

---

# **dessn Documentation**

***Release 0.0.1***

**dessn**

March 28, 2016



## CONTENTS

<b>1 Examples</b>	<b>3</b>
<b>2 Core</b>	<b>5</b>
<b>3 Utilities</b>	<b>7</b>
<b>4 Implementations</b>	<b>9</b>
<b>5 General</b>	<b>11</b>
5.1 dessn package . . . . .	11
5.1.1 Subpackages . . . . .	11
dessn.chain package . . . . .	11
dessn.examples package . . . . .	29
dessn.model package . . . . .	41
dessn.simulation package . . . . .	48
dessn.toy package . . . . .	48
dessn.utility package . . . . .	55
<b>Python Module Index</b>	<b>57</b>
<b>Index</b>	<b>59</b>



This document should detail the code structure, examples, and all other goodies.

You can find PDF documentation [here](#).



---

**CHAPTER  
ONE**

---

**EXAMPLES**

Learning from examples is my preferred method.

For the primary motivation example, explaining how the models work and tie together, and why certain design choices have been chosen - primarily marginalisation over numerical integration, see the `dessn.examples.simple` package.

For a different example, which shows how to use discrete parameters, see the `dessn.examples.discrete` package.



---

**CHAPTER  
TWO**

---

**CORE**

To learn how the underlying models function, and specific details on the sorts of parameters and edges allowed, please see the documentation located at [\*deessn.model\*](#).



---

**CHAPTER  
THREE**

---

**UTILITIES**

For examples and usage instructions on how to use the plotting library included in this project, see [\*desn.chain\*](#).



---

**CHAPTER  
FOUR**

---

## **IMPLEMENTATIONS**

Finally, for the toy model implementation, see *dessn.toy*



## GENERAL

The general project structure is as follows:

### 5.1 dessn package

Welcome to the **DESSN** code base.

#### 5.1.1 Subpackages

##### **dessn.chain package**

###### **Submodules**

###### **dessn.chain.chain module**

```
class dessn.chain.chain.ChainConsumer
    Bases: object
```

A class for consuming chains produced by an MCMC walk

```
add_chain (chain, parameters=None, name=None)
    Add a chain to the consumer.
```

**Parameters** `chain` : str|ndarray

The chain to load. Normally a numpy.ndarray, but can also accept a string. If a string is found, it interprets the string as a filename and attempts to load it in.

**parameters** : list[str], optional

A list of parameter names, one for each column (dimension) in the chain.

**name** : str, optional

The name of the chain. Used when plotting multiple chains at once.

**Returns** ChainConsumer

Itself, to allow chaining calls.

```
configure_bar (summary=None)
```

Configure the bar plots showing the marginalised distributions. If you do not call this explicitly, the `plot()` method will invoke this method automatically.

**summary** [bool, optional] If overridden, sets whether parameter summaries should be set as axis titles.  
Will not work if you have multiple chains

**configure\_contour** (*sigmas=None, cloud=None, contourf=None, contourf\_alpha=1.0*)

Configure the default variables for the contour plots. If you do not call this explicitly, the `plot()` method will invoke this method automatically.

Please ensure that you call this method after adding all the relevant data to the chain consumer, as the consume changes configuration values depending on the presupplied data.

**Parameters** **sigmas** : np.array, optional

The  $\sigma$  contour levels to plot. Defaults to [0.5, 1, 2, 3]. Number of contours shown decreases with the number of chains to show.

**cloud** : bool, optional

If set, overrides the default behaviour and plots the cloud or not

**contourf** : bool, optional

If set, overrides the default behaviour and plots filled contours or not

**contourf\_alpha** : float, optional

Filled contour alpha value override.

**configure\_general** (*bins=None, flip=True, rainbow=None, colours=None, serif=True, plot\_hists=True, max\_ticks=5*)

Configure the general plotting parameters common across the bar and contour plots. If you do not call this explicitly, the `plot()` method will invoke this method automatically.

**Parameters** **bins** : int|float, optional

The number of bins to use. By default uses  $\frac{\sqrt{n}}{10}$ , where  $n$  are the number of data points.  
Giving an integer will set the number of bins to the given value. Giving a float will scale  
the number of bins, such that giving `bins=1.5` will result in using  $\frac{1.5\sqrt{n}}{10}$  bins.

**flip** : bool, optional

Set to false if, when plotting only two parameters, you do not want it to rotate the histogram so that it is horizontal.

**rainbow** : bool, optional

Set to True to force use of rainbow colours

**colours** : list[str(hex)], optional

Provide a list of colours to use for each chain. If you provide more chains than colours, you *will* get the rainbow colour spectrum.

**serif** : bool, optional

Whether to display ticks and labels with serif font.

**plot\_hists** : bool, optional

Whether to plot marginalised distributions or not

**max\_ticks** : int, optional

The maximum number of ticks to use on the plots

**configure\_truth** (\*\*kwargs)

Configure the arguments passed to the `axvline` and `axhline` methods when plotting truth values. If you do not call this explicitly, the `plot()` method will invoke this method automatically.

Recommended to set the parameters `linestyle`, `color` and/or `alpha` if you want some basic control.  
 Default is to use an opaque black dashed line.

**get\_latex\_table** (`parameters=None`, `transpose=False`, `caption=None`, `label=None`, `hlines=True`,  
`blank_fill=''`)

Generates a LaTeX table from parameter summaries.

For an example output, see the image below:

Table 1: The maximum likelihood results for the tested models

Model	$x$	$y$	$\alpha$	$\beta$	$\gamma$
Model A	$0.1^{+0.9}_{-1.1}$	$5.1^{+2.9}_{-3.1}$	$-0.1^{+1.2}_{-1.0}$	$0.0 \pm 0.2$	—
Model B	$-0.8^{+0.8}_{-1.2}$	$-5.3^{+2.3}_{-1.7}$	$0.5^{+1.5}_{-1.3}$	—	$1.8^{+1.7}_{-1.4}$

**Parameters** `parameters` : list[str], optional

A list of what parameters to include in the table. By default, includes all parameters

**transpose** : bool, optional

Defaults to False, which gives each column as a parameter, each chain (model) as a row.  
 You can swap it so that you have a parameter each row and a model each column by  
 setting this to True

**caption** : str, optional

If you want to generate a caption for the table through Python, use this. Defaults to an  
 empty string

**label** : str, optional

If you want to generate a label for the table through Python, use this. Defaults to an  
 empty string

**hlines** : bool, optional

Inserts \hline before and after the header, and at the end of table.

**blank\_fill** : str, optional

If a model does not have a particular parameter, will fill that cell of the table with this  
 string.

**Returns** str

the LaTeX table.

**get\_parameter\_text** (`lower, maximum, upper, wrap=False`)

Generates LaTeX appropriate text from marginalised parameter bounds.

**Parameters** `lower` : float

The lower bound on the parameter

`maximum` : float

The value of the parameter with maximum probability

`upper` : float

The upper bound on the parameter

**wrap** : bool

Wrap output text in dollar signs for LaTeX

**get\_summary()**

Gets a summary of the marginalised parameter distributions.

**Returns** list of dictionaries

One entry per chain, parameter bounds stored in dictionary with parameter as key

**plot** (figsize='GROW', parameters=None, extents=None, filename=None, display=False, truth=None, legend=True)

Plot the chain

**Parameters** **figsize** : str|tuple(float), optional

The figure size to generate. Accepts a regular two tuple of size in inches, or one of several key words. The default value of COLUMN creates a figure of appropriate size of insertion into an A4 LaTeX document in two-column mode. PAGE creates a full page width figure. GROW creates an image that scales with parameters (1.5 inches per parameter). String arguments are not case sensitive.

**parameters** : list[str], optional

If set, only creates a plot for those specific parameters

**extents** : list[tuple[float]] or dict[str], optional

Extents are given as two-tuples. You can pass in a list the same size as parameters (or default parameters if you don't specify parameters), or as a dictionary.

**filename** : str, optional

If set, saves the figure to this location

**display** : bool, optional

If True, shows the figure using `plt.show()`.

**truth** : list[float] or dict[str], optional

A list of truth values corresponding to parameters, or a dictionary of truth values indexed by key

**legend** : bool, optional

If true, creates a legend in your plot using the chain names.

**Returns** figure

the matplotlib figure

**plot\_walks** (parameters=None, truth=None, extents=None, display=False, filename=None, chain=None, convolve=None, figsize=None)

Plots the chain walk; the parameter values as a function of step index.

This plot is more for a sanity or consistency check than for use with final results. Plotting this before plotting with `plot()` allows you to quickly see if the chains are well behaved, or if certain parameters are suspect or require a greater burn in period.

The desired outcome is to see an unchanging distribution along the x-axis of the plot. If there are obvious tails or features in the parameters, you probably want to investigate.

See `dessn.chain.demoWalk.DemoWalk` for example usage.

**Parameters** `parameters` : list[str], optional

Specify a subset of parameters to plot. If not set, all parameters are plotted.

**truth** : list[float]|dict[str], optional

A list of truth values corresponding to parameters, or a dictionary of truth values keyed by the parameter.

**extents** : list[tuple]|dict[str], optional

A list of two-tuples for plot extents per parameter, or a dictionary of extents keyed by the parameter.

**display** : bool, optional

If set, shows the plot using `plt.show()`

**filename** : str, optional

If set, saves the figure to the filename

**chain** : int|str, optional

Used to specify which chain to show if more than one chain is loaded in. Can be an integer, specifying the chain index, or a str, specifying the chain name.

**convolve** : int, optional

If set, overplots a smoothed version of the steps using `convolve` as the width of the smoothing filter.

**figsize** : tuple, optional

If set, sets the created figure size.

**Returns** figure

the matplotlib figure created

## dессн.chain.demoOneChain module

**class** `dессн.chain.demoOneChain.DemoOneChain`

The single chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it servers no other purpose.

Running this file in python creates a random data set, representing a single MCMC chain, such as you might get from emcee.

First, we create a consumer and load the chain, and tell it to plot the chain without knowing the parameter labels. It is set to so that the plot should pop up. To continue running the code, close the plot.

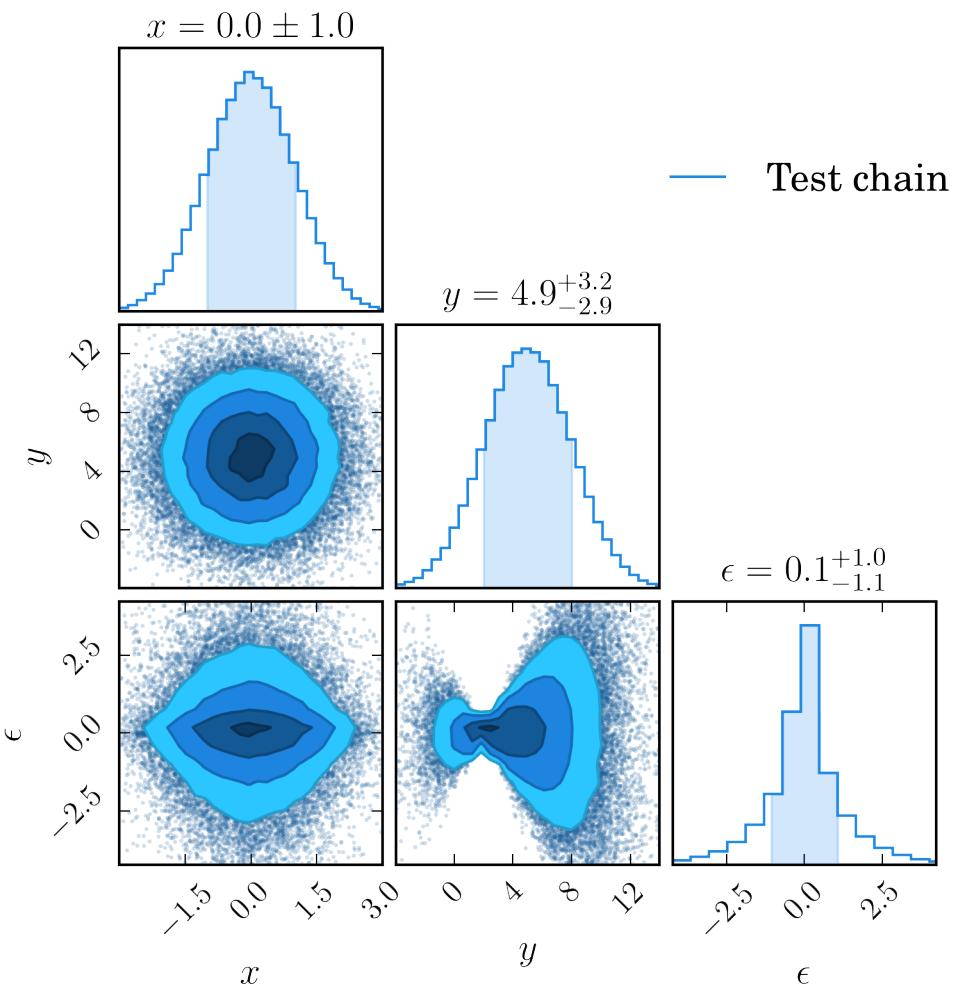
The second thing we do is create a different consumer, and load the chain into it. We also supply the parameter labels. By default, as we only have a single chain, contours are filled, the marginalised histograms are shaded, and the best fit parameter bounds are shown as axis titles.

The plot for this is saved to the png file below:

## dессн.chain.demoTable module

**class** `dессн.chain.demoTable.DemoTable`

The multiple chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it servers no other purpose.



Running this file in python creates two random data sets, representing two separate chains, *for two separate models*.

This example shows the output of calling the `get_latex_table()` method.

Table 1: The maximum likelihood results for the tested models

Model	$x$	$y$	$\alpha$	$\beta$	$\gamma$
Model A	$0.1^{+0.9}_{-1.1}$	$5.1^{+2.9}_{-3.1}$	$-0.1^{+1.2}_{-1.0}$	$0.0 \pm 0.2$	—
Model B	$-0.8^{+0.8}_{-1.2}$	$-5.3^{+2.3}_{-1.7}$	$0.5^{+1.5}_{-1.3}$	—	$1.8^{+1.7}_{-1.4}$

### dessn.chain.demoThreeChains module

#### class dessn.chain.demoThreeChains.DemoThreeChains

The multiple chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it servers no other purpose.

Running this file in python creates three random data sets, representing three separate chains.

First, we create a consumer and load the first two chains, and tell it to plot with filled contours.

The second thing we do is create a different consumer, and load all three chains into it. We also supply the parameter labels the first time we load in a chain. The plot for this is saved to the png file below:

### dessn.chain.demoTwoDisjointChains module

#### class dessn.chain.demoTwoDisjointChains.DemoTwoDisjointChains

The multiple chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it servers no other purpose.

Running this file in python creates two random data sets, representing two separate chains, *for two separate models*.

It is sometimes the case that we wish to compare models which have partially overlapping parameters. For example, we might fit a model which depends has cosmology dependend on  $\Omega_m$  and  $\Omega_\Lambda$ , where we assume  $w = 1$ . Alternatively, we might assume flatness, and therefore fix  $\Omega_\Lambda$  but instead vary the equation of state  $w$ . The good news is, you can visualise them both at once!

The second thing we do is create a consumer, and load both chains into it. As we have different parameters for each chain we supply the right parameters for each chain. The plot for this is saved to the png file below:

### dessn.chain.demoVarious module

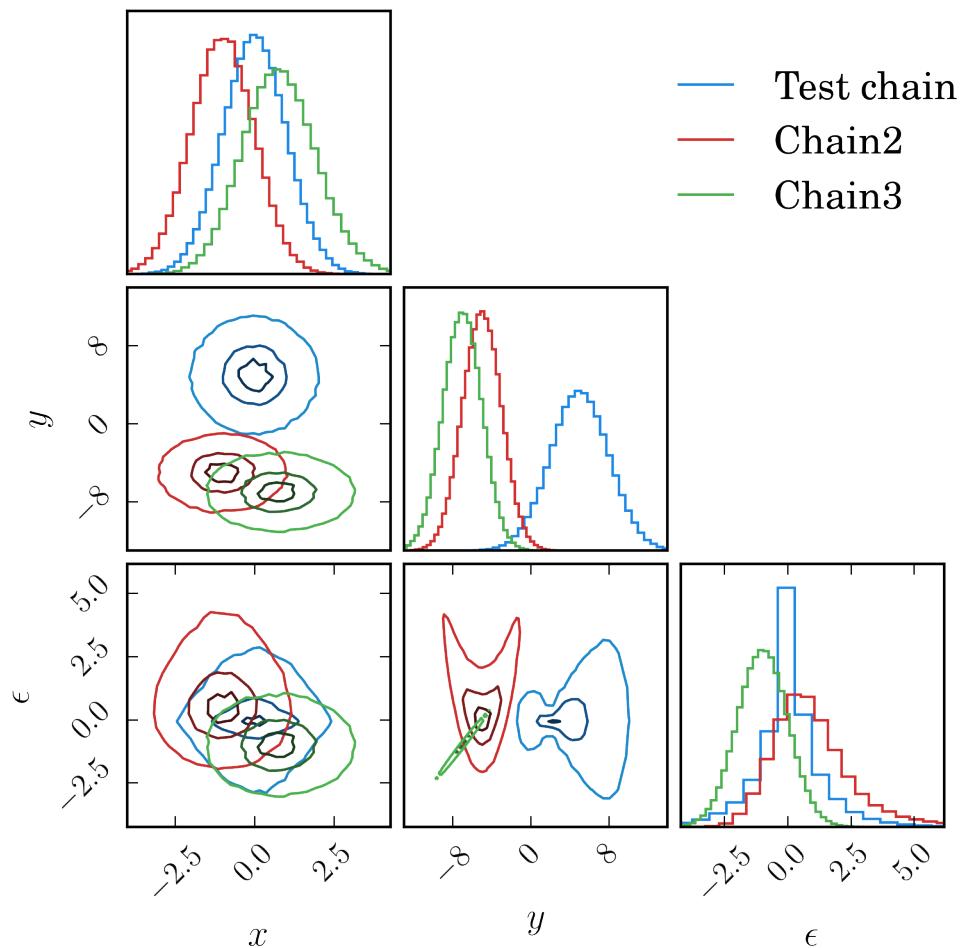
#### class dessn.chain.demoVarious.DemoVarious

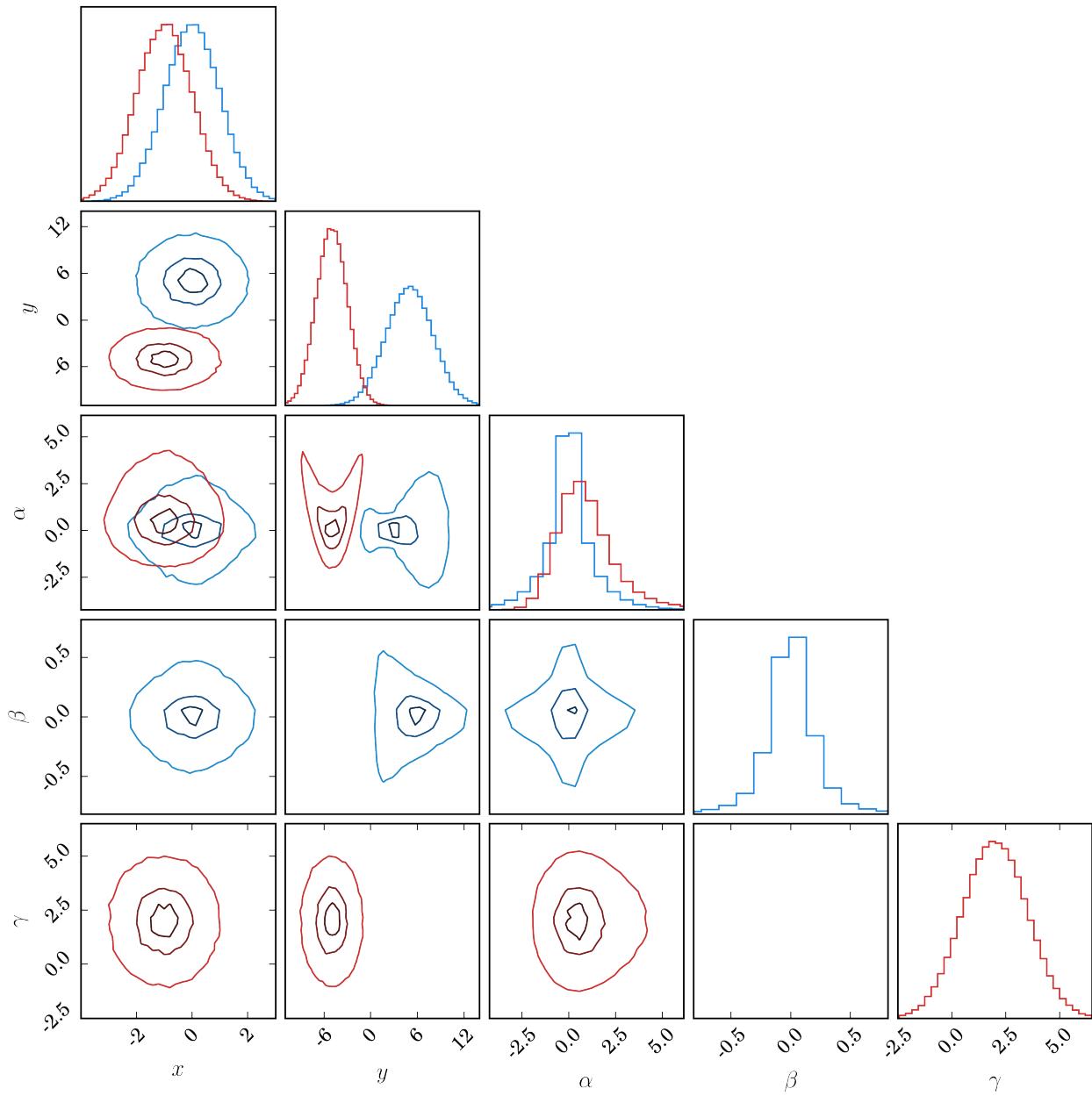
Bases: object

The demo for various functions and usages of Chain Consumer.

This file should show some examples of how to use ChainConsumer in more unusual ways with extra customisation.

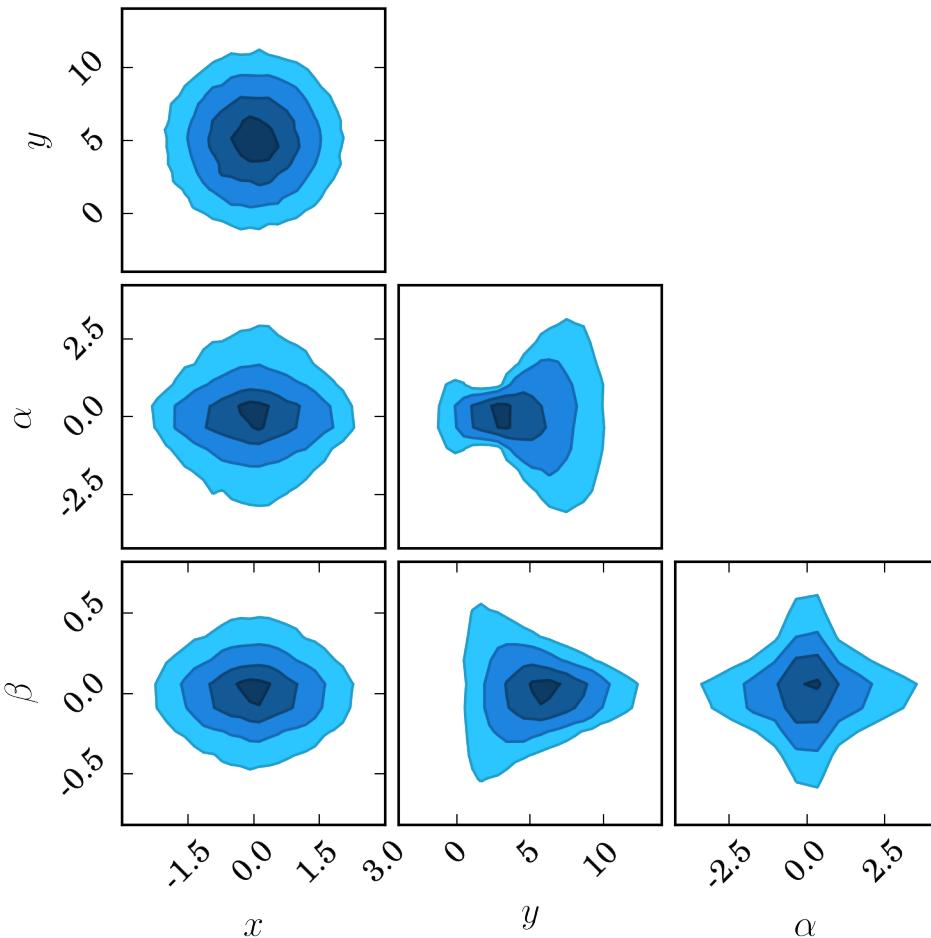
The methods of this class should provide context as to what is being done.





**various1\_no\_histogram()**

Plot data without histogram or cloud. For those liking the minimalistic approach

**various2\_select\_parameters()**

You can chose to only display a select number of parameters. Here the  $\beta$  parameter is not displayed.

**various3\_flip\_histogram()**

When you only display two parameters and don't disable histograms, your plot will look different.

You can suppress this by passing to `flip=False` to `ChainConsumer.configure_general()`. See the commented out line in code for the actual line to disable this.

The max number of ticks is also modified in this example.

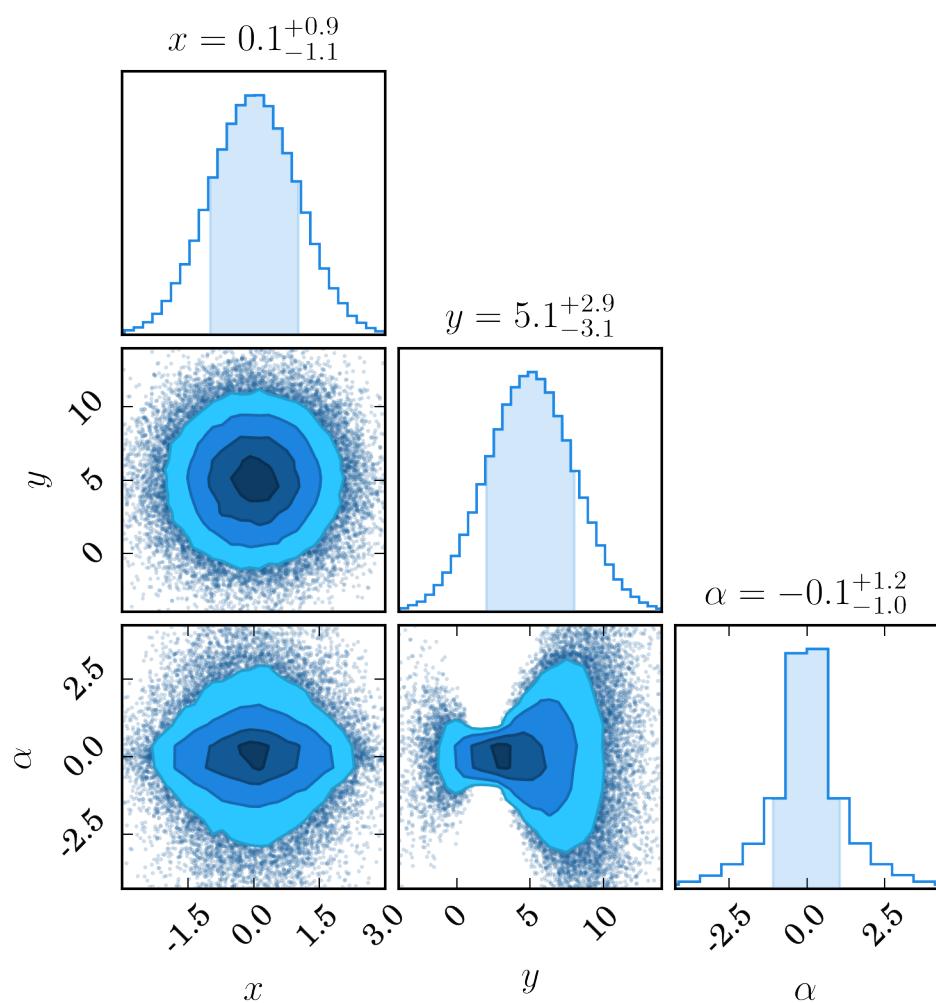
**various4\_summaries()**

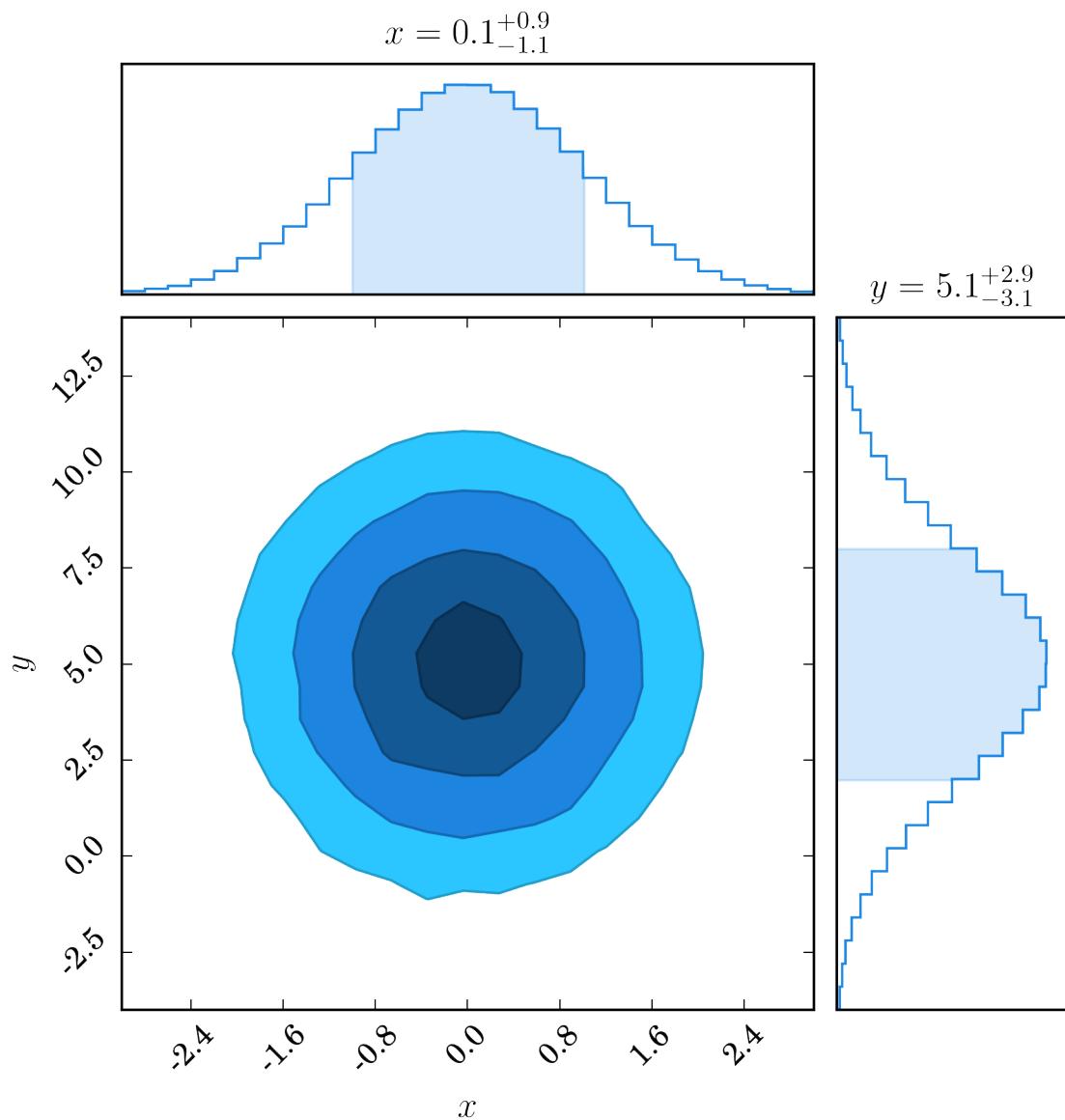
If there is only one chain to analyse, and you only chose to plot a small number of parameters, the parameter summary will be shown above the relevant axis. You can set this to always show or always not show by using the `force_summary` flag. Also, here we demonstrate more  $\sigma$  levels!

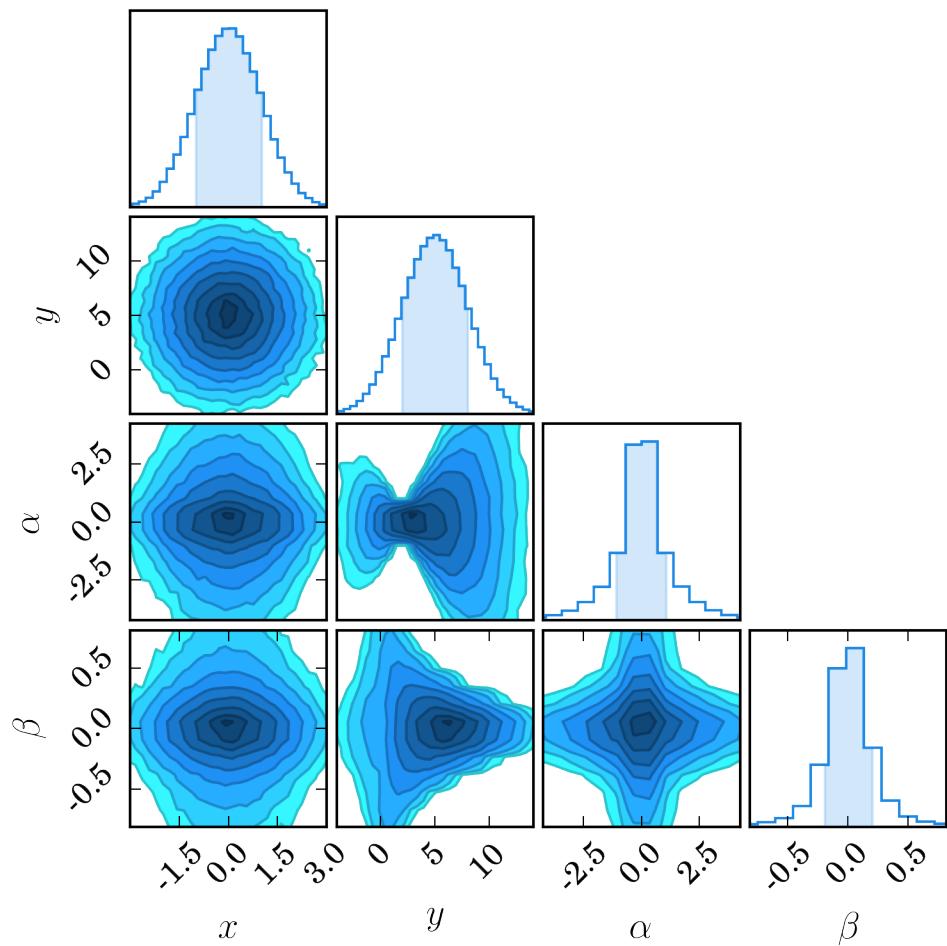
**various5\_custom\_colours()**

You can supply custom colours to the plotting. Be warned, if you have more chains than colours, you *will* get a rainbow instead!

Note that, due to colour scaling, you **must** supply custom colours as full six digit hex colours, such as #A87B20.

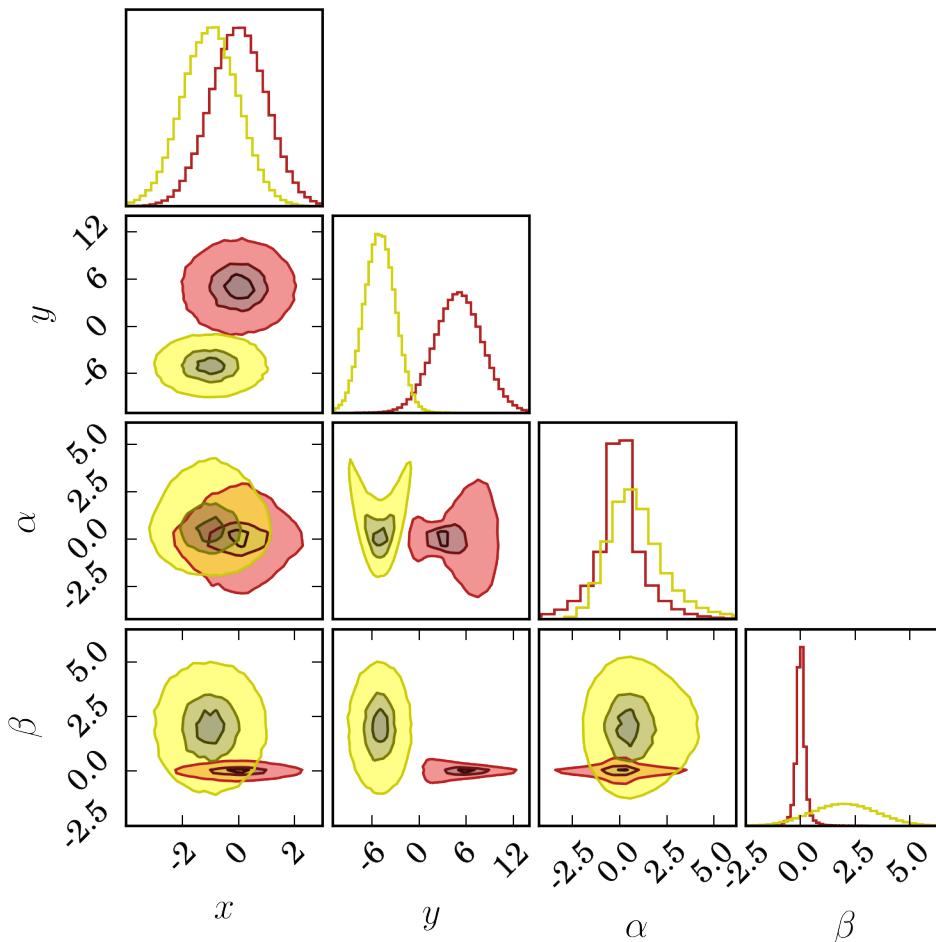






As colours get scaled, it is a good idea to pick something neither too light, dark, or saturated.

In this example, I also force contour filling and set contour filling opacity to 0.5, so we can see overlap.



#### **various6\_truth\_values()**

The reward for scrolling down so far, the first customised argument that will be frequently used; truth values.

Truth values can be given as a list the same length of the input parameters, or as a dictionary, keyed by the parameters.

In the code there are two examples. The first, where a list is passed in, and the second, where an incomplete dictionary of truth values is passed in. In the second case, customised values for truth line plotting are used. The figures are respectively

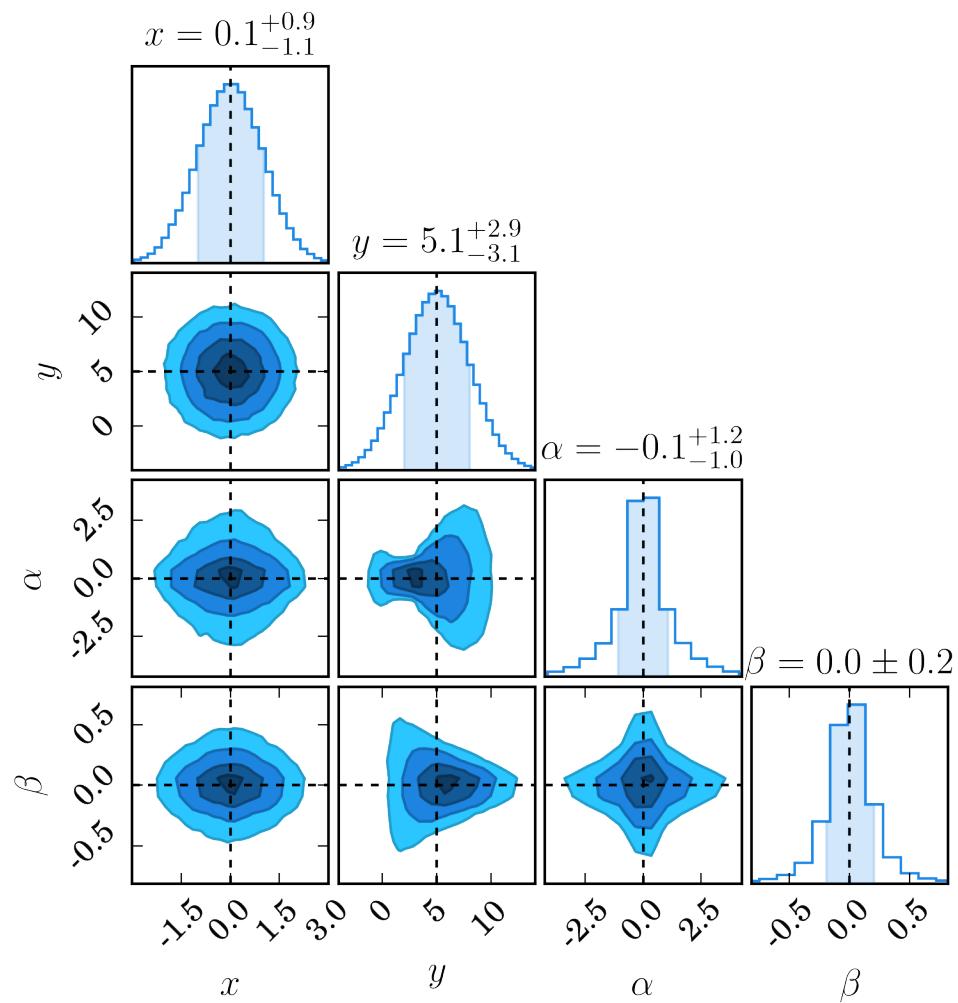
#### **various7\_rainbow()**

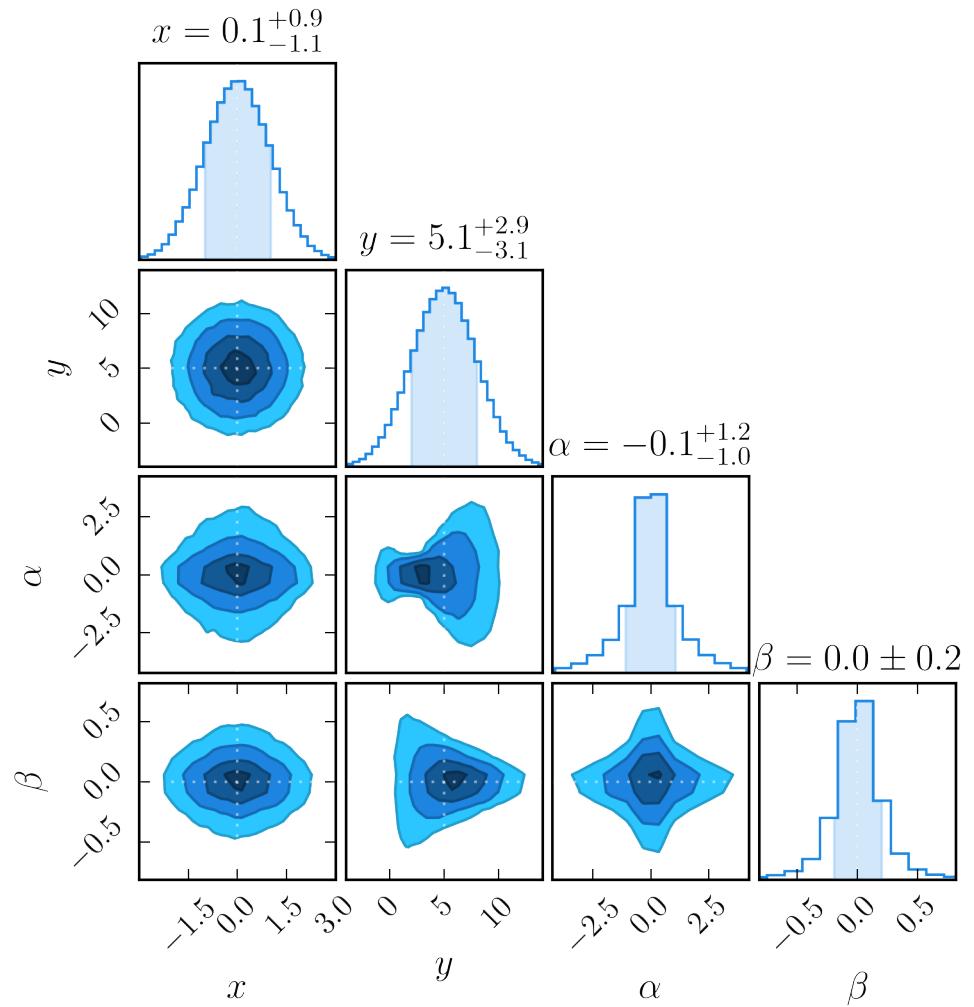
An example on using the rainbow with sans-serif fonts and too many bins!

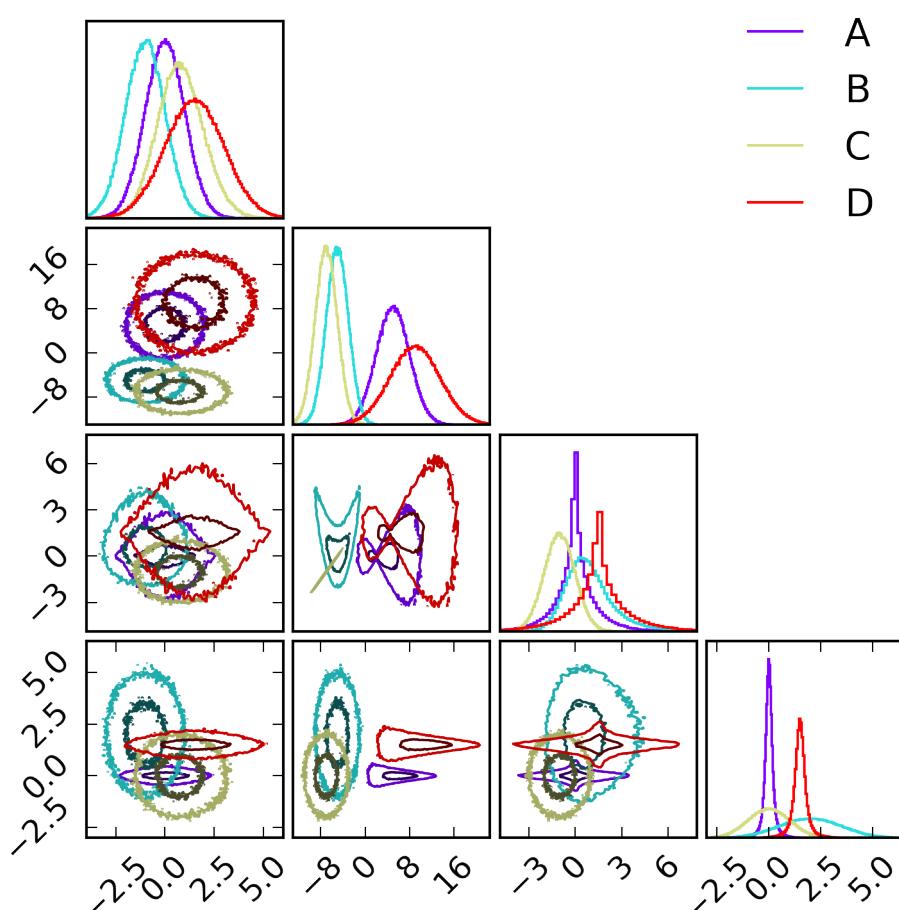
#### **various8\_extents()**

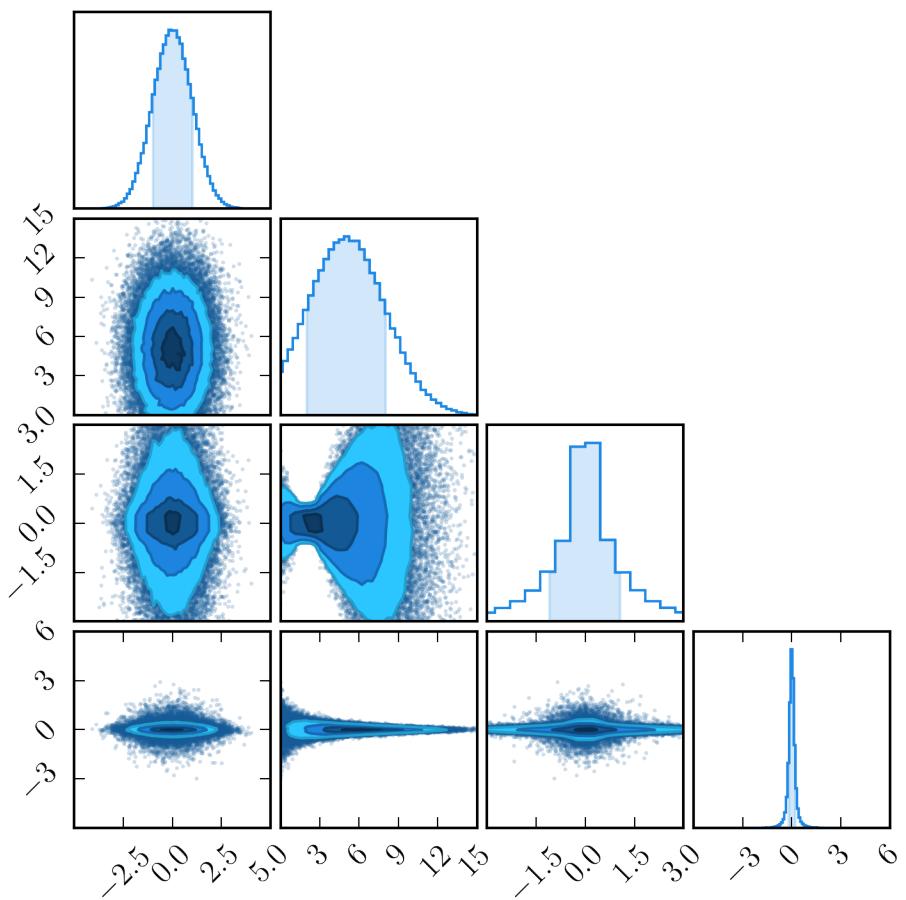
An example on using customised extents. Similarly to the example for truth values in `various7_truth_values()`, you can pass a list in, or a dictionary.

Also modifying the number of bins using a float value to scale, rather than set, the number of bins.









## dessn.chain.demoWalk module

### class dessn.chain.demoWalk.DemoWalk

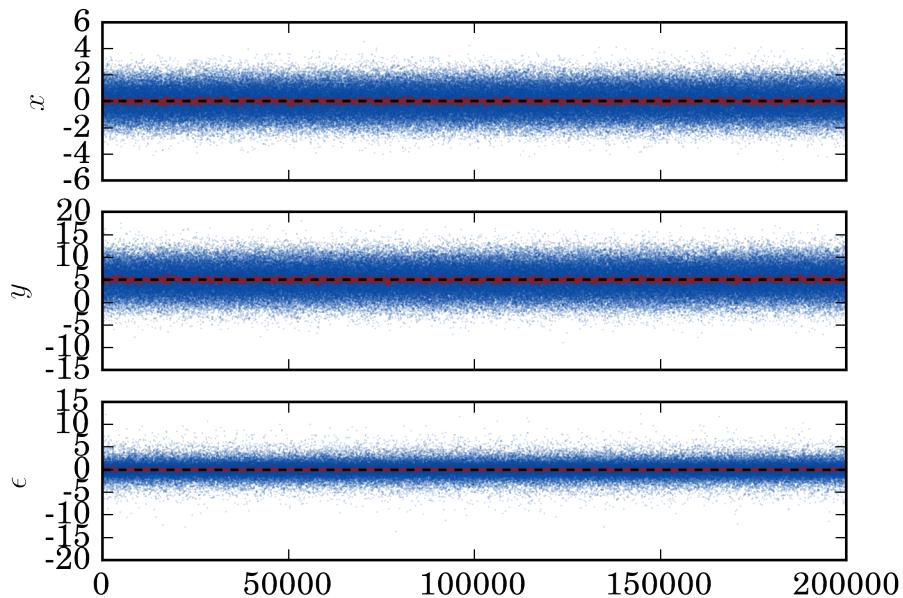
The single chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it servers no other purpose.

Running this file in python creates a random data set, representing a single MCMC chain, such as you might get from emcee.

We want to see if our walks are behaving as expected, which means we should see them scattered around the underlying truth value (or actual truth value if supplied). This is a good consistency check, because if the burn in period (which should have already been removed from the chain) was insufficiently long, you would expect to see tails appear in this plot.

Because individual samples can be noisy, there is also the option to pass an integer via parameter `convolve`, which overplots a boxcar smoothed version of the steps, where `convolve` sets the smooth window size.

The plot for this is saved to the png file below:



## dessn.examples package

### Subpackages

#### dessn.examples.discrete package

An example used to prototype the inclusion of discrete parameters into the mix.

The scenario being modelled is thus:

A bag is filled with an infinite amount of coloured balls, either red or blue, where the total fraction of the balls that are red is given by the variable  $r$ . We remove only a few balls from this infinite bag and record the colour and size of the ball.

However, the person writing down the colour is mostly colour blind, and so mistakes do happen (at a known rate). Luckily, there is information contained in the physical size of the balls (which is measured perfectly), as red balls are generally found to be larger in radius than the blue balls. Knowing the size distribution, we can use this information to potentially correct for some misclassifications, and thus determine the actual fraction  $r$  of red balls in the bag.

Firstly, we model misidentification as:

$$P(c_i|c) = \begin{cases} 0.9, & \text{if } c_i = c \\ 0.1, & \text{otherwise} \end{cases}$$

We model the sizes as being Gaussian distributed based on the colour.

$$P(s_i|c) = \begin{cases} \frac{1}{\sqrt{2\pi}0.3} \exp\left(-\frac{(s_i-2)^2}{2\times0.3^2}\right), & \text{if } c = \text{red} \\ \frac{1}{\sqrt{2\pi}0.3} \exp\left(-\frac{(s_i-1)^2}{2\times0.3^2}\right), & \text{if } c = \text{blue} \end{cases}$$

Following a basic binomial process, given a total fraction rate of  $r$ , we have

$$P(c|r) = \begin{cases} r, & \text{if } c = \text{red} \\ 1 - r, & \text{if } c = \text{blue} \end{cases}$$

Putting a flat prior on the rate between 0 and 1 ( $P(r) = 1$ ), the total probability for  $n$  data points is

$$P(r|s_o, c_o) \propto \prod_{i=1}^n \sum_c P(s_i|c) P(c_i|c) P(c|r) P(r)$$

With three conditional probabilities, we will have three edges in our node, one discrete parameter  $c$ , one underlying node  $r$ , and two observed parameters  $s_o$  and  $c_o$ .

## Submodules

### **dessn.examples.discrete.discrete module**

```
class dessn.examples.discrete.discrete.Colour
    Bases: dessn.model.parameter.ParameterDiscrete
        get_discrete(data)
        get_discrete_requirements()
class dessn.examples.discrete.discrete.DiscreteModel
    Bases: dessn.model.model.Model
```

A small example model illustrating how to use discrete parameters.

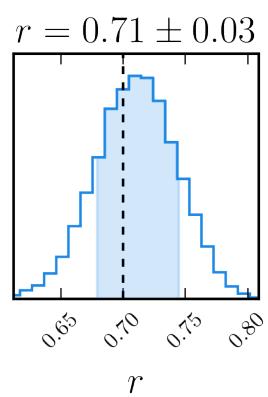
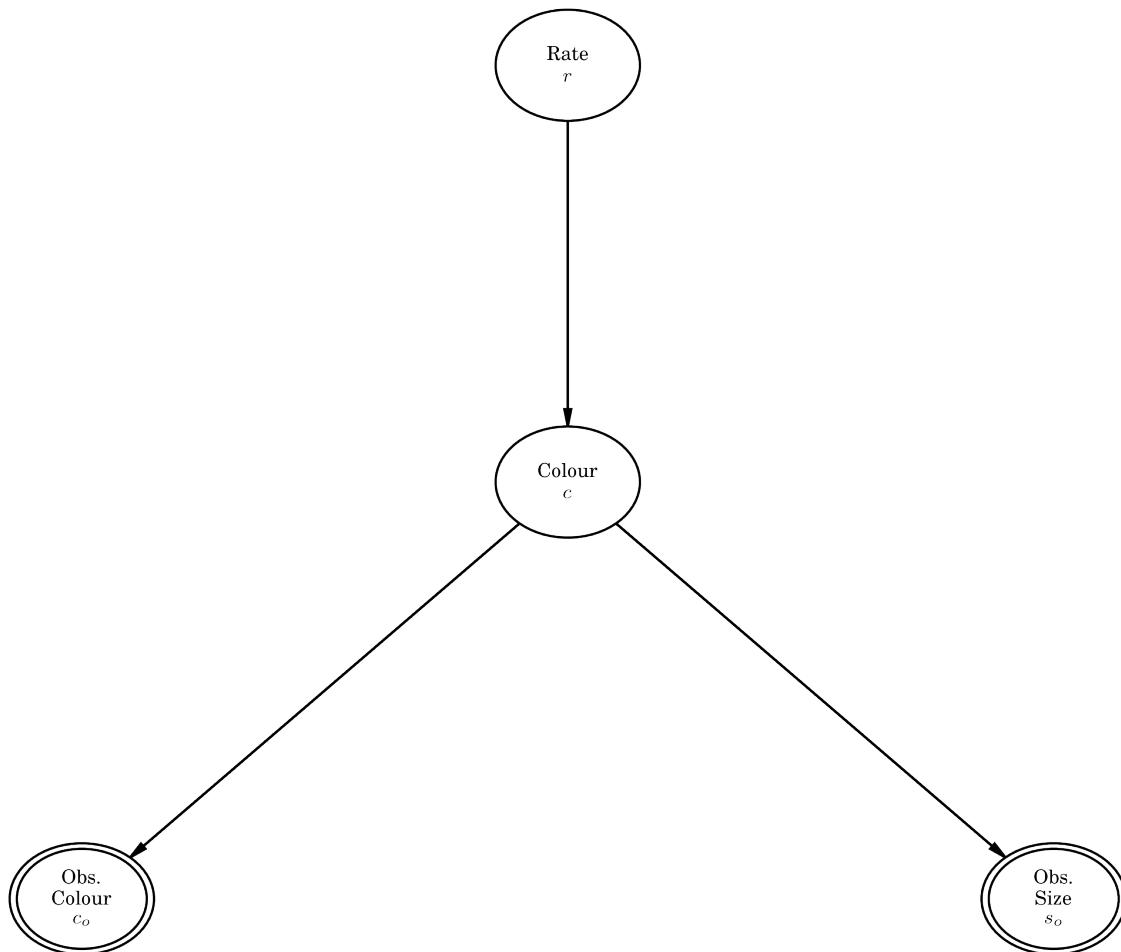
As normal, the model is set up by declaring parameters (which can be thought of like nodes on a PGM), and declaring the edges between parameters (the conditional probabilities).

This is the primary class in this package, and you can see that other classes inherit from either `Parameter` or from `Edge`.

I leave the documentation for `Parameter` and `Edge` to those classes, and encourage viewing the code directly to understand exactly what is happening.

Running this file in python first generates a PGM of the model, and then runs `emcee` and creates a corner plot:

```
class dessn.examples.discrete.ObservedColour
    Bases: dessn.model.parameter.ParameterObserved
class dessn.examples.discrete.ObservedSize
    Bases: dessn.model.parameter.ParameterObserved
class dessn.examples.discrete.ToColour
    Bases: dessn.model.edge.Edge
        get_log_likelihood(data)
```



```
class dessn.examples.discrete.discrete.ToColour2
    Bases: dессн.model.edge.Edge
```

```
    get_log_likelihood(data)
```

```
class dessn.examples.discrete.discrete.ToRate
```

```
    Bases: dессн.model.edge.Edge
```

```
    get_log_likelihood(data)
```

```
class dessn.examples.discrete.discrete.UnderlyingRate
```

```
    Bases: dессн.model.parameter.ParameterUnderlying
```

```
    get_log_prior(data)
```

```
    get_suggestion(data)
```

```
    get_suggestion_requirements()
```

```
    get_suggestion_sigma(data)
```

```
dессн.examples.discrete.discrete.get_data(n=200)
```

**dессн.examples.simple package** This module is designed to give a step by step overview of a very simplified example Bayesian model.

The basic example model is laid out in the parent class *Example*, and there are three implementations. The first implementation, *ExampleIntegral*, shows how the problem might be approached in a simple model, where numerical integration is simply done as part of the likelihood calculation.

However, if there are multiple latent parameters, we get polynomial growth of the number of numerical integrations we have to do, and so this does not scale well at all.

This leads us to the implementation in *ExampleLatent*, where we use the MCMC algorithm to essentially do Monte Carlo integration via marginalisation. This means we do not need to perform the numerical integration in the likelihood calculation, however the cost of doing so is increase dimensionality of our MCMC.

Finally, the *ExampleModel* implementation shows how the *ExampleLatent* class might be written to make use of Nodes. This is done in preparation for more complicated models, which will have more than one layer and needs to be configurable.

## Subpackages

**dессн.examples.simple.modelbased package** I have placed the class based example for implementing the simplified model into its own module, so that the documentation generating for the `simple` module does not get cluttered with all the small classes this module will have.

The primary class to look at in code is the *ExampleModel* class.

I should finally note that in order to demonstrate parameter transformations, I have modified the model used in the previous two examples (*ExampleIntegral* and *ExampleLatent*) to also include a luminosity transformation, where I simply halve the luminosity before converting it to flux. Physically, this could represent a perfect 50% mirror absorption on the primary telescope mirror.

## Submodules

**dessn.examples.simple.modelbased.exampleModel module**

**class** dessn.examples.simple.modelbased.exampleModel.ExampleModel  
 Bases: *dessn.model.model.Model*

An implementation of *ExampleLatent* using classes instead of procedural code.

The model is set up by declaring nodes, the edges between nodes, and then calling `finalise` on the model to verify its correctness.

This is the primary class in this package, and you can see that other classes inherit from either *Parameter* or from *Edge*.

I leave the documentation for *Parameter* and *Edge* to those classes, and encourage viewing the code directly to understand exactly what is happening.

Running this file in python first generates a PGM of the model, and then runs `emcee` and creates a corner plot:

We could also run the example model using the PT sampler by specifying a number of temperature to the `fit_model` method. You would get similar results.

**class** dessn.examples.simple.modelbased.exampleModel.FluxToLuminosity  
 Bases: *dessn.model.edge.Edge*

**get\_log\_likelihood**(*data*)

**class** dessn.examples.simple.modelbased.exampleModel.LatentLuminosity(*n=100*)  
 Bases: *dessn.model.parameter.ParameterLatent*

**get\_num\_latent**()

**get\_suggestion**(*data*)

**get\_suggestion\_requirements**()

**get\_suggestion\_sigma**(*data*)

**class** dessn.examples.simple.modelbased.exampleModel.LuminosityToAdjusted  
 Bases: *dessn.model.edge.EdgeTransformation*

**get\_transformation**(*data*)

**class** dessn.examples.simple.modelbased.exampleModel.LuminosityToSupernovaDistribution  
 Bases: *dessn.model.edge.Edge*

**get\_log\_likelihood**(*data*)

**class** dessn.examples.simple.modelbased.exampleModel.ObservedFlux(*n=100*)  
 Bases: *dessn.model.parameter.ParameterObserved*

**class** dessn.examples.simple.modelbased.exampleModel.ObservedFluxError(*n=100*)  
 Bases: *dessn.model.parameter.ParameterObserved*

**class** dessn.examples.simple.modelbased.exampleModel.UnderlyingSupernovaDistribution1  
 Bases: *dessn.model.parameter.ParameterUnderlying*

**get\_log\_prior**(*data*)

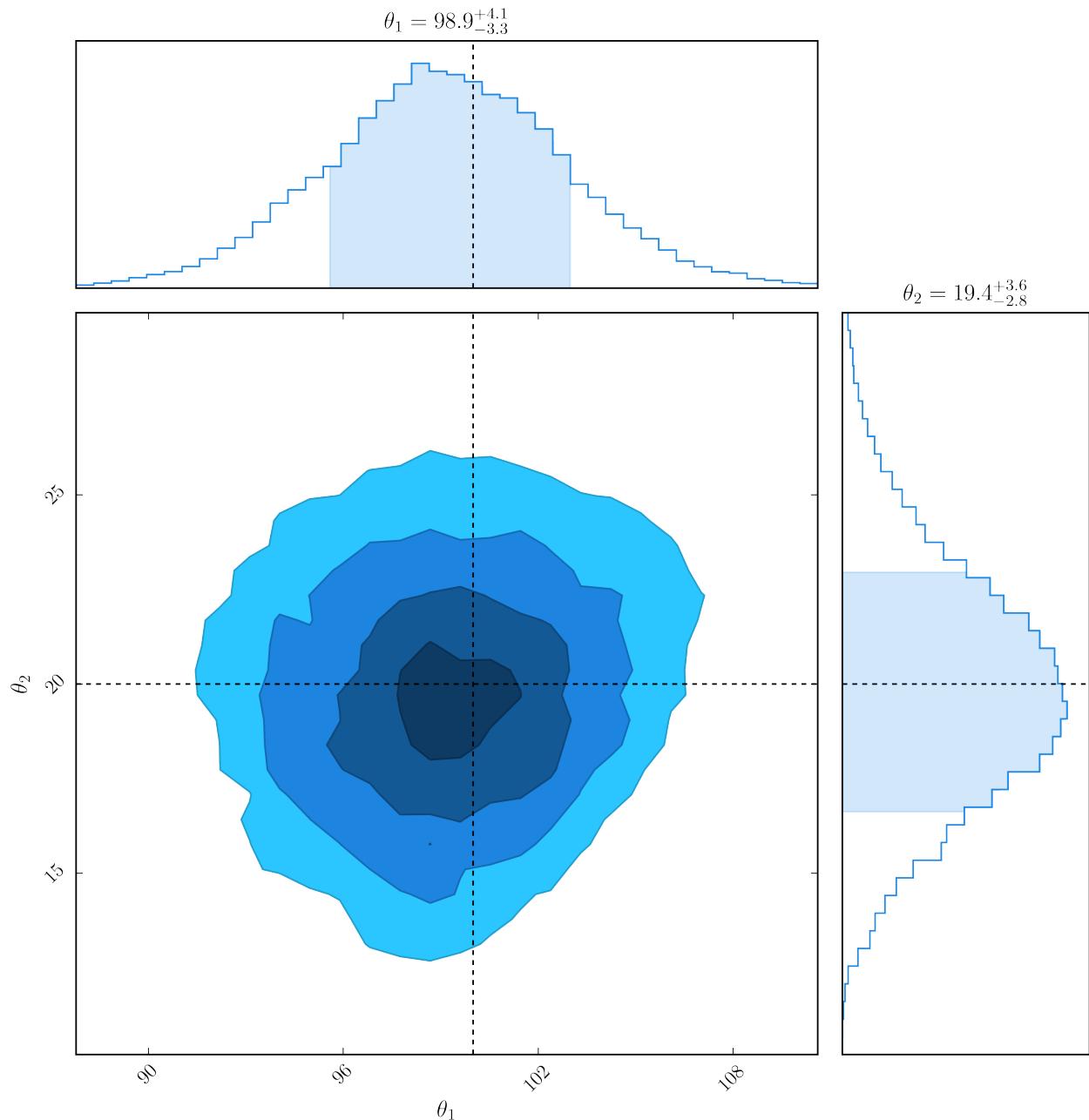
We model the prior enforcing realistic values

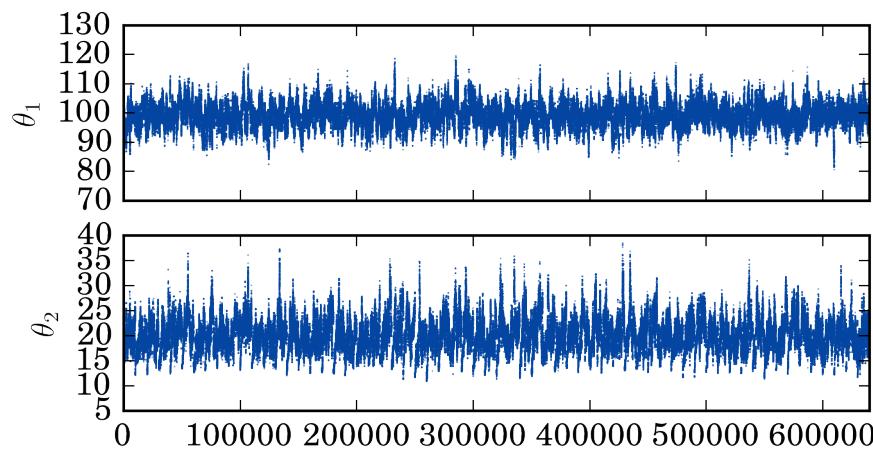
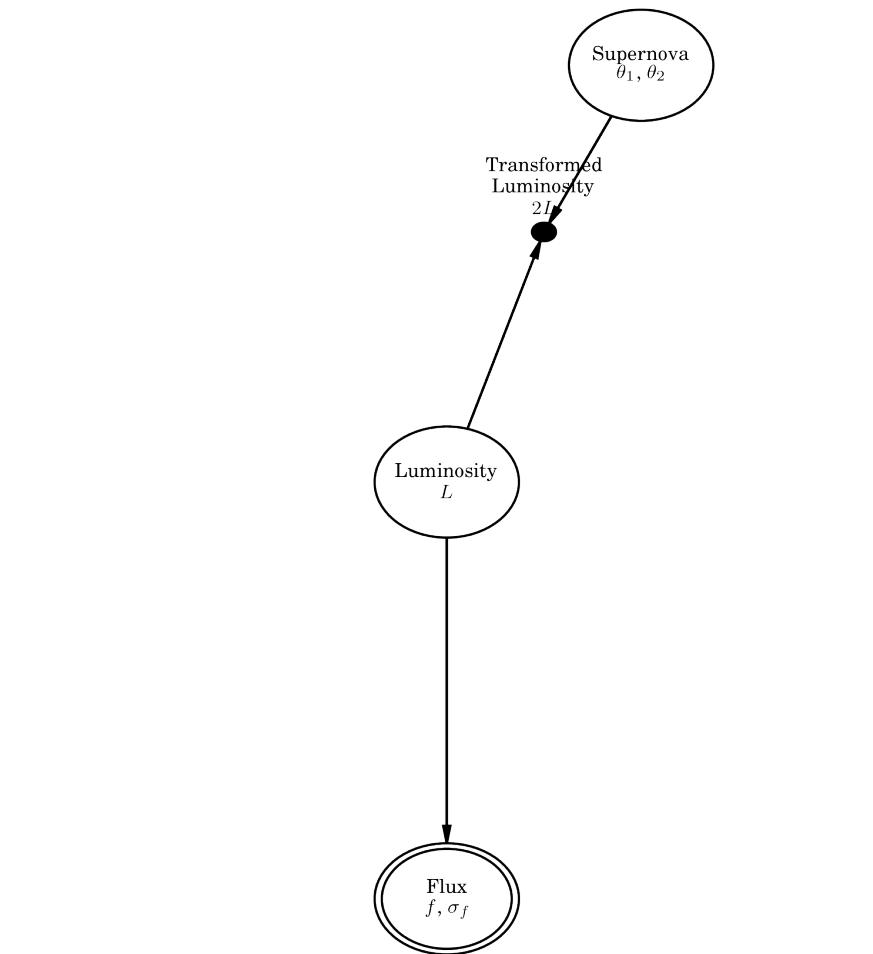
**get\_suggestion**(*data*)

**get\_suggestion\_requirements**()

**get\_suggestion\_sigma**(*data*)

**class** dessn.examples.simple.modelbased.exampleModel.UnderlyingSupernovaDistribution2  
 Bases: *dessn.model.parameter.ParameterUnderlying*





```
get_log_prior(data)
    We model the prior enforcing realistic values

get_suggestion(data)

get_suggestion_requirements()

get_suggestion_sigma(data)

class dessn.examples.simple.modelbased.exampleModel.UselessTransformation
    Bases: dessn.model.parameter.ParameterTransformation
```

## Submodules

### dессн.examples.simple.example module

```
class dessn.examples.simple.example.Example(n=30, theta_1=100.0, theta_2=20.0)
    Bases: object
```

Setting up the math for some examples.

Let us assume that we are observing supernova that are drawn from an underlying supernova distribution parameterised by  $\theta$ , where the supernova itself simply a luminosity  $L$ . We measure the luminosity of multiple supernovas, giving us an array of measurements  $D$ . If we wish to recover the underlying distribution of supernovas from our measurements, we wish to find  $P(\theta|D)$ , which is given by

$$P(\theta|D) \propto P(D|\theta)P(\theta)$$

Note that in the above equation, we realise that  $P(D|L) = \prod_{i=1}^N P(D_i|L_i)$  as our measurements are independent. The likelihood  $P(D|\theta)$  is given by

$$P(D|\theta) = \prod_{i=1}^N \int_{-\infty}^{\infty} P(D_i|L_i)P(L_i|\theta)dL_i$$

We now have two distributions to characterise. Let us assume both are gaussian, that is our observed luminosity  $x_i$  has gaussian error  $\sigma_i$  from the actual supernova luminosity, and the supernova luminosity is drawn from an underlying gaussian distribution parameterised by  $\theta$ .

$$\begin{aligned} P(D_i|L_i) &= \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(x_i - L_i)^2}{2\sigma_i^2}\right) \\ P(L_i|\theta) &= \frac{1}{\sqrt{2\pi}\theta_2} \exp\left(-\frac{(L_i - \theta_1)^2}{2\theta_2^2}\right) \end{aligned}$$

This gives us a likelihood of

$$P(D|\theta) = \prod_{i=1}^N \frac{1}{2\pi\theta_2\sigma_i} \int_{-\infty}^{\infty} \exp\left(-\frac{(x_i - L_i)^2}{2\sigma_i^2} - \frac{(L_i - \theta_1)^2}{2\theta_2^2}\right) dL_i$$

Working in log space for as much as possible will assist in numerical precision, so we can rewrite this as

$$\log(P(D|\theta)) = \sum_{i=1}^N \left[ \log\left(\int_{-\infty}^{\infty} \exp\left(-\frac{(x_i - L_i)^2}{2\sigma_i^2} - \frac{(L_i - \theta_1)^2}{2\theta_2^2}\right) dL_i\right) - \log(2\pi\theta_2\sigma_i) \right]$$

**Parameters** n : int, optional

The number of supernova to ‘observe’

---

**theta\_1** : float, optional  
The mean of the underlying supernova luminosity distribution

**theta\_2** : float, optional  
The standard deviation of the underlying supernova luminosity distribution

**do\_emcee** (*nwalkers=None*, *nburn=None*, *nsteps=None*)  
Abstract method to configure the emcee parameters

**static get\_data** (*n=50*, *theta\_1=100.0*, *theta\_2=20.0*, *scale=1.0*, *seed=1*)

**get\_likelihood** (*theta*, *data*, *error*)  
Abstract method to return the log likelihood

**get\_posterior** (*theta*, *data*, *error*)  
Gives the log posterior probability given the supplied input parameters.

**Parameters** **theta** : array of model parameters  
**data** : array of length *n*  
An array of observed luminosities  
**error** : array of length *n*  
An array of observed luminosity errors

**Returns** float  
the log posterior probability

**get\_prior** (*theta*)  
Get the log prior probability given the input.  
The prior distribution is currently implemented as flat prior.

**Parameters** **theta** : array of model parameters  
**Returns** float  
the log prior probability

**plot\_observations** ()  
Plot the observations and observation distribution.

## dессн.examples.simple.exampleIntegral module

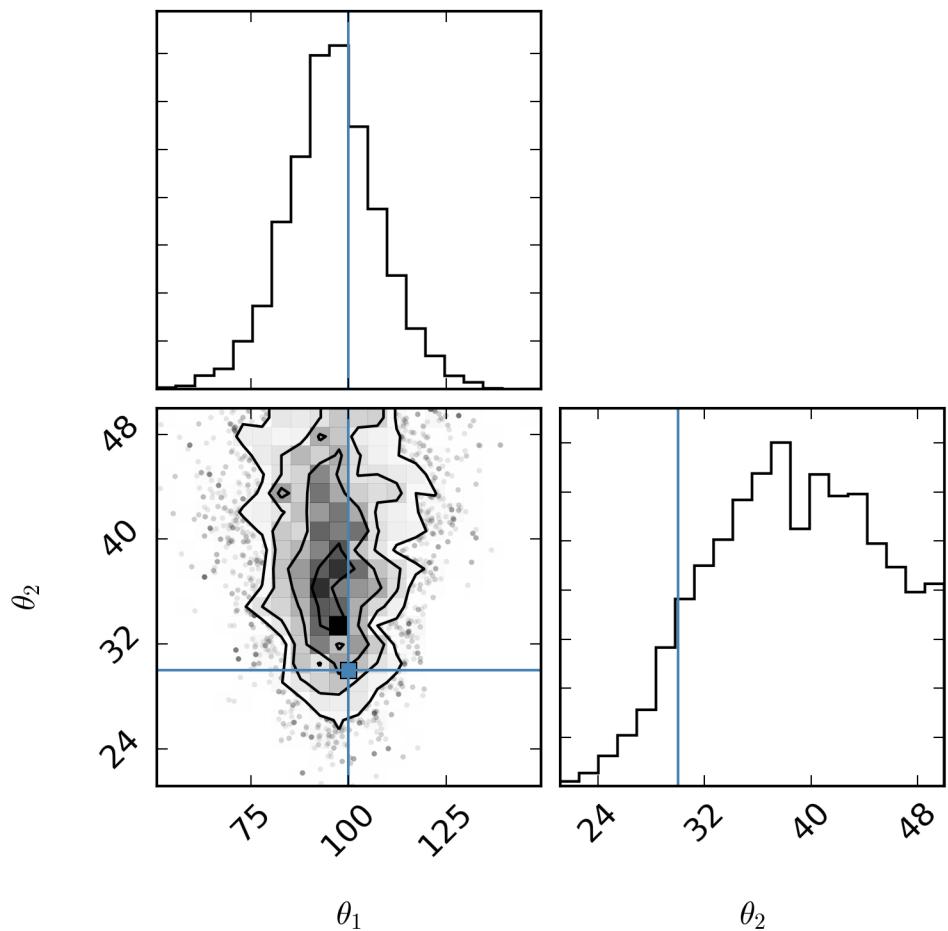
**class** dессн.examples.simple.exampleIntegral.**ExampleIntegral** (*n=10*, *theta\_1=100.0*, *theta\_2=30.0*)  
Bases: [dессн.examples.simple.example.Example](#)

An example implementation using integration over a latent parameter.

Building off the math from [Example](#) Creating this class will set up observations from an underlying distribution. Invoke `emcee` by calling the object. In this example, we perform the marginalisation inside the likelihood calculation, which gives us dimensionality only of two (the length of the  $\theta$  array). However, this is at the expense of performing the marginalisation over  $dL_i$ , as this requires computing *n* integrals for each step in the MCMC.

Note that I believe my numerical integration is not working properly, hence the weird output results. The moral of the story - it takes far, far longer to run than any other way of doing it, should still be the take home message from this.

**Parameters** **n** : int, optional  
The number of supernova to ‘observe’



**theta\_1** : float, optional

The mean of the underlying supernova luminosity distribution

**theta\_2** : float, optional

The standard deviation of the underlying supernova luminosity distribution

**do\_emcee** (*nwalkers*=16, *nburn*=500, *nsteps*=1000)

Run the *emcee* chain and produce a corner plot.

Saves a png image of the corner plot to plots/exampleIntegration.png.

**Parameters** **nwalkers** : int, optional

The number of walkers to use. Minimum of four.

**nburn** : int, optional

The burn in period of the chains.

**nsteps** : int, optional

The number of steps to run

**get\_likelihood** (*theta*, *data*, *error*)

Gets the log likelihood given the supplied input parameters.

**Parameters** **theta** : array of size 2

An array representing  $[\theta_1, \theta_2]$

**data** : array of length *n*

An array of observed luminosities

**error** : array of length *n*

An array of observed luminosity errors

**Returns** float

the log likelihood probability

## dessn.examples.simple.exampleLatent module

```
class dessn.examples.simple.exampleLatent.ExampleLatent(n=30, theta_1=100.0, theta_2=20.0)
```

Bases: [dessn.examples.simple.example.Example](#)

An example implementation using marginalisation over latent parameters.

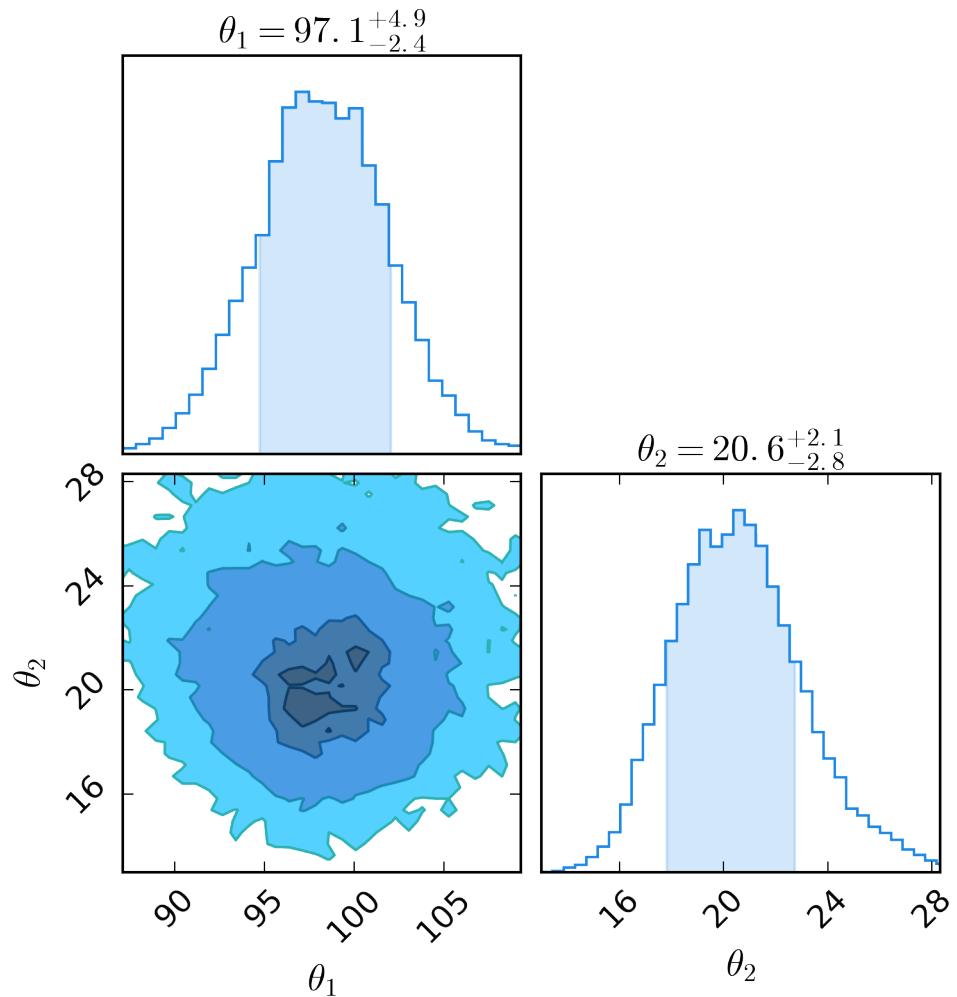
Building off the math from [Example](#), instead of performing the integration numerically in the computation of the likelihood, we can instead use Monte Carlo integration by simply setting the latent parameters  $\vec{L}$  as free parameters, giving us

$$\log \left( P(D|\theta, \vec{L}) \right) = - \sum_{i=1}^N \left[ \frac{(x_i - L_i)^2}{\sigma_i^2} + \frac{(L_i - \theta_1)^2}{\theta_2^2} + \log(2\pi\theta_2\sigma_i) \right]$$

Creating this class will set up observations from an underlying distribution. Invoke *emcee* by calling the object. In this example, we marginalise over  $L_i$  after running our MCMC, and so we no longer have to compute integrals in our chain, but instead have dimensionality of  $2 + n$ , where *n* are the number of observations.

**Parameters** **n** : int, optional

The number of supernova to ‘observe’



**theta\_1** : float, optional

The mean of the underlying supernova luminosity distribution

**theta\_2** : float, optional

The standard deviation of the underlying supernova luminosity distribution

**do\_emcee** (*nwalkers*=500, *nburn*=500, *nsteps*=1000)

Run the *emcee* chain and produce a corner plot.

Saves a png image of the corner plot to plots/exampleLatent.png.

**Parameters** **nwalkers** : int, optional

The number of walkers to use.

**nburn** : int, optional

The burn in period of the chains.

**nsteps** : int, optional

The number of steps to run

**get\_likelihood** (*theta*, *data*, *error*)

Gets the log likelihood given the supplied input parameters.

**Parameters** **theta** : array of length  $2 + n$

An array representing  $[\theta_1, \theta_2, \vec{L}]$

**data** : array of length  $n$

An array of observed luminosities

**error** : array of length  $n$

An array of observed luminosity errors

**Returns** float

the log likelihood probability

## dессн.model package

### Submodules

#### dессн.model.edge module

**class** dессн.model.edge.Edge (*probability\_of*, *given*)

Bases: object

An edge connection one or more parameters to one or more different parameters.

An edge is a connection between parameters (*not* Nodes), and signifies a joint probability distribution. That is, if in our mathematical definition of our model, we find the term  $P(a, b|c, d, e)$ , this would be represented by a single edge. Similarly,  $P(a|b)P(b|c, d)$  would be two edges.

**Parameters** **probability\_of** : str or list[str]

The dependent parameters. With the example  $P(a, b|c, d)$ , this input would be `['a', 'b']`.

**given** : str or list[str]

In the example  $P(a, b|c, d)$ , this input would be `['c', 'd']`.

**get\_log\_likelihood(data)**

Gets the log likelihood of this edge.

For example, if we had

$$P(a, b|c, d) = \frac{1}{\sqrt{2\pi}d} \exp\left(-\frac{(ab - c)^2}{d^2}\right),$$

we could implement this function as `return -np.log(np.sqrt(2*np.pi)*data['d']) - (data['a']*data['b'] - data['c'])**2/(data['d']**2)`

**Returns float**

the log likelihood given the supplied data and the model parametrisation.

**class dessn.model.edge.EdgeTransformation(probability\_of, given)**

Bases: `dessn.model.edge.Edge`

This specialised edge is used to connect to transformation nodes.

A transformation edge does not give a likelihood, but - as it is a known transformation - returns a dictionary when `get_transformation` is invoked that is injected into the data dictionary given to regular edges.

See [LuminosityToAdjusted](#) for a simple example.

**Parameters probability\_of : str or list[str]**

The dependent parameters. With the example  $P(a, b|c, d)$ , (assuming the functional form is a delta), this input would be `['a', 'b']`.

**given : str or list[str]**

In the example  $P(a, b|c, d)$ , (assuming the functional form is a delta), this input would be `['c', 'd']`.

**get\_log\_likelihood(data)**

**get\_transformation(data)**

Calculates the new parameters from the given data

**Returns dict**

a dictionary containing a value for each parameter given in `transform_to`

**dessn.model.model module**

**class dessn.model.model.Model(model\_name)**

Bases: `object`

A generalised model for use in arbitrary situations.

A model is, at heart, simply a collection of nodes and edges. Apart from simply being a container in which to place nodes and edges, the model is also responsible for figuring out how to connect edges (which map to parameters) with the right nodes, for sorting edges such that when an edge is evaluated all its required data has been generated by other nodes or edges, for managing the emcee running, and also for generating the visual PGMs.

It is thus a complex class, and I expect, as of writing this summary, it contains numerous bugs.

**Parameters model\_name : str**

The model name, used for serialisation

**add\_edge**(*edge*)

Adds an edge into the models collection of edges

**Parameters** *edge* : *Edge*

**add\_node**(*node*)

Adds a node into the models collection of nodes.

**Parameters** *node* : *Parameter*

**finalise**()

Finalises the model.

This method runs consistency checks on the model (making sure there are not orphaned nodes, edges to parameters that do not exist, etc), and in doing so links the right edges to the right nodes and determines the order in which edges should be evaluated.

You can manually call this method after setting all nodes and edges to confirm as early as possible that the model is valid. If you do not call it manually, this method is invoked by the model when requesting concrete information, such as the PGM or model fits.

**fit\_model**(*num\_temps=None*, *num\_walkers=None*, *num\_steps=5000*, *num\_burn=3000*,  
*temp\_dir=None*, *save\_interval=300*, *p0=None*)

Uses emcee to fit the supplied model.

This method sets an emcee run using the EnsembleSampler and manual chain management to allow for very high dimension models. MPI running is detected automatically for less hassle, and chain progress is serialised to disk automatically for convenience.

This method works... but is still a work in progress

**Parameters** *num\_temps* : int, optional

The number of temperature to run. If none, does not use PTSampler

**num\_walkers** : int, optional

The number of walkers to run. If not supplied, it defaults to eight times the model dimensionality

**num\_steps** : int, optional

The number of steps to run

**num\_burn** : int, optional

The number of steps to discard for burn in

**temp\_dir** : str

If set, specifies a directory in which to save temporary results, like the emcee chain

**save\_interval** : float

The amount of seconds between saving the chain to file. Setting to None disables serialisation.

**Returns** ndarray

The final flattened chain of dimensions (*num\_dimensions*, *num\_walkers* \* (*num\_steps* - *num\_burn*))

fig

The corner plot figure returned from `corner.corner(...)`

**get\_consumer**()

```
get_log_likelihood(theta)
get_log_prior(theta)
get_pgm(filename=None)
```

Renders (and returns) a PGM of the current model.

**Parameters** filename : str, optional

if the filename is set, the PGM is saved to file in the top level plots directory.

**Returns** daft.PGM

The daft PGM class, for further customisation if required.

## **dessn.model.parameter module**

```
class dessn.model.parameter.Parameter(name, label, group=None)
```

Bases: object

A parameter represented on a node on a PGM model. Multiple parameters can be assigned to the same node on a PGM by giving them the same group.

The Parameter class can essentially be thought of as a wrapper around a parameter or variable in your model.

This class is an abstract class, and cannot be directly instantiated. Instead, instantiate one of the provided subclasses, as detailed below.

**Parameters** name : str

The parameter name, used as the key to access this parameter in the data object

**label** : str

The parameter label, for use in plotting and PGM creation.

**group** : str, optional

The group in the PGM that this parameter belongs to. Will replace name on the PGM if set.

**get\_suggestion**(data)

This function is used when finding a good starting position for the sampler.

The better this suggestion is, the less burn in time is needed. As parameters can vary by orders of magnitude, and have allowed ranges (some parameters cannot be negative for example), local optimisation methods do not always work when starting with some arbitrary and random initial condition. As such, overriding this parameter is required for all latent and underlying parameters.

**Parameters** data : dictionary

The parameters and observed data to generate a suggested parameter from

**Returns** float

a suggested parameter

**get\_suggestion\_requirements**()

Returns suggestion parameter requirements.

**get\_suggestion\_sigma**(data)

Starting all walkers from the same position is not a good thing to do, so the suggested starting positions given by the `get_suggestion()` need to be randomised slightly so that the walkers start in different positions. This is done by taking the suggested parameter and adding gaussian noise to it, where the

scale/width/standard deviation of the spread is given by this function. Overestimating this value to try and ensure a proper spread of walker positions can lead to complications and increased convergence, so don't always think bigger is better!

**Parameters** `data` : dictionary

The parameters and observed data to generate a suggested parameter from

**Returns** float

a suggested parameter sigma, used to randomise the suggest parameter

```
class dessn.model.parameter.ParameterDiscrete(name, label, group=None)
Bases: dessn.model.parameter.Parameter
```

A parameter representing a discrete variable in our model.

Unlike latent variables which can be easily marginalised over, discrete variables simply create more issues than they are worth. Algorithms like Hamiltonian Monte Carlo require continuous posterior surfaces, which - off the bat - simply rule out discrete parameters. As such, discrete parameters in the model are integrated out (really, they are summed over). Examples of discrete parameters might be supernova types, which galaxy is the actual transient host, or trivially whether a coin was flipped to be heads or tails.

Discrete parameters must implement a `get_discrete()` method, which needs to return the discrete options for the particular step in the model. Some discrete options may be global (for example, we would consider all supernova type combinations with each supernova), however some can be dependent on the current observation (some supernova might have only one possible host, others might have two or more). As the possible types can be dependent on observation, the `get_discrete_requirements()` method also exists and can be overridden. It functions identically to the `get_suggestion_requirements` method in the `ParameterLatent` class.

For examples how to use latent parameters, see the example give by [DiscreteModel](#).

**Parameters** `name` : str

The parameter name, used as the key to access this parameter in the data object

`label` : str

The parameter label, for use in plotting and PGM creation.

`group` : str, optional

The group in the PGM that this parameter belongs to. Will replace `name` on the PGM if set.

`get_discrete(data)`

Returns the possible discrete types for this parameter.

**Parameters** `data` : dictionary

Contains parameter values and observations for the particular step in the chain and observation

**Returns** list

A list of types which are iterated over.

`get_discrete_requirements()`

Gets the data and parameters required for generating the discrete values for this parameters

**Returns** list

Defaults to an empty list.

`get_suggestion(data)`

```
get_suggestion_requirements()
get_suggestion_sigma(data)

class dessn.model.parameter.ParameterLatent(name, label, group=None)
Bases: dessn.model.parameter.Parameter

A parameter representing a latent, or hidden, variable in our model.

Given infinitely powerful computers, these nodes would not be necessary, for they represent marginalisation over unknown / hidden / latent parameters in the model, and we would simple integrate them out when computing the likelihood probability. However, this is not the case, and it is more efficient to simply incorporate latent parameters into our model and essentially marginalise over them using Monte Carlo integration. We thus trade explicit numerical integration in each step of our calculation for increased dimensionality.

For examples on why and how to use latent parameters, see the examples beginning in Example.
```

**Parameters** `name` : str  
The parameter name, used as the key to access this parameter in the data object

`label` : str  
The parameter label, for use in plotting and PGM creation.

`group` : str, optional  
The group in the PGM that this parameter belongs to. Will replace `name` on the PGM if set.

```
get_num_latent()  
The number of latent parameters to include in the model.  
Running MCMC requires knowing the dimensionality of our model, which means knowing how many latent parameters (realisations of an underlying hidden distribution) we require.  
For example, if we observe a hundred supernova drawn from an underlying supernova distribution, we would have to realise a hundred latent variables - one per data point.
```

**Returns** int  
the number of latent parameters required by this node

```
class dessn.model.parameter.ParameterObserved(name, label, data, group=None)
Bases: dessn.model.parameter.Parameter

A parameter representing an observed variables

This parameter is used for all observables in the model. In addition to a normal parameter, it also contains data, which should be a list of n elements (for n data points), with each element allowed to be an arbitrary data type. This data is what is given to the incoming and outgoing node edges to calculate likelihoods. It is very important that, if your model has multiple observed parameters, each observed parameter returns lists of the same length.

Parameters name : str  
The parameter name, used as the key to access this parameter in the data object

label : str  
The parameter label, for use in plotting and PGM creation.



data : list[object]  
The data list to supply to the edges.



group : str, optional


```

---

The group in the PGM that this parameter belongs to. Will replace name on the PGM if set.

**get\_data()**

Returns a dictionary containing keys of the parameter names and values of the parameter data object

**get\_suggestion(data)****get\_suggestion\_requirements()****get\_suggestion\_sigma(data)**

**class** dessn.model.parameter.**ParameterTransformation**(name, label, group=None)

Bases: *dessn.model.parameter.Parameter*

A parameter representing a variable transformation.

This parameter essentially represents latent variables which are fully determined - their probability is given by a delta function. Examples of this might be the luminosity distance, as it is known exactly when given cosmology and redshift. Or it might represent a conversion between observed flux and actual flux, given we have a well defined flux correction.

On a PGM, this parameter would be represented by a point, not an ellipse.

**Parameters** **name** : str

The parameter name, used as the key to access this parameter in the data object

**label** : str

The parameter label, for use in plotting and PGM creation.

**group** : str, optional

The group in the PGM that this parameter belongs to. Will replace name on the PGM if set.

**get\_suggestion(data)****get\_suggestion\_requirements()****get\_suggestion\_sigma(data)**

**class** dessn.model.parameter.**ParameterUnderlying**(name, label, group=None)

Bases: *dessn.model.parameter.Parameter*

A parameter representing an underlying parameter in your model.

On the PGM, these nodes would be at the very top, and would represent the variables we are trying to fit for, such as  $\Omega_M$ .

These nodes are required to implement the abstract method `get_log_prior`. In addition, they must also implement the abstract method `get_suggestion`, which returns a suggested value for the parameter, given some input specified by the `get_suggestion_requirements`. By default `get_suggestion_requirements` specifies no requirements, however you can override this method, such that it returns a list of required data to generate a suggestion.

**Parameters** **name** : str

The parameter name, used as the key to access this parameter in the data object

**label** : str

The parameter label, for use in plotting and PGM creation.

**group** : str, optional

The group in the PGM that this parameter belongs to. Will replace name on the PGM if set.

**get\_log\_prior**(*data*)

Returns the log prior for the parameter.

**Parameters** *data* : dic

A dictionary containing all data and the model parameters being tested at a given step in the MCMC chain. For this class, if the class was instantiated with a name of “omega\_m”, the input dictionary would have the key “omega\_m”, and the value of “omega\_m” at that particular step in your chain.

**Returns** float

the log prior probability given the current value of the parameters

**get\_suggestion\_requirements()**

## dessn.simulation package

### Submodules

#### dessn.simulation.simulation module

**class** dessn.simulation.simulation.**Simulation**

Bases: object

**get\_simulation**(*omega\_m*, *h\_0*, *snIa\_mean*, *snIa\_sigma*, *snII\_mean*, *snII\_sigma*, *r*, *num\_trans*=50)

## dessn.toy package

This module will contain the original toy model when implemented.

### Submodules

#### dessn.toy.edges module

**class** dessn.toy.edges.**ToCount**

Bases: *dessn.model.edge.Edge*

**get\_log\_likelihood**(*data*)

Given CCD efficiency, convert from count to flux *f*. We go from counts, assume Poisson error, to get an observed flux and flux error from counts. From that, we can calculate the likelihood of an actual flux, given our observed flux and observed flux error, assuming a normal distribution:

$$f_{\text{observed}} = \frac{\text{count}}{\text{conversion} \times \text{efficiency}}$$
$$\sigma_f = \frac{\sqrt{\text{count}}}{\text{conversion} \times \text{efficiency}}$$
$$P(\text{count}_o | f) = \frac{1}{\sqrt{2\pi}\sigma_f} \exp\left(-\frac{(f_{\text{observed}} - f)^2}{2\sigma_f^2}\right)$$

---

```
class dessn.toy.edges.ToFlux
```

Bases: `dessn.model.edge.EdgeTransformation`

```
get_transformation(data)
```

Gets flux from the luminosity distance and luminosity. Note that the luminosity here is actually the **log luminosity**

$$f = \frac{L}{4\pi D_L^2}$$

```
class dessn.toy.edges.ToLuminosity
```

Bases: `dessn.model.edge.Edge`

```
get_log_likelihood(data)
```

Assume type is 0 for a Type SnIa, or 1 for SnII. It will be continuous, so we round the variable.

If we have a type SnIa supernova, we use the type SnIa distribution, which is modelled as a gaussian.

We should also note clearly that luminosity here is actually **log luminosity**, we work in log space.

$$P(L|\mu_{\text{SnIa}}, \sigma_{\text{SnIa}}) = \frac{1}{\sqrt{2\pi}\sigma_{\text{SnIa}}} \exp\left(-\frac{(L - \mu_{\text{SnIa}})^2}{2\sigma_{\text{SnIa}}^2}\right)$$

If we have a type SnII supernova, we use the type SnII distribution, which is also modelled as a gaussian.

$$P(L|\mu_{\text{SnII}}, \sigma_{\text{SnII}}) = \frac{1}{\sqrt{2\pi}\sigma_{\text{SnII}}} \exp\left(-\frac{(L - \mu_{\text{SnII}})^2}{2\sigma_{\text{SnII}}^2}\right)$$

```
class dessn.toy.edges.ToLuminosityDistance
```

Bases: `dessn.model.edge.EdgeTransformation`

```
get_transformation(data)
```

```
class dessn.toy.edges.ToRate
```

Bases: `dessn.model.edge.Edge`

```
get_log_likelihood(data)
```

The likelihood of having the supernova types  $T$  given supernova rate  $r$ .

We model the supernova rate as a binomial process, with rate  $r$ . That is, given  $x$  type Ia supernova and  $y$  type II supernova, our pdf is given by

$$P(T|r) = \begin{cases} r, & \text{if } T = \text{SnIa} \\ 1 - r, & \text{if } T = \text{SnII} \end{cases}$$

```
class dessn.toy.edges.ToRedshift
```

Bases: `dessn.model.edge.EdgeTransformation`

```
get_transformation(data)
```

```
class dessn.toy.edges.ToType
```

Bases: `dessn.model.edge.Edge`

```
get_log_likelihood(data)
```

Gets the probability of the actual object being of one type, given we observe a singular other type.

That is, if we think we observe a type Ia supernova, what is the probability it is actually a type Ia, and what is the probability it is a different type of supernova.

At the moment, this is a trivial function, where we assume that we are correct 90% of the time.

Also note that the input types (accessed by the `type` key) are continuous, and we therefore round them to get the discrete type. The method of changing from continuous to discrete will probably update in the future.

$$P(T_o|T) = 0.01 + 0.98\delta_{T_o,T}$$

### **dessn.toy.latent module**

```
class dessn.toy.latent.Luminosity(n)
    Bases: dessn.model.parameter.ParameterLatent
        get_num_latent()
        get_suggestion(data)
        get_suggestion_requirements()
        get_suggestion_sigma(data)

class dessn.toy.latent.Redshift(n)
    Bases: dessn.model.parameter.ParameterTransformation
        get_num_latent()

class dessn.toy.latent.Type(n)
    Bases: dessn.model.parameter.ParameterDiscrete
        get_discrete(data)
        get_discrete_requirements()
        get_num_latent()
        get_suggestion(data)
        get_suggestion_requirements()
```

### **dessn.toy.observed module**

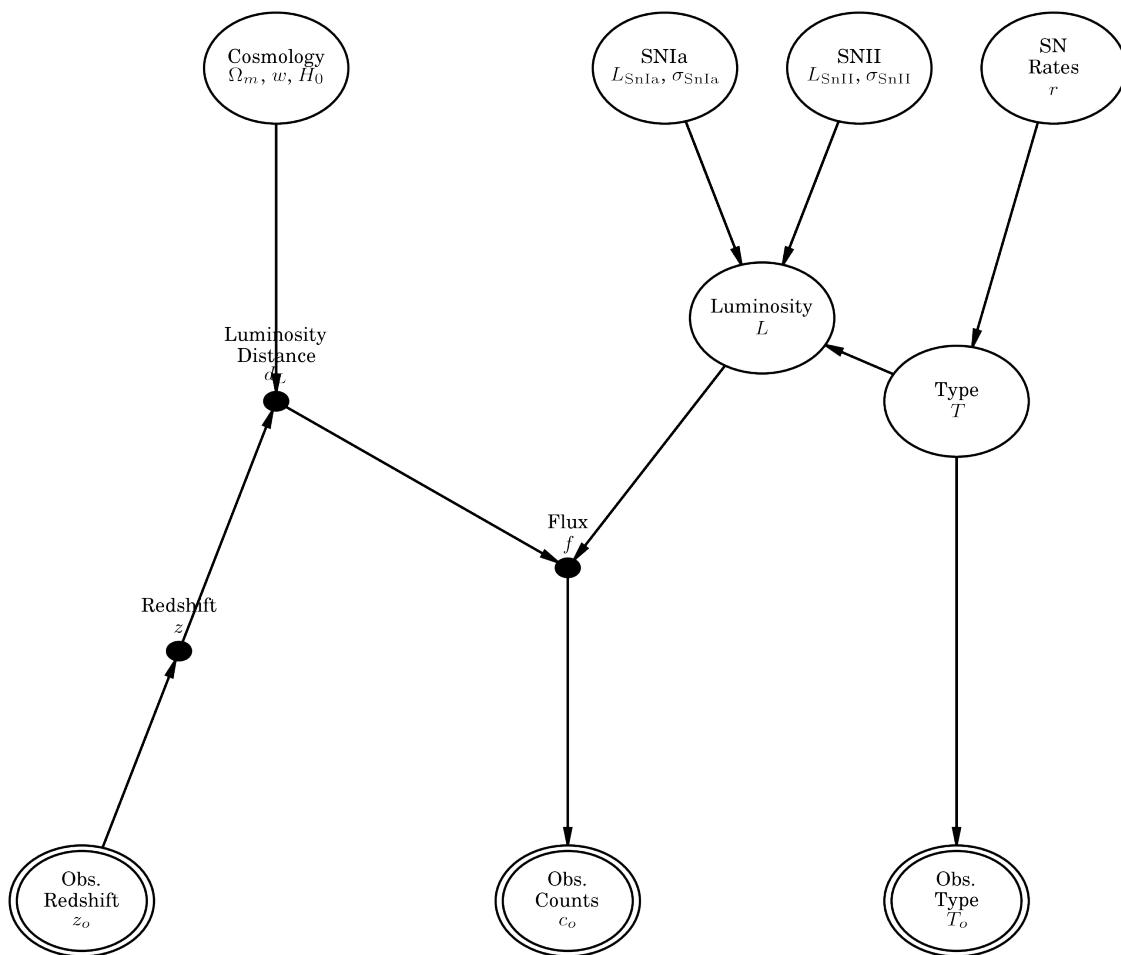
```
class dessn.toy.observed.ObservedCounts(counts)
    Bases: dessn.model.parameter.ParameterObserved
class dessn.toy.observed.ObservedRedshift(redshifts)
    Bases: dessn.model.parameter.ParameterObserved
class dessn.toy.observed.ObservedType(types)
    Bases: dessn.model.parameter.ParameterObserved
```

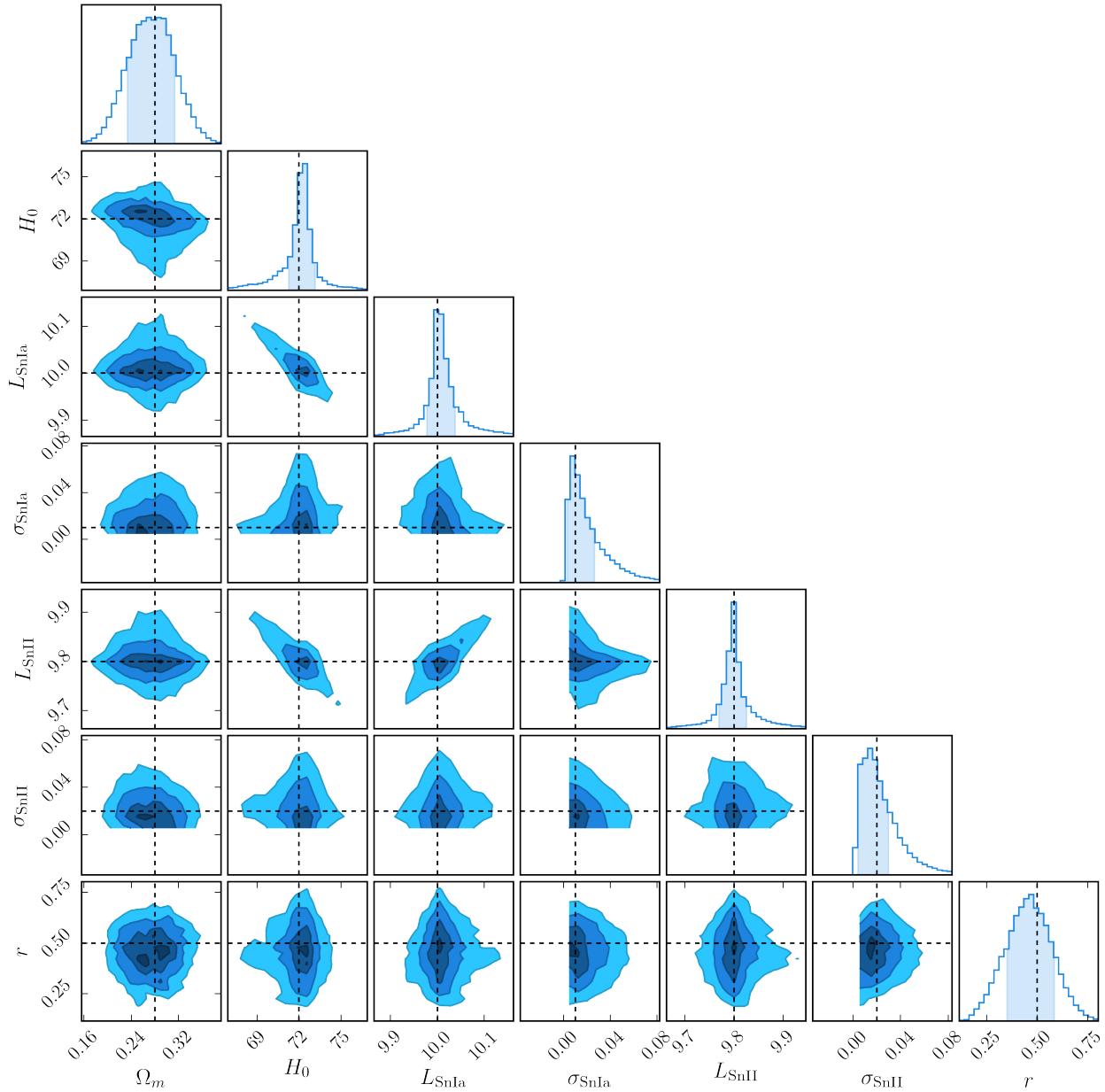
### **dessn.toy.toyModel module**

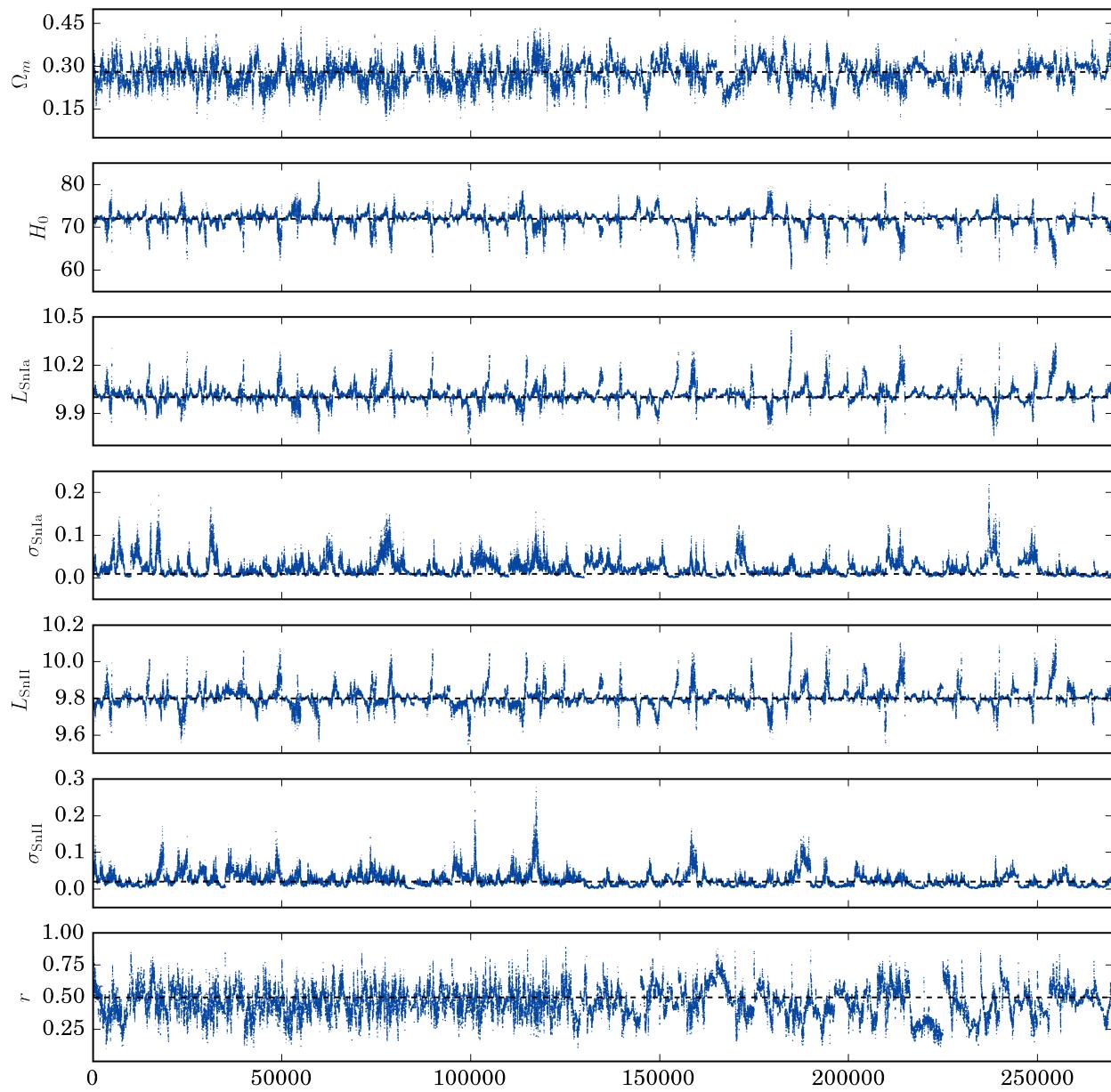
```
class dessn.toy.toyModel.ToyModel(observations)
    Bases: dessn.model.model.Model
```

A modified toy model. The likelihood surfaces and PGM model are given below.

Probabilities and model details are recorded in the model parameter and edge classes.







**dessn.toy.transformations module**

```
class dessn.toy.transformations.Flux
    Bases: dessn.model.parameter.ParameterTransformation

class dessn.toy.transformations.LuminosityDistance
    Bases: dessn.model.parameter.ParameterTransformation
```

**dessn.toy.underlying module**

```
class dessn.toy.underlying.Hubble
    Bases: dessn.model.parameter.ParameterUnderlying

    get_log_prior(data)
    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.OmegaM
    Bases: dessn.model.parameter.ParameterUnderlying

    get_log_prior(data)
    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.SupernovaIIDist1
    Bases: dessn.model.parameter.ParameterUnderlying

    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.SupernovaIIDist2
    Bases: dessn.model.parameter.ParameterUnderlying

    get_log_prior(data)
    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.SupernovaIaDist1
    Bases: dessn.model.parameter.ParameterUnderlying

    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.SupernovaIaDist2
    Bases: dessn.model.parameter.ParameterUnderlying

    get_log_prior(data)
    get_suggestion(data)
    get_suggestion_sigma(data)

class dessn.toy.underlying.SupernovaRate
    Bases: dessn.model.parameter.ParameterUnderlying

    get_log_prior(data)
```

```
get_suggestion(data)
get_suggestion_sigma(data)

class dessn.toy.underlying.W
    Bases: dessn.model.parameter.ParameterUnderlying
    get_log_prior(data)
    get_suggestion(data)
    get_suggestion_sigma(data)
```

## dessn.utility package

### Submodules

#### dessn.utility.hdemcee module

```
class dessn.utility.hdemcee.EmceeWrapper(sampler)
    Bases: object
    get_results(num_burn)
    run_chain(num_steps, num_burn, num_walkers, num_dim, start=None, save_interval=300,
              save_dim=None, temp_dir=None)
```

#### dessn.utility.newtonian module

```
class dessn.utility.newtonian.NewtonianPosition(nodes, edges, top=None, bottom=None)
    Bases: object
    fit(plot=False)
    iterate(p, v, i)
```



**d**

dessn, 11  
dessn.chain, 11  
dessn.chain.chain, 11  
dessn.chain.demoOneChain, 15  
dessn.chain.demoTable, 15  
dessn.chain.demoThreeChains, 17  
dessn.chain.demoTwoDisjointChains, 17  
dessn.chain.demoVarious, 17  
dessn.chain.demoWalk, 29  
dessn.examples, 29  
dessn.examples.discrete, 29  
dessn.examples.discrete.discrete, 30  
dessn.examples.simple, 32  
dessn.examples.simple.example, 36  
dessn.examples.simple.exampleIntegral,  
    37  
dessn.examples.simple.exampleLatent, 39  
dessn.examples.simple.modelbased, 32  
dessn.examples.simple.modelbased.exampleModel,  
    33  
dessn.model, 41  
dessn.model.edge, 41  
dessn.model.model, 42  
dessn.model.parameter, 44  
dessn.simulation, 48  
dessn.simulation.simulation, 48  
dessn.toy, 48  
dessn.toy.edges, 48  
dessn.toy.latent, 50  
dessn.toy.observed, 50  
dessn.toy.toyModel, 50  
dessn.toy.transformations, 54  
dessn.toy.underlying, 54  
dessn.utility, 55  
dessn.utility.hdemcee, 55  
dessn.utility.newtonian, 55



**A**

add\_chain() (dessn.chain.chain.ChainConsumer method),  
   11  
 add\_edge() (dessn.model.model.Model method), 42  
 add\_node() (dessn.model.model.Model method), 43

**C**

ChainConsumer (class in dessn.chain.chain), 11  
 Colour (class in dessn.examples.discrete.discrete), 30  
 configure\_bar() (dessn.chain.chain.ChainConsumer  
   method), 11  
 configure\_contour() (dessn.chain.chain.ChainConsumer  
   method), 12  
 configure\_general() (dessn.chain.chain.ChainConsumer  
   method), 12  
 configure\_truth() (dessn.chain.chain.ChainConsumer  
   method), 12

**D**

DemoOneChain (class in dessn.chain.demoOneChain),  
   15  
 DemoTable (class in dessn.chain.demoTable), 15  
 DemoThreeChains (class  
   dессн.chain.demoThreeChains), 17  
 DemoTwoDisjointChains (class  
   dессн.chain.demoTwoDisjointChains), 17  
 DemoVarious (class in dessn.chain.demoVarious), 17  
 DemoWalk (class in dessn.chain.demoWalk), 29  
 dessn (module), 11  
 dessn.chain (module), 11  
 dessn.chain.chain (module), 11  
 dessn.chain.demoOneChain (module), 15  
 dessn.chain.demoTable (module), 15  
 dessn.chain.demoThreeChains (module), 17  
 dessn.chain.demoTwoDisjointChains (module), 17  
 dessn.chain.demoVarious (module), 17  
 dessn.chain.demoWalk (module), 29  
 dessn.examples (module), 29  
 dessn.examples.discrete (module), 29  
 dessn.examples.discrete.discrete (module), 30  
 dessn.examples.simple (module), 32  
 dessn.examples.simple.example (module), 36

dessn.examples.simple.exampleIntegral (module), 37  
 dessn.examples.simple.exampleLatent (module), 39  
 dessn.examples.simple.modelbased (module), 32  
 dessn.examples.simple.modelbased.exampleModel  
   (module), 33  
 dessn.model (module), 41  
 dessn.model.edge (module), 41  
 dessn.model.model (module), 42  
 dessn.model.parameter (module), 44  
 dessn.simulation (module), 48  
 dessn.simulation.simulation (module), 48  
 dessn.toy (module), 48  
 dessn.toy.edges (module), 48  
 dessn.toy.latent (module), 50  
 dessn.toy.observed (module), 50  
 dessn.toy.toyModel (module), 50  
 dessn.toy.transformations (module), 54  
 dessn.toy.underlying (module), 54  
 dessn.utility (module), 55  
 dessn.utility.hdemcee (module), 55  
 dessn.utility.newtonian (module), 55  
 DiscreteModel (class  
   in  
     dессн.examples.discrete.discrete), 30  
 do\_emcee() (dессн.examples.simple.example.Example  
   method), 37  
 do\_emcee() (dессн.examples.simple.exampleIntegral.ExampleIntegral  
   method), 39  
 do\_emcee() (dессн.examples.simple.exampleLatent.ExampleLatent  
   method), 41

**E**

Edge (class in dessn.model.edge), 41  
 EdgeTransformation (class in dessn.model.edge), 42  
 EmceeWrapper (class in dessn.utility.hdemcee), 55  
 Example (class in dessn.examples.simple.example), 36  
 ExampleIntegral (class  
   in  
     dессн.examples.simple.exampleIntegral),  
   37  
 ExampleLatent (class  
   in  
     dессн.examples.simple.exampleLatent), 39  
 ExampleModel (class  
   in  
     dессн.examples.simple.modelbased.exampleModel),  
   33

## F

finalise() (dessn.model.model.Model method), 43  
fit() (dessn.utility.newtonian.NewtonianPosition method), 55  
fit\_model() (dessn.model.model.Model method), 43  
Flux (class in dessn.toy.transformations), 54  
FluxToLuminosity (class in dessn.examples.simple.modelbased.exampleModel), 33

## G

get\_consumer() (dessn.model.model.Model method), 43  
get\_data() (dessn.examples.simple.example.Example static method), 37  
get\_data() (dessn.model.parameter.ParameterObserved method), 47  
get\_data() (in module dessn.examples.discrete.discrete), 32  
get\_discrete() (dessn.examples.discrete.discrete.Colour method), 30  
get\_discrete() (dessn.model.parameter.ParameterDiscrete method), 45  
get\_discrete() (dessn.toy.latent.Type method), 50  
get\_discrete\_requirements()  
    (dessn.examples.discrete.discrete.Colour method), 30  
get\_discrete\_requirements()  
    (dessn.model.parameter.ParameterDiscrete method), 45  
get\_discrete\_requirements()  
    (dessn.toy.latent.Type method), 50  
get\_latex\_table() (dessn.chain.chain.ChainConsumer method), 13  
get\_likelihood() (dessn.examples.simple.example.Example method), 37  
get\_likelihood() (dessn.examples.simple.exampleIntegral.ExampleIntegral method), 39  
get\_likelihood() (dessn.examples.simple.exampleLatent.ExampleLatent method), 41  
get\_log\_likelihood() (dessn.examples.discrete.discrete.ToColour method), 30  
get\_log\_likelihood() (dessn.examples.discrete.discrete.ToColour<sub>2</sub> method), 32  
get\_log\_likelihood() (dessn.examples.discrete.discrete.ToRate method), 32  
get\_log\_likelihood() (dessn.examples.simple.modelbased.exampleModel.Luminosity method), 33  
get\_log\_likelihood() (dessn.examples.simple.modelbased.exampleModel), 33  
get\_log\_likelihood() (dessn.model.edge.Edge method), 42  
get\_log\_likelihood() (dessn.model.edge.EdgeTransformation method), 42

get\_log\_likelihood() (dessn.model.model.Model method), 43  
get\_log\_likelihood() (dessn.toy.edges.ToCount method), 48  
get\_log\_likelihood() (dessn.toy.edges.ToLuminosity method), 49  
get\_log\_likelihood() (dessn.toy.edges.ToRate method), 49  
get\_log\_likelihood() (dessn.toy.edges.ToType method), 49  
get\_log\_prior() (dessn.examples.discrete.discrete.UnderlyingRate method), 32  
get\_log\_prior() (dessn.examples.simple.modelbased.exampleModel.Underlying method), 33  
get\_log\_prior() (dessn.examples.simple.modelbased.exampleModel.Underlying method), 33  
get\_log\_prior() (dessn.model.model.Model method), 44  
get\_log\_prior() (dessn.model.parameter.ParameterUnderlying method), 48  
get\_log\_prior() (dessn.toy.underlying.Hubble method), 54  
get\_log\_prior() (dessn.toy.underlying.OmegaM method), 54  
get\_log\_prior() (dessn.toy.underlying.SupernovaIaDist2 method), 54  
get\_log\_prior() (dessn.toy.underlying.SupernovaIIDist2 method), 54  
get\_log\_prior() (dessn.toy.underlying.SupernovaRate method), 54  
get\_log\_prior() (dessn.toy.underlying.W method), 55  
get\_num\_latent() (dessn.examples.simple.modelbased.exampleModel.Latent method), 33  
get\_num\_latent() (dessn.model.parameter.ParameterLatent method), 46  
get\_num\_latent() (dessn.toy.latent.Luminosity method), 50  
get\_num\_latent() (dessn.toy.latent.Redshift method), 50  
get\_num\_latent() (dessn.toy.latent.Type method), 50  
get\_parameter\_text() (dessn.chain.chain.ChainConsumer method), 13  
get\_pgm() (dessn.model.model.Model method), 44  
get\_posterior() (dessn.examples.simple.example.Example method), 37  
get\_prior() (dessn.examples.simple.example.Example method), 37  
get\_results0() (dessn.utility.hdemcee.EmceeWrapper method), 55  
get\_simulation() (dessn.simulation.simulation.Simulation method), 48  
get\_suggestion() (dessn.examples.discrete.discrete.UnderlyingRate method), 32  
get\_suggestion() (dessn.examples.simple.modelbased.exampleModel.Latent method), 33

```

get_suggestion() (dессн.examples.simple.modelbased.exampleModelUnderlyingSupernovaDistribution1
    method), 33
get_suggestion() (dессн.examples.simple.modelbased.exampleModelUnderlyingSupernovaDistribution2
    method), 36
get_suggestion() (dессн.model.parameter.Parameter
    method), 44
get_suggestion() (dессн.model.parameter.ParameterDiscrete
    method), 45
get_suggestion() (dессн.model.parameter.ParameterObserved
    method), 47
get_suggestion() (dессн.model.parameter.ParameterTransformation
    method), 47
get_suggestion() (dессн.toy.latent.Luminosity
    method), 50
get_suggestion() (dессн.toy.latent.Type
    method), 50
get_suggestion() (dессн.toy.underlying.Hubble
    method), 54
get_suggestion() (dессн.toy.underlying.OmegaM
    method), 54
get_suggestion() (dессн.toy.underlying.SupernovaIaDist1
    method), 54
get_suggestion() (dессн.toy.underlying.SupernovaIaDist2
    method), 54
get_suggestion() (dессн.toy.underlying.SupernovaIIdist1
    method), 54
get_suggestion() (dессн.toy.underlying.SupernovaIIdist2
    method), 54
get_suggestion() (dессн.toy.underlying.SupernovaRate
    method), 54
get_suggestion() (dессн.toy.underlying.W
    method), 55
get_suggestion_requirements()
    (dессн.examples.discrete.discrete.UnderlyingRate
        method), 32
get_suggestion_requirements()
    (dессн.examples.simple.modelbased.exampleModel.LatentType
        method), 33
get_suggestion_requirements()
    (dессн.examples.simple.modelbased.exampleModel.UnderlyingSupernovaDistribution)
        (dессн.toy.underlying.SupernovaRate
            method), 55
get_suggestion_requirements()
    (dессн.model.parameter.Parameter
        method), 44
get_suggestion_requirements()
    (dессн.model.parameter.ParameterDiscrete
        method), 45
get_suggestion_requirements()
    (dессн.model.parameter.ParameterObserved
        method), 47
get_suggestion_requirements()
    (dессн.model.parameter.ParameterTransformation
        method), 47
get_suggestion_requirements()
    (dессн.model.parameter.UnderlyingSupernovaDistribution)
        (dессн.toy.underlying.SupernovaRate
            method), 55
get_suggestion_sigma()
    (dессн.examples.discrete.discrete.UnderlyingRate
        method), 32
get_suggestion_sigma()
    (dессн.examples.simple.modelbased.exampleModel.UnderlyingType
        method), 33
get_suggestion_sigma()
    (dессн.examples.simple.modelbased.exampleModel.UnderlyingType
        method), 33
get_suggestion_sigma()
    (dессн.model.parameter.Parameter
        method), 44
get_suggestion_sigma()
    (dессн.model.parameter.ParameterDiscrete
        method), 46
get_suggestion_sigma()
    (dессн.model.parameter.ParameterObserved
        method), 47
get_suggestion_sigma()
    (dессн.model.parameter.ParameterTransformation
        method), 47
get_suggestion_sigma()
    (dессн.toy.latent.Luminosity
        method), 50
get_suggestion_sigma()
    (dессн.toy.underlying.Hubble
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.OmegaM
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.SupernovaIaDist1
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.SupernovaIaDist2
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.SupernovaIIdist1
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.SupernovaIIdist2
        method), 54
get_suggestion_sigma()
    (dессн.toy.underlying.SupernovaRate
        method), 55
get_suggestion_sigma()
    (dессн.toy.underlying.W
        method), 55
get_summary()
    (dессн.chain.chain.ChainConsumer
        method), 14
get_transformation()
    (dессн.examples.simple.modelbased.exampleModel.LatentType
        method), 33
get_transformation()
    (dессн.model.edge.EdgeTransformation
        method), 42
get_transformation()
    (dессн.toy.edges.ToFlux
        method), 49
get_transformation()
    (dессн.toy.edges.ToLuminosityDistance
        method), 49
get_transformation()
    (dессн.toy.edges.ToRedshift
        method), 49

```

## H

Hubble (class in dessn.toy.underlying), 54

## I

iterate() (dessn.utility.newtonian.NewtonianPosition method), 55

## L

LatentLuminosity (class in dessn.examples.simple.modelbased.exampleModel), 33

Luminosity (class in dessn.toy.latent), 50

LuminosityDistance (class in dessn.toy.transformations), 54

LuminosityToAdjusted (class in dessn.examples.simple.modelbased.exampleModel), 33

LuminosityToSupernovaDistribution (class in dessn.examples.simple.modelbased.exampleModel), 33

## M

Model (class in dessn.model.model), 42

## N

NewtonianPosition (class in dessn.utility.newtonian), 55

## O

ObservedColour (class in dessn.examples.discrete.discrete), 30

ObservedCounts (class in dessn.toy.observed), 50

ObservedFlux (class in dessn.examples.simple.modelbased.exampleModel), 33

ObservedFluxError (class in dessn.examples.simple.modelbased.exampleModel), 33

ObservedRedshift (class in dessn.toy.observed), 50

ObservedSize (class in dessn.examples.discrete.discrete), 30

ObservedType (class in dessn.toy.observed), 50

OmegaM (class in dessn.toy.underlying), 54

## P

Parameter (class in dessn.model.parameter), 44

ParameterDiscrete (class in dessn.model.parameter), 45

ParameterLatent (class in dessn.model.parameter), 46

ParameterObserved (class in dessn.model.parameter), 46

ParameterTransformation (class in dessn.model.parameter), 47

ParameterUnderlying (class in dessn.model.parameter), 47

plot() (dessn.chain.chain.ChainConsumer method), 14

plot\_observations() (dessn.examples.simple.example.Example method), 37

plot\_walks() (dessn.chain.chain.ChainConsumer method), 14

## R

Redshift (class in dessn.toy.latent), 50

run\_chain() (dessn.utility.hdemcee.EmceeWrapper method), 55

## S

Simulation (class in dessn.simulation.simulation), 48

SupernovaIaDist1 (class in dessn.toy.underlying), 54

SupernovaIaDist2 (class in dessn.toy.underlying), 54

SupernovaIIDist1 (class in dessn.toy.underlying), 54

SupernovaIIDist2 (class in dessn.toy.underlying), 54

SupernovaRate (class in dessn.toy.underlying), 54

## T

ToColour (class in dessn.examples.discrete.discrete), 30

ToColour2 (class in dessn.examples.discrete.discrete), 30

ToCount (class in dessn.toy.edges), 48

ToFlux (class in dessn.toy.edges), 48

ToLuminosity (class in dessn.toy.edges), 49

ToLuminosityDistance (class in dessn.toy.edges), 49

ToRate (class in dessn.examples.discrete.discrete), 32

ToRate (class in dessn.toy.edges), 49

ToRedshift (class in dessn.toy.edges), 49

ToType (class in dessn.toy.edges), 49

ToyModel (class in dessn.toy.toyModel), 50

Type (class in dessn.toy.latent), 50

## U

UnderlyingRate (class in dessn.examples.discrete.discrete), 32

UnderlyingSupernovaDistribution1 (class in dessn.examples.simple.modelbased.exampleModel), 33

UnderlyingSupernovaDistribution2 (class in dessn.examples.simple.modelbased.exampleModel), 33

UselessTransformation (class in dessn.examples.simple.modelbased.exampleModel), 36

## V

various1\_no\_histogram()

(dessn.chain.demoVarious.DemoVarious method), 17

various2\_select\_parameters()

(dessn.chain.demoVarious.DemoVarious method), 20

various3\_flip\_histogram()  
    (dessn.chain.demoVarious.DemoVarious  
        method), 20  
various4\_summaries() (dessn.chain.demoVarious.DemoVarious  
    method), 20  
various5\_custom\_colours()  
    (dessn.chain.demoVarious.DemoVarious  
        method), 20  
various6\_truth\_values() (dessn.chain.demoVarious.DemoVarious  
    method), 24  
various7\_rainbow() (dessn.chain.demoVarious.DemoVarious  
    method), 24  
various8\_extents() (dessn.chain.demoVarious.DemoVarious  
    method), 24

## W

W (class in dessn.toy.underlying), 55