243197
-0.01922804 -0.03532052 -0.0229837  -0.02118347 -0.00624797
-0.02247772 -0.02899336 -0.02943324 -0.02318504 -0.02445557
-0.02455501 -0.00661906 -0.029969   -0.01755215 -0.01973601]

Variance:

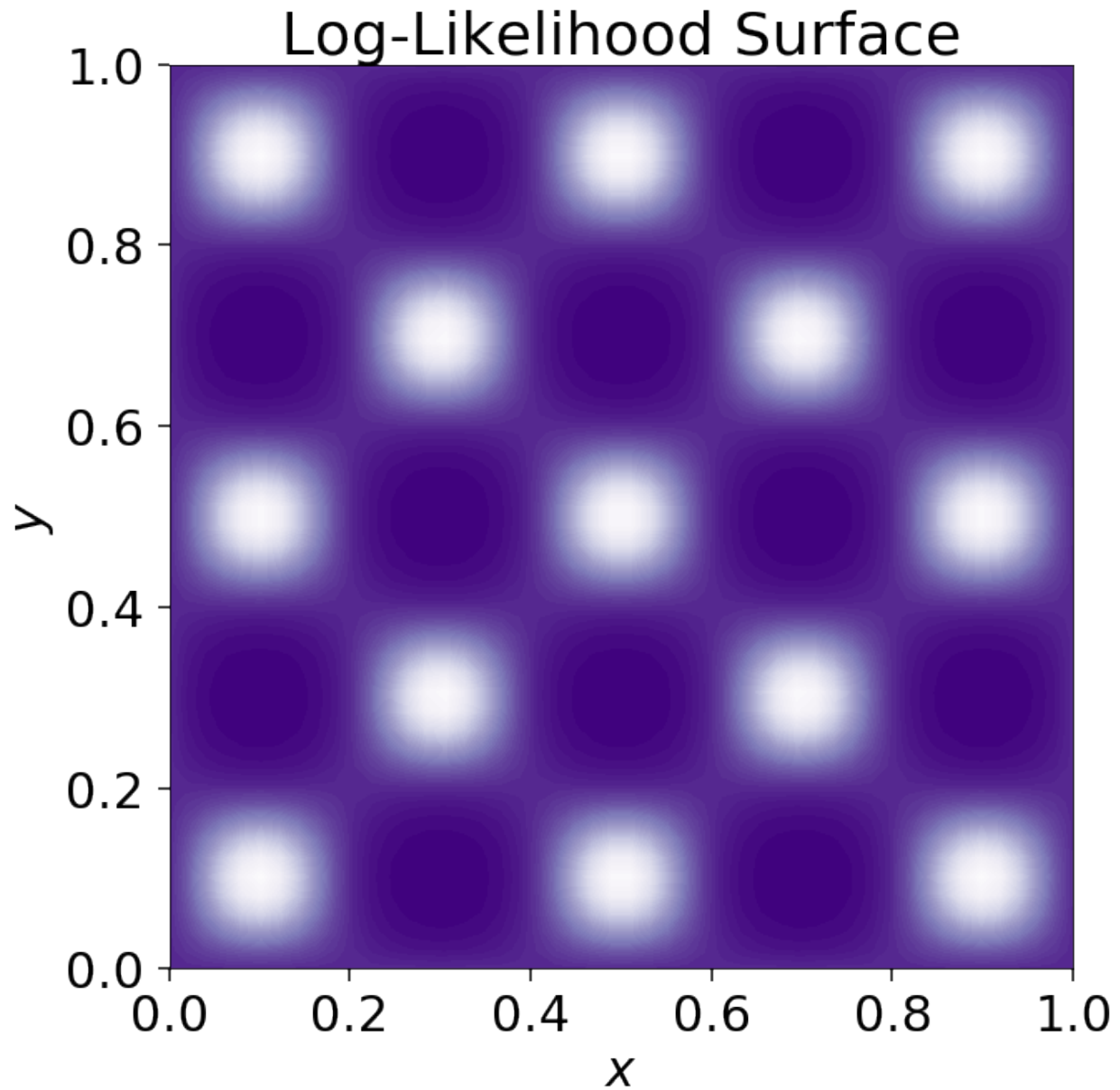
slice: [0.99505849 0.97200556 1.03652393 0.99880137 0.98862659
1.00075338 0.99494738 0.99976605 0.9965513  1.01882192
0.97857377 0.99662175 0.98167938 0.98594533 0.99283048
1.01748035 0.97116046 1.00298012 1.0111866  1.0202167
0.99495185 1.04121714 0.99569076 1.00889279 0.97541806]

rslice: [0.99821846 1.02606283 0.99985712 0.99358811 1.00021096
0.98121015 0.99658629 0.99363295 1.00926491 0.99788756
0.98109713 0.93410622 1.02039019 1.01227682 0.97890678
1.03568037 1.01749501 0.98945627 0.99539522 0.98519908
0.97363697 0.99418089 0.99500449 0.92339752 0.9456492 ]

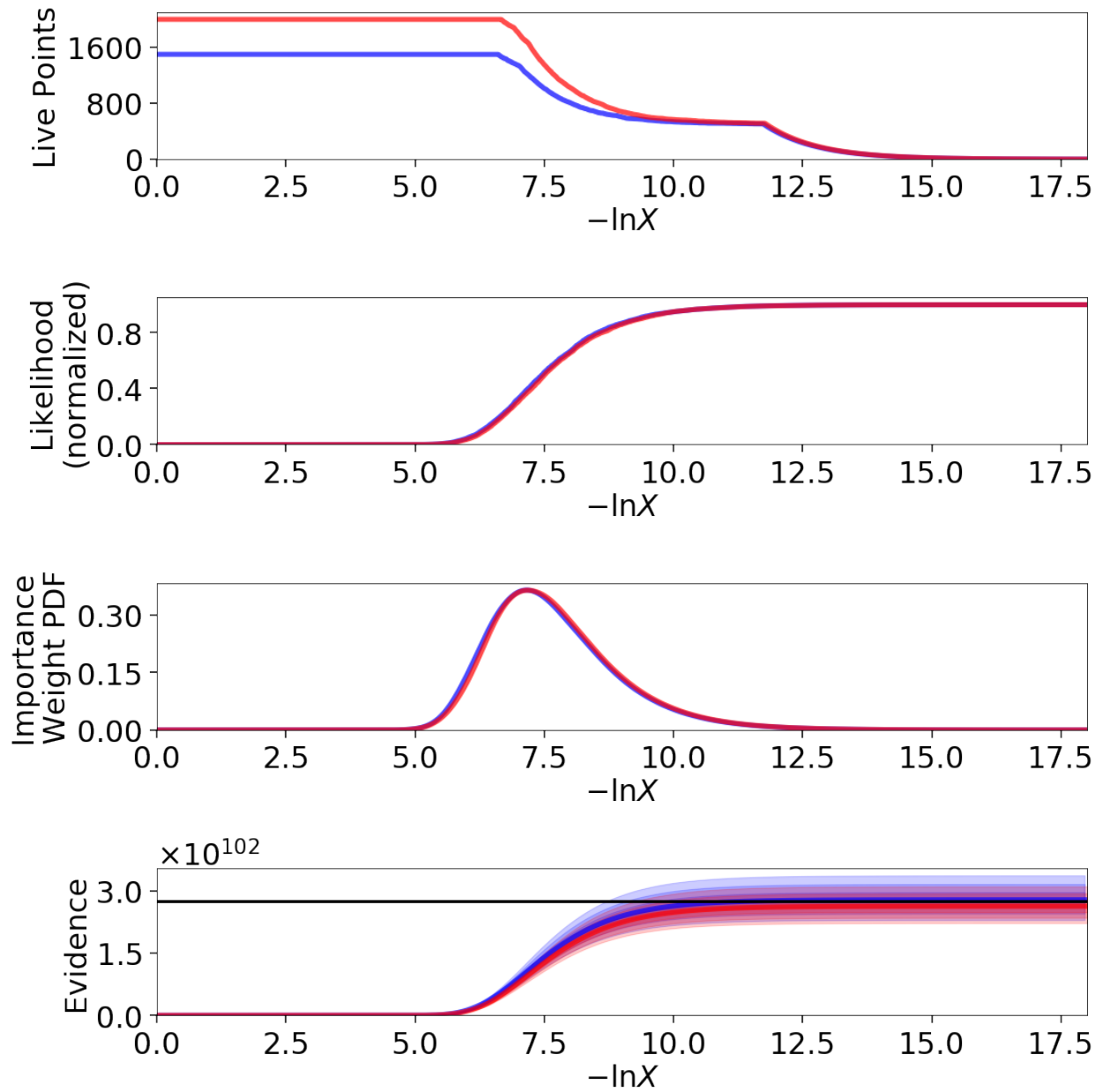
hslice: [0.99093439 1.00783986 1.00908646 0.98642076 1.02137535
1.00685834 0.98783381 1.02676786 1.021385  1.00869302
0.96427675 0.96278338 0.98425856 1.00001262 1.00796364
1.01546776 1.01142401 1.01775994 0.99990323 1.00081825
1.00426292 1.00153755 0.99376306 0.99011333 0.98264584]
```

## Eggbox

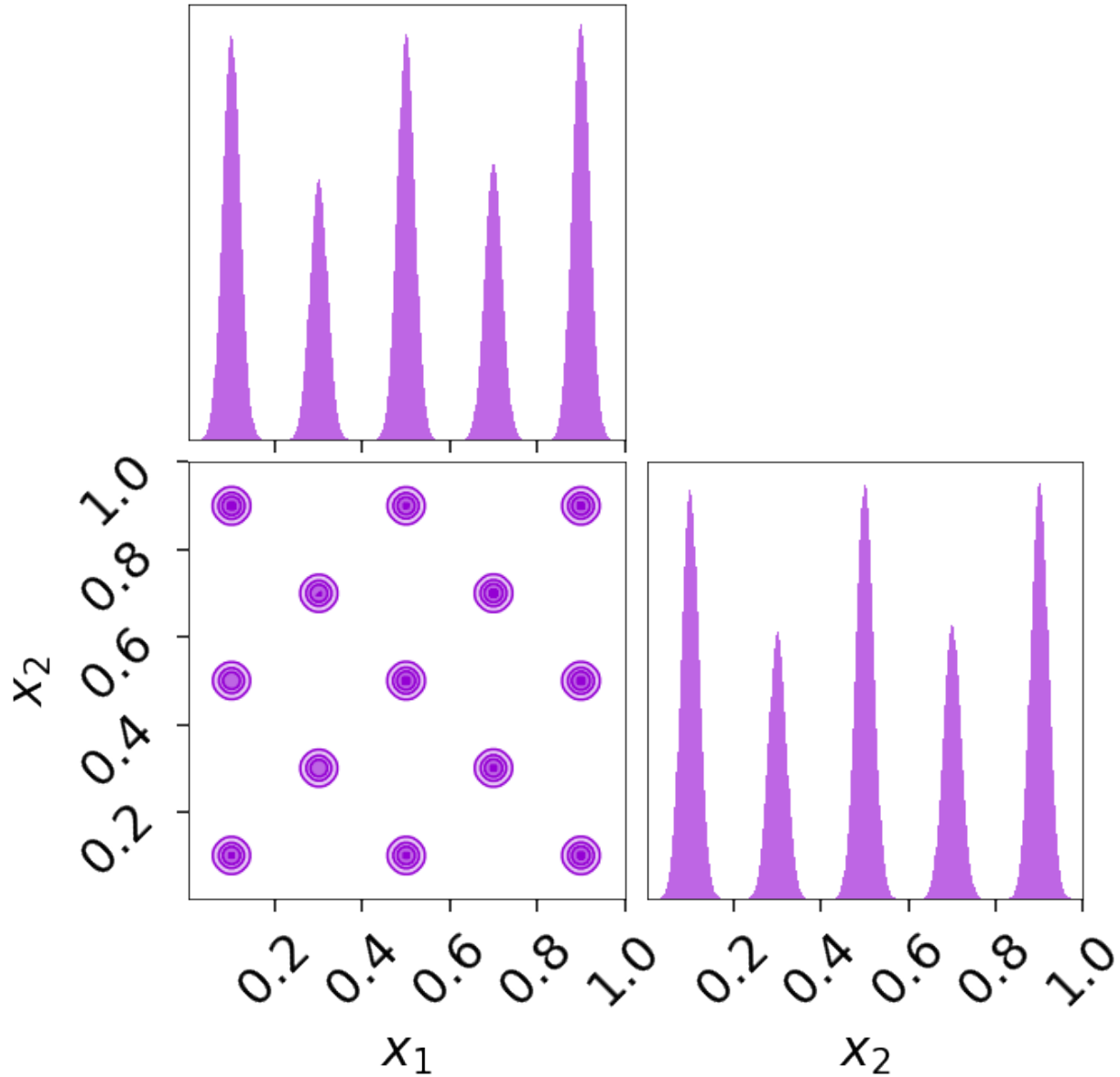
The “Eggbox” likelihood is a useful test case that demonstrates Nested Sampling’s ability to properly sample/integrate over multi-modal distributions.



The evidence estimates from two independent runs look reasonable:



The posterior estimate also looks quite good:

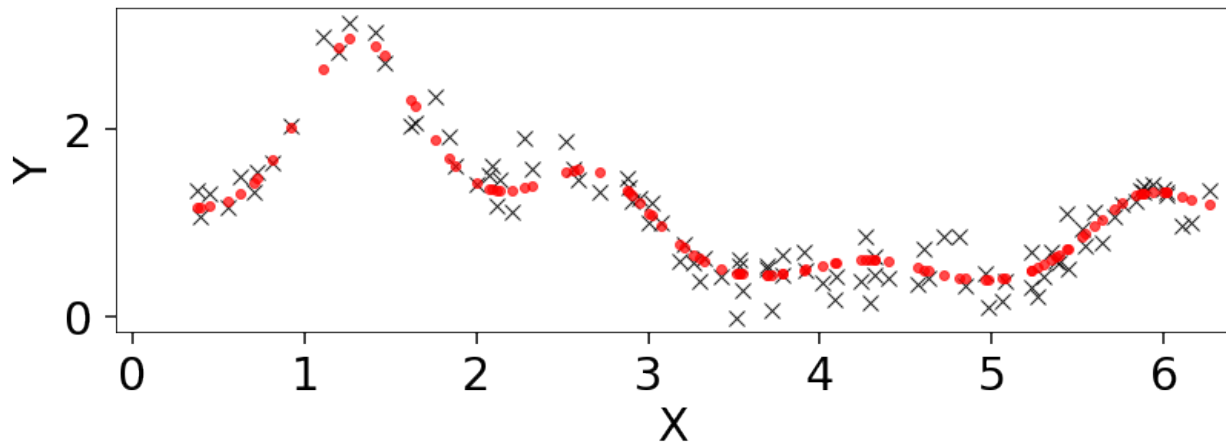


### Exponential Wave

This toy problem was originally suggested by [suggested](#) by Johannes Buchner for being multimodal with two roughly equal-amplitude solutions. We are interested in modeling periodic data of the form:

$$y(x) = \exp [n_a \sin(f_a x + p_a) + n_b \sin(f_b x + p_b)]$$

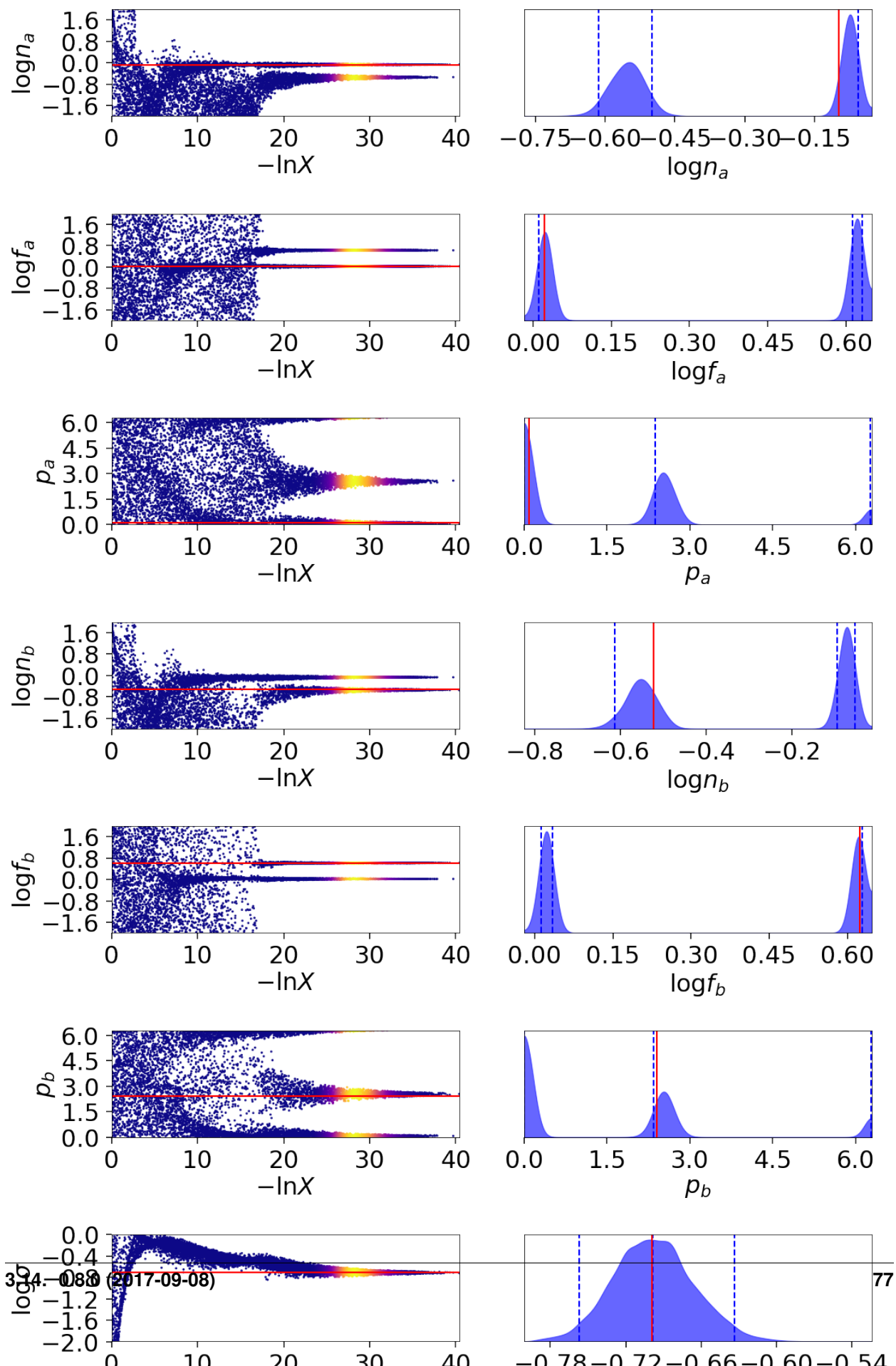
where  $x$  goes from 0 to  $2\pi$ .

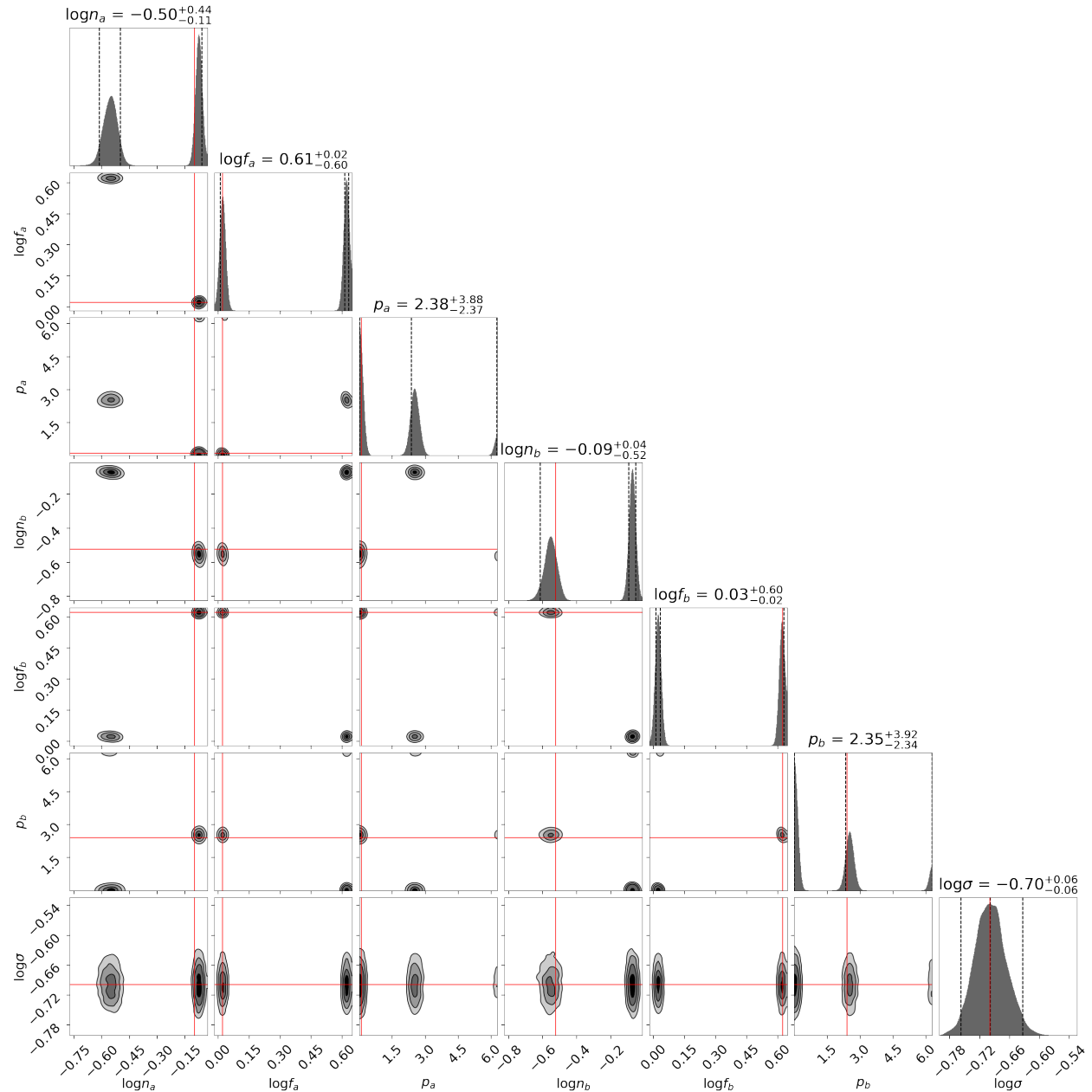


This model has six free parameters controlling the relevant amplitude, period, and phase of each component (which have periodic boundary conditions). We also have a seventh,  $\sigma$ , corresponding to the amount of scatter.

The results are shown below.





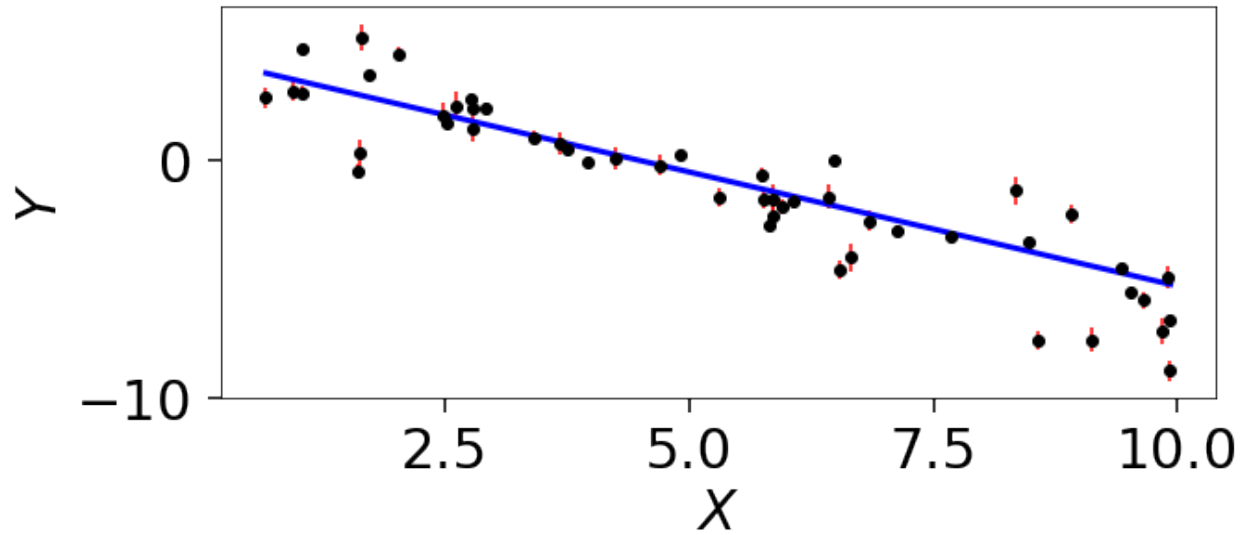


## Linear Regression

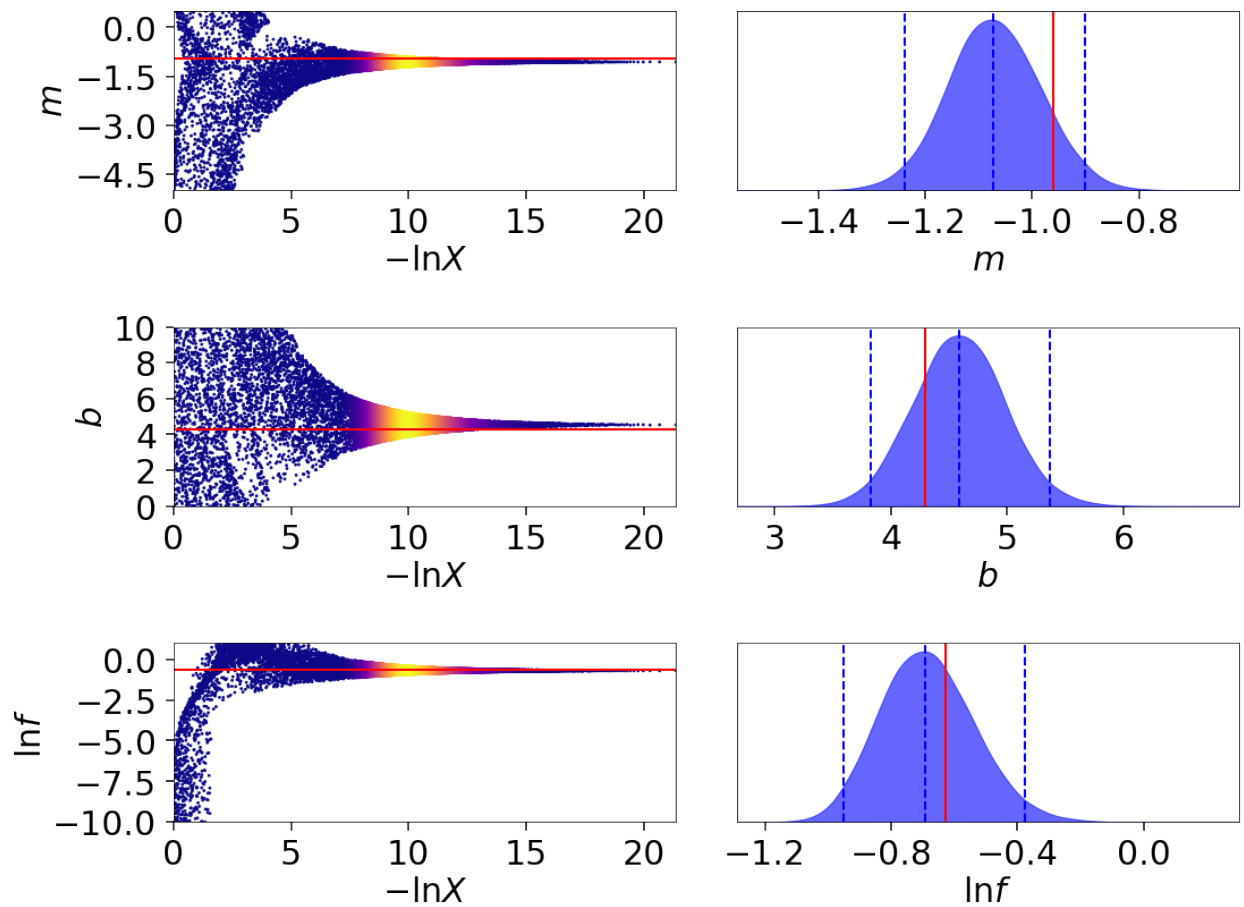
Linear regression is ubiquitous in research. In this example we'll fit a line

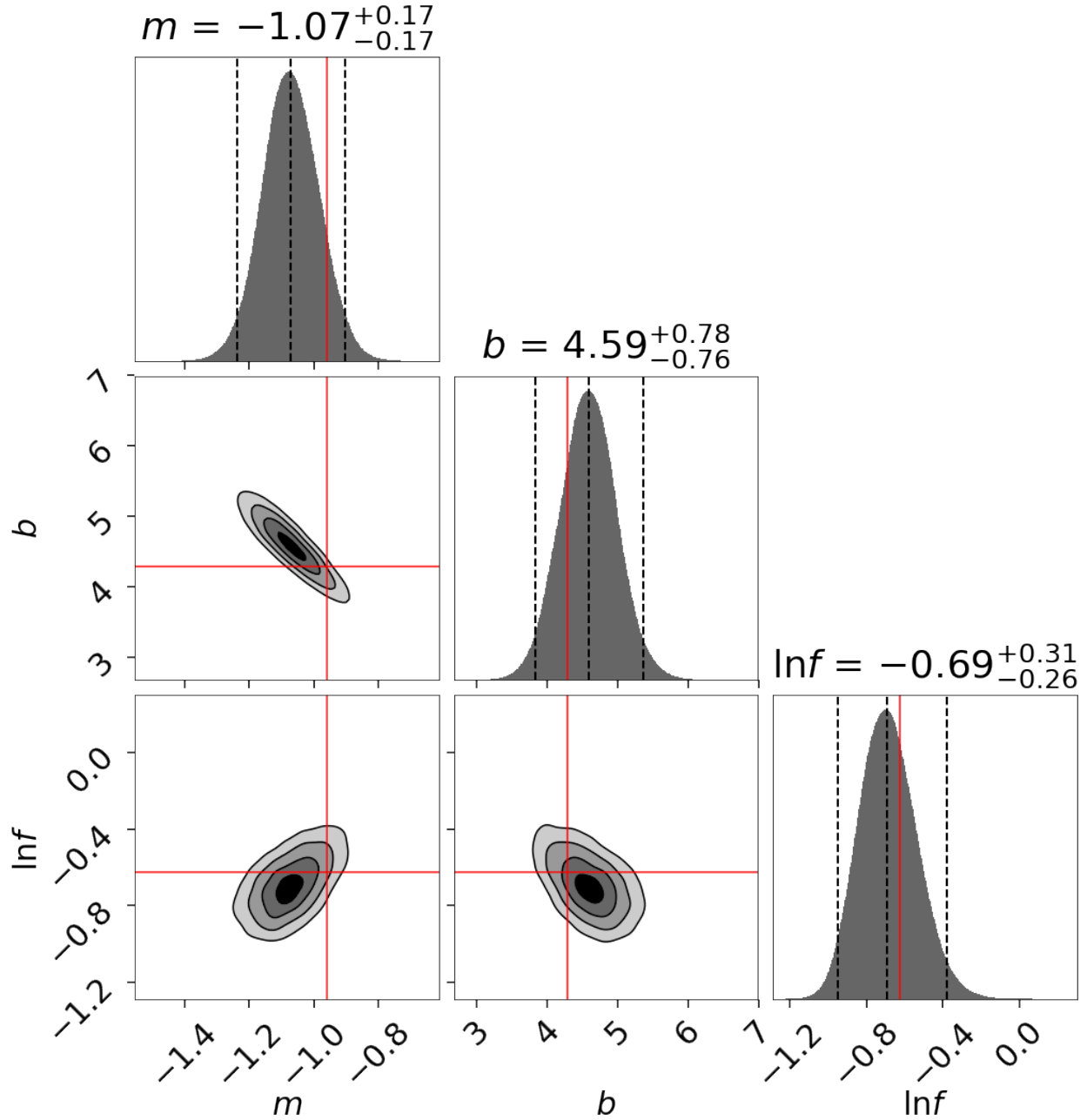
$$y = mx + b$$

to data where the error bars have been over/underestimated by some fraction of the observed value  $f$  and need to be decreased/increased. Note that this example is taken directly from the [emcee documentation](#).



The trace plot and corner plot show reasonable parameter recovery.





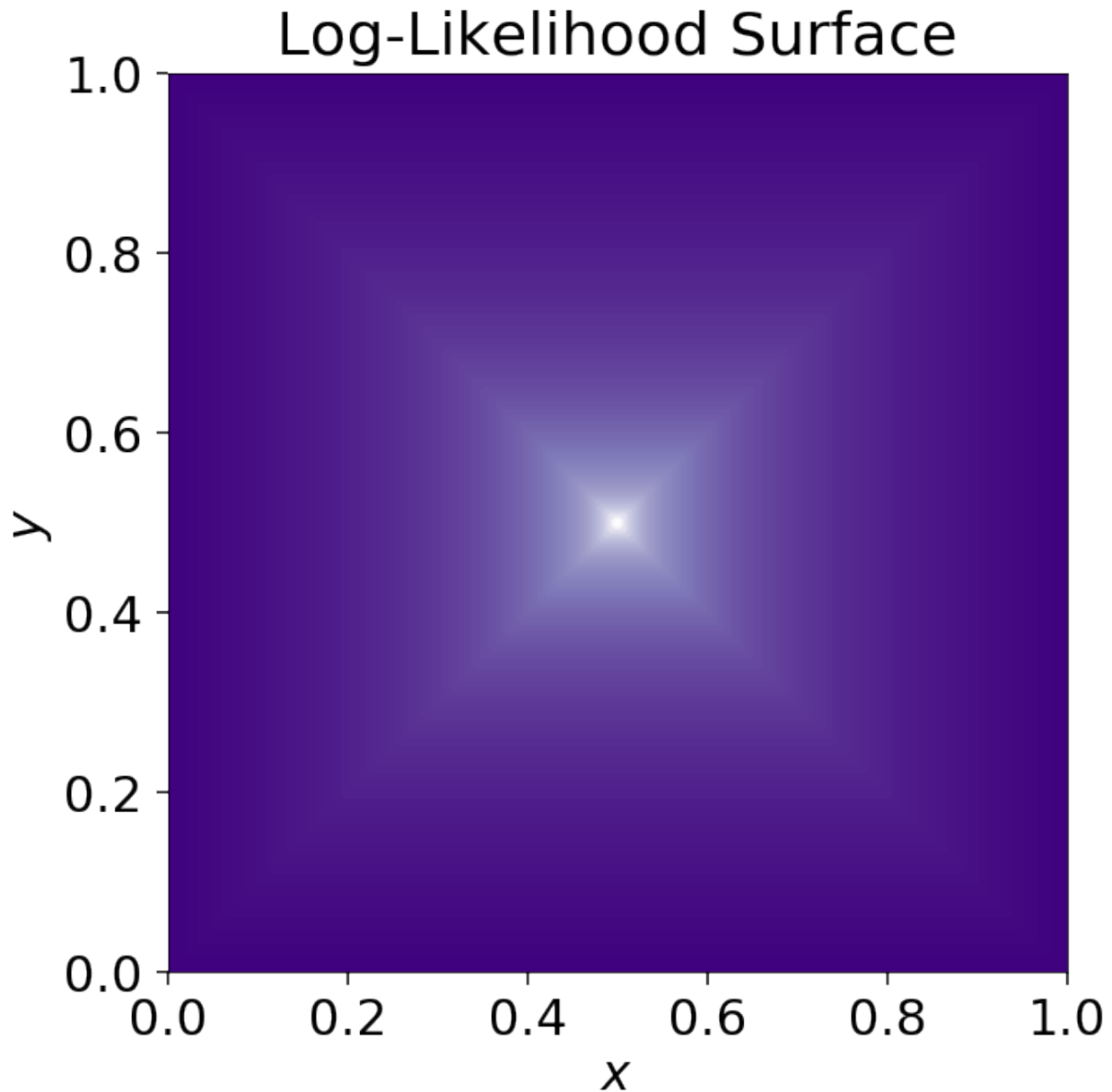
### Hyper-Pyramid

One of the key assumptions of *Static Nested Sampling* (extended by *Dynamic Nested Sampling*) is that we “shrink” the prior volume  $X_i$  at each iteration  $i$  as

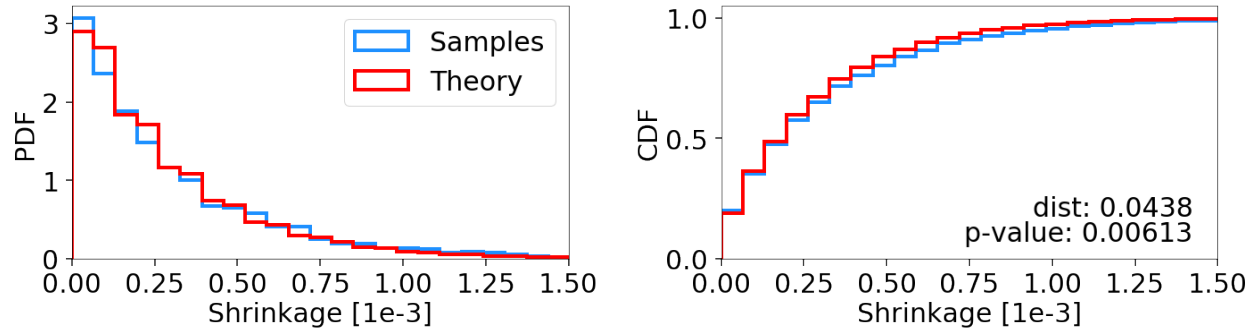
$$X_i = t_i X_{i-1}, \quad t_i \sim \text{Beta}(K, 1)$$

at each iteration with  $t_i$  a random variable with distribution  $\text{Beta}(K, 1)$  where  $K$  is the total number of live points. We can empirically test this assumption by using functions whose volumes can be analytically computed directly from the position/likelihood of a sample.

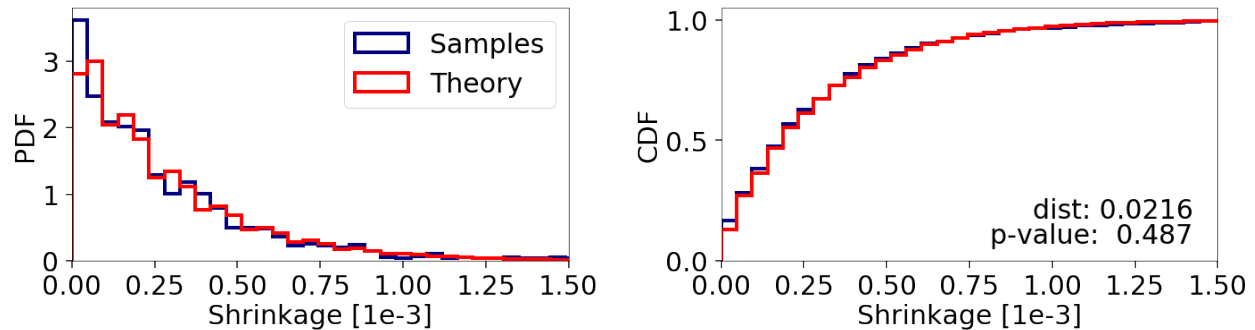
One example of this is the “hyper-pyramid” function from [Buchner \(2014\)](#).



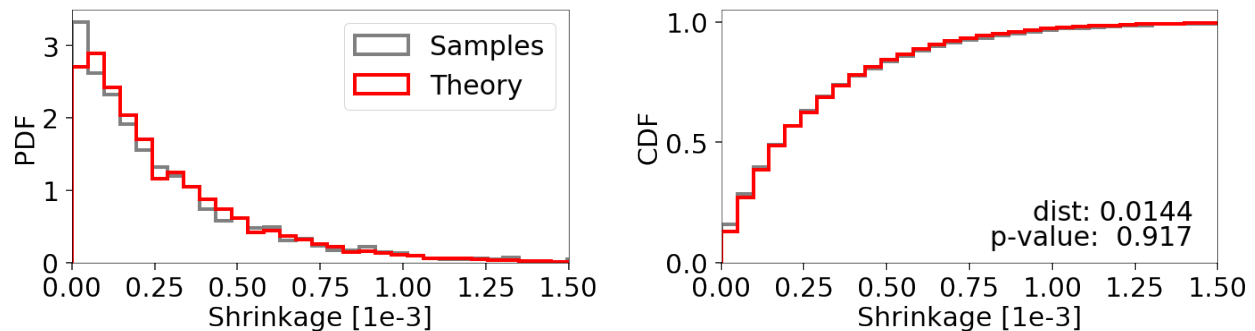
We can compare the set of samples generated from `dynesty` with the expected theoretical shrinkage using a [Kolmogorov-Smirnov \(KS\) Test](#). When sampling uniformly from a set of bounding ellipsoids, we expect to be more sensitive to whether they fully encompass the bounding volume. Indeed, running on default settings in higher dimensions yields shrinkages that are inconsistent with our theoretical expectation (i.e. we shrink too fast):



If bootstrapping is enabled so that ellipsoid expansion factors are determined “on the fly”, we can mitigate this problem:

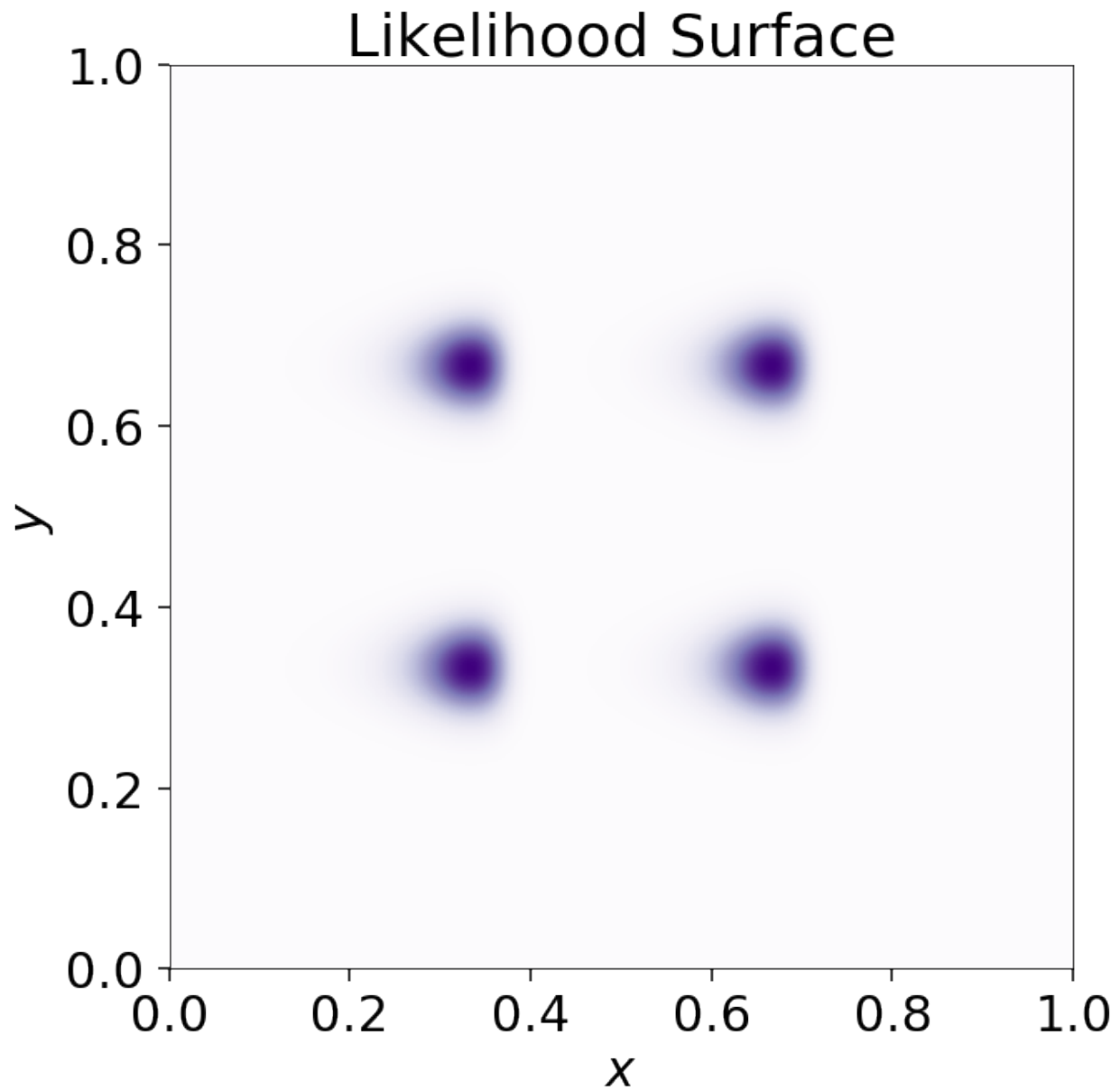


Alternately, using a sampling method other than 'unif' can also avoid this issue by making our proposals less sensitive to the exact size/coverage of the bounding ellipsoids:

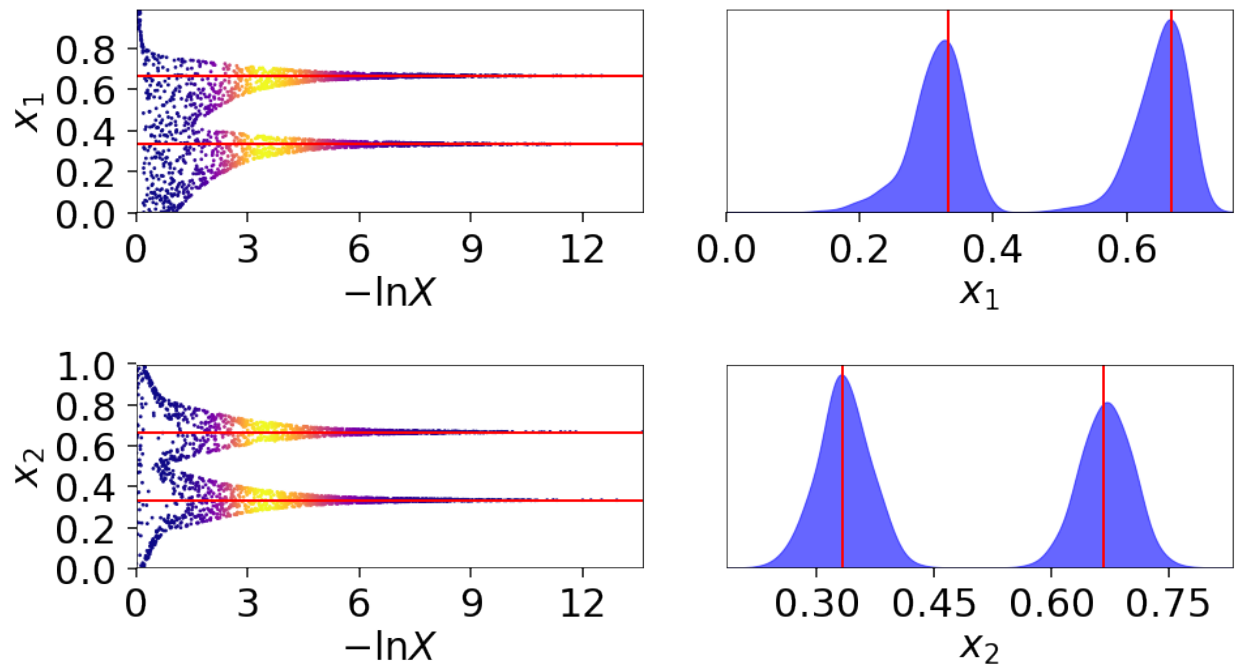


## LogGamma

The multi-modal Log-Gamma distribution is useful for stress testing the effectiveness of bounding distributions since it contains multiple modes coupled with long tails.

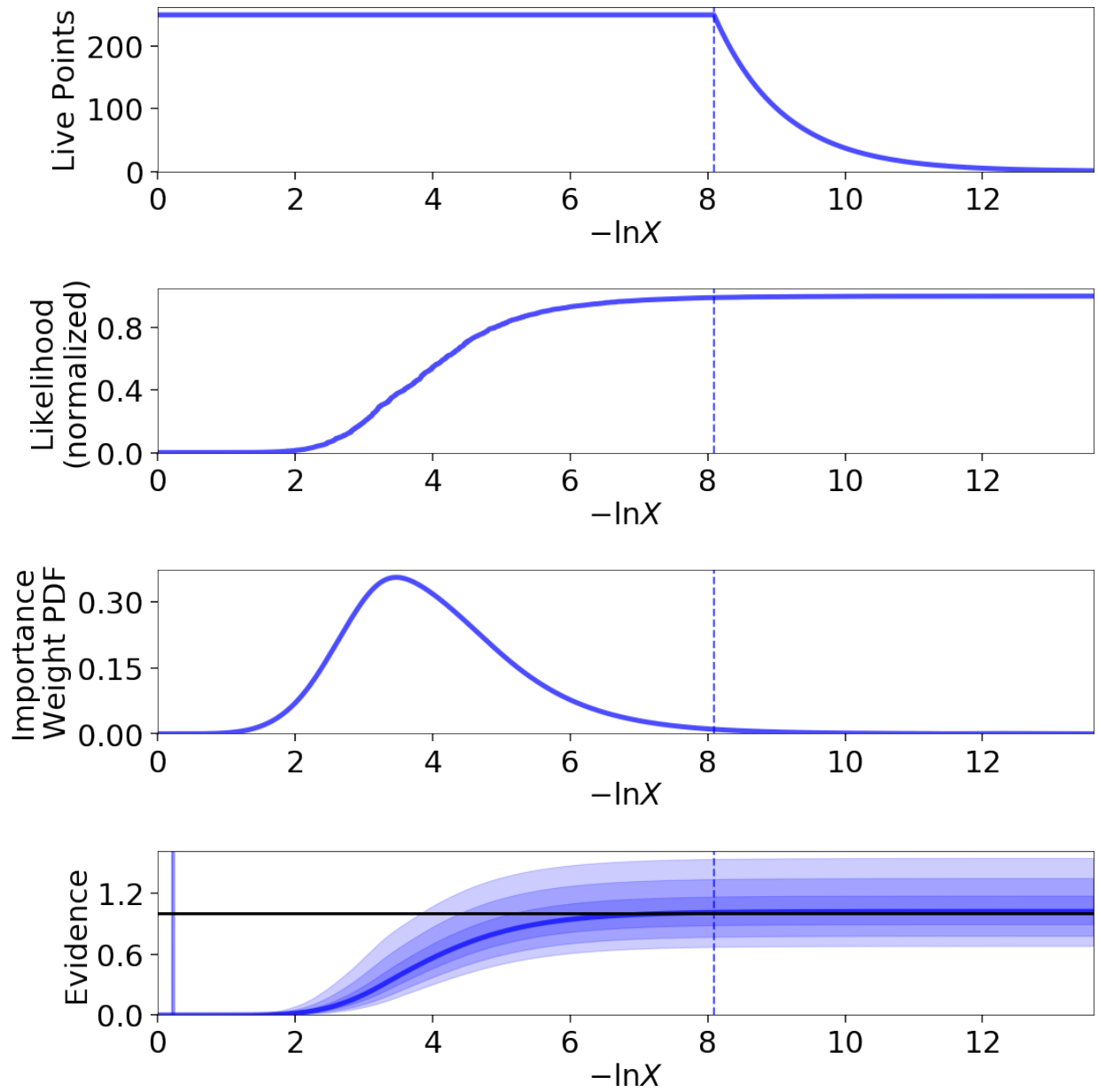


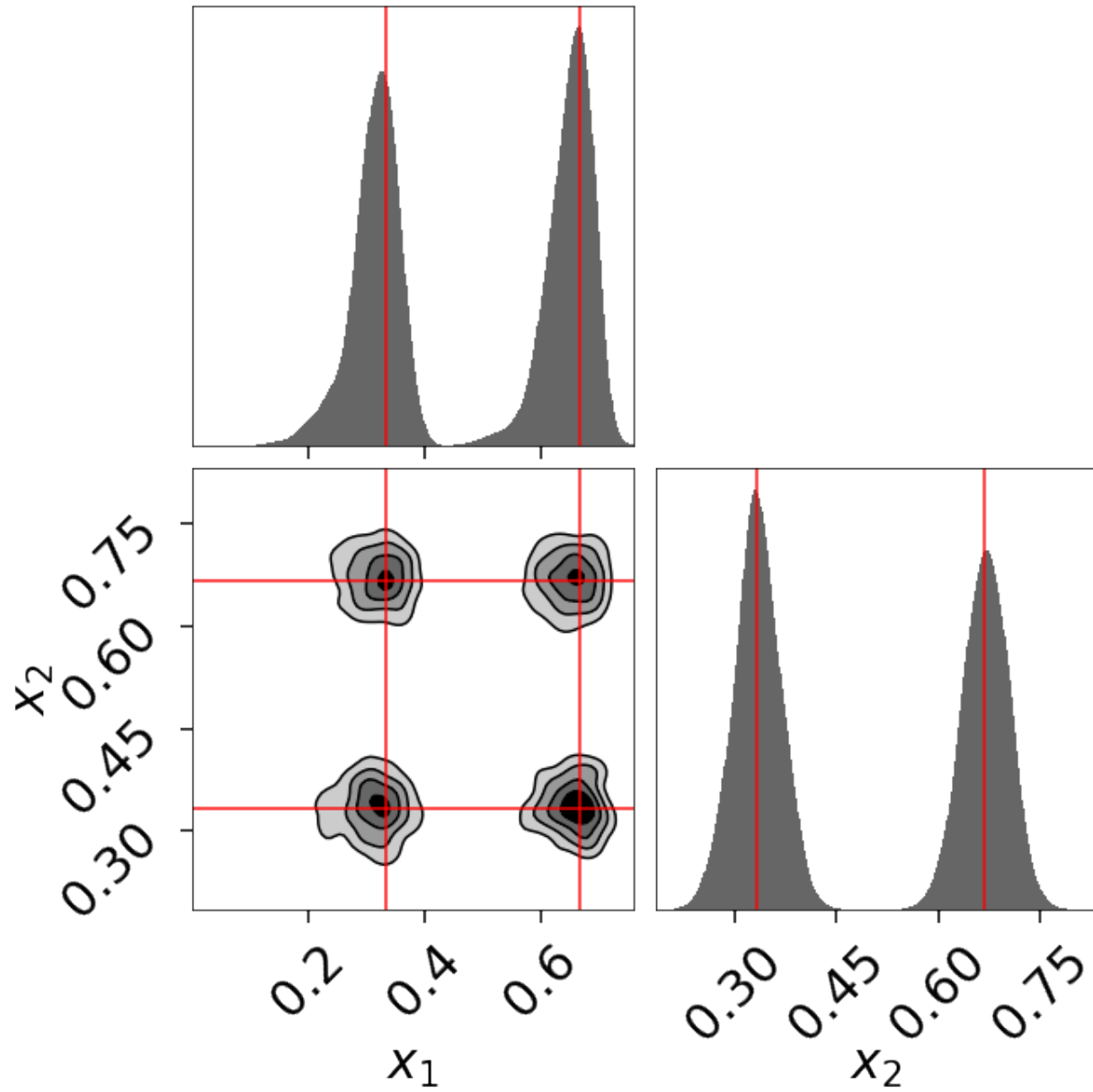
`dynesty` is able to sample from this distribution in  $d = 2$  dimensions without too much difficulty:



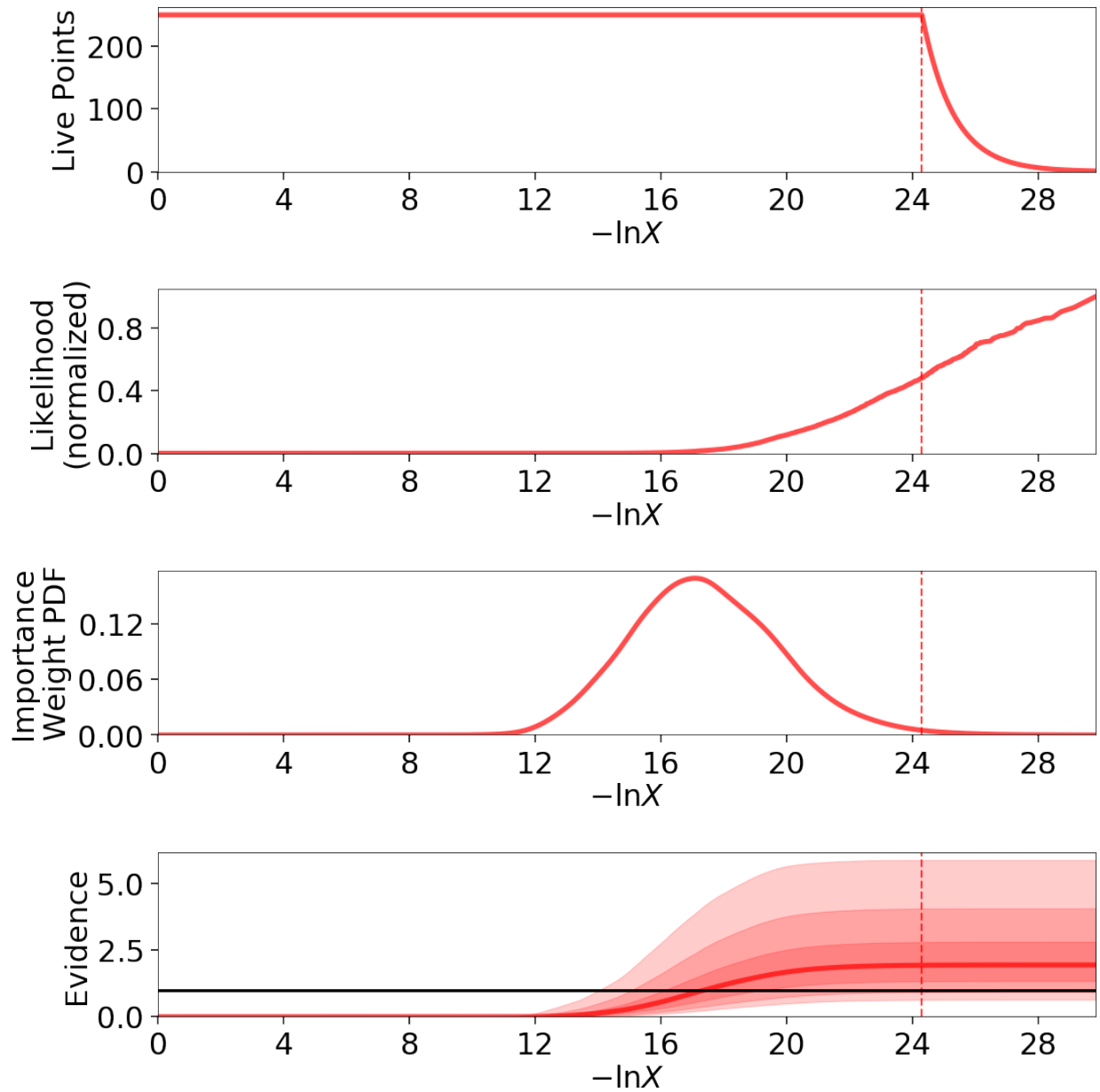
Although the analytic estimate of the evidence error diverges (requiring us to compute it numerically following *Nested Sampling Errors*), we are able to recover the evidence and the shape of the posterior quite well:





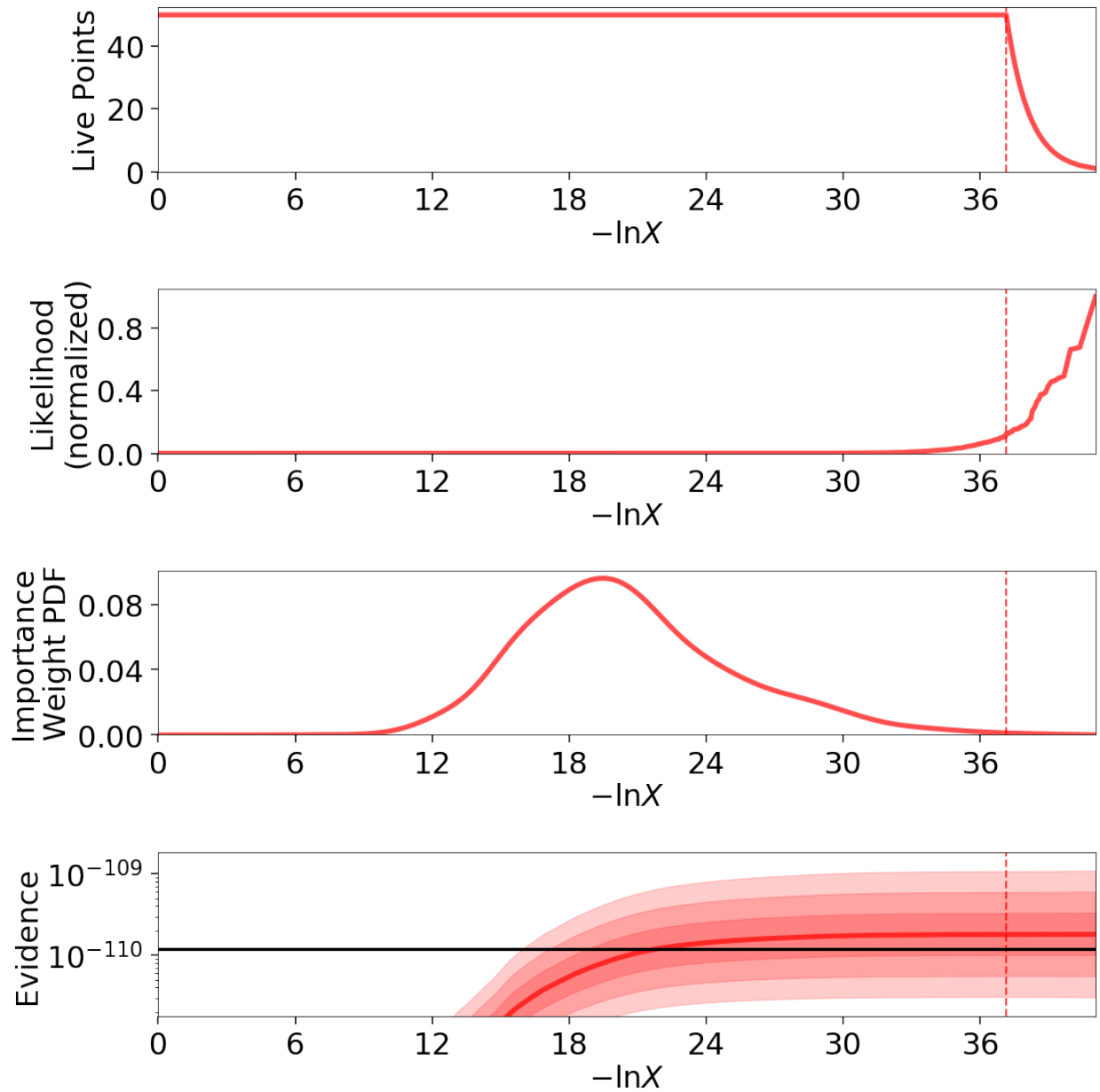


Our results in  $d = 10$  dimensions are also consistent with the expected theoretical value:

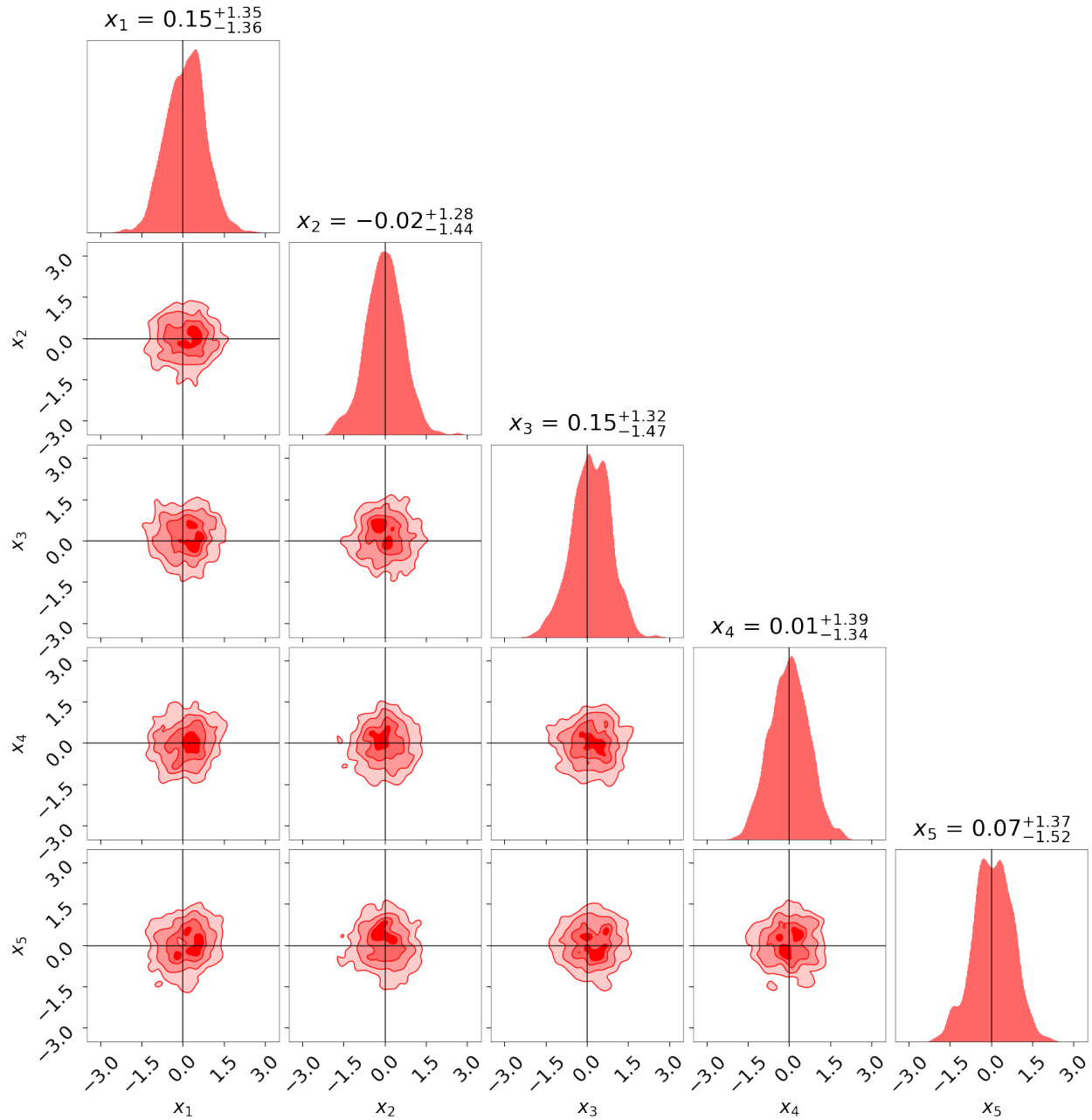


### 200-D Normal

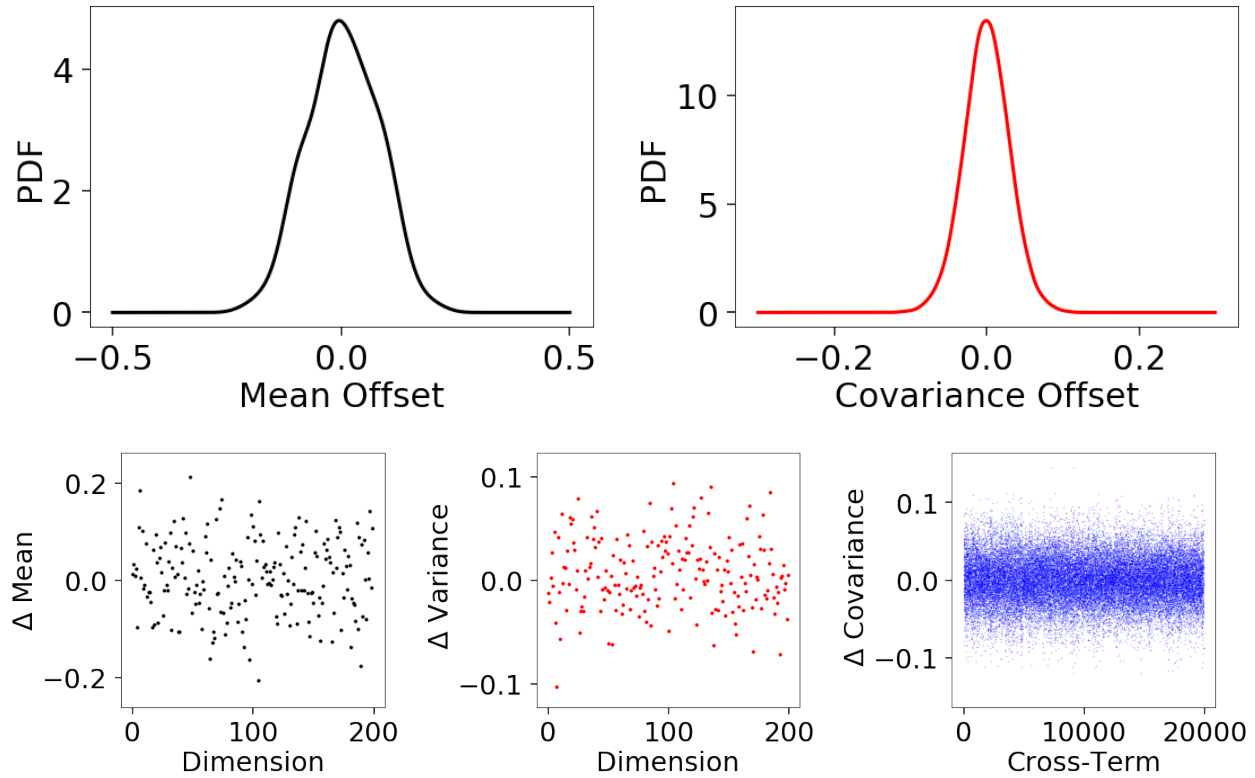
We examine the impact of gradients for sampling from high-dimensional problems using a 200-D iid normal distribution with an associated 200-D iid normal prior. With Hamiltonian slice sampling (`'hslice'`), we find we are able to recover the appropriate evidence:



Our posterior recovery also appears reasonable, as evidenced by the small snapshot below:

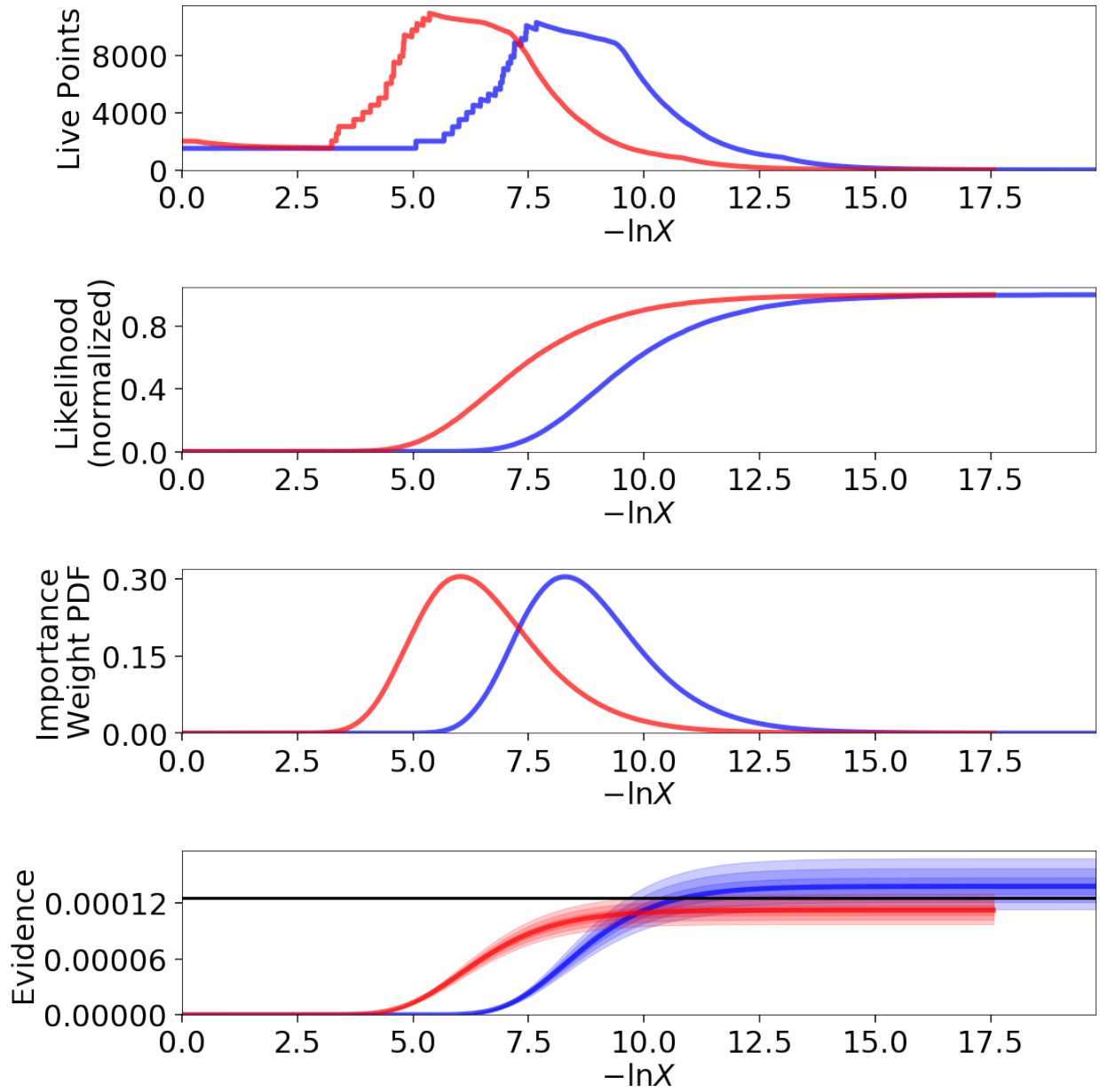


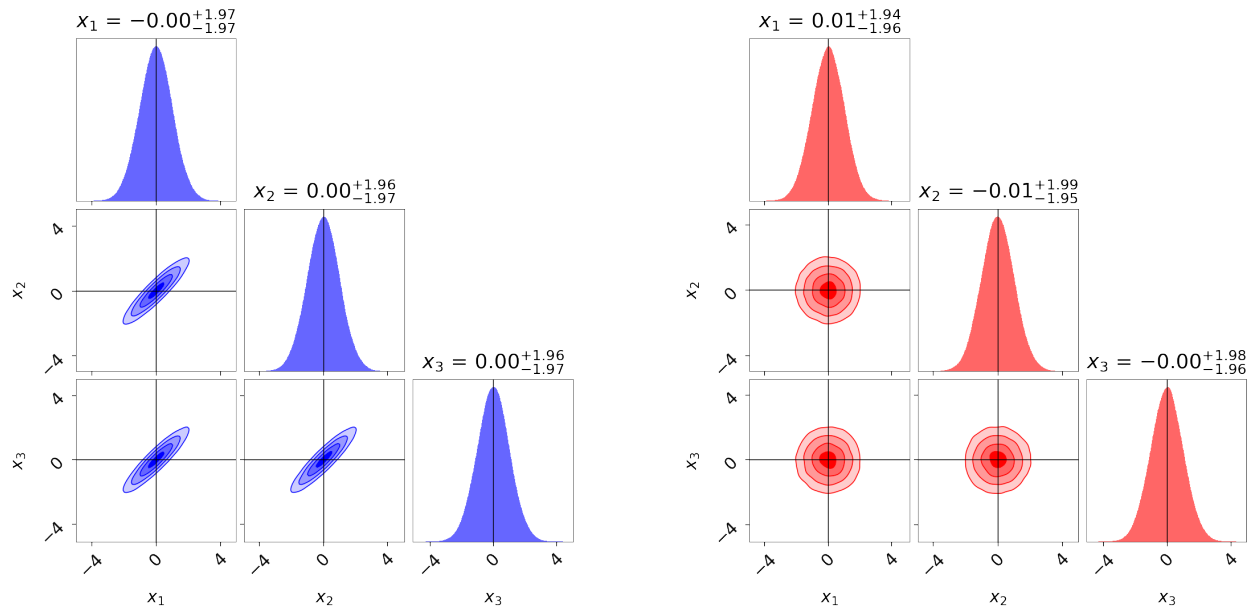
We also find unbiased recovery of the mean and covariances in line with the accuracy we'd expect given the amount of live points used:



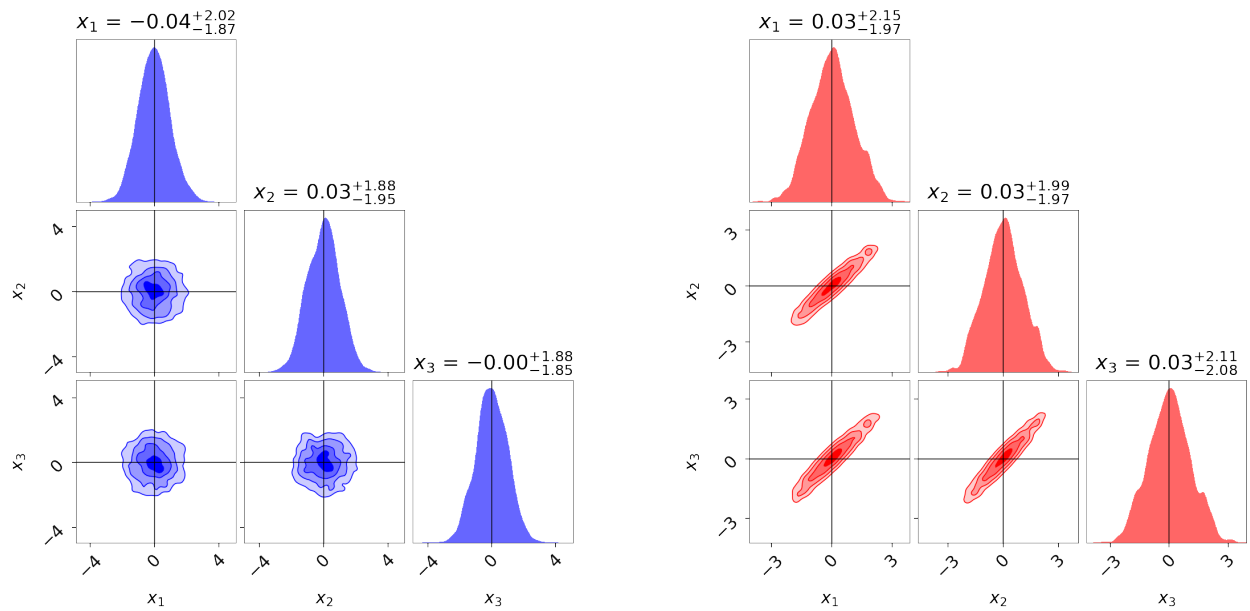
### Importance Reweighting

Nested sampling generates a set of samples and associated importance weights, which can be used to estimate the posterior. As such, it is trivial to re-weight our samples to target a slightly different distribution using **importance reweighting**. To illustrate this, we run `dynesty` on two 3-D multivariate Normal distributions with and without strong covariances.

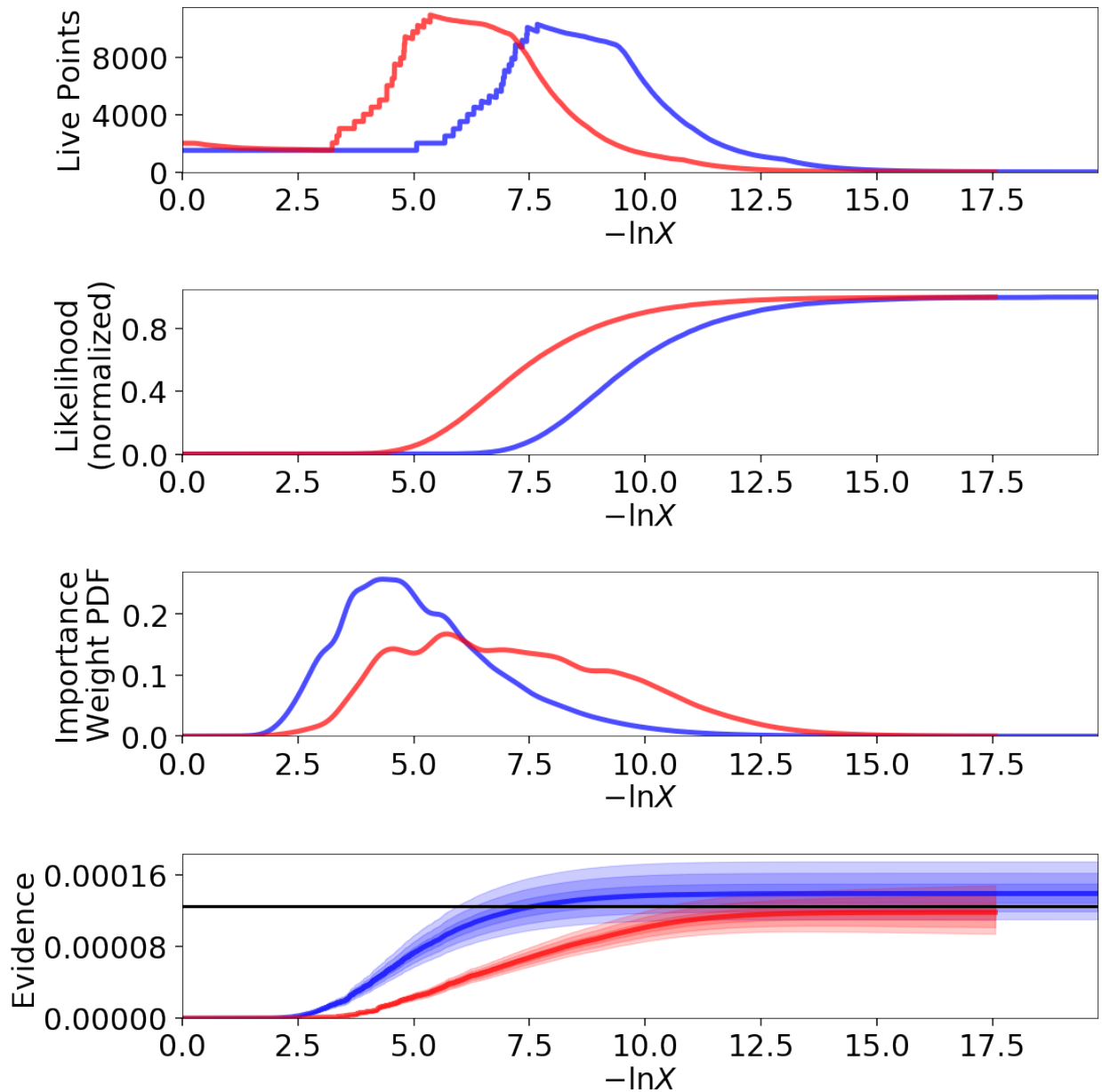




We then use the built-in utilities in `dynesty` to reweight each set of samples to approximate the other distribution. Given that both samples have non-zero coverage over each target distribution, we find that the results are quite reasonable:



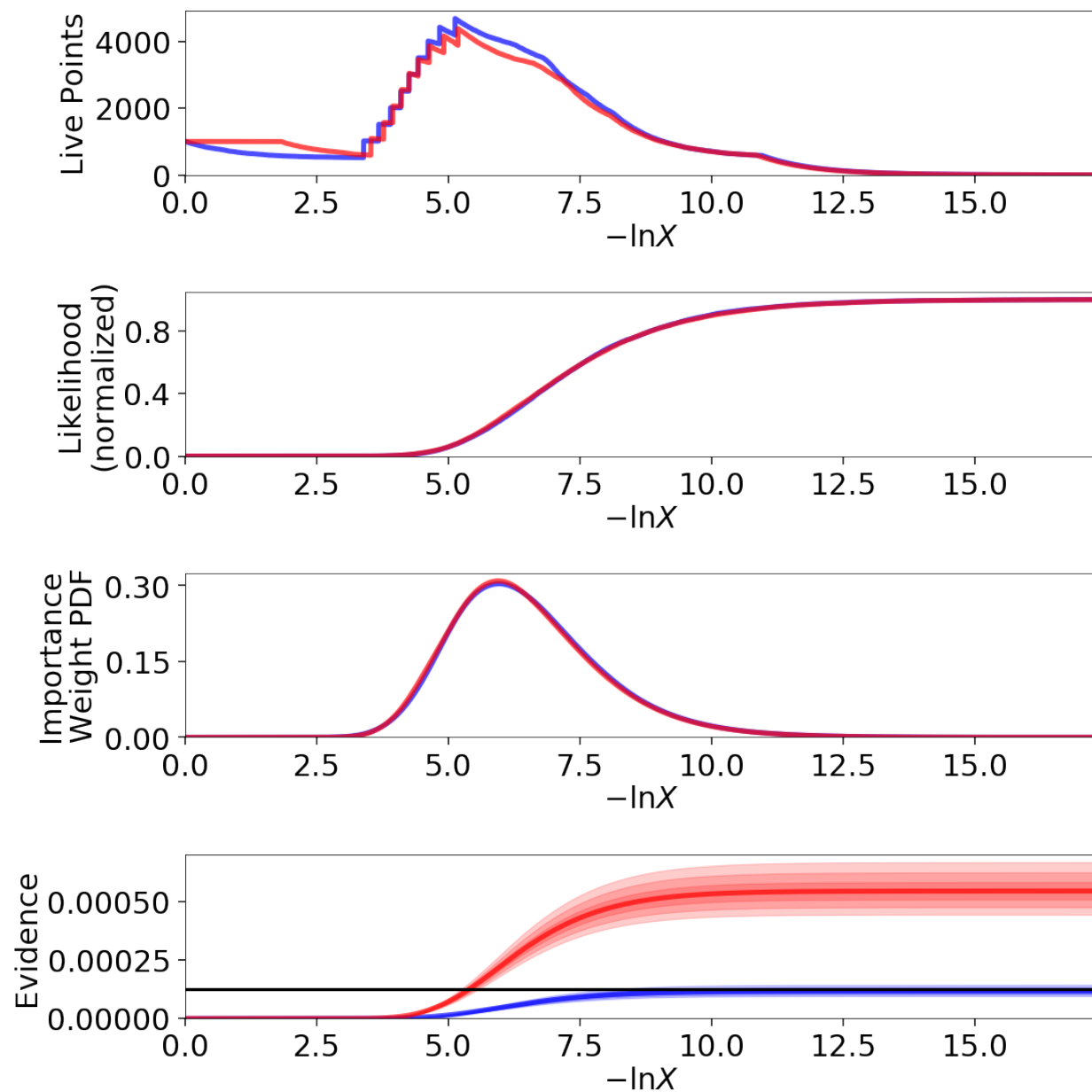


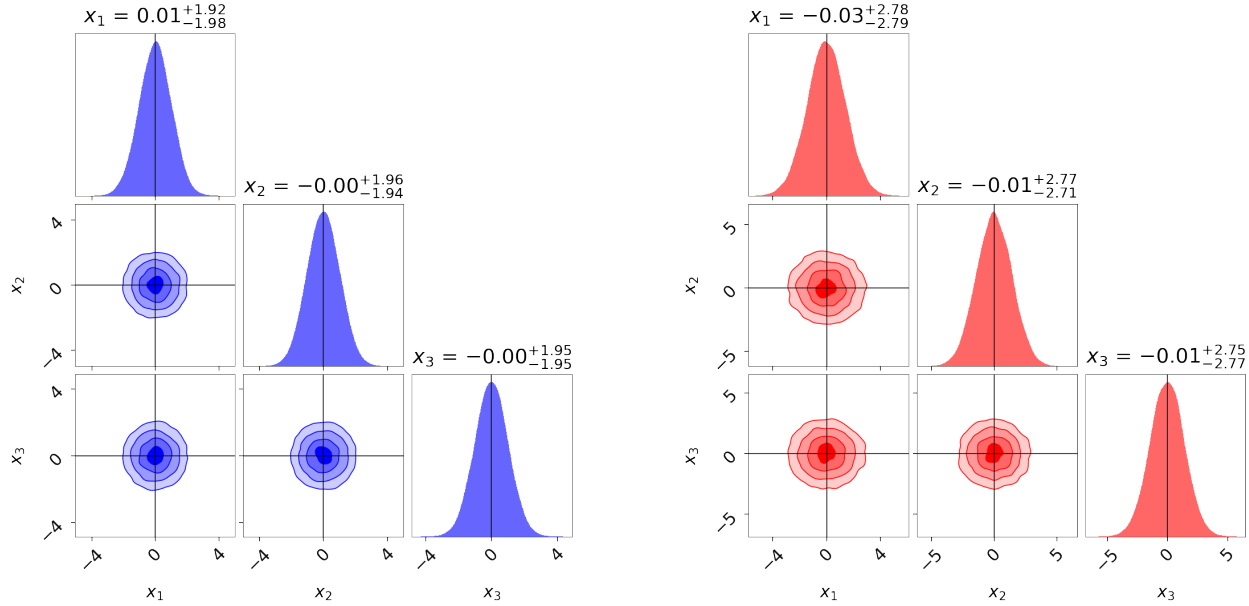


### Noisy Likelihoods

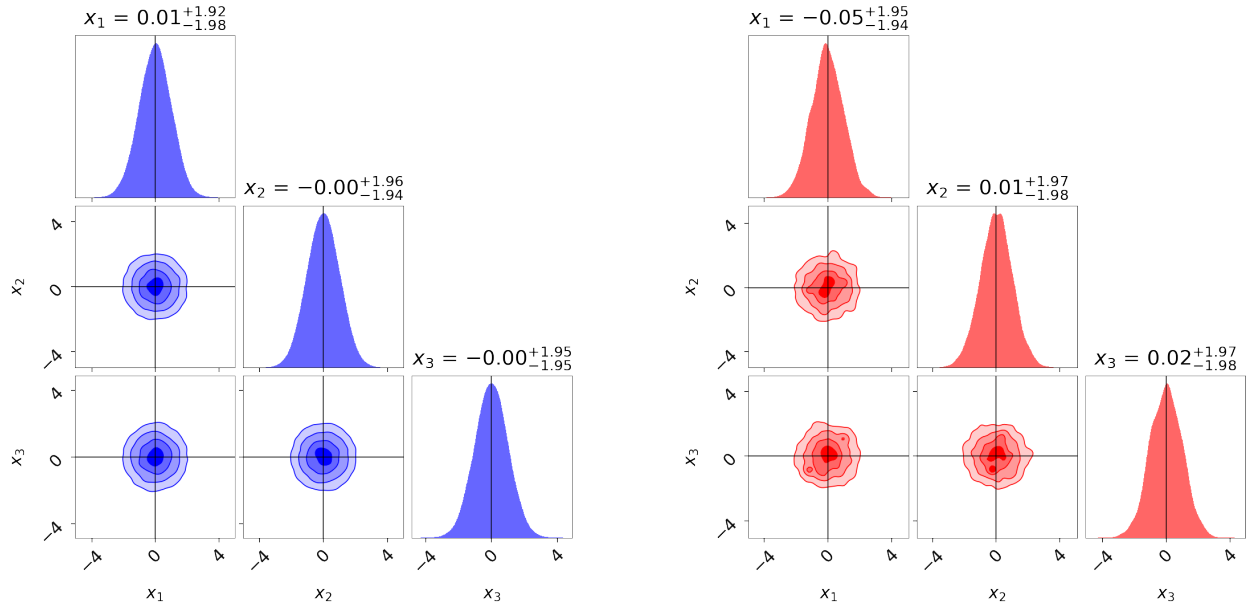
It is possible to sample from noisy likelihoods in `dynesty` just like with MCMC provided they are *unbiased*. While there are additional challenges to sampling from noisy likelihood surfaces, the largest is the fact that over time we expect the likelihoods to be biased high due to the biased impact of random fluctuations on sampling: while fluctuations to lower values get quickly replaced, fluctuations to higher values can only be replaced by fluctuations to higher values elsewhere. This leads to a natural bias that gets “locked in” while sampling, which can substantially broaden the likelihood surface and thus the inferred posterior.

We illustrate this by adding in some random noise to a 3-D iid Normal distribution. While the allocation of samples is almost identical, the estimated evidence is substantially larger and the posterior substantially broader due to the impact of these positive fluctuations.

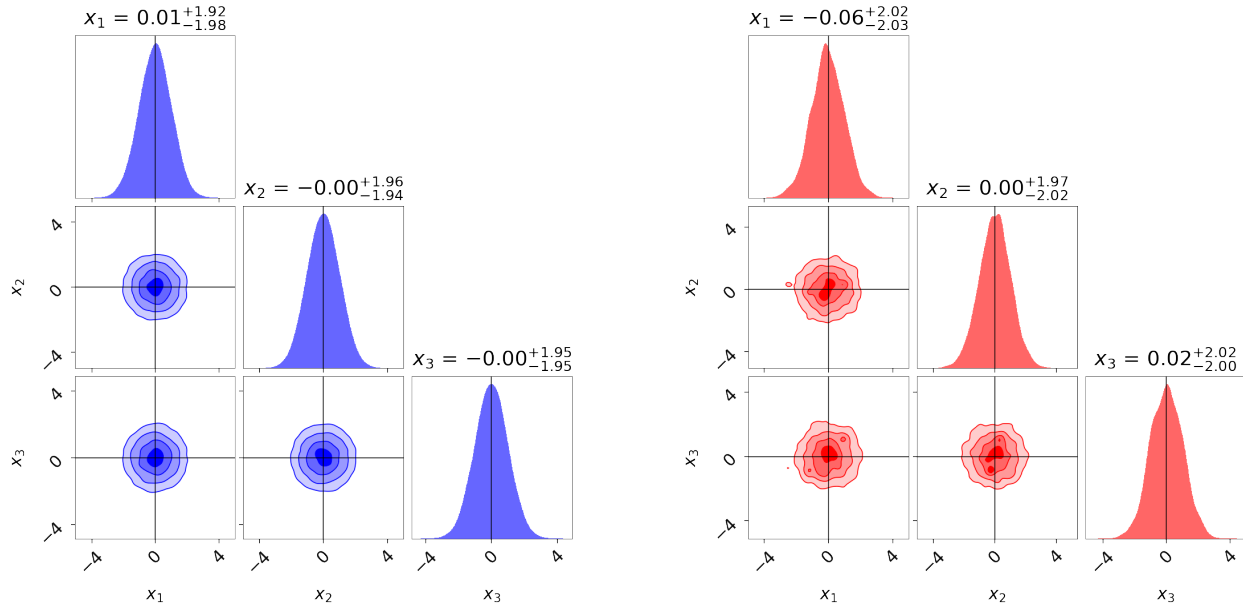




If we know the “true” underlying likelihood, it is straightforward to use *Importance Reweighting* to adjust the distribution to match:



However, in most cases these are not available. In that case, we have to rely on being able to generate multiple realizations of the noisy likelihood at the set of evaluated positions in order to obtain more accurate (but still noisy) estimates of the underlying likelihood. These can then be used to get an estimate of the true distribution through the appropriate importance reweighting scheme:



### 3.14.7 FAQ

This page contains a collection of frequently asked questions from `dynesty` users, along with some answers that hopefully are helpful to you. If you don't see your particular issue addressed here, feel free to [open an issue](#).

For citation information, see the `:ref: 'Citations'` section on the homepage.

#### Sampling Questions

##### What sampling method should I be using?

This is always problem-dependent, but some general advice is to select one based on the dimensionality of your problem. In low dimensions, uniform sampling is often quite efficient since the bounding distributions can often encompass the majority of the prior volume. In moderate dimensions, random walks often can serve as an effective way to propose new points without relying on the exact shape/size of the bounds being correct. In higher dimensions, we generally need non-rejection methods such as slice sampling to generate samples efficiently since the prior volume is so large. Using gradients can also help generate efficient proposals in this regime. `dynesty` uses these rules-of-thumb by default to choose a sampling option with `'auto'`.

##### Sampling seems to freeze around an efficiency of 10%. Is this a bug?

This isn't a bug, but probably just a consequence of the first bounding update. By default, `dynesty` waits to actually start sampling using the proposed sampling/bounding methods passed to the sampler until a set of conditions specified in `first_update` are satisfied. This lets the live points somewhat move away from the edges of the prior and begin to adapt to the shape of the target distribution, which helps to avoid problems such as the bounds “shredding” the live points into lots of tiny islands. The basic heuristic used is to wait until uniform proposals from the prior hit a cumulative efficiency of 10%, but that threshold can be adjusted using the `first_update` argument.

##### Is there an easy way to add more samples to an existing set of results?

Yes! There are actually a bunch of ways to do this. If you have the `NestedSampler` currently initialized, just executing `run_nested()` will start adding samples where you left off. If you're instead interested in adding more samples to a previous part of the run, the best strategy is to just start a new independent run and then “combine” the old and new runs together into a single (improved) run using the `merge_runs()` function.

If you're using the `DynamicNestedSampler`, executing `run_nested` will automatically add more dynamically-allocated samples based on your target weight function as long as the stopping criteria hasn't been met. If you would like to add a new batch of samples manually, running `add_batch` will assign a new set of samples. Finally, `merge_runs()` also works with results generated from Dynamic Nested Sampling, so it is just as easy to set off a new run and combine it with your original result.

### There are `inf` values in my lower/upper log-likelihood bounds! Should I be concerned?

In most cases no. As mentioned in [Running Internally](#), these values are just the lower and upper limits of the log-likelihood used to limit your sampling. If you're sampling starting from the prior, you're starting out from a likelihood of 0 and therefore a log-likelihood of `-inf`. If you haven't specified a particular `logl_max` to terminate sampling, the default value is set to be `+inf` so it will never prematurely terminate sampling. These values can change during Dynamic Nested Sampling, at which point they serve as the endpoints between which a new batch of live points is allocated.

In rare cases, errors in these bounds can be signs of Bad Things that may have happened while sampling. This is often the case if the log-likelihood values being sampled (and displayed) are also non-sensical (e.g., involve `nan` or `inf` values, etc.). In that case, it is often useful to terminate the run early and examine the set of samples to see if there are any possible issues.

### Sometimes while sampling my estimated evidence errors become undefined! Should I be concerned?

Most often this is *not* a cause for concern. As mentioned in [Approximate Evidence Errors](#), `dynesty` uses an approximate method to estimate evidence errors in real time based on the KL divergence ("information gain") and the current number of live points. Sometimes this approximation can lead to improper results (i.e. negative variances), which can often occur early in the run when there is a lot of uncertainty in the prior volume. While this often "corrects" itself later in the run, sometimes the effect can persist. Regardless of whether the approximation converges, however, errors can still be computed using the functions described in [Nested Sampling Errors](#) as normal. I am currently working on developing a more robust approximation that avoids some of these issues.

In rare cases, issues with the evidence error approximation can be a sign that something has gone Terribly Wrong during the sampling phase. This is often the case if the log-likelihood values being output also are non-sensical (e.g., involve `nan` or `inf` values). In that case, it is often useful to terminate the run early and examine the set of samples to see if there are any possible issues.

### When adding batches of live points sometimes the log-likelihoods being displayed don't monotonically increase as I expect. What's going on?

When points are added in each batch, they are allocated randomly between the lower and upper log-likelihood bounds (since they are being sampled randomly). These values are the ones being output to the terminal. Once all the points have been allocated, then nested sampling can begin by replacing each of the lowest log-likelihood values with a better one.

### Sampling is taking much longer than I'd like. What should I do?!

Unfortunately, there's no catch-all solution to this. The most important first step is to make sure you're examining real-time outputs using the `print_progress=True` option (enabled by default) if you're sampling internally using `run_nested()` and printing out progress if sampling externally using, e.g., `sample()`.

If the bounding distribution is updating frequently and you're using more computationally intensive methods such as `'multi'`, some of this might be due to excessive overhead associated with constructing the bounds. This can be reduced by increasing `update_interval`.

If the overall sampling efficiency is low (*relative to what you'd expect*), it might indicate that the distribution used (e.g., `'single'`) isn't effective and more complex ones such as `'multi'` should be used instead. If you're already using those but still getting inefficient proposals, that might indicate that the bounding distribution are struggling to capture the target distribution. This can happen if, e.g., the posterior occupies a thin, strongly-curved manifold in several dimensions, which is hard to model with a series of overlapping ellipsoids or other similar distributions.

Another possible culprit might be the enlargement factors. While the default 25% value usually doesn't significantly

decrease the efficiency, there some exceptions. If you are instead deriving expansion factors from bootstrapping, it's possible you're experiencing severe Monte Carlo noise (see [Bounding Questions](#)). You could try to resolve this by either using more live points or switching to an alternate sampling method less sensitive to the size of the bounding distributions such as `'rwalk'` or `'slice'`.

If sampling progresses efficiently after the first bounding update (i.e. when `bound > 0`) for the majority of the run but becomes substantially less efficient near the final `dlogz` stopping criterion, that could be a sign that the the current set of live points are unable to give rise to bounding distributions that are detailed enough to track the shape of the remaining prior volume. As above, this behavior could be remedied by using more live points or alternate sampling methods. Depending on the goal, the `dlogz` tolerance could also be adjusted.

Finally, if sampling seems to be progressing efficiently but is just taking a long time, it might be because the high-likelihood regions of parameter space are small compared to the prior volume. As discussed in [Role of Priors in Nested Sampling](#), the time it takes to sample to a given `dlogz` tolerance scales as the “information” gained by updating from the prior to the posterior. Since Nested Sampling starts by sampling from the entire prior volume, having overly-broad priors will increase the runtime.

**I noticed that the number of iterations and/or function calls during a run don't exactly match up with the limits I specify using, e.g., `maxiter` or `maxcall`. Is this a bug?**

No, this is not a bug (i.e. this behavior is not unintended). When proposing a new point, `dynesty` currently only checks the stopping criterion specified (whether iterations or function calls) *after* that point has been accepted. This can also happen when using the [DynamicSampler](#) to propose a new batch of points, since the first batch of points need to be allocated before checking the stopping criterion.

**I find other sampling are inefficient relative to ‘unif’. Why would I ever want to use them?**

The main reason these methods are more inefficient than uniform sampling is that they are designed to sample from higher-dimensional (and somewhat more “difficult”) distributions, which is inherently challenging due to the behavior of [Typical Sets](#). Broadly speaking, these methods are actually reasonably efficient when compared to other (non-gradient) sampling methods on similar problems (see, e.g., [here](#)).

In addition, it is also important to keep in mind that samples from `dynesty` are nominally *independent* (i.e. already “thinned”). As a reference point, consider an MCMC algorithm with a sampling efficiency of 20%. While this might seem more efficient than the 4% default target efficiency of `'rwalk'` in `dynesty`, the output samples from MCMC are (by design) correlated. If the resulting MCMC chain needs to be thinned by more than a factor of 5 to ensure independent samples, its “real” sampling efficiency is actually then below the 4% nominally achieved by `dynesty`. This is discussed further in the [release paper](#).

**How many walks (steps) do you need to use for `'rwalk'` ?**

In general, random walk behavior leads to excursions from the mean at a rate that scales as (roughly)  $\sqrt{n}\sigma$  where  $n$  is the number of walks and  $\sigma$  is the typical length scale. The number of steps needed then roughly scales as  $d^2$ . In general this behavior doesn't dominate unless sampling in high ( $d \gtrsim 20$ ) dimensions. In lower dimensions ( $d \lesssim 10$ ), `walks=25` is often sufficient, while in moderate dimensions ( $d \sim 10 - 20$ ) `walks=50` or greater are often necessary to maintain independent samples.

**What are the differences between `'slice'` and PolyChord?**

Our implementation of multivariate slice sampling more closely follows the prescription in [Neal \(2003\)](#) than the algorithm outlined in the [PolyChord](#) paper. We conservatively enforce a strict Gibbs updating scheme that requires sampling from *all* 1-D conditional distributions (in random order); we term this entire update a “slice”. This enables us to rigorously satisfy detailed balance at the cost of being less efficient.

We also treat mode identification and sampling a little differently than PolyChord. In `dynesty` our bounding objects are used to track modes as well as a set of orthogonal basis vectors characterizing that mode. Slicing then takes place along that specific basis, allowing us to sample efficiently even in a multi-modal context. For PolyChord, mode identification works using a slightly different clustering algorithm and sampling takes place in a “pre-whitened” space based on the derived orthogonal basis.

Our implementation of `'rslice'` more closely follows the method employed in PolyChord.

### How many slices (“repeats”) do you need to use for `'slice'` ?

Since slice sampling is a form of non-rejection sampling, the number of “slices” requires for Nested Sampling is (in theory) independent of dimensionality and can remain relatively constant. This is especially true if there are a set of local principle axes that can be effectively captured by the bounding distributions (e.g., `'multi'`). There are more pathological cases, however, where the number of slices can weakly scale with dimensionality. In general we find that the default (and conservative) `slices=5` is robust under a wide variety of circumstances.

### The stopping criterion for Dynamic Nested Sampling is taking a long time to evaluate. Is that normal?

For large numbers of samples with a large number of varying live points, this is normal. Every new particle increases the complexity of simulating the errors used in the stopping criterion (see [Nested Sampling Errors](#)), so the time required tends to scale with the number of batches added. This is especially true if the “full” live point simulation is being used (via the `error = 'simulate'` argument) rather than the approximation enabled by default (`error = 'sim_approx'`).

### I’m trying to sample using gradients but getting extremely poor performance. I thought gradients were supposed to make sampling more efficient! What gives?

While gradients are extremely useful in terms of substantially improving the scaling of most sampling methods with dimensionality (gradient-based methods have better polynomial scaling than non-gradient slice sampling, both of which are *substantially* better over the runaway exponential scaling of random walks), it can take a while for these benefits to really kick in. These scaling arguments generally ignore the constant prefactor, which can be quite large for many gradient-based approaches that require integrating along some trajectory, often resulting in (at least) dozens of function calls per sample. This often makes it more efficient to run simpler sampling techniques on lower-dimensional problems.

If you feel like your performance is poorer than expected even given this, or if you notice other results that make you highly suspicious of the resulting samples, please double-check the [Sampling with Gradients](#) page to make sure you’ve passed in the correct log-likelihood gradient and are dealing with the unit cube Jacobian properly. Failing to apply this (or applying it twice) violates conservation of energy and momentum and leads to the integration timesteps along the trajectories changing in undesirable ways. It’s also possible the numerical errors in the Jacobian (if you’ve set `compute_jac=True`) might be propagating through to the computed trajectories. If so, consider trying to compute the analytic Jacobian by hand to reduce the impact of numerical errors.

If you still find subpar performance, please feel free to [open an issue](#).

## Live Point Questions

### How many live points should I use?

Short answer: **it depends**.

Longer answer: Unfortunately, there’s no easy answer here. Increasing the number of live points helps establish more flexible and robust bounds, improving the overall sampling efficiency and prior volume resolution. However, it simultaneously increases the runtime. These competing behaviors mean that compromises need to be made which are problem-dependent.

In general, for ellipsoid-based bounds an absolute minimum of  $\text{ndim} + 1$  live points is “required”, with  $2 * \text{ndim}$  being a (roughly) “safe” threshold. If bootstraps are used to establish bounds while sampling uniformly, however, many (many) more live points should be used. Around  $50 * \text{ndim}$  points are recommended *for each expected mode*.

Methods that do not depend on the absolute size of the bounds (but instead rely on their shape) can use fewer live points. Their main restriction is that new live point proposals (which “evolve” a copy of an existing live point to a new position) must be independent of their starting point. Using too few points can require excessive thinning, which quickly negates the benefit of using fewer points if speed is an issue.  $10 * \text{ndim}$  per mode seems to work reasonably well, although this depends sensitively on the amount of prior volume that has to be traversed: if the likelihood is a



set of tiny islands in an ocean of prior volume, then you'll need to use more live points to avoid missing them. See [LogGamma](#), [Eggbox](#), or [Exponential Wave](#) for some examples of this in practice.

## Bounding Questions

### What bounds should I be using?

Generally, `'multi'` (multiple ellipsoid decomposition) is the most adaptive, being able to model a wide variety of behaviors and complex distributions. It is enabled in `dynesty` by default.

For simple unimodal problems, `'single'` (a single bounding ellipsoid) can often do quite well. It also helps to guard against cases where methods like `'multi'` can accidentally “shred” the posterior into many pieces if the ellipsoid decompositions are too aggressive.

For low-dimensional problems, ensemble methods like `'balls'` and `'cubes'` can be quite effective by allowing live points themselves to create “emergent” structure. These can create more flexible shapes than `'multi'`, although they have trouble modeling separate structures with wildly different shapes.

In almost all cases, using no bound (`'none'`) should be seen as a fallback option. It is mostly useful for systematics checks or in cases where the number of live points is small relative to the number of dimensions.

### What are the differences between `'multi'` and MultiNest?

The multi-ellipsoid decomposition/bounding method implemented in `dynesty` is entirely based on the algorithm implemented in `nestle` which itself is based on the algorithm *described* in [Feroz, Hobson & Bridges \(2009\)](#). As such, it doesn't include any improvements, changes, etc. that may or may not be included in `MultiNest`.

In addition, there are a few differences in the portion of the algorithm that decides when to split an ellipsoid into multiple ellipsoids. As with `nestle`, the implementation in `dynesty` is more conservative about splitting ellipsoids to avoid over-constraining the remaining prior volume and also enlarges all the resulting ellipsoids by a constant volume prefactor. In general this results in a slightly lower sampling efficiency but greater overall robustness. These defaults can be changed through the [Top-Level Interface](#) via the `enlarge`, `vol_dec` and `vol_check` keywords if you would like to experiment with more conservative/aggressive behavior.

`dynesty` also uses different heuristics than `MultiNest` when deciding, e.g., when to first construct bounds. See [Bounding Options](#) for additional details.

### No matter what bounds, options, etc. I pick, the initial samples all come from ‘bound = 0’ and continue until the overall efficiency is quite low. What’s going on here?

By default, `dynesty` opts to wait until some time has passed until constructing the first bounding distribution. This behavior is designed to avoid constructing overly large bounds that often significantly exceed the confines of the unit cube, which can lead to excessive time spent generating random numbers early in a given run. Prior to constructing the initial bound, samples are proposed from the unit cube, which is taken to be `bound = 0`. The options that control these heuristics can be modified using the `first_update` argument.

### During a run I sometimes see the bound index jump forward several places. Is this normal?

To avoid getting stuck sampling from bad bounding distributions (see above), `dynesty` automatically triggers a bounding update whenever the number of likelihood calls exceeds `update_interval` while sampling from a particular bound. This can lead to multiple bounds being constructed before the sample is accepted.

### A constant expansion factor seems arbitrary and I want to try out bootstrapping. How many bootstrap realizations do I need?

Sec. 6.1 of [Buchner \(2014\)](#) discusses the basic behavior of bootstrapping and how many iterations are needed to ensure that realizations do not include the same live point over some number of realizations. `bootstrap = 20` appears to work well in practice, although this is more aggressive than the `bootstrap = 50` recommended by Buchner.



**When bootstrapping is on, sometimes during a run a bound will be really large. This then leads to a large number of log-likelihood calls before the bound shrinks back to a reasonable size again. Why is this happening? Is this a bug?**

This isn't (technically) a bug, but rather Monte Carlo noise associated with the bootstrapping process. Depending on the chosen method, sometimes bounds can be unstable, leading to large variations between bootstraps and subsequently large expansions factors. Some of this is explored in the [Gaussian Shells](#) and [Hyper-Pyramid](#) examples. In general, this is a sign that you don't have enough live points to robustly determine your log-likelihood bounds at a given iteration, and should likely be running with more. Note that "robustly" is the key word here, since it can often take a (some might find "excessively") large number of live points to confidently determine that you aren't missing any hidden prior volume.

## Pool Questions

**My provided pool is crashing. What do I do?**

First, check that all relevant variables, functions, etc. are properly accessible and that the `pool.map` function is working as intended. Sometimes pools can have issues passing variables to/from members or executing tasks (a)synchronously depending on the setup.

Second, check if your pool has issues pickling some types of functions or evaluating some of the functions in [sampling](#). In general, nested functions require more advanced pickling (e.g., `dill`), which is not enabled with some pools by default.

If those quick fixes don't work, feel free to raise an issue. However, as multi-threading and multi-processing are notoriously difficult to debug, especially on a problem I'm not familiar with, it's likely that I might not be able to help all that much.

## 3.14.8 References and Acknowledgements

The release paper describing the code can be found [here](#).

### Code

`dynesty` is the spiritual successor to Nested Sampling package [nestle](#) and has benefited enormously from the work put in by [Kyle Barbary](#) and [other contributors](#).

Much of the API is inspired by the ensemble MCMC package [emcee](#) as well as other work by [Daniel Foreman-Mackey](#).

Many of the plotting utilities draw heavily upon Daniel Foreman-Mackey's wonderful [corner](#) package.

Several other plotting utilities as well as the real-time status outputs are inspired in part by features available in the statistical modeling package [PyMC3](#).

### Papers and Texts

The dynamic sampling framework was entirely inspired by:

[Higson et al. 2017b](#). *Dynamic nested sampling: an improved algorithm for parameter estimation and evidence calculation*. ArXiv e-prints, 1704.03459.

Much of the nested sampling error analysis is based on:

[Higson et al. 2017a](#). *Sampling errors in nested sampling parameter estimation*. ArXiv e-prints, 1703.09701.

[Chopin & Robert 2010](#). *Properties of Nested Sampling*. Biometrika, 97, 741.

The nested sampling algorithms in `RadFriendsSampler` and `SupFriendsSampler` are based on:

Buchner 2016. *A statistical test for Nested Sampling algorithms*. Statistics and Computing, 26, 383.

Slice sampling and its implementations in nested sampling are based on:

Handley, Hobson & Lasenby 2015b. *POLYCHORD: next-generation nested sampling*. MNRAS, 453, 4384.

Handley, Hobson & Lasenby 2015a. *POLYCHORD: nested sampling for cosmology*. MNRASL, 450, L61.

Neal 2003. *Slice sampling*. Ann. Statist., 31, 705.

The implementation of multi-ellipsoidal decomposition are based in part on:

Feroz et al. 2013. *Importance Nested Sampling and the MultiNest Algorithm*. ArXiv e-prints, 1306.2144.

Feroz, Hobson & Bridges 2009. *MultiNest: an efficient and robust Bayesian inference tool for cosmology and particle physics*. MNRAS, 398, 1601.

Several useful reference texts include:

Salomone et al. 2018. *Unbiased and Consistent Nested Sampling via Sequential Monte Carlo*. ArXiv e-prints, 1805.03924.

Walter 2015. *Point Process-based Monte Carlo estimation*. ArXiv e-prints, 1412.6368.

Shaw, Bridges & Hobson 2007. *Efficient Bayesian inference for multimodal problems in cosmology*. MNRAS, 378, 1365.

Mukherjee, Parkinson & Liddle 2006. *A Nested sampling algorithm for cosmological model selection*. ApJ, 638, L51.

Silvia & Skilling 2006. *Data Analysis: A Bayesian Tutorial, 2nd Edition*. Oxford University Press.

Skilling 2006. *Nested sampling for general Bayesian computation*. Bayesian Anal., 1, 833.

Skilling 2004. *Nested Sampling*. In Maximum entropy and Bayesian methods in science and engineering (ed. G. Erickson, J.T. Rychert, C.R. Smith). AIP Conf. Proc., 735, 395.

### 3.14.9 API

This page details the methods and classes provided by the `dynesty` module.

#### Top-Level Interface

The top-level interface (defined natively upon initialization) that provides access to the two main sampler “super-classes” via `NestedSampler()` and `DynamicNestedSampler()`.

```
dynesty.dynesty.NestedSampler(loglikelihood, prior_transform, ndim, nlive=500, bound='multi',
                               sample='auto', periodic=None, reflective=None, update_interval=None,
                               first_update=None, npdim=None, rstate=None, queue_size=None, pool=None,
                               use_pool=None, live_points=None, logl_args=None, logl_kwargs=None, ptform_args=None,
                               ptform_kwargs=None, gradient=None, grad_args=None, grad_kwargs=None,
                               compute_jac=False, enlarge=None, bootstrap=0, vol_dec=0.5, vol_check=2.0,
                               walks=25, facc=0.5, slices=5, fmove=0.9, max_move=100, **kwargs)
```

Initializes and returns a sampler object for Static Nested Sampling.

## Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function translating a unit cube to the parameter space according to the prior. The input is a 1-d `numpy` array with length `ndim`, where each value is in the range  $[0, 1)$ . The return value should also be a 1-d `numpy` array with length `ndim`, where each value is a parameter. The return value is passed to the `loglikelihood` function. For example, for a 2 parameter model with flat priors in the range  $[0, 2)$ , the function would be:

```
def prior_transform(u):
    return 2.0 * u
```

**ndim** [int] Number of parameters returned by `prior_transform` and accepted by `loglikelihood`.

**nlive** [int, optional] Number of “live” points. Larger numbers result in a more finely sampled posterior (more accurate evidence), but also a larger number of iterations required to converge. Default is 500.

**bound** [{`'none'`, `'single'`, `'multi'`, `'balls'`, `'cubes'`}, optional] Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points. Choices are no bound (`'none'`), a single bounding ellipsoid (`'single'`), multiple bounding ellipsoids (`'multi'`), balls centered on each live point (`'balls'`), and cubes centered on each live point (`'cubes'`). Default is `'multi'`.

**sample** [{`'auto'`, `'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds. Unique methods available are: uniform sampling within the bounds (`'unif'`), random walks with fixed proposals (`'rwalk'`), random walks with variable (“staggering”) proposals (`'rstagger'`), multivariate slice sampling along preferred orientations (`'slice'`), “random” slice sampling along all orientations (`'rslice'`), and “Hamiltonian” slices along random trajectories (`'hslice'`). `'auto'` selects the sampling method based on the dimensionality of the problem (from `ndim`). When `ndim < 10`, this defaults to `'unif'`. When  $10 \leq \text{ndim} \leq 20$ , this defaults to `'rwalk'`. When `ndim > 20`, this defaults to `'hslice'` if a gradient is provided and `'slice'` otherwise. `'rstagger'` and `'rslice'` are provided as alternatives for `'rwalk'` and `'slice'`, respectively. Default is `'auto'`.

**periodic** [iterable, optional] A list of indices for parameters with periodic boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may wrap around the edge. Default is `None` (i.e. no periodic boundary conditions).

**reflective** [iterable, optional] A list of indices for parameters with reflective boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may reflect at the edge. Default is `None` (i.e. no reflective boundary conditions).

**update\_interval** [int or float, optional] If an integer is passed, only update the proposal distribution every `update_interval`-th likelihood call. If a float is passed, update the proposal after every `round(update_interval * nlive)`-th likelihood call. Larger update intervals larger can be more efficient when the likelihood function is quick to evaluate. Default behavior is to target a roughly constant change in prior volume, with 1.

5 for 'unif',  $0.15 * \text{walks}$  for 'rwalk' and 'rstagger',  $0.9 * \text{ndim} * \text{slices}$  for 'slice',  $2.0 * \text{slices}$  for 'rslice', and  $25.0 * \text{slices}$  for 'hslice'.

**first\_update** [dict, optional] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube ('none') to the one specified by `sample`. Options are the minimum number of likelihood calls ('min\_ncall') and the minimum allowed overall efficiency in percent ('min\_eff'). Defaults are  $2 * \text{nlive}$  and 10., respectively.

**npdim** [int, optional] Number of parameters accepted by `prior_transform`. This might differ from `ndim` in the case where a parameter of loglikelihood is dependent upon multiple independently distributed parameters, some of which may be nuisance parameters.

**rstate** [`RandomState`, optional]

**RandomState instance.** If not given, the global random state of the `random` module will be used.

**queue\_size** [int, optional] Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) `queue_size` many threads. Each thread independently proposes new live points until the proposal distribution is updated. If no value is passed, this defaults to `pool.size` (if a `pool` has been provided) and 1 otherwise (no parallelism).

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where a pool should be used to execute operations in parallel. These govern whether `prior_transform` is executed in parallel during initialization ('prior\_transform'), loglikelihood is executed in parallel during initialization ('loglikelihood'), live points are proposed in parallel during a run ('propose\_point'), and bounding distributions are updated in parallel during a run ('update\_bound'). Default is `True` for all options.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] A set of live points used to initialize the nested sampling run. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods. By default, if these are not provided the initial set of live points will be drawn uniformly from the unit `npdim`-cube. **WARNING: It is crucial that the initial set of live points have been sampled from the prior. Failure to provide a set of valid live points will result in incorrect results.**

**logl\_args** [iterable, optional] Additional arguments that can be passed to loglikelihood.

**logl\_kwargs** [dict, optional] Additional keyword arguments that can be passed to loglikelihood.

**ptform\_args** [iterable, optional] Additional arguments that can be passed to `prior_transform`.

**ptform\_kwargs** [dict, optional] Additional keyword arguments that can be passed to `prior_transform`.

**gradient** [function, optional] A function which returns the gradient corresponding to the provided loglikelihood *with respect to the unit cube*. If provided, this will be used when computing reflections when sampling with 'hslice'. If not provided, gradients are approximated numerically using 2-sided differencing.

**grad\_args** [iterable, optional] Additional arguments that can be passed to `gradient`.

**grad\_kwargs** [dict, optional] Additional keyword arguments that can be passed to `gradient`.

**compute\_jac** [bool, optional] Whether to compute and apply the Jacobian  $dv/du$  from the target space  $v$  to the unit cube  $u$  when evaluating the gradient. If `False`, the gradient provided is assumed to be already defined with respect to the unit cube. If `True`, the gradient provided is assumed to be defined with respect to the target space so the Jacobian needs to be numerically computed and applied. Default is `False`.

**enlarge** [float, optional] Enlarge the volumes of the specified bounding object(s) by this fraction. The preferred method is to determine this organically using bootstrapping. If `bootstrap > 0`, this defaults to 1.0. If `bootstrap = 0`, this instead defaults to 1.25.

**bootstrap** [int, optional] Compute this many bootstrapped realizations of the bounding objects. Use the maximum distance found to the set of points left out during each iteration to enlarge the resulting volumes. Can lead to unstable bounding ellipsoids. Default is 0 (no bootstrap).

**vol\_dec** [float, optional] For the 'multi' bounding option, the required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is 0.5.

**vol\_check** [float, optional] For the 'multi' bounding option, the factor used when checking if the volume of the original bounding ellipsoid is large enough to warrant  $> 2$  splits via `ell.vol > vol_check * nlive * pointvol`. Default is 2.0.

**walks** [int, optional] For the 'rwalk' sampling option, the minimum number of steps (minimum 2) before proposing a new live point. Default is 25.

**facc** [float, optional] The target acceptance fraction for the 'rwalk' sampling option. Default is 0.5. Bounded to be between `[1. / walks, 1.]`.

**slices** [int, optional] For the 'slice', 'rslice', and 'hslice' sampling options, the number of times to execute a “slice update” before proposing a new live point. Default is 5. Note that 'slice' cycles through **all dimensions** when executing a “slice update”.

**fmove** [float, optional] The target fraction of samples that are proposed along a trajectory (i.e. not reflecting) for the 'hslice' sampling option. Default is 0.9.

**max\_move** [int, optional] The maximum number of timesteps allowed for 'hslice' per proposal forwards and backwards in time. Default is 100.

## Returns

**sampler** [sampler from [nestedsamplers](#)] An initialized instance of the chosen sampler specified via `bound`.

```
dynesty.dynesty.DynamicNestedSampler(loglikelihood, prior_transform, ndim, bound='multi',
                                     sample='auto', periodic=None, reflective=None, update_interval=None, first_update=None, npdim=None,
                                     rstate=None, queue_size=None, pool=None, use_pool=None, logl_args=None, logl_kwargs=None,
                                     ptform_args=None, ptform_kwargs=None, gradient=None, grad_args=None, grad_kwargs=None,
                                     compute_jac=False, enlarge=None, bootstrap=0, vol_dec=0.5, vol_check=2.0, walks=25, facc=0.5,
                                     slices=5, fmove=0.9, max_move=100, **kwargs)
```

Initializes and returns a sampler object for Dynamic Nested Sampling.

## Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function translating a unit cube to the parameter space according to the prior. The input is a 1-d `numpy` array with length `ndim`, where each value is in the

range [0, 1). The return value should also be a 1-d `numpy` array with length `ndim`, where each value is a parameter. The return value is passed to the loglikelihood function. For example, for a 2 parameter model with flat priors in the range [0, 2), the function would be:

```
def prior_transform(u):  
    return 2.0 * u
```

**ndim** [int] Number of parameters returned by `prior_transform` and accepted by `loglikelihood`.

**bound** [['none', 'single', 'multi', 'balls', 'cubes'], optional] Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points. Choices are no bound ('none'), a single bounding ellipsoid ('single'), multiple bounding ellipsoids ('multi'), balls centered on each live point ('balls'), and cubes centered on each live point ('cubes'). Default is 'multi'.

**sample** [['auto', 'unif', 'rwalk', 'rstagger',]  
          'slice', 'rslice', 'hslice'], optional

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds. Unique methods available are: uniform sampling within the bounds ('unif'), random walks with fixed proposals ('rwalk'), random walks with variable (“staggering”) proposals ('rstagger'), multivariate slice sampling along preferred orientations ('slice'), “random” slice sampling along all orientations ('rslice'), and “Hamiltonian” slices along random trajectories ('hslice'). 'auto' selects the sampling method based on the dimensionality of the problem (from `ndim`). When `ndim` < 10, this defaults to 'unif'. When `10` <= `ndim` <= `20`, this defaults to 'rwalk'. When `ndim` > `20`, this defaults to 'hslice' if a gradient is provided and 'slice' otherwise. 'rstagger' and 'rslice' are provided as alternatives for 'rwalk' and 'slice', respectively. Default is 'auto'.

**periodic** [iterable, optional] A list of indices for parameters with periodic boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may wrap around the edge. Default is `None` (i.e. no periodic boundary conditions).

**reflective** [iterable, optional] A list of indices for parameters with reflective boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may reflect at the edge. Default is `None` (i.e. no reflective boundary conditions).

**update\_interval** [int or float, optional] If an integer is passed, only update the proposal distribution every `update_interval`-th likelihood call. If a float is passed, update the proposal after every `round(update_interval * nlive)`-th likelihood call. Larger update intervals larger can be more efficient when the likelihood function is quick to evaluate. Default behavior is to target a roughly constant change in prior volume, with `1.5` for 'unif', `0.15 * walks` for 'rwalk' and 'rstagger', `0.9 * ndim * slices` for 'slice', `2.0 * slices` for 'rslice', and `25.0 * slices` for 'hslice'.

**first\_update** [dict, optional] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube ('none') to the one specified by `sample`. Options are the minimum number of likelihood calls ('min\_ncall') and the minimum allowed overall efficiency in percent ('min\_eff'). Defaults are `2 * nlive` and `10.`, respectively.



**npdim** [int, optional] Number of parameters accepted by `prior_transform`. This might differ from `ndim` in the case where a parameter of loglikelihood is dependent upon multiple independently distributed parameters, some of which may be nuisance parameters.

**rstate** [`RandomState`, optional]

**RandomState instance.** If not given, the global random state of the `random` module will be used.

**queue\_size** [int, optional] Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) `queue_size` many threads. Each thread independently proposes new live points until the proposal distribution is updated. If no value is passed, this defaults to `pool.size` (if a `pool` has been provided) and 1 otherwise (no parallelism).

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where a pool should be used to execute operations in parallel. These govern whether `prior_transform` is executed in parallel during initialization ('`prior_transform`'), loglikelihood is executed in parallel during initialization ('`loglikelihood`'), live points are proposed in parallel during a run ('`propose_point`'), bounding distributions are updated in parallel during a run ('`update_bound`'), and the stopping criteria is evaluated in parallel during a run ('`stop_function`'). Default is `True` for all options.

**logl\_args** [iterable, optional] Additional arguments that can be passed to `loglikelihood`.

**logl\_kwargs** [dict, optional] Additional keyword arguments that can be passed to `loglikelihood`.

**ptform\_args** [iterable, optional] Additional arguments that can be passed to `prior_transform`.

**ptform\_kwargs** [dict, optional] Additional keyword arguments that can be passed to `prior_transform`.

**gradient** [function, optional] A function which returns the gradient corresponding to the provided `loglikelihood` *with respect to the unit cube*. If provided, this will be used when computing reflections when sampling with '`hslice`'. If not provided, gradients are approximated numerically using 2-sided differencing.

**grad\_args** [iterable, optional] Additional arguments that can be passed to `gradient`.

**grad\_kwargs** [dict, optional] Additional keyword arguments that can be passed to `gradient`.

**compute\_jac** [bool, optional] Whether to compute and apply the Jacobian  $dv/du$  from the target space  $v$  to the unit cube  $u$  when evaluating the `gradient`. If `False`, the gradient provided is assumed to be already defined with respect to the unit cube. If `True`, the gradient provided is assumed to be defined with respect to the target space so the Jacobian needs to be numerically computed and applied. Default is `False`.

**enlarge** [float, optional] Enlarge the volumes of the specified bounding object(s) by this fraction. The preferred method is to determine this organically using bootstrapping. If `bootstrap > 0`, this defaults to 1.0. If `bootstrap = 0`, this instead defaults to 1.25.

**bootstrap** [int, optional] Compute this many bootstrapped realizations of the bounding objects. Use the maximum distance found to the set of points left out during each iteration to enlarge the resulting volumes. Can lead to unstable bounding ellipsoids. Default is 0 (no bootstrap).

**vol\_dec** [float, optional] For the '`multi`' bounding option, the required fractional reduction in volume after splitting an ellipsoid in order to accept the split. Default is 0.5.

**vol\_check** [float, optional] For the 'multi' bounding option, the factor used when checking if the volume of the original bounding ellipsoid is large enough to warrant  $> 2$  splits via  $\text{ell.vol} > \text{vol\_check} * \text{nlive} * \text{pointvol}$ . Default is 2.0.

**walks** [int, optional] For the 'rwalk' sampling option, the minimum number of steps (minimum 2) before proposing a new live point. Default is 25.

**facc** [float, optional] The target acceptance fraction for the 'rwalk' sampling option. Default is 0.5. Bounded to be between  $[1. / \text{walks}, 1.]$ .

**slices** [int, optional] For the 'slice', 'rslice', and 'hslice' sampling options, the number of times to execute a “slice update” before proposing a new live point. Default is 5. Note that 'slice' cycles through **all dimensions** when executing a “slice update”.

**fmove** [float, optional] The target fraction of samples that are proposed along a trajectory (i.e. not reflecting) for the 'hslice' sampling option. Default is 0.9.

**max\_move** [int, optional] The maximum number of timesteps allowed for 'hslice' per proposal forwards and backwards in time. Default is 100.

### Returns

**sampler** [a `dynesty.DynamicSampler` instance] An initialized instance of the dynamic nested sampler.

**class** `dynesty.dynesty._function_wrapper` (*func, args, kwargs, name='input'*)

Bases: `object`

A hack to make functions pickleable when `args` or `kwargs` are also included. Based on the implementation in [emcee](#).

## Bounding

Bounding classes used when proposing new live points, along with a number of useful helper functions. Bounding objects include:

**UnitCube:** The unit N-cube (unconstrained draws from the prior).

**Ellipsoid:** Bounding ellipsoid.

**MultiEllipsoid:** A set of (possibly overlapping) bounding ellipsoids.

**RadFriends:** A set of (possibly overlapping) balls centered on each live point.

**SupFriends:** A set of (possibly overlapping) cubes centered on each live point.

**class** `dynesty.bounding.UnitCube` (*ndim*)

Bases: `object`

An N-dimensional unit cube.

### Parameters

**ndim** [int] The number of dimensions of the unit cube.

**contains** (*self, x*)

Checks if unit cube contains the point `x`.

**randoffset** (*self, rstate=None*)

Draw a random offset from the center of the unit cube.

**sample** (*self, rstate=None*)

Draw a sample uniformly distributed within the unit cube.



**Returns**

**x** [`ndarray` with shape (ndim,)] A coordinate within the unit cube.

**samples** (*self*, *nsamples*, *rstate=None*)

Draw *nsamples* samples randomly distributed within the unit cube.

**Returns**

**x** [`ndarray` with shape (nsamples, ndim)] A collection of coordinates within the unit cube.

**update** (*self*, *points*, *pointvol=0.0*, *rstate=None*, *bootstrap=0*, *pool=None*)

Filler function.

**class** `dynesty.bounding.Ellipsoid` (*ctr*, *cov*)

Bases: `object`

An N-dimensional ellipsoid defined by:

$$(\mathbf{x} - \mathbf{v})^T \mathbf{A} (\mathbf{x} - \mathbf{v}) = 1$$

where the vector *v* is the center of the ellipsoid and *A* is a symmetric, positive-definite  $N \times N$  matrix.

**Parameters**

**ctr** [`ndarray` with shape (N,)] Coordinates of ellipsoid center.

**cov** [`ndarray` with shape (N, N)] Covariance matrix describing the axes.

**contains** (*self*, *x*)

Checks if ellipsoid contains *x*.

**distance** (*self*, *x*)

Compute the normalized distance to *x* from the center of the ellipsoid.

**major\_axis\_endpoints** (*self*)

Return the endpoints of the major axis.

**randoffset** (*self*, *rstate=None*)

Return a random offset from the center of the ellipsoid.

**sample** (*self*, *rstate=None*)

Draw a sample uniformly distributed within the ellipsoid.

**Returns**

**x** [`ndarray` with shape (ndim,)] A coordinate within the ellipsoid.

**samples** (*self*, *nsamples*, *rstate=None*)

Draw *nsamples* samples uniformly distributed within the ellipsoid.

**Returns**

**x** [`ndarray` with shape (nsamples, ndim)] A collection of coordinates within the ellipsoid.

**scale\_to\_vol** (*self*, *vol*)

Scale ellipsoid to a target volume.

**unitcube\_overlap** (*self*, *ndraws=10000*, *rstate=None*)

Using *ndraws* Monte Carlo draws, estimate the fraction of overlap between the ellipsoid and the unit cube.

**update** (*self*, *points*, *pointvol=0.0*, *rstate=None*, *bootstrap=0*, *pool=None*, *mc\_integrate=False*)

Update the ellipsoid to bound the collection of points.

**Parameters**

**points** [`ndarray` with shape (npoints, ndim)] The set of points to bound.

**pointvol** [float, optional] The minimum volume associated with each point. Default is 0 ..

**rstate** [`RandomState`, optional] `RandomState` instance.

**bootstrap** [int, optional] The number of bootstrapped realizations of the ellipsoid. The maximum distance to the set of points “left out” during each iteration is used to enlarge the resulting volumes. Default is 0.

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**mc\_integrate** [bool, optional] Whether to use Monte Carlo methods to compute the effective overlap of the final ellipsoid with the unit cube. Default is `False`.

**class** `dynesty.bounding.MultiEllipsoid` (*ells=None, ctrs=None, covs=None*)

Bases: `object`

A collection of M N-dimensional ellipsoids.

#### Parameters

**ells** [list of `Ellipsoid` objects with length M, optional] A set of `Ellipsoid` objects that make up the collection of N-ellipsoids. Used to initialize `MultiEllipsoid` if provided.

**ctrs** [`ndarray` with shape (M, N), optional] Collection of coordinates of ellipsoid centers. Used to initialize `MultiEllipsoid` if `ams` is also provided.

**covs** [`ndarray` with shape (M, N, N), optional] Collection of matrices describing the axes of the ellipsoids. Used to initialize `MultiEllipsoid` if `ctrs` also provided.

**contains** (*self, x*)

Checks if the set of ellipsoids contains *x*.

**major\_axis\_endpoints** (*self*)

Return the endpoints of the major axis of each ellipsoid.

**monte\_carlo\_vol** (*self, ndraws=10000, rstate=None, return\_overlap=True*)

Using *ndraws* Monte Carlo draws, estimate the volume of the *union* of ellipsoids. If *return\_overlap=True*, also returns the estimated fractional overlap with the unit cube.

**overlap** (*self, x, j=None*)

Checks how many ellipsoid(s) *x* falls within, skipping the *j*-th ellipsoid.

**sample** (*self, rstate=None, return\_q=False*)

Sample a point uniformly distributed within the *union* of ellipsoids.

#### Returns

**x** [`ndarray` with shape (ndim,)] A coordinate within the set of ellipsoids.

**idx** [int] The index of the ellipsoid *x* was sampled from.

**q** [int, optional] The number of ellipsoids *x* falls within.

**samples** (*self, nsamples, rstate=None*)

Draw *nsamples* samples uniformly distributed within the *union* of ellipsoids.

#### Returns

**xs** [`ndarray` with shape (nsamples, ndim)] A collection of coordinates within the set of ellipsoids.

**scale\_to\_vols** (*self, vols*)

Scale ellipsoids to a corresponding set of target volumes.

**update** (*self*, *points*, *pointvol*=0.0, *vol\_dec*=0.5, *vol\_check*=2.0, *rstate*=None, *bootstrap*=0, *pool*=None, *mc\_integrate*=False)

Update the set of ellipsoids to bound the collection of points.

#### Parameters

**points** [ndarray with shape (npoints, ndim)] The set of points to bound.

**pointvol** [float, optional] The minimum volume associated with each point. Default is 0 . .

**vol\_dec** [float, optional] The required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is 0 . 5.

**vol\_check** [float, optional] The factor used when checking if the volume of the original bounding ellipsoid is large enough to warrant  $> 2$  splits via `ell.vol > vol_check * nlive * pointvol`. Default is 2 . 0.

**rstate** [RandomState, optional] RandomState instance.

**bootstrap** [int, optional] The number of bootstrapped realizations of the ellipsoids. The maximum distance to the set of points “left out” during each iteration is used to enlarge the resulting volumes. Default is 0.

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**mc\_integrate** [bool, optional] Whether to use Monte Carlo methods to compute the effective volume and fractional overlap of the final union of ellipsoids with the unit cube. Default is False.

**within** (*self*, *x*, *j*=None)

Checks which ellipsoid(s) *x* falls within, skipping the *j*-th ellipsoid.

**class** dynesty.bounding.RadFriends (*ndim*, *cov*=None)

Bases: object

A collection of N-balls of identical size centered on each live point.

#### Parameters

**ndim** [int] The number of dimensions of each ball.

**cov** [ndarray with shape (ndim, ndim), optional] Covariance structure (correlation and size) of each ball.

**\_get\_covariance\_from\_all\_points** (*self*, *points*)

Compute covariance using all points.

**\_get\_covariance\_from\_clusters** (*self*, *points*)

Compute covariance from re-centered clusters.

**contains** (*self*, *x*, *ctrs*)

Check if the set of balls contains *x*.

**monte\_carlo\_vol** (*self*, *ctrs*, *ndraws*=10000, *rstate*=None, *return\_overlap*=True)

Using *ndraws* Monte Carlo draws, estimate the volume of the *union* of balls. If *return\_overlap*=True, also returns the estimated fractional overlap with the unit cube.

**overlap** (*self*, *x*, *ctrs*)

Check how many balls *x* falls within.

**sample** (*self*, *ctrs*, *rstate*=None, *return\_q*=False)

Sample a point uniformly distributed within the *union* of balls.

#### Returns

**x** [`ndarray` with shape (ndim,)] A coordinate within the set of balls.

**q** [int, optional] The number of balls **x** falls within.

**samples** (*self*, *nsamples*, *ctrs*, *rstate=None*)

Draw *nsamples* samples uniformly distributed within the *union* of balls.

#### Returns

**xs** [`ndarray` with shape (nsamples, ndim)] A collection of coordinates within the set of balls.

**scale\_to\_vol** (*self*, *vol*)

Scale ball to encompass a target volume.

**update** (*self*, *points*, *pointvol=0.0*, *rstate=None*, *bootstrap=0*, *pool=None*, *mc\_integrate=False*, *use\_clustering=True*)

Update the radii of our balls.

#### Parameters

**points** [`ndarray` with shape (npoints, ndim)] The set of points to bound.

**pointvol** [float, optional] The minimum volume associated with each point. Default is 0 . .

**rstate** [`RandomState`, optional] `RandomState` instance.

**bootstrap** [int, optional] The number of bootstrapped realizations of the ellipsoids. The maximum distance to the set of points “left out” during each iteration is used to enlarge the resulting volumes. Default is 0.

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**mc\_integrate** [bool, optional] Whether to use Monte Carlo methods to compute the effective volume and fractional overlap of the final union of balls with the unit cube. Default is `False`.

**use\_clustering** [bool, optional] Whether to use clustering to avoid issues with widely-separated modes. Default is `True`.

**within** (*self*, *x*, *ctrs*)

Check which balls **x** falls within.

**class** `dynesty.bounding.SuperFriends` (*ndim*, *cov=None*)

Bases: `object`

A collection of N-cubes of identical size centered on each live point.

#### Parameters

**ndim** [int] The number of dimensions of the cube.

**cov** [`ndarray` with shape (ndim, ndim), optional] Covariance structure (correlation and size) of each cube.

**\_\_get\_covariance\_from\_all\_points** (*self*, *points*)

Compute covariance using all points.

**\_\_get\_covariance\_from\_clusters** (*self*, *points*)

Compute covariance from re-centered clusters.

**contains** (*self*, *x*, *ctrs*)

Checks if the set of cubes contains **x**.

**monte\_carlo\_vol** (*self*, *ctr*s, *ndraws*=10000, *rstate*=None, *return\_overlap*=False)

Using *ndraws* Monte Carlo draws, estimate the volume of the *union* of cubes. If *return\_overlap*=True, also returns the estimated fractional overlap with the unit cube.

**overlap** (*self*, *x*, *ctr*s)

Checks how many cubes *x* falls within, skipping the *j*-th cube.

**sample** (*self*, *ctr*s, *rstate*=None, *return\_q*=False)

Sample a point uniformly distributed within the *union* of cubes.

#### Returns

**x** [ndarray with shape (ndim,)] A coordinate within the set of cubes.

**q** [int, optional] The number of cubes *x* falls within.

**samples** (*self*, *nsamples*, *ctr*s, *rstate*=None)

Draw *nsamples* samples uniformly distributed within the *union* of cubes.

#### Returns

**xs** [ndarray with shape (nsamples, ndim)] A collection of coordinates within the set of cubes.

**scale\_to\_vol** (*self*, *vol*)

Scale cube to encompass a target volume.

**update** (*self*, *points*, *pointvol*=0.0, *rstate*=None, *bootstrap*=0, *pool*=None, *mc\_integrate*=False, *use\_clustering*=True)

Update the half-side-lengths of our cubes.

#### Parameters

**points** [ndarray with shape (npoints, ndim)] The set of points to bound.

**pointvol** [float, optional] The minimum volume associated with each point. Default is 0.

**rstate** [RandomState, optional] RandomState instance.

**bootstrap** [int, optional] The number of bootstrapped realizations of the ellipsoids. The maximum distance to the set of points “left out” during each iteration is used to enlarge the resulting volumes. Default is 0.

**pool** [user-provided pool, optional] Use this pool of workers to execute operations in parallel.

**mc\_integrate** [bool, optional] Whether to use Monte Carlo methods to compute the effective volume and fractional overlap of the final union of cubes with the unit cube. Default is False.

**use\_clustering** [bool, optional] Whether to use clustering to avoid issues with widely-separated modes. Default is True.

**within** (*self*, *x*, *ctr*s)

Checks which cubes *x* falls within.

**dynesty.bounding.vol\_prefactor** (*n*, *p*=2.0)

Returns the volume constant for an *n*-dimensional sphere with an  $L^p$  norm. The constant is defined as:

$$f = (2. * \text{Gamma}(1./p + 1))^{*n} / \text{Gamma}(n/p + 1.)$$

By default the  $p=2.$  norm is used (i.e. the standard Euclidean norm).

**dynesty.bounding.logvol\_prefactor** (*n*, *p*=2.0)

Returns the  $\ln(\text{volume constant})$  for an *n*-dimensional sphere with an  $L^p$  norm. The constant is defined as:

```
lnf = n * ln(2.) + n * LogGamma(1./p + 1) - LogGamma(n/p + 1.)
```

By default the `p=2.` norm is used (i.e. the standard Euclidean norm).

`dynesty.bounding.randsphere` (*n*, *rstate=None*)

Draw a point uniformly within an *n*-dimensional unit sphere.

`dynesty.bounding.bounding_ellipsoid` (*points*, *pointvol=0.0*)

Calculate the bounding ellipsoid containing a collection of points.

#### Parameters

**points** [`ndarray` with shape (*npoints*, *ndim*)] A set of coordinates.

**pointvol** [float, optional] The minimum volume occupied by a single point. When provided, used to set a minimum bound on the ellipsoid volume as *npoints* \* *pointvol*. Default is 0..

#### Returns

**ellipsoid** [`Ellipsoid`] The bounding `Ellipsoid` object.

`dynesty.bounding.bounding_ellipsoids` (*points*, *pointvol=0.0*, *vol\_dec=0.5*, *vol\_check=2.0*)

Calculate a set of ellipsoids that bound the collection of points.

#### Parameters

**points** [`ndarray` with shape (*npoints*, *ndim*)] A set of coordinates.

**pointvol** [float, optional] Volume represented by a single point. When provided, used to set a minimum bound on the ellipsoid volume as *npoints* \* *pointvol*. Default is 0..

**vol\_dec** [float, optional] The required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is 0.5.

**vol\_check** [float, optional] The factor used to when checking whether the volume of the original bounding ellipsoid is large enough to warrant more trial splits via *ell.vol* > *vol\_check* \* *npoints* \* *pointvol*. Default is 2.0.

#### Returns

**mell** [`MultiEllipsoid` object] The `MultiEllipsoid` object used to bound the collection of points.

`dynesty.bounding._bounding_ellipsoids` (*points*, *ell*, *pointvol=0.0*, *vol\_dec=0.5*, *vol\_check=2.0*)

Internal method used to compute a set of bounding ellipsoids when a bounding ellipsoid for the entire set has already been calculated.

#### Parameters

**points** [`ndarray` with shape (*npoints*, *ndim*)] A set of coordinates.

**ell** [`Ellipsoid`] The bounding ellipsoid containing *points*.

**pointvol** [float, optional] Volume represented by a single point. When provided, used to set a minimum bound on the ellipsoid volume as *npoints* \* *pointvol*. Default is 0..

**vol\_dec** [float, optional] The required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is 0.5.

**vol\_check** [float, optional] The factor used to when checking whether the volume of the original bounding ellipsoid is large enough to warrant more trial splits via *ell.vol* > *vol\_check* \* *npoints* \* *pointvol*. Default is 2.0.

### Returns

**ells** [list of *Ellipsoid* objects] List of *Ellipsoid* objects used to bound the collection of points. Used to initialize the *MultiEllipsoid* object returned in *bounding\_ellipsoids()*.

`dynesty.bounding._ellipsoid_bootstrap_expand(args)`

Internal method used to compute the expansion factor for a bounding ellipsoid based on bootstrapping.

`dynesty.bounding._ellipsoids_bootstrap_expand(args)`

Internal method used to compute the expansion factor(s) for a collection of bounding ellipsoids using bootstrapping.

`dynesty.bounding._friends_bootstrap_radius(args)`

Internal method used to compute the radius (half-side-length) for each ball (cube) used in *RadFriends* (*SupFriends*) using bootstrapping.

`dynesty.bounding._friends_leaveoneout_radius(points, ftype)`

Internal method used to compute the radius (half-side-length) for each ball (cube) used in *RadFriends* (*SupFriends*) using leave-one-out (LOO) cross-validation.

### Sampling

Functions for proposing new live points used by *Sampler* (and its children from *nestedsamplers*) and *DynamicSampler*.

`dynesty.sampling.sample_unif(args)`

Evaluate a new point sampled uniformly from a bounding proposal distribution. Parameters are zipped within `args` to utilize `pool.map`-style functions.

### Parameters

**u** [*ndarray* with shape (npdim,)] Position of the initial sample.

**loglstar** [float] Ln(likelihood) bound. **Not applicable here.**

**axes** [*ndarray* with shape (ndim, ndim)] Axes used to propose new points. **Not applicable here.**

**scale** [float] Value used to scale the provided axes. **Not applicable here.**

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning ln(likelihood) given parameters as a 1-d *numpy* array of length *ndim*.

**kwargs** [dict] A dictionary of additional method-specific parameters. **Not applicable here.**

### Returns

**u** [*ndarray* with shape (npdim,)] Position of the final proposed point within the unit cube. **For uniform sampling this is the same as the initial input position.**

**v** [*ndarray* with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample. For uniform sampling this is 1 by construction.

**blob** [dict] Collection of ancillary quantities used to tune `scale`. **Not applicable for uniform sampling.**

`dynesty.sampling.sample_rwalk(args)`

Return a new live point proposed by random walking away from an existing live point.

#### Parameters

**u** [ndarray with shape (npdim,)] Position of the initial sample. **This is a copy of an existing live point.**

**loglstar** [float] Ln(likelihood) bound.

**axes** [ndarray with shape (ndim, ndim)] Axes used to propose new points. For random walks new positions are proposed using the *Ellipsoid* whose shape is defined by axes.

**scale** [float] Value used to scale the provided axes.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning ln(likelihood) given parameters as a 1-d `numpy` array of length `ndim`.

**kwargs** [dict] A dictionary of additional method-specific parameters.

#### Returns

**u** [ndarray with shape (npdim,)] Position of the final proposed point within the unit cube.

**v** [ndarray with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample.

**blob** [dict] Collection of ancillary quantities used to tune `scale`.

`dynesty.sampling.sample_rstagger(args)`

Return a new live point proposed by random “staggering” away from an existing live point. The difference between this and the random walk is the step size is exponentially adjusted to reach a target acceptance rate *during* each proposal (in addition to *between* proposals).

#### Parameters

**u** [ndarray with shape (npdim,)] Position of the initial sample. **This is a copy of an existing live point.**

**loglstar** [float] Ln(likelihood) bound.

**axes** [ndarray with shape (ndim, ndim)] Axes used to propose new points. For random walks new positions are proposed using the *Ellipsoid* whose shape is defined by axes.

**scale** [float] Value used to scale the provided axes.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning ln(likelihood) given parameters as a 1-d `numpy` array of length `ndim`.

**kwargs** [dict] A dictionary of additional method-specific parameters.

#### Returns

**u** [ndarray with shape (npdim,)] Position of the final proposed point within the unit cube.



**v** [`ndarray` with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample.

**blob** [dict] Collection of ancillary quantities used to tune `scale`.

`dynesty.sampling.sample_slice(args)`

Return a new live point proposed by a series of random slices away from an existing live point. Standard “Gibbs-like” implementation where a single multivariate “slice” is a combination of `ndim` univariate slices through each axis.

#### Parameters

**u** [`ndarray` with shape (npdim,)] Position of the initial sample. **This is a copy of an existing live point.**

**loglstar** [float] Ln(likelihood) bound.

**axes** [`ndarray` with shape (ndim, ndim)] Axes used to propose new points. For slices new positions are proposed along the arthogonal basis defined by `axes`.

**scale** [float] Value used to scale the provided axes.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning ln(likelihood) given parameters as a 1-d `numpy` array of length `ndim`.

**kwargs** [dict] A dictionary of additional method-specific parameters.

#### Returns

**u** [`ndarray` with shape (npdim,)] Position of the final proposed point within the unit cube.

**v** [`ndarray` with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample.

**blob** [dict] Collection of ancillary quantities used to tune `scale`.

`dynesty.sampling.sample_rslice(args)`

Return a new live point proposed by a series of random slices away from an existing live point. Standard “random” implementation where each slice is along a random direction based on the provided axes.

#### Parameters

**u** [`ndarray` with shape (npdim,)] Position of the initial sample. **This is a copy of an existing live point.**

**loglstar** [float] Ln(likelihood) bound.

**axes** [`ndarray` with shape (ndim, ndim)] Axes used to propose new slice directions.

**scale** [float] Value used to scale the provided axes.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning ln(likelihood) given parameters as a 1-d `numpy` array of length `ndim`.

**kwargs** [dict] A dictionary of additional method-specific parameters.

#### Returns

**u** [`ndarray` with shape (npdim,)] Position of the final proposed point within the unit cube.

**v** [`ndarray` with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample.

**blob** [dict] Collection of ancillary quantities used to tune `scale`.

`dynesty.sampling.sample_hslice` (*args*)

Return a new live point proposed by “Hamiltonian” Slice Sampling using a series of random trajectories away from an existing live point. Each trajectory is based on the provided axes and samples are determined by moving forwards/backwards in time until the trajectory hits an edge and approximately reflecting off the boundaries. Once a series of reflections has been established, we propose a new live point by slice sampling across the entire path.

#### Parameters

**u** [`ndarray` with shape (npdim,)] Position of the initial sample. **This is a copy of an existing live point.**

**loglstar** [float] Ln(likelihood) bound.

**axes** [`ndarray` with shape (ndim, ndim)] Axes used to propose new slice directions.

**scale** [float] Value used to scale the provided axes.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**loglikelihood** [function] Function returning Ln(likelihood) given parameters as a 1-d `numpy` array of length `ndim`.

**kwargs** [dict] A dictionary of additional method-specific parameters.

#### Returns

**u** [`ndarray` with shape (npdim,)] Position of the final proposed point within the unit cube.

**v** [`ndarray` with shape (ndim,)] Position of the final proposed point in the target parameter space.

**logl** [float] Ln(likelihood) of the final proposed point.

**nc** [int] Number of function calls used to generate the sample.

**blob** [dict] Collection of ancillary quantities used to tune `scale`.

## Baseline Sampler

The base `Sampler` class containing various helpful functions. All other samplers inherit this class either explicitly or implicitly.

**class** `dynesty.sampler.Sampler` (*loglikelihood*, *prior\_transform*, *npdim*, *live\_points*, *update\_interval*, *first\_update*, *rstate*, *queue\_size*, *pool*, *use\_pool*)

Bases: `object`

The basic sampler object that performs the actual nested sampling.

**Parameters**

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int, optional] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**`_beyond_unit_bound (self, loglstar)`**

Check whether we should update our bound beyond the initial unit cube.

**`_empty_queue (self)`**

Dump all live point proposals currently on the queue.

**`_fill_queue (self, loglstar)`**

Sequentially add new live point proposals to the queue.

**`_get_point_value (self, loglstar)`**

Grab the first live point proposal in the queue.

**`_get_print_func (self, print_func, print_progress)`**

**`_new_point (self, loglstar, logvol)`**

Propose points until a new point that satisfies the log-likelihood constraint `loglstar` is found.

**`_remove_live_points (self)`**

Remove the final set of live points if they were previously added to the current set of dead points.

**`add_final_live (self, print_progress=True, print_func=None)`**

**A wrapper that executes the loop adding the final live points.** Adds the final set of live points to the pre-existing sequence of dead points from the current nested sampling run.

**Parameters**

**print\_progress** [bool, optional] Whether or not to output a simple summary of the current run that updates with each iteration. Default is `True`.

**print\_func** [function, optional] A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.

**`add_live_points (self)`**

Add the remaining set of live points to the current set of dead points. Instantiates a generator that will be called by the user. Returns the same outputs as `sample()`.

**n\_effective**

Estimate the effective number of posterior samples using the Kish Effective Sample Size (ESS) where  $ESS = \text{sum}(wts)^2 / \text{sum}(wts^2)$ . Note that this is  $\text{len}(wts)$  when  $wts$  are uniform and 1 if there is only one non-zero element in  $wts$ .

**reset** (*self*)

Re-initialize the sampler.

**results**

Saved results from the nested sampling run. If bounding distributions were saved, those are also returned.

**run\_nested** (*self*, *maxiter*=None, *maxcall*=None, *dlogz*=None, *logl\_max*=inf, *n\_effective*=None, *add\_live*=True, *print\_progress*=True, *print\_func*=None, *save\_bounds*=True)

**A wrapper that executes the main nested sampling loop.** Iteratively replace the worst live point with a sample drawn uniformly from the prior until the provided stopping criteria are reached.

**Parameters**

**maxiter** [int, optional] Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations. Iteration may stop earlier if termination condition is reached. Default is `sys.maxsize` (no limit).

**dlogz** [float, optional] Iteration will stop when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is  $\ln(z + z_{\text{est}}) - \ln(z) < dlogz$ , where  $z$  is the current evidence from all saved samples and  $z_{\text{est}}$  is the estimated contribution from the remaining volume. If *add\_live* is `True`, the default is  $1e-3 * (n_{\text{live}} - 1) + 0.01$ . Otherwise, the default is 0.01.

**logl\_max** [float, optional] Iteration will stop when the sampled  $\ln(\text{likelihood})$  exceeds the threshold set by *logl\_max*. Default is no bound (`np.inf`).

**n\_effective: int, optional** Minimum number of effective posterior samples. If the estimated “effective sample size” (ESS) exceeds this number, sampling will terminate. Default is no ESS (`np.inf`).

**add\_live** [bool, optional] Whether or not to add the remaining set of live points to the list of samples at the end of each run. Default is `True`.

**print\_progress** [bool, optional] Whether or not to output a simple summary of the current run that updates with each iteration. Default is `True`.

**print\_func** [function, optional] A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.

**save\_bounds** [bool, optional] Whether or not to save past bounding distributions used to bound the live points internally. Default is `True`.

**sample** (*self*, *maxiter*=None, *maxcall*=None, *dlogz*=0.01, *logl\_max*=inf, *n\_effective*=inf, *add\_live*=True, *save\_bounds*=True, *save\_samples*=True)

**The main nested sampling loop.** Iteratively replace the worst live point with a sample drawn uniformly from the prior until the provided stopping criteria are reached. Instantiates a generator that will be called by the user.

**Parameters**

**maxiter** [int, optional] Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations. Iteration may stop earlier if termination condition is reached. Default is `sys.maxsize` (no limit).

**dlogz** [float, optional] Iteration will stop when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is  $\ln(z + z_{\text{est}}) - \ln(z) < \text{dlogz}$ , where  $z$  is the current evidence from all saved samples and  $z_{\text{est}}$  is the estimated contribution from the remaining volume. Default is 0.01.

**logl\_max** [float, optional] Iteration will stop when the sampled  $\ln(\text{likelihood})$  exceeds the threshold set by `logl_max`. Default is no bound (`np.inf`).

**n\_effective: int, optional** Minimum number of effective posterior samples. If the estimated “effective sample size” (ESS) exceeds this number, sampling will terminate. Default is no ESS (`np.inf`).

**add\_live** [bool, optional] Whether or not to add the remaining set of live points to the list of samples when calculating `n_effective`. Default is `True`.

**save\_bounds** [bool, optional] Whether or not to save past distributions used to bound the live points internally. Default is `True`.

**save\_samples** [bool, optional] Whether or not to save past samples from the nested sampling run (along with other ancillary quantities) internally. Default is `True`.

## Returns

**worst** [int] Index of the live point with the worst likelihood. This is our new dead point sample.

**ustar** [ndarray with shape (npdim,)] Position of the sample.

**vstar** [ndarray with shape (ndim,)] Transformed position of the sample.

**loglstar** [float]  $\ln(\text{likelihood})$  of the sample.

**logvol** [float]  $\ln(\text{prior volume})$  within the sample.

**logwt** [float]  $\ln(\text{weight})$  of the sample.

**logz** [float] Cumulative  $\ln(\text{evidence})$  up to the sample (inclusive).

**logzvar** [float] Estimated cumulative variance on `logz` (inclusive).

**h** [float] Cumulative information up to the sample (inclusive).

**nc** [int] Number of likelihood calls performed before the new live point was accepted.

**worst\_it** [int] Iteration when the live (now dead) point was originally proposed.

**boundidx** [int] Index of the bound the dead point was originally drawn from.

**bounditer** [int] Index of the bound being used at the current iteration.

**eff** [float] The cumulative sampling efficiency (in percent).

**delta\_logz** [float] The estimated remaining evidence expressed as the  $\ln(\text{ratio})$  of the current evidence.

## Static Nested Samplers

Children of `dynesty.sampler` used to proposing new live points. Includes:

**UnitCubeSampler:** Uses the unit cube to bound the set of live points (i.e. no bound).

**SingleEllipsoidSampler:** Uses a single ellipsoid to bound the set of live points.

**MultiEllipsoidSampler:** Uses multiple ellipsoids to bound the set of live points.

**RadFriendsSampler:** Uses an N-sphere of fixed radius centered on each live point to bound the set of live points.

**SupFriendsSampler:** Uses an N-cube of fixed length centered on each live point to bound the set of live points.

```
class dynesty.nestedsamplers.UnitCubeSampler(loglikelihood, prior_transform, npdim,
                                             live_points, method, update_interval,
                                             first_update, rstate, queue_size, pool,
                                             use_pool, kwargs={})
```

Bases: `dynesty.sampler.Sampler`

Samples conditioned on the unit N-cube (i.e. with no bounds).

### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape  $(n_{\text{live}}, \text{ndim})$ ] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**method** [{`'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters.

**propose\_live** (*self*)

Return a live point/axes to be used by other sampling methods.

**propose\_unif** (*self*)

Propose a new live point by sampling *uniformly* within the unit cube.

**update** (*self*, *pointvol*)

Update the unit cube bound.

**update\_hslice** (*self*, *blob*)

Update the Hamiltonian slice proposal scale based on the relative amount of time spent moving vs reflecting.

**update\_rwalk** (*self*, *blob*)

Update the random walk proposal scale based on the current number of accepted/rejected steps.

**update\_slice** (*self*, *blob*)

Update the slice proposal scale based on the relative size of the slices compared to our initial guess.

**update\_unif** (*self*, *blob*)

Filler function.

```
class dynesty.nestedsamplers.SingleEllipsoidSampler(loglikelihood, prior_transform,
                                                    npdim, live_points, method, update_interval,
                                                    first_update, rstate, queue_size,
                                                    pool, use_pool, kwargs={})
```

Bases: `dynesty.sampler.Sampler`

Samples conditioned on a single ellipsoid used to bound the set of live points.

#### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**method** [{`'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters.

**propose\_live** (*self*)

Return a live point/axes to be used by other sampling methods.

**propose\_unif** (*self*)

Propose a new live point by sampling *uniformly* within the ellipsoid.

**update** (*self*, *pointvol*)

Update the bounding ellipsoid using the current set of live points.

**update\_hslice** (*self*, *blob*)

Update the Hamiltonian slice proposal scale based on the relative amount of time spent moving vs reflecting.

**update\_rwalk** (*self*, *blob*)

Update the random walk proposal scale based on the current number of accepted/rejected steps.

**update\_slice** (*self*, *blob*)

Update the slice proposal scale based on the relative size of the slices compared to our initial guess.

**update\_unif** (*self*, *blob*)

Filler function.

```
class dynesty.nestedsamplers.MultiEllipsoidSampler (loglikelihood, prior_transform,
                                                npdim, live_points, method,
                                                update_interval, first_update,
                                                rstate, queue_size, pool, use_pool,
                                                kwargs={})
```

Bases: `dynesty.sampler.Sampler`

Samples conditioned on the union of multiple (possibly overlapping) ellipsoids used to bound the set of live points.

#### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**method** [{`'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters.

**propose\_live** (*self*)

Return a live point/axes to be used by other sampling methods.



**propose\_unif** (*self*)

Propose a new live point by sampling *uniformly* within the union of ellipsoids.

**update** (*self*, *pointvol*)

Update the bounding ellipsoids using the current set of live points.

**update\_hslice** (*self*, *blob*)

Update the Hamiltonian slice proposal scale based on the relative amount of time spent moving vs reflecting.

**update\_rwalk** (*self*, *blob*)

Update the random walk proposal scale based on the current number of accepted/rejected steps.

**update\_slice** (*self*, *blob*)

Update the slice proposal scale based on the relative size of the slices compared to our initial guess.

**update\_unif** (*self*, *blob*)

Filler function.

```
class dynesty.nestedsamplers.RadFriendsSampler (loglikelihood, prior_transform, npdim,
                                              live_points, method, update_interval,
                                              first_update, rstate, queue_size, pool,
                                              use_pool, kwargs={})
```

Bases: `dynesty.sampler.Sampler`

Samples conditioned on the union of (possibly overlapping) N-spheres centered on the current set of live points.

#### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**method** [{`'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters.

**propose\_live** (*self*)

Propose a live point/axes to be used by other sampling methods.

**propose\_unif** (*self*)

Propose a new live point by sampling *uniformly* within the union of N-spheres defined by our live points.

**update** (*self*, *pointvol*)

Update the N-sphere radii using the current set of live points.

**update\_hslice** (*self*, *blob*)

Update the Hamiltonian slice proposal scale based on the relative amount of time spent moving vs reflecting.

**update\_rwalk** (*self*, *blob*)

Update the random walk proposal scale based on the current number of accepted/rejected steps.

**update\_slice** (*self*, *blob*)

Update the slice proposal scale based on the relative size of the slices compared to our initial guess.

**update\_unif** (*self*, *blob*)

Filler function.

```
class dynesty.nestedsamplers.SupFriendsSampler (loglikelihood, prior_transform, npdim,
                                              live_points, method, update_interval,
                                              first_update, rstate, queue_size, pool,
                                              use_pool, kwargs={})
```

Bases: `dynesty.sampler.Sampler`

Samples conditioned on the union of (possibly overlapping) N-cubes centered on the current set of live points.

#### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d `numpy` array of length `ndim`.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int] Number of parameters accepted by `prior_transform`.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] Initial set of “live” points. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods.

**method** [{`'unif'`, `'rwalk'`, `'rstagger'`,  
`'slice'`, `'rslice'`, `'hslice'`}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters.

**propose\_live** (*self*)

Return a live point/axes to be used by other sampling methods.

**propose\_unif** (*self*)

Propose a new live point by sampling *uniformly* within the collection of N-cubes defined by our live points.

**update** (*self*, *pointvol*)

Update the N-cube side-lengths using the current set of live points.

**update\_hslice** (*self*, *blob*)

Update the Hamiltonian slice proposal scale based on the relative amount of time spent moving vs reflecting.

**update\_rwalk** (*self*, *blob*)

Update the random walk proposal scale based on the current number of accepted/rejected steps.

**update\_slice** (*self*, *blob*)

Update the slice proposal scale based on the relative size of the slices compared to our initial guess.

**update\_unif** (*self*, *blob*)

Filler function.

## Dynamic Nested Sampler

Contains the dynamic nested sampler class *DynamicSampler* used to dynamically allocate nested samples. Note that *DynamicSampler* implicitly wraps a sampler from *nestedsamplers*. Also contains the weight function *weight\_function()* and stopping function *stopping\_function()*. These are used by default within *DynamicSampler* if corresponding functions are not provided by the user.

```
class dynesty.dynamicsampler.DynamicSampler (loglikelihood, prior_transform, npdim,
                                             bound, method, update_interval,
                                             first_update, rstate, queue_size, pool,
                                             use_pool, kwargs)
```

Bases: *object*

A dynamic nested sampler that allocates live points adaptively during a single run according to a specified weight function until a specified stopping criteria is reached.

### Parameters

**loglikelihood** [function] Function returning  $\ln(\text{likelihood})$  given parameters as a 1-d *numpy* array of length *ndim*.

**prior\_transform** [function] Function transforming a sample from the a unit cube to the parameter space of interest according to the prior.

**npdim** [int, optional] Number of parameters accepted by *prior\_transform*.

**bound** [{*'none'*, *'single'*, *'multi'*, *'balls'*, *'cubes'*}, optional] Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points.

**method** [{*'unif'*, *'rwalk'*, *'rstagger'*,  
*'slice'*, *'rslice'*, *'hslice'*}, optional]

Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds.

**update\_interval** [int] Only update the bounding distribution every `update_interval`-th likelihood call.

**first\_update** [dict] A dictionary containing parameters governing when the sampler should first update the bounding distribution from the unit cube to the one specified by the user.

**rstate** [`RandomState`] `RandomState` instance.

**queue\_size: int** Carry out likelihood evaluations in parallel by queueing up new live point proposals using (at most) this many threads/members.

**pool: pool** Use this pool of workers to execute operations in parallel.

**use\_pool** [dict, optional] A dictionary containing flags indicating where the provided `pool` should be used to execute operations in parallel.

**kwargs** [dict, optional] A dictionary of additional parameters (described below).

**\_get\_print\_func** (*self*, *print\_func*, *print\_progress*)

**add\_batch** (*self*, *nlive*=500, *wt\_function*=None, *wt\_kwargs*=None, *maxiter*=None, *maxcall*=None, *logl\_bounds*=None, *save\_bounds*=True, *print\_progress*=True, *print\_func*=None, *stop\_val*=None)

Allocate an additional batch of (nested) samples based on the combined set of previous samples using the specified weight function.

### Parameters

**nlive** [int, optional] The number of live points used when adding additional samples in the batch. Default is 500.

**wt\_function** [func, optional] A cost function that takes a `Results` instance and returns a log-likelihood range over which a new batch of samples should be generated. The default function simply computes a weighted average of the posterior and evidence information content as:

$$\text{weight} = \text{pfrac} * \text{pweight} + (1. - \text{pfrac}) * \text{zweight}$$

**wt\_kwargs** [dict, optional] Extra arguments to be passed to the weight function.

**maxiter** [int, optional] Maximum number of iterations allowed. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations allowed. Default is `sys.maxsize` (no limit).

**logl\_bounds** [tuple of size (2,), optional] The  $\ln(\text{likelihood})$  bounds used to bracket the run. If `None`, the provided `wt_function` will be used to determine the bounds (this is the default behavior).

**save\_bounds** [bool, optional] Whether or not to save distributions used to bound the live points internally during dynamic live point allocations. Default is `True`.

**print\_progress** [bool, optional] Whether to output a simple summary of the current run that updates each iteration. Default is `True`.

**print\_func** [function, optional] A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.

**stop\_val** [float, optional] The value of the stopping criteria to be passed to `print_func()`. Used internally within `run_nested()` to keep track of progress.

**combine\_runs** (*self*)

Merge the most recent run into the previous (combined) run by “stepping through” both runs simultaneously.

**n\_effective**

Estimate the effective number of posterior samples using the Kish Effective Sample Size (ESS) where  $ESS = \text{sum}(wts)^2 / \text{sum}(wts^2)$ . Note that this is  $\text{len}(wts)$  when *wts* are uniform and 1 if there is only one non-zero element in *wts*.

**reset** (*self*)

Re-initialize the sampler.

**results**

Saved results from the dynamic nested sampling run. All saved bounds are also returned.

**run\_nested** (*self*, *nlive\_init*=500, *maxiter\_init*=None, *maxcall\_init*=None, *dlogz\_init*=0.01, *logl\_max\_init*=inf, *n\_effective\_init*=inf, *nlive\_batch*=500, *wt\_function*=None, *wt\_kwargs*=None, *maxiter\_batch*=None, *maxcall\_batch*=None, *maxiter*=None, *maxcall*=None, *maxbatch*=None, *n\_effective*=inf, *stop\_function*=None, *stop\_kwargs*=None, *use\_stop*=True, *save\_bounds*=True, *print\_progress*=True, *print\_func*=None, *live\_points*=None)

**The main dynamic nested sampling loop.** After an initial “baseline” run using a constant number of live points, dynamically allocates additional (nested) samples to optimize a specified weight function until a specified stopping criterion is reached.

**Parameters**

**nlive\_init** [int, optional] The number of live points used during the initial (“baseline”) nested sampling run. Default is 500.

**maxiter\_init** [int, optional] Maximum number of iterations for the initial baseline nested sampling run. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall\_init** [int, optional] Maximum number of likelihood evaluations for the initial baseline nested sampling run. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**dlogz\_init** [float, optional] The baseline run will stop when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is  $\ln(z + z_{\text{est}}) - \ln(z) < dlogz$ , where *z* is the current evidence from all saved samples and *z<sub>est</sub>* is the estimated contribution from the remaining volume. The default is 0.01.

**logl\_max\_init** [float, optional] The baseline run will stop when the sampled  $\ln(\text{likelihood})$  exceeds this threshold. Default is no bound (`np.inf`).

**n\_effective\_init: int, optional** Minimum number of effective posterior samples needed during the baseline run. If the estimated “effective sample size” (ESS) exceeds this number, sampling will terminate. Default is no ESS (`np.inf`).

**nlive\_batch** [int, optional] The number of live points used when adding additional samples from a nested sampling run within each batch. Default is 500.

**wt\_function** [func, optional] A cost function that takes a `Results` instance and returns a log-likelihood range over which a new batch of samples should be generated. The default function simply computes a weighted average of the posterior and evidence information content as:

$$\text{weight} = \text{pfrac} * \text{pweight} + (1. - \text{pfrac}) * \text{zweight}$$

**wt\_kwargs** [dict, optional] Extra arguments to be passed to the weight function.

**maxiter\_batch** [int, optional] Maximum number of iterations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall\_batch** [int, optional] Maximum number of likelihood evaluations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxiter** [int, optional] Maximum number of iterations allowed. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations allowed. Default is `sys.maxsize` (no limit).

**maxbatch** [int, optional] Maximum number of batches allowed. Default is `sys.maxsize` (no limit).

**n\_effective: int, optional** Minimum number of effective posterior samples needed during the entire run. If the estimated “effective sample size” (ESS) exceeds this number, sampling will terminate. Default is no ESS (`np.inf`).

**stop\_function** [func, optional] A function that takes a `Results` instance and returns a boolean indicating that we should terminate the run because we’ve collected enough samples.

**stop\_kwargs** [float, optional] Extra arguments to be passed to the stopping function.

**use\_stop** [bool, optional] Whether to evaluate our stopping function after each batch. Disabling this can improve performance if other stopping criteria such as `maxcall` are already specified. Default is `True`.

**save\_bounds** [bool, optional] Whether or not to save distributions used to bound the live points internally during dynamic live point allocation. Default is `True`.

**print\_progress** [bool, optional] Whether to output a simple summary of the current run that updates each iteration. Default is `True`.

**print\_func** [function, optional] A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] A set of live points used to initialize the nested sampling run. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods. By default, if these are not provided the initial set of live points will be drawn from the unit `npdim`-cube. **WARNING: It is crucial that the initial set of live points have been sampled from the prior. Failure to provide a set of valid live points will result in biased results.**

**sample\_batch** (*self*, *nlive\_new*=500, *update\_interval*=None, *logl\_bounds*=None, *maxiter*=None, *maxcall*=None, *save\_bounds*=True)

Generate an additional series of nested samples that will be combined with the previous set of dead points. Works by hacking the internal `sampler` object. Instantiates a generator that will be called by the user.

#### Parameters

**nlive\_new** [int] Number of new live points to be added. Default is 500.

**update\_interval** [int or float, optional] If an integer is passed, only update the bounding distribution every `update_interval`-th likelihood call. If a float is passed, update the bound after every `round(update_interval * nlive)`-th likelihood call. Larger

update intervals can be more efficient when the likelihood function is quick to evaluate. If no value is provided, defaults to the value passed during initialization.

**logl\_bounds** [tuple of size (2,), optional] The  $\ln(\text{likelihood})$  bounds used to bracket the run. If `None`, the default bounds span the entire range covered by the original run.

**maxiter** [int, optional] Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations. Iteration may stop earlier if termination condition is reached. Default is `sys.maxsize` (no limit).

**save\_bounds** [bool, optional] Whether or not to save past distributions used to bound the live points internally. Default is `True`.

### Returns

**worst** [int] Index of the live point with the worst likelihood. This is our new dead point sample. **Negative values indicate the index of a new live point generated when initializing a new batch.**

**ustar** [ndarray with shape (npdim,)] Position of the sample.

**vstar** [ndarray with shape (ndim,)] Transformed position of the sample.

**loglstar** [float]  $\ln(\text{likelihood})$  of the sample.

**nc** [int] Number of likelihood calls performed before the new live point was accepted.

**worst\_it** [int] Iteration when the live (now dead) point was originally proposed.

**boundidx** [int] Index of the bound the dead point was originally drawn from.

**bounditer** [int] Index of the bound being used at the current iteration.

**eff** [float] The cumulative sampling efficiency (in percent).

**sample\_initial** (*self*, *nlive*=500, *update\_interval*=None, *first\_update*=None, *maxiter*=None, *maxcall*=None, *logl\_max*=inf, *dlogz*=0.01, *n\_effective*=inf, *live\_points*=None)

Generate a series of initial samples from a nested sampling run using a fixed number of live points using an internal sampler from *nestedsamplers*. Instantiates a generator that will be called by the user.

### Parameters

**nlive** [int, optional] The number of live points to use for the baseline nested sampling run. Default is 500.

**update\_interval** [int or float, optional] If an integer is passed, only update the bounding distribution every *update\_interval*-th likelihood call. If a float is passed, update the bound after every  $\text{round}(\text{update\_interval} * \text{nlive})$ -th likelihood call. Larger update intervals can be more efficient when the likelihood function is quick to evaluate. If no value is provided, defaults to the value passed during initialization.

**first\_update** [dict, optional] A dictionary containing parameters governing when the sampler will first update the bounding distribution from the unit cube ('none') to the one specified by *sample*.

**maxiter** [int, optional] Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is `sys.maxsize` (no limit).

**maxcall** [int, optional] Maximum number of likelihood evaluations. Iteration may stop earlier if termination condition is reached. Default is `sys.maxsize` (no limit).



**dlogz** [float, optional] Iteration will stop when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is  $\ln(z + z_{\text{est}}) - \ln(z) < \text{dlogz}$ , where  $z$  is the current evidence from all saved samples and  $z_{\text{est}}$  is the estimated contribution from the remaining volume. The default is 0.01.

**logl\_max** [float, optional] Iteration will stop when the sampled  $\ln(\text{likelihood})$  exceeds the threshold set by `logl_max`. Default is no bound (`np.inf`).

**n\_effective: int, optional** Target number of effective posterior samples. If the estimated “effective sample size” (ESS) exceeds this number, sampling will terminate. Default is no ESS (`np.inf`).

**live\_points** [list of 3 `ndarray` each with shape (nlive, ndim)] A set of live points used to initialize the nested sampling run. Contains `live_u`, the coordinates on the unit cube, `live_v`, the transformed variables, and `live_logl`, the associated loglikelihoods. By default, if these are not provided the initial set of live points will be drawn from the unit `npdim`-cube. **WARNING: It is crucial that the initial set of live points have been sampled from the prior. Failure to provide a set of valid live points will lead to incorrect results.**

### Returns

**worst** [int] Index of the live point with the worst likelihood. This is our new dead point sample.

**ustar** [`ndarray` with shape (npdim,)] Position of the sample.

**vstar** [`ndarray` with shape (ndim,)] Transformed position of the sample.

**loglstar** [float]  $\ln(\text{likelihood})$  of the sample.

**logvol** [float]  $\ln(\text{prior volume})$  within the sample.

**logwt** [float]  $\ln(\text{weight})$  of the sample.

**logz** [float] Cumulative  $\ln(\text{evidence})$  up to the sample (inclusive).

**logzvar** [float] Estimated cumulative variance on `logz` (inclusive).

**h** [float] Cumulative information up to the sample (inclusive).

**nc** [int] Number of likelihood calls performed before the new live point was accepted.

**worst\_it** [int] Iteration when the live (now dead) point was originally proposed.

**boundidx** [int] Index of the bound the dead point was originally drawn from.

**bounditer** [int] Index of the bound being used at the current iteration.

**eff** [float] The cumulative sampling efficiency (in percent).

**delta\_logz** [float] The estimated remaining evidence expressed as the  $\ln(\text{ratio})$  of the current evidence.

`dynesty.dynamicsampler.weight_function(results, args=None, return_weights=False)`

The default weight function utilized by `DynamicSampler`. Zipped parameters are passed to the function via `args`. Assigns each point a weight based on a weighted average of the posterior and evidence information content:

```
weight = pfrac * pweight + (1. - pfrac) * zweight
```

where `pfrac` is the fractional importance placed on the posterior, the evidence weight `zweight` is based on the estimated remaining posterior mass, and the posterior weight `pweight` is the sample’s importance weight.



Returns a set of log-likelihood bounds set by the earliest/latest samples where `weight > maxfrac * max(weight)`, with additional left/right padding based on `pad`.

#### Parameters

**results** [Results instance] Results instance.

**args** [dictionary of keyword arguments, optional] Arguments used to set the log-likelihood bounds used for sampling, as described above. Default values are `pfrac = 0.8`, `maxfrac = 0.8`, and `pad = 1`.

**return\_weights** [bool, optional] Whether to return the individual weights (and their components) used to compute the log-likelihood bounds. Default is `False`.

#### Returns

**logl\_bounds** [tuple with shape (2,)] Log-likelihood bounds (`logl_min`, `logl_max`) determined by the weights.

**weights** [tuple with shape (3,), optional] The individual weights (`pweight`, `zweight`, `weight`) used to determine `logl_bounds`.

`dynesty.dynamicsampler.stopping_function(results, args=None, rstate=None, M=None, return_vals=False)`

The default stopping function utilized by *DynamicSampler*. Zipped parameters are passed to the function via `args`. Assigns the run a stopping value based on a weighted average of the stopping values for the posterior and evidence:

```
stop = pfrac * stop_post + (1.- pfrac) * stop_evid
```

The evidence stopping value is based on the estimated evidence error (i.e. standard deviation) relative to a given threshold:

```
stop_evid = evid_std / evid_thresh
```

The posterior stopping value is based on the fractional error (i.e. standard deviation / mean) in the Kullback-Leibler (KL) divergence relative to a given threshold:

```
stop_post = (kld_std / kld_mean) / post_thresh
```

Estimates of the mean and standard deviation are computed using `n_mc` realizations of the input using a provided 'error' keyword (either 'jitter' or 'simulate', which call related functions `jitter_run()` and `simulate_run()` in *dynesty.utils*, respectively, or 'sim\_approx', which boosts 'jitter' by a factor of two).

Returns the boolean `stop <= 1`. If `True`, the *DynamicSampler* will stop adding new samples to our results.

#### Parameters

**results** [Results instance] Results instance.

**args** [dictionary of keyword arguments, optional] Arguments used to set the stopping values. Default values are `pfrac = 1.0`, `evid_thresh = 0.1`, `post_thresh = 0.02`, `n_mc = 128`, `error = 'sim_approx'`, and `approx = True`.

**rstate** [RandomState, optional] RandomState instance.

**M** [map function, optional] An alias to a map-like function. This allows users to pass functions from pools (e.g., `pool.map`) to compute realizations in parallel. By default the standard `map` function is used.

**return\_vals** [bool, optional] Whether to return the stopping value (and its components). Default is `False`.

#### Returns

**stop\_flag** [bool] Boolean flag indicating whether we have passed the desired stopping criteria.

**stop\_vals** [tuple of shape (3,), optional] The individual stopping values (`stop_post`, `stop_evid`, `stop`) used to determine the stopping criteria.

`dynesty.dynamicsampler._kld_error(args)`

Internal pool.map-friendly wrapper for `kld_error()` used by `stopping_function()`.

## Sampling Results

Utilities for handling results.

**class** `dynesty.results.Results`

Bases: `dict`

Contains the full output of a run along with a set of helper functions for summarizing the output.

**summary** (*self*)

Return a formatted string giving a quick summary of the results.

`dynesty.results.print_fn(results, niter, ncall, add_live_it=None, dlogz=None, stop_val=None, nbatch=None, logl_min=-inf, logl_max=inf, pbar=None)`

The default function used to print out results in real time.

#### Parameters

**results** [tuple] Collection of variables output from the current state of the sampler. Currently includes: (1) particle index, (2) unit cube position, (3) parameter position, (4)  $\ln(\text{likelihood})$ , (5)  $\ln(\text{volume})$ , (6)  $\ln(\text{weight})$ , (7)  $\ln(\text{evidence})$ , (8)  $\text{Var}[\ln(\text{evidence})]$ , (9) information, (10) number of (current) function calls, (11) iteration when the point was originally proposed, (12) index of the bounding object originally proposed from, (13) index of the bounding object active at a given iteration, (14) cumulative efficiency, and (15) estimated remaining  $\ln(\text{evidence})$ .

**niter** [int] The current iteration of the sampler.

**ncall** [int] The total number of function calls at the current iteration.

**add\_live\_it** [int, optional] If the last set of live points are being added explicitly, this quantity tracks the sorted index of the current live point being added.

**dlogz** [float, optional] The evidence stopping criterion. If not provided, the provided stopping value will be used instead.

**stop\_val** [float, optional] The current stopping criterion (for dynamic nested sampling). Used if the `dlogz` value is not specified.

**nbatch** [int, optional] The current batch (for dynamic nested sampling).

**logl\_min** [float, optional] The minimum log-likelihood used when starting sampling. Default is `-np.inf`.

**logl\_max** [float, optional] The maximum log-likelihood used when stopping sampling. Default is `np.inf`.

## Useful Helper Functions

A collection of useful functions.

`dynesty.utils.unitcheck(u, nonbounded=None)`

Check whether `u` is inside the unit cube. Given a masked array `nonbounded`, also allows periodic boundaries conditions to exceed the unit cube.

`dynesty.utils.resample_equal(samples, weights, rstate=None)`

Resample a new set of points from the weighted set of inputs such that they all have equal weight.

Each input sample appears in the output array either `floor(weights[i] * nsamples)` or `ceil(weights[i] * nsamples)` times, with `floor` or `ceil` randomly selected (weighted by proximity).

### Parameters

**samples** [`ndarray` with shape (nsamples,)] Set of unequally weighted samples.

**weights** [`ndarray` with shape (nsamples,)] Corresponding weight of each sample.

**rstate** [`RandomState`, optional] `RandomState` instance.

### Returns

**equal\_weight\_samples** [`ndarray` with shape (nsamples,)] New set of samples with equal weights.

## Notes

Implements the systematic resampling method described in [Hol, Schon, and Gustafsson \(2006\)](#).

## Examples

```
>>> x = np.array([[1., 1.], [2., 2.], [3., 3.], [4., 4.]])
>>> w = np.array([0.6, 0.2, 0.15, 0.05])
>>> utils.resample_equal(x, w)
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 3.,  3.]])
```

`dynesty.utils.mean_and_cov(samples, weights)`

Compute the weighted mean and covariance of the samples.

### Parameters

**samples** [`ndarray` with shape (nsamples, ndim)] 2-D array containing data samples. This ordering is equivalent to using `rowvar=False` in `cov`.

**weights** [`ndarray` with shape (nsamples,)] 1-D array of sample weights.

### Returns

**mean** [`ndarray` with shape (ndim,)] Weighted sample mean vector.

**cov** [`ndarray` with shape (ndim, ndim)] Weighted sample covariance matrix.

## Notes

Implements the formulae found [here](#).

`dynesty.utils.quantile(x, q, weights=None)`

Compute (weighted) quantiles from an input set of samples.

### Parameters

**x** [`ndarray` with shape (nsamps,)] Input samples.

**q** [`ndarray` with shape (nquantiles,)] The list of quantiles to compute from `[0., 1.]`.

**weights** [`ndarray` with shape (nsamps,), optional] The associated weight from each sample.

### Returns

**quantiles** [`ndarray` with shape (nquantiles,)] The weighted sample quantiles computed at `q`.

`dynesty.utils.jitter_run(res, rstate=None, approx=False)`

Probes **statistical uncertainties** on a nested sampling run by explicitly generating a *realization* of the prior volume associated with each sample (dead point). Companion function to `resample_run()` and `simulate_run()`.

### Parameters

**res** [`Results` instance] The `Results` instance taken from a previous nested sampling run.

**rstate** [`RandomState`, optional] `RandomState` instance.

**approx** [bool, optional] Whether to approximate all sets of uniform order statistics by their associated marginals (from the Beta distribution). Default is `False`.

### Returns

**new\_res** [`Results` instance] A new `Results` instance with corresponding weights based on our “jittered” prior volume realizations.

`dynesty.utils.resample_run(res, rstate=None, return_idx=False)`

Probes **sampling uncertainties** on a nested sampling run using bootstrap resampling techniques to generate a *realization* of the (expected) prior volume(s) associated with each sample (dead point). This effectively splits a nested sampling run with `K` particles (live points) into a series of `K` “strands” (i.e. runs with a single live point) which are then bootstrapped to construct a new “resampled” run. Companion function to `jitter_run()` and `simulate_run()`.

### Parameters

**res** [`Results` instance] The `Results` instance taken from a previous nested sampling run.

**rstate** [`RandomState`, optional] `RandomState` instance.

**return\_idx** [bool, optional] Whether to return the list of resampled indices used to construct the new run. Default is `False`.

### Returns

**new\_res** [`Results` instance] A new `Results` instance with corresponding samples and weights based on our “bootstrapped” samples and (expected) prior volumes.

`dynesty.utils.simulate_run(res, rstate=None, return_idx=False, approx=False)`

Probes **combined uncertainties** (statistical and sampling) on a nested sampling run by wrapping `jitter_run()` and `resample_run()`.

### Parameters

**res** [`Results` instance] The `Results` instance taken from a previous nested sampling run.

**rstate** [`RandomState`, optional] `RandomState` instance.

**return\_idx** [`bool`, optional] Whether to return the list of resampled indices used to construct the new run. Default is `False`.

**approx** [`bool`, optional] Whether to approximate all sets of uniform order statistics by their associated marginals (from the Beta distribution). Default is `False`.

#### Returns

**new\_res** [`Results` instance] A new `Results` instance with corresponding samples and weights based on our “simulated” samples and prior volumes.

`dynesty.utils.reweight_run(res, logp_new, logp_old=None)`

Reweight a given run based on a new target distribution.

#### Parameters

**res** [`Results` instance] The `Results` instance taken from a previous nested sampling run.

**logp\_new** [`ndarray` with shape (nsamps,)] New target distribution evaluated at the location of the samples.

**logp\_old** [`ndarray` with shape (nsamps,)] Old target distribution evaluated at the location of the samples. If not provided, the `logl` values from `res` will be used.

#### Returns

**new\_res** [`Results` instance] A new `Results` instance with corresponding weights based on our reweighted samples.

`dynesty.utils.unravel_run(res, save_proposals=True, print_progress=True)`

Unravels a run with  $K$  live points into  $K$  “strands” (a nested sampling run with only 1 live point). **WARNING: the ancillary quantities provided with each unraveled “strand” are only valid if the point was initialized from the prior.**

#### Parameters

**res** [`Results` instance] The `Results` instance taken from a previous nested sampling run.

**save\_proposals** [`bool`, optional] Whether to save a reference to the proposal distributions from the original run in each unraveled strand. Default is `True`.

**print\_progress** [`bool`, optional] Whether to output the current progress to `stderr`. Default is `True`.

#### Returns

**new\_res** [list of `Results` instances] A list of new `Results` instances for each individual strand.

`dynesty.utils.merge_runs(res_list, print_progress=True)`

Merges a set of runs with differing (possibly variable) numbers of live points into one run.

#### Parameters

**res\_list** [list of `Results` instances] A list of `Results` instances returned from previous runs.

**print\_progress** [`bool`, optional] Whether to output the current progress to `stderr`. Default is `True`.

#### Returns

**combined\_res** [`Results` instance] The `Results` instance for the combined run.

`dynesty.utils.kl_divergence(res1, res2)`

Computes the **Kullback-Leibler (KL) divergence** from the discrete probability distribution defined by `res2` to the discrete probability distribution defined by `res1`.

#### Parameters

- res1** [*Results* instance] *Results* instance for the distribution we are computing the KL divergence to. **Note that, by construction, the samples in ‘res1’ \*must\* be a subset of the samples in ‘res2’.**
- res2** [*Results* instance] *Results* instance for the distribution we are computing the KL divergence from. **Note that, by construction, the samples in ‘res2’ \*must\* be a superset of the samples in ‘res1’.**

#### Returns

**kld** [*ndarray* with shape (nsamps,)] The cumulative KL divergence defined over `res1`.

`dynesty.utils.kld_error(res, error='simulate', rstate=None, return_new=False, approx=False)`

Computes the **Kullback-Leibler (KL) divergence** from the discrete probability distribution defined by `res` to the discrete probability distribution defined by a **realization** of `res`.

#### Parameters

- res** [*Results* instance] *Results* instance for the distribution we are computing the KL divergence from.
- error** [{'jitter', 'resample', 'simulate'}, optional] The error method employed, corresponding to `jitter_run()`, `resample_run()`, and `simulate_run()`, respectively. Default is 'simulate'.
- rstate** [*RandomState*, optional] *RandomState* instance.
- return\_new** [bool, optional] Whether to return the realization of the run used to compute the KL divergence. Default is `False`.
- approx** [bool, optional] Whether to approximate all sets of uniform order statistics by their associated marginals (from the Beta distribution). Default is `False`.

#### Returns

**kld** [*ndarray* with shape (nsamps,)] The cumulative KL divergence defined from `res` to a random realization of `res`.

**new\_res** [*Results* instance, optional] The *Results* instance corresponding to the random realization we computed the KL divergence to.

`dynesty.utils._merge_two(res1, res2, compute_aux=False)`

Internal method used to merges two runs with differing (possibly variable) numbers of live points into one run.

#### Parameters

- res1** [*Results* instance] The “base” nested sampling run.
- res2** [*Results* instance] The “new” nested sampling run.
- compute\_aux** [bool, optional] Whether to compute auxiliary quantities (evidences, etc.) associated with a given run. **WARNING: these are only valid if ‘res1’ or ‘res2’ was initialized from the prior \*and\* their sampling bounds overlap.** Default is `False`.

#### Returns

**res** [*Results* instances] *Results* instance from the newly combined nested sampling run.

`dynesty.utils._get_nsamps_samples_n(res)`  
 Helper function for calculating the number of samples

#### Parameters

**res** [*Results* instance] The *Results* instance taken from a previous nested sampling run.

#### Returns

**nsamps: int** The total number of samples  
**samples\_n: array** Number of live points at a given iteration

## Plotting Utilities

A set of built-in plotting functions to help visualize dynesty nested sampling *Results*.

`dynesty.plotting.runplot(results, span=None, logplot=False, kde=True, nkde=1000, color='blue', plot_kwargs=None, label_kwargs=None, lnz_error=True, lnz_truth=None, truth_color='red', truth_kwargs=None, max_x_ticks=8, max_y_ticks=3, use_math_text=True, mark_final_live=True, fig=None)`  
 Plot live points, ln(likelihood), ln(weight), and ln(evidence) as a function of ln(prior volume).

#### Parameters

**results** [*Results* instance] A *Results* instance from a nested sampling run.

**span** [iterable with shape (4,), optional] A list where each element is either a length-2 tuple containing lower and upper bounds *or* a float from `(0., 1.]` giving the fraction below the maximum. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

```
span = [(0., 10.), 0.001, 0.2, (5., 6.)]
```

Default is `(0., 1.05 * max(data))` for each element.

**logplot** [bool, optional] Whether to plot the evidence on a log scale. Default is `False`.

**kde** [bool, optional] Whether to use kernel density estimation to estimate and plot the PDF of the importance weights as a function of log-volume (as opposed to the importance weights themselves). Default is `True`.

**nkde** [int, optional] The number of grid points used when plotting the kernel density estimate. Default is 1000.

**color** [str or iterable with shape (4,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting the lines in each subplot. Default is `'blue'`.

**plot\_kwargs** [dict, optional] Extra keyword arguments that will be passed to `plot`.

**label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.

**lnz\_error** [bool, optional] Whether to plot the 1, 2, and 3-sigma approximate error bars derived from the ln(evidence) error approximation over the course of the run. Default is `True`.

**lnz\_truth** [float, optional] A reference value for the evidence that will be overplotted on the evidence subplot if provided.

**truth\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color used when plotting `lnz_truth`. Default is `'red'`.

- truth\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting `lnz_truth`.
- max\_x\_ticks** [int, optional] Maximum number of ticks allowed for the x axis. Default is 8.
- max\_y\_ticks** [int, optional] Maximum number of ticks allowed for the y axis. Default is 4.
- use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using `e`. Default is `False`.
- mark\_final\_live** [bool, optional] Whether to indicate the final addition of recycled live points (if they were added to the resulting samples) using a dashed vertical line. Default is `True`.
- fig** [(Figure, Axes), optional] If provided, overplot the run onto the provided figure. Otherwise, by default an internal figure is generated.

### Returns

**runplot** [(Figure, Axes)] Output summary plot.

`dynesty.plotting.traceplot` (*results*, *span=None*, *quantiles=[0.025, 0.5, 0.975]*, *smooth=0.02*, *post\_color='blue'*, *post\_kwargs=None*, *kde=True*, *nkde=1000*, *trace\_cmap='plasma'*, *trace\_color=None*, *trace\_kwargs=None*, *connect=False*, *connect\_highlight=10*, *connect\_color='red'*, *connect\_kwargs=None*, *max\_n\_ticks=5*, *use\_math\_text=False*, *labels=None*, *label\_kwargs=None*, *show\_titles=False*, *title\_fmt='.2f'*, *title\_kwargs=None*, *truths=None*, *truth\_color='red'*, *truth\_kwargs=None*, *verbose=False*, *fig=None*)

Plot traces and marginalized posteriors for each parameter.

### Parameters

**results** [*Results* instance] A *Results* instance from a nested sampling run. **Compatible with results derived from `nestle`.**

**span** [iterable with shape (ndim,), optional] A list where each element is either a length-2 tuple containing lower and upper bounds or a float from `(0., 1.]` giving the fraction of (weighted) samples to include. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

```
span = [(0., 10.), 0.95, (5., 6.)]
```

Default is 0.999999426697 (5-sigma credible interval) for each parameter.

**quantiles** [iterable, optional] A list of fractional quantiles to overplot on the 1-D marginalized posteriors as vertical dashed lines. Default is `[0.025, 0.5, 0.975]` (the 95%/2-sigma credible interval).

**smooth** [float or iterable with shape (ndim,), optional] The standard deviation (either a single value or a different value for each subplot) for the Gaussian kernel used to smooth the 1-D marginalized posteriors, expressed as a fraction of the span. Default is 0.02 (2% smoothing). If an integer is provided instead, this will instead default to a simple (weighted) histogram with `bins=smooth`.

**post\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting the histograms. Default is `'blue'`.

**post\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the marginalized 1-D posteriors.



- kde** [bool, optional] Whether to use kernel density estimation to estimate and plot the PDF of the importance weights as a function of log-volume (as opposed to the importance weights themselves). Default is `True`.
- nkde** [int, optional] The number of grid points used when plotting the kernel density estimate. Default is 1000.
- trace\_cmap** [str or iterable with shape (ndim,), optional] A `matplotlib`-style colormap (either a single colormap or a different colormap for each subplot) used when plotting the traces, where each point is colored according to its weight. Default is `'plasma'`.
- trace\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different color for each subplot) used when plotting the traces. This overrides the `trace_cmap` option by giving all points the same color. Default is `None` (not used).
- trace\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the traces.
- connect** [bool, optional] Whether to draw lines connecting the paths of unique particles. Default is `False`.
- connect\_highlight** [int or iterable, optional] If `connect=True`, highlights the paths of a specific set of particles. If an integer is passed, `connect_highlight` random particle paths will be highlighted. If an iterable is passed, then the particle paths corresponding to the provided indices will be highlighted.
- connect\_color** [str, optional] The color of the highlighted particle paths. Default is `'red'`.
- connect\_kwargs** [dict, optional] Extra keyword arguments used for plotting particle paths.
- max\_n\_ticks** [int, optional] Maximum number of ticks allowed. Default is 5.
- use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using `e`. Default is `False`.
- labels** [iterable with shape (ndim,), optional] A list of names for each parameter. If not provided, the default name used when plotting will follow  $x_i$  style.
- label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.
- show\_titles** [bool, optional] Whether to display a title above each 1-D marginalized posterior showing the 0.5 quantile along with the upper/lower bounds associated with the 0.025 and 0.975 (95%/2-sigma credible interval) quantiles. Default is `True`.
- title\_fmt** [str, optional] The format string for the quantiles provided in the title. Default is `'.2f'`.
- title\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_title` command.
- truths** [iterable with shape (ndim,), optional] A list of reference values that will be overplotted on the traces and marginalized 1-D posteriors as solid horizontal/vertical lines. Individual values can be exempt using `None`. Default is `None`.
- truth\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting `truths`. Default is `'red'`.
- truth\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the vertical and horizontal lines with `truths`.

**verbose** [bool, optional] Whether to print the values of the computed quantiles associated with each parameter. Default is `False`.

**fig** [(Figure, Axes), optional] If provided, overplot the traces and marginalized 1-D posteriors onto the provided figure. Otherwise, by default an internal figure is generated.

## Returns

**traceplot** [(Figure, Axes)] Output trace plot.

`dynesty.plotting.cornerpoints`(*results*, *dims=None*, *thin=1*, *span=None*, *cmap='plasma'*, *color=None*, *kde=True*, *nkde=1000*, *plot\_kwargs=None*, *labels=None*, *label\_kwargs=None*, *truths=None*, *truth\_color='red'*, *truth\_kwargs=None*, *max\_n\_ticks=5*, *use\_math\_text=False*, *fig=None*)

Generate a (sub-)corner plot of (weighted) samples.

## Parameters

**results** [*Results* instance] A *Results* instance from a nested sampling run. **Compatible with results derived from nestle.**

**dims** [iterable of shape (ndim,), optional] The subset of dimensions that should be plotted. If not provided, all dimensions will be shown.

**thin** [int, optional] Thin the samples so that only each *thin*-th sample is plotted. Default is 1 (no thinning).

**span** [iterable with shape (ndim,), optional] A list where each element is either a length-2 tuple containing lower and upper bounds or a float from (0., 1.] giving the fraction of (weighted) samples to include. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

```
span = [(0., 10.), 0.95, (5., 6.)]
```

Default is 1. for all parameters (no bound).

**cmap** [str, optional] A `matplotlib`-style colormap used when plotting the points, where each point is colored according to its weight. Default is 'plasma'.

**color** [str, optional] A `matplotlib`-style color used when plotting the points. This overrides the `cmap` option by giving all points the same color. Default is `None` (not used).

**kde** [bool, optional] Whether to use kernel density estimation to estimate and plot the PDF of the importance weights as a function of log-volume (as opposed to the importance weights themselves). Default is `True`.

**nkde** [int, optional] The number of grid points used when plotting the kernel density estimate. Default is 1000.

**plot\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the points.

**labels** [iterable with shape (ndim,), optional] A list of names for each parameter. If not provided, the default name used when plotting will follow  $x_i$  style.

**label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.

**truths** [iterable with shape (ndim,), optional] A list of reference values that will be overplotted on the traces and marginalized 1-D posteriors as solid horizontal/vertical lines. Individual values can be exempt using `None`. Default is `None`.

**truth\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting `truths`. Default is 'red'.

**truth\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the vertical and horizontal lines with `truths`.

**max\_n\_ticks** [int, optional] Maximum number of ticks allowed. Default is 5.

**use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using `e`. Default is `False`.

**fig** [(`Figure`, `Axes`), optional] If provided, overplot the points onto the provided figure object. Otherwise, by default an internal figure is generated.

## Returns

**cornerpoints** [(`Figure`, `Axes`)] Output (sub-)corner plot of (weighted) samples.

```
dynesty.plotting.cornerplot(results, dims=None, span=None, quantiles=[0.025, 0.5, 0.975], color='black', smooth=0.02, quantiles_2d=None, hist_kwargs=None, hist2d_kwargs=None, labels=None, label_kwargs=None, show_titles=False, title_fmt='.2f', title_kwargs=None, truths=None, truth_color='red', truth_kwargs=None, max_n_ticks=5, top_ticks=False, use_math_text=False, verbose=False, fig=None)
```

Generate a corner plot of the 1-D and 2-D marginalized posteriors.

## Parameters

**results** [`Results` instance] A `Results` instance from a nested sampling run. **Compatible with results derived from `nestle`.**

**dims** [iterable of shape (ndim,), optional] The subset of dimensions that should be plotted. If not provided, all dimensions will be shown.

**span** [iterable with shape (ndim,), optional] A list where each element is either a length-2 tuple containing lower and upper bounds or a float from `(0., 1.]` giving the fraction of (weighted) samples to include. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

```
span = [(0., 10.), 0.95, (5., 6.)]
```

Default is `0.999999426697` (5-sigma credible interval).

**quantiles** [iterable, optional] A list of fractional quantiles to overplot on the 1-D marginalized posteriors as vertical dashed lines. Default is `[0.025, 0.5, 0.975]` (spanning the 95%/2-sigma credible interval).

**color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting the histograms. Default is 'black'.

**smooth** [float or iterable with shape (ndim,), optional] The standard deviation (either a single value or a different value for each subplot) for the Gaussian kernel used to smooth the 1-D and 2-D marginalized posteriors, expressed as a fraction of the span. Default is `0.02` (2% smoothing). If an integer is provided instead, this will instead default to a simple (weighted) histogram with `bins=smooth`.

**quantiles\_2d** [iterable with shape (nquant,), optional] The quantiles used for plotting the smoothed 2-D distributions. If not provided, these default to 0.5, 1, 1.5, and 2-sigma contours roughly corresponding to quantiles of `[0.1, 0.4, 0.65, 0.85]`.

- hist\_kwargs** [dict, optional] Extra keyword arguments to send to the 1-D (smoothed) histograms.
- hist2d\_kwargs** [dict, optional] Extra keyword arguments to send to the 2-D (smoothed) histograms.
- labels** [iterable with shape (ndim,), optional] A list of names for each parameter. If not provided, the default name used when plotting will follow  $x_i$  style.
- label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.
- show\_titles** [bool, optional] Whether to display a title above each 1-D marginalized posterior showing the 0.5 quantile along with the upper/lower bounds associated with the 0.025 and 0.975 (95%/2-sigma credible interval) quantiles. Default is `True`.
- title\_fmt** [str, optional] The format string for the quantiles provided in the title. Default is `'.2f'`.
- title\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_title` command.
- truths** [iterable with shape (ndim,), optional] A list of reference values that will be overplotted on the traces and marginalized 1-D posteriors as solid horizontal/vertical lines. Individual values can be exempt using `None`. Default is `None`.
- truth\_color** [str or iterable with shape (ndim,), optional] A `matplotlib`-style color (either a single color or a different value for each subplot) used when plotting `truths`. Default is `'red'`.
- truth\_kwargs** [dict, optional] Extra keyword arguments that will be used for plotting the vertical and horizontal lines with `truths`.
- max\_n\_ticks** [int, optional] Maximum number of ticks allowed. Default is 5.
- top\_ticks** [bool, optional] Whether to label the top (rather than bottom) ticks. Default is `False`.
- use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using `e`. Default is `False`.
- verbose** [bool, optional] Whether to print the values of the computed quantiles associated with each parameter. Default is `False`.
- fig** [(`Figure`, `Axes`), optional] If provided, overplot the traces and marginalized 1-D posteriors onto the provided figure. Otherwise, by default an internal figure is generated.

## Returns

**cornerplot** [(`Figure`, `Axes`)] Output corner plot.

`dynesty.plotting.boundplot` (*results*, *dims*, *it=None*, *idx=None*, *prior\_transform=None*, *periodic=None*, *reflective=None*, *ndraws=5000*, *color='gray'*, *plot\_kwargs=None*, *labels=None*, *label\_kwargs=None*, *max\_n\_ticks=5*, *use\_math\_text=False*, *show\_live=False*, *live\_color='darkviolet'*, *live\_kwargs=None*, *span=None*, *fig=None*)

Return the bounding distribution used to propose either (1) live points at a given iteration or (2) a specific dead point during the course of a run, projected onto the two dimensions specified by *dims*.

## Parameters

- results** [`Results` instance] A `Results` instance from a nested sampling run.
- dims** [length-2 tuple] The dimensions used to plot the bounding.

- it** [int, optional] If provided, returns the bounding distribution at the specified iteration of the nested sampling run. **Note that this option and ‘idx’ are mutually exclusive.**
- idx** [int, optional] If provided, returns the bounding distribution used to propose the dead point at the specified iteration of the nested sampling run. **Note that this option and ‘it’ are mutually exclusive.**
- prior\_transform** [func, optional] The function transforming samples within the unit cube back to samples in the native model space. If provided, the transformed bounding distribution will be plotted in the native model space.
- periodic** [iterable, optional] A list of indices for parameters with periodic boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may wrap around the edge. Default is `None` (i.e. no periodic boundary conditions).
- reflective** [iterable, optional] A list of indices for parameters with reflective boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may reflect at the edge. Default is `None` (i.e. no reflective boundary conditions).
- ndraws** [int, optional] The number of random samples to draw from the bounding distribution when plotting. Default is 5000.
- color** [str, optional] The color of the points randomly sampled from the bounding distribution. Default is 'gray'.
- plot\_kwargs** [dict, optional] Extra keyword arguments used when plotting the bounding draws.
- labels** [iterable with shape (ndim,), optional] A list of names for each parameter. If not provided, the default name used when plotting will follow  $x_i$  style.
- label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.
- max\_n\_ticks** [int, optional] Maximum number of ticks allowed. Default is 5.
- use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using  $e$ . Default is `False`.
- show\_live** [bool, optional] Whether the live points at a given iteration (for `it`) or associated with the bounding (for `idx`) should be highlighted. Default is `False`. In the dynamic case, only the live points associated with the batch used to construct the relevant bound are plotted.
- live\_color** [str, optional] The color of the live points. Default is 'darkviolet'.
- live\_kwargs** [dict, optional] Extra keyword arguments used when plotting the live points.
- span** [iterable with shape (2,), optional] A list where each element is a length-2 tuple containing lower and upper bounds. Default is `None` (no bound).
- fig** [(`Figure`, `Axes`), optional] If provided, overplot the draws onto the provided figure. Otherwise, by default an internal figure is generated.

## Returns

- bounding\_plot** [(`Figure`, `Axes`)] Output plot of the bounding distribution.

```
dynesty.plotting.cornerbound(results, it=None, idx=None, dims=None, prior_transform=None,
                             periodic=None, reflective=None, ndraws=5000, color='gray',
                             plot_kwargs=None, labels=None, label_kwargs=None,
                             max_n_ticks=5, use_math_text=False, show_live=False,
                             live_color='darkviolet', live_kwargs=None, span=None,
                             fig=None)
```

Return the bounding distribution used to propose either (1) live points at a given iteration or (2) a specific dead point during the course of a run, projected onto all pairs of dimensions.

### Parameters

- results** [*Results* instance] A *Results* instance from a nested sampling run.
- it** [int, optional] If provided, returns the bounding distribution at the specified iteration of the nested sampling run. **Note that this option and ‘idx’ are mutually exclusive.**
- idx** [int, optional] If provided, returns the bounding distribution used to propose the dead point at the specified iteration of the nested sampling run. **Note that this option and ‘it’ are mutually exclusive.**
- dims** [iterable of shape (ndim,), optional] The subset of dimensions that should be plotted. If not provided, all dimensions will be shown.
- prior\_transform** [func, optional] The function transforming samples within the unit cube back to samples in the native model space. If provided, the transformed bounding distribution will be plotted in the native model space.
- periodic** [iterable, optional] A list of indices for parameters with periodic boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may wrap around the edge. Default is *None* (i.e. no periodic boundary conditions).
- reflective** [iterable, optional] A list of indices for parameters with reflective boundary conditions. These parameters *will not* have their positions constrained to be within the unit cube, enabling smooth behavior for parameters that may reflect at the edge. Default is *None* (i.e. no reflective boundary conditions).
- ndraws** [int, optional] The number of random samples to draw from the bounding distribution when plotting. Default is 5000.
- color** [str, optional] The color of the points randomly sampled from the bounding distribution. Default is 'gray'.
- plot\_kwargs** [dict, optional] Extra keyword arguments used when plotting the bounding draws.
- labels** [iterable with shape (ndim,), optional] A list of names for each parameter. If not provided, the default name used when plotting will be in  $x_i$  style.
- label\_kwargs** [dict, optional] Extra keyword arguments that will be sent to the `set_xlabel` and `set_ylabel` methods.
- max\_n\_ticks** [int, optional] Maximum number of ticks allowed. Default is 5.
- use\_math\_text** [bool, optional] Whether the axis tick labels for very large/small exponents should be displayed as powers of 10 rather than using e. Default is *False*.
- show\_live** [bool, optional] Whether the live points at a given iteration (for *it*) or associated with the bounding (for *idx*) should be highlighted. Default is *False*. In the dynamic case, only the live points associated with the batch used to construct the relevant bound are plotted.
- live\_color** [str, optional] The color of the live points. Default is 'darkviolet'.

**live\_kwargs** [dict, optional] Extra keyword arguments used when plotting the live points.

**span** [iterable with shape (2,), optional] A list where each element is a length-2 tuple containing lower and upper bounds. Default is `None` (no bound).

**fig** [(`Figure`, `Axes`), optional] If provided, overplot the draws onto the provided figure. Otherwise, by default an internal figure is generated.

### Returns

**cornerbound** [(`Figure`, `Axes`)] Output corner plot of the bounding distribution.

`dynesty.plotting._hist2d(x, y, smooth=0.02, span=None, weights=None, levels=None, ax=None, color='gray', plot_datapoints=False, plot_density=True, plot_contours=True, no_fill_contours=False, fill_contours=True, contour_kwargs=None, contourf_kwargs=None, data_kwargs=None, **kwargs)`

Internal function called by `cornerplot()` used to generate a 2-D histogram/contour of samples.

### Parameters

**x** [iterable with shape (nsamps,)] Sample positions in the first dimension.

**y** [iterable with shape (nsamps,)] Sample positions in the second dimension.

**span** [iterable with shape (ndim,), optional] A list where each element is either a length-2 tuple containing lower and upper bounds or a float from `(0., 1.]` giving the fraction of (weighted) samples to include. If a fraction is provided, the bounds are chosen to be equal-tailed. An example would be:

```
span = [(0., 10.), 0.95, (5., 6.)]
```

Default is `0.999999426697` (5-sigma credible interval).

**weights** [iterable with shape (nsamps,)] Weights associated with the samples. Default is `None` (no weights).

**levels** [iterable, optional] The contour levels to draw. Default are `[0.5, 1, 1.5, 2]`-sigma.

**ax** [`Axes`, optional] An `axes` instance on which to add the 2-D histogram. If not provided, a figure will be generated.

**color** [str, optional] The `matplotlib`-style color used to draw lines and color cells and contours. Default is `'gray'`.

**plot\_datapoints** [bool, optional] Whether to plot the individual data points. Default is `False`.

**plot\_density** [bool, optional] Whether to draw the density colormap. Default is `True`.

**plot\_contours** [bool, optional] Whether to draw the contours. Default is `True`.

**no\_fill\_contours** [bool, optional] Whether to add absolutely no filling to the contours. This differs from `fill_contours=False`, which still adds a white fill at the densest points. Default is `False`.

**fill\_contours** [bool, optional] Whether to fill the contours. Default is `True`.

**contour\_kwargs** [dict] Any additional keyword arguments to pass to the `contour` method.

**contourf\_kwargs** [dict] Any additional keyword arguments to pass to the `contourf` method.

**data\_kwargs** [dict] Any additional keyword arguments to pass to the `plot` method when adding the individual data points.





### d

- `dynesty.bounding`, [108](#)
- `dynesty.dynamicsampler`, [127](#)
- `dynesty.dynesty`, [102](#)
- `dynesty.nestedsamplers`, [121](#)
- `dynesty.plotting`, [139](#)
- `dynesty.results`, [134](#)
- `dynesty.sampler`, [118](#)
- `dynesty.sampling`, [115](#)
- `dynesty.utils`, [135](#)



## Symbols

- `_beyond_unit_bound()` (*dynesty.sampler.Sampler method*), 119
  - `_bounding_ellipsoids()` (*in module dynesty.bounding*), 114
  - `_ellipsoid_bootstrap_expand()` (*in module dynesty.bounding*), 115
  - `_ellipsoids_bootstrap_expand()` (*in module dynesty.bounding*), 115
  - `_empty_queue()` (*dynesty.sampler.Sampler method*), 119
  - `_fill_queue()` (*dynesty.sampler.Sampler method*), 119
  - `_friends_bootstrap_radius()` (*in module dynesty.bounding*), 115
  - `_friends_leaveoneout_radius()` (*in module dynesty.bounding*), 115
  - `_function_wrapper` (*class in dynesty.dynesty*), 108
  - `_get_covariance_from_all_points()` (*dynesty.bounding.RadFriends method*), 111
  - `_get_covariance_from_all_points()` (*dynesty.bounding.SupFriends method*), 112
  - `_get_covariance_from_clusters()` (*dynesty.bounding.RadFriends method*), 111
  - `_get_covariance_from_clusters()` (*dynesty.bounding.SupFriends method*), 112
  - `_get_nsamps_samples_n()` (*in module dynesty.utils*), 138
  - `_get_point_value()` (*dynesty.sampler.Sampler method*), 119
  - `_get_print_func()` (*dynesty.dynamicsampler.DynamicSampler method*), 128
  - `_get_print_func()` (*dynesty.sampler.Sampler method*), 119
  - `_hist2d()` (*in module dynesty.plotting*), 147
  - `_kld_error()` (*in module dynesty.dynamicsampler*), 134
  - `_merge_two()` (*in module dynesty.utils*), 138
  - `_new_point()` (*dynesty.sampler.Sampler method*), 119
  - `_remove_live_points()` (*dynesty.sampler.Sampler method*), 119
- ## A
- `add_batch()` (*dynesty.dynamicsampler.DynamicSampler method*), 128
  - `add_final_live()` (*dynesty.sampler.Sampler method*), 119
  - `add_live_points()` (*dynesty.sampler.Sampler method*), 119
- ## B
- `bounding_ellipsoid()` (*in module dynesty.bounding*), 114
  - `bounding_ellipsoids()` (*in module dynesty.bounding*), 114
  - `boundplot()` (*in module dynesty.plotting*), 144
- ## C
- `combine_runs()` (*dynesty.dynamicsampler.DynamicSampler method*), 128
  - `contains()` (*dynesty.bounding.Ellipsoid method*), 109
  - `contains()` (*dynesty.bounding.MultiEllipsoid method*), 110
  - `contains()` (*dynesty.bounding.RadFriends method*), 111
  - `contains()` (*dynesty.bounding.SupFriends method*), 112
  - `contains()` (*dynesty.bounding.UnitCube method*), 108
  - `cornerbound()` (*in module dynesty.plotting*), 145
  - `cornerplot()` (*in module dynesty.plotting*), 143
  - `cornerpoints()` (*in module dynesty.plotting*), 142
- ## D
- `distance()` (*dynesty.bounding.Ellipsoid method*), 109

`DynamicNestedSampler()` (in module `dynesty.dynesty`), 105  
`DynamicSampler` (class in `dynesty.dynamicsampler`), 127  
`dynesty.bounding` (module), 108  
`dynesty.dynamicsampler` (module), 127  
`dynesty.dynesty` (module), 102  
`dynesty.nestedsamplers` (module), 121  
`dynesty.plotting` (module), 139  
`dynesty.results` (module), 134  
`dynesty.sampler` (module), 118  
`dynesty.sampling` (module), 115  
`dynesty.utils` (module), 135

## E

`Ellipsoid` (class in `dynesty.bounding`), 109

## J

`jitter_run()` (in module `dynesty.utils`), 136

## K

`kl_divergence()` (in module `dynesty.utils`), 137  
`kld_error()` (in module `dynesty.utils`), 138

## L

`logvol_prefactor()` (in module `dynesty.bounding`), 113

## M

`major_axis_endpoints()` (`dynesty.bounding.Ellipsoid` method), 109  
`major_axis_endpoints()` (`dynesty.bounding.MultiEllipsoid` method), 110  
`mean_and_cov()` (in module `dynesty.utils`), 135  
`merge_runs()` (in module `dynesty.utils`), 137  
`monte_carlo_vol()` (`dynesty.bounding.MultiEllipsoid` method), 110  
`monte_carlo_vol()` (`dynesty.bounding.RadFriends` method), 111  
`monte_carlo_vol()` (`dynesty.bounding.SupFriends` method), 112  
`MultiEllipsoid` (class in `dynesty.bounding`), 110  
`MultiEllipsoidSampler` (class in `dynesty.nestedsamplers`), 124

## N

`n_effective` (`dynesty.dynamicsampler.DynamicSampler` attribute), 129  
`n_effective` (`dynesty.sampler.Sampler` attribute), 119  
`NestedSampler()` (in module `dynesty.dynesty`), 102

## O

`overlap()` (`dynesty.bounding.MultiEllipsoid` method), 110  
`overlap()` (`dynesty.bounding.RadFriends` method), 111  
`overlap()` (`dynesty.bounding.SupFriends` method), 113

## P

`print_fn()` (in module `dynesty.results`), 134  
`propose_live()` (`dynesty.nestedsamplers.MultiEllipsoidSampler` method), 124  
`propose_live()` (`dynesty.nestedsamplers.RadFriendsSampler` method), 125  
`propose_live()` (`dynesty.nestedsamplers.SingleEllipsoidSampler` method), 123  
`propose_live()` (`dynesty.nestedsamplers.SupFriendsSampler` method), 127  
`propose_live()` (`dynesty.nestedsamplers.UnitCubeSampler` method), 122  
`propose_unif()` (`dynesty.nestedsamplers.MultiEllipsoidSampler` method), 124  
`propose_unif()` (`dynesty.nestedsamplers.RadFriendsSampler` method), 126  
`propose_unif()` (`dynesty.nestedsamplers.SingleEllipsoidSampler` method), 123  
`propose_unif()` (`dynesty.nestedsamplers.SupFriendsSampler` method), 127  
`propose_unif()` (`dynesty.nestedsamplers.UnitCubeSampler` method), 122

## Q

`quantile()` (in module `dynesty.utils`), 136

## R

`RadFriends` (class in `dynesty.bounding`), 111  
`RadFriendsSampler` (class in `dynesty.nestedsamplers`), 125  
`randoffset()` (`dynesty.bounding.Ellipsoid` method), 109  
`randoffset()` (`dynesty.bounding.UnitCube` method), 108  
`randsphere()` (in module `dynesty.bounding`), 114  
`resample_equal()` (in module `dynesty.utils`), 135  
`resample_run()` (in module `dynesty.utils`), 136  
`reset()` (`dynesty.dynamicsampler.DynamicSampler` method), 129  
`reset()` (`dynesty.sampler.Sampler` method), 120  
`Results` (class in `dynesty.results`), 134  
`results` (`dynesty.dynamicsampler.DynamicSampler` attribute), 129  
`results` (`dynesty.sampler.Sampler` attribute), 120  
`reweight_run()` (in module `dynesty.utils`), 137

`run_nested()` (*dynesty.dynamicsampler.DynamicSampler* method), 129

`run_nested()` (*dynesty.sampler.Sampler* method), 120

`runplot()` (*in module dynesty.plotting*), 139

## S

`sample()` (*dynesty.bounding.Ellipsoid* method), 109

`sample()` (*dynesty.bounding.MultiEllipsoid* method), 110

`sample()` (*dynesty.bounding.RadFriends* method), 111

`sample()` (*dynesty.bounding.SupFriends* method), 113

`sample()` (*dynesty.bounding.UnitCube* method), 108

`sample()` (*dynesty.sampler.Sampler* method), 120

`sample_batch()` (*dynesty.dynamicsampler.DynamicSampler* method), 130

`sample_hslice()` (*in module dynesty.sampling*), 118

`sample_initial()` (*dynesty.dynamicsampler.DynamicSampler* method), 131

`sample_rslic()` (*in module dynesty.sampling*), 117

`sample_rstagger()` (*in module dynesty.sampling*), 116

`sample_rwalk()` (*in module dynesty.sampling*), 116

`sample_slice()` (*in module dynesty.sampling*), 117

`sample_unif()` (*in module dynesty.sampling*), 115

`Sampler` (*class in dynesty.sampler*), 118

`samples()` (*dynesty.bounding.Ellipsoid* method), 109

`samples()` (*dynesty.bounding.MultiEllipsoid* method), 110

`samples()` (*dynesty.bounding.RadFriends* method), 112

`samples()` (*dynesty.bounding.SupFriends* method), 113

`samples()` (*dynesty.bounding.UnitCube* method), 109

`scale_to_vol()` (*dynesty.bounding.Ellipsoid* method), 109

`scale_to_vol()` (*dynesty.bounding.RadFriends* method), 112

`scale_to_vol()` (*dynesty.bounding.SupFriends* method), 113

`scale_to_vols()` (*dynesty.bounding.MultiEllipsoid* method), 110

`simulate_run()` (*in module dynesty.utils*), 136

`SingleEllipsoidSampler` (*class in dynesty.nestedsamplers*), 123

`stopping_function()` (*in module dynesty.dynamicsampler*), 133

`summary()` (*dynesty.results.Results* method), 134

`SupFriends` (*class in dynesty.bounding*), 112

`SupFriendsSampler` (*class in dynesty.nestedsamplers*), 126

## T

`traceplot()` (*in module dynesty.plotting*), 140

`unitcheck()` (*in module dynesty.utils*), 135

`UnitCube` (*class in dynesty.bounding*), 108

`unitcube_overlap()` (*dynesty.bounding.Ellipsoid* method), 109

`UnitCubeSampler` (*class in dynesty.nestedsamplers*), 122

`unravel_run()` (*in module dynesty.utils*), 137

`update()` (*dynesty.bounding.Ellipsoid* method), 109

`update()` (*dynesty.bounding.MultiEllipsoid* method), 110

`update()` (*dynesty.bounding.RadFriends* method), 112

`update()` (*dynesty.bounding.SupFriends* method), 113

`update()` (*dynesty.bounding.UnitCube* method), 109

`update()` (*dynesty.nestedsamplers.MultiEllipsoidSampler* method), 125

`update()` (*dynesty.nestedsamplers.RadFriendsSampler* method), 126

`update()` (*dynesty.nestedsamplers.SingleEllipsoidSampler* method), 123

`update()` (*dynesty.nestedsamplers.SupFriendsSampler* method), 127

`update()` (*dynesty.nestedsamplers.UnitCubeSampler* method), 122

`update_hslice()` (*dynesty.nestedsamplers.MultiEllipsoidSampler* method), 125

`update_hslice()` (*dynesty.nestedsamplers.RadFriendsSampler* method), 126

`update_hslice()` (*dynesty.nestedsamplers.SingleEllipsoidSampler* method), 123

`update_hslice()` (*dynesty.nestedsamplers.SupFriendsSampler* method), 127

`update_hslice()` (*dynesty.nestedsamplers.UnitCubeSampler* method), 122

`update_rwalk()` (*dynesty.nestedsamplers.MultiEllipsoidSampler* method), 125

`update_rwalk()` (*dynesty.nestedsamplers.RadFriendsSampler* method), 126

`update_rwalk()` (*dynesty.nestedsamplers.SingleEllipsoidSampler* method), 124

`update_rwalk()` (*dynesty.nestedsamplers.SupFriendsSampler* method), 127

`update_rwalk()` (*dynesty.nestedsamplers.UnitCubeSampler* method), 122

`update_slice()` (*dynesty.nestedsamplers.MultiEllipsoidSampler* method), 125

`update_slice()` (*dynesty.nestedsamplers.RadFriendsSampler* method), 126

`update_slice()` (*dynesty.nestedsamplers.SingleEllipsoidSampler* method), 124

`update_slice()` (*dynesty.nestedsamplers.SupFriendsSampler* method), 127

`update_slice()` (*dynesty.nestedsamplers.UnitCubeSampler* method), 123

`update_unif()` (*dynesty.nestedsamplers.MultiEllipsoidSampler method*), [125](#)  
`update_unif()` (*dynesty.nestedsamplers.RadFriendsSampler method*), [126](#)  
`update_unif()` (*dynesty.nestedsamplers.SingleEllipsoidSampler method*), [124](#)  
`update_unif()` (*dynesty.nestedsamplers.SupFriendsSampler method*), [127](#)  
`update_unif()` (*dynesty.nestedsamplers.UnitCubeSampler method*), [123](#)

## V

`vol_prefactor()` (*in module dynesty.bounding*), [113](#)

## W

`weight_function()` (*in module dynesty.dynamicsampler*), [132](#)  
`within()` (*dynesty.bounding.MultiEllipsoid method*), [111](#)  
`within()` (*dynesty.bounding.RadFriends method*), [112](#)  
`within()` (*dynesty.bounding.SupFriends method*), [113](#)