

---

# **dessn Documentation**

***Release 0.0.1***

**dessn**

March 10, 2016



<b>1</b>	<b>dessn package</b>	<b>1</b>
1.1	Subpackages	1
1.1.1	dessn.chain package	1
	Submodules	1
	dessn.chain.chain module	1
	dessn.chain.demoOneChain module	3
	dessn.chain.demoThreeChains module	5
	dessn.chain.demoTwoDisjointChains module	5
1.1.2	dessn.entry package	7
	Submodules	7
	dessn.entry.sim module	7
1.1.3	dessn.model package	7
	Submodules	7
	dessn.model.edge module	7
	dessn.model.model module	8
	dessn.model.node module	10
1.1.4	dessn.simple package	12
	Subpackages	12
	Submodules	16
	dessn.simple.example module	16
	dessn.simple.exampleIntegral module	17
	dessn.simple.exampleLatent module	19
1.1.5	dessn.simulation package	21
	Submodules	21
	dessn.simulation.observationFactory module	21
	dessn.simulation.simulation module	21
1.1.6	dessn.toy package	21
	Submodules	22
	dessn.toy.edges module	22
	dessn.toy.latent module	22
	dessn.toy.observed module	22
	dessn.toy.toyModel module	23
	dessn.toy.transformations module	23
	dessn.toy.underlying module	24
1.1.7	dessn.utility package	24
	Submodules	24
	dessn.utility.hdemcee module	24
	dessn.utility.newtonian module	24
	<b>Python Module Index</b>	<b>25</b>



## DESSN PACKAGE

Welcome to the **DESSN** code base.

## 1.1 Subpackages

### 1.1.1 `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.

**get\_parameter\_text** (*lower*, *maximum*, *upper*)

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

**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='COLUMN', filename=None, display=False, rainbow=False, contour\_kwargs=None*)

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. String arguments are not case sensitive.

**filename** : str, optional

If set, saves the figure to this location

**display** : bool

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

**rainbow** : bool

If true, forces the use of rainbow colours when displaying multiple chains. By default, under a certain number of chains to show, this method uses a predefined list of colours.

**contour\_kwargs** : dict

A dictionary of optional arguments to pass to the `plot_contour()` function.

**Returns** figure

the matplotlib figure

**plot\_bars** (*ax, parameter, chain\_row, bins=50, colour='#222222', fit\_values=None*)

Method responsible for plotting the marginalised distributions

**Parameters** **ax** : matplotlib axis

Upon which the plot is drawn

**parameter** : str

The parameter label, if it exists

**chain\_row** : np.ndarray

The data corresponding to the parameter

**bins** : int, optional

The number of bins to use. Default value is overridden by `plot()`

**colour** : str

The colour to use when plotting. Default value is overridden

**Returns** float

the maximum value of the histogram plot (used to ensure vertical spacing)

```
plot_contour (ax, x, y, bins=50, sigmas=None, colour='#222222', fit_values=None,
               force_contourf=False, cloud=True)
```

Plots contours of the probability surface between two parameters

**Parameters** **ax** : figure.axis

The axis to plot to

**x** : np.ndarray

The x axis array of data

**y** : np.ndarray

The y axis array of data

**bins** : int, optional

The number of bins to use. Overridden by the `plot()` method.

**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.

**colour** : str(hex code), optional

The colour to plot the contours in. Overridden by the `plot()` method.

**fit\_values** : np.array, optional

An array representing the lower bound, maximum, and upper bound of the marginalised parameters

**force\_contourf** : bool

Can force the plotting method to plot filled contours even when it would normally be disabled. It is normally disabled when plotting multiple chains.

**cloud** : bool

If true and there is only one chain, plots the cloud of points. If false, or there are more than one chain, does not plot the cloud.

## dessn.chain.demoOneChain module

**class** `dessn.chain.demoOneChain.DemoOneChain`

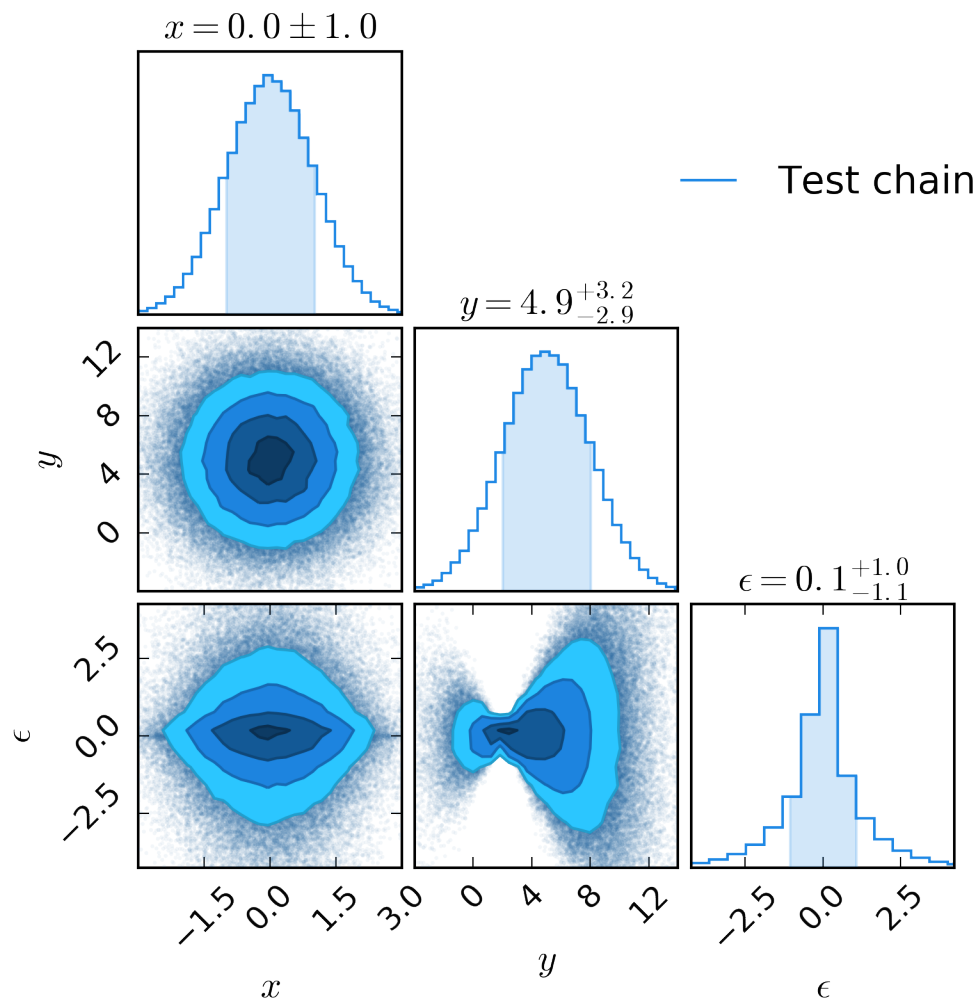
The single chain demo for Chain Consumer. Dummy class used to get documentation caught by sphinx-apidoc, it serves no other purpose.

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:





**dessn.chain.demoThreeChains module**

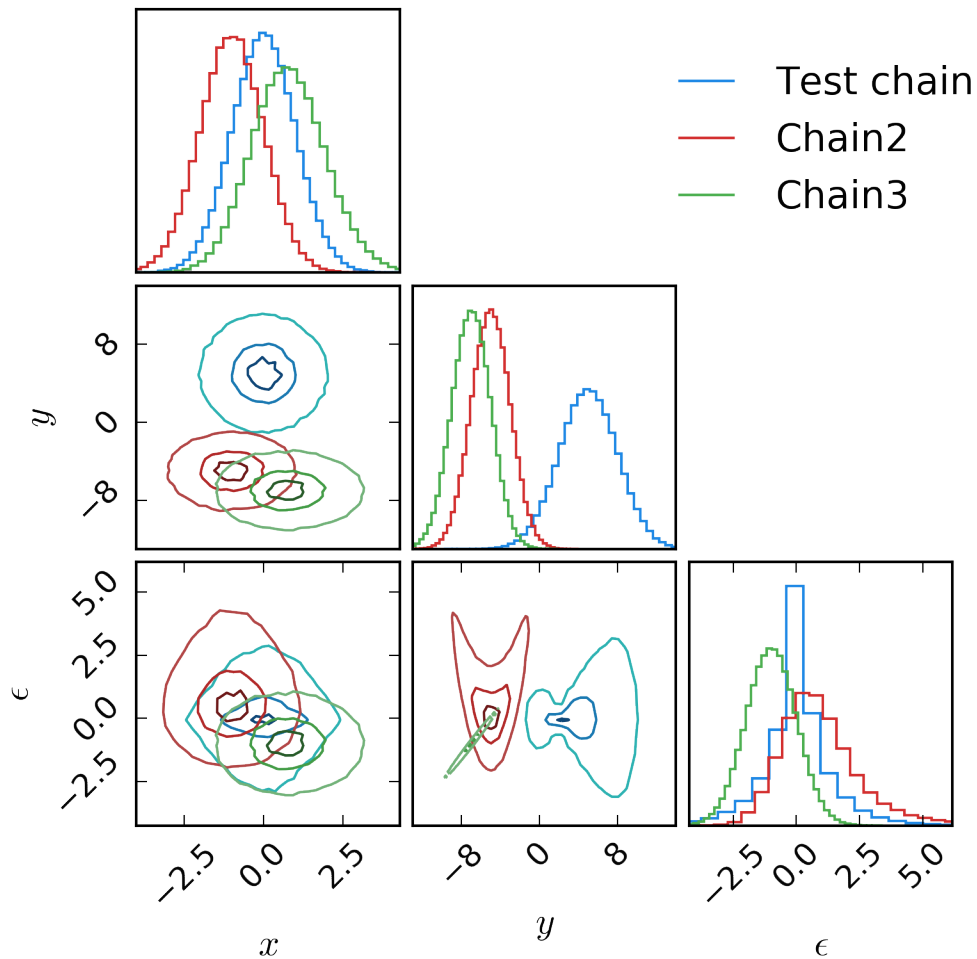
**class** `dessn.chain.demoThreeChains.DemoThreeChains`

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**

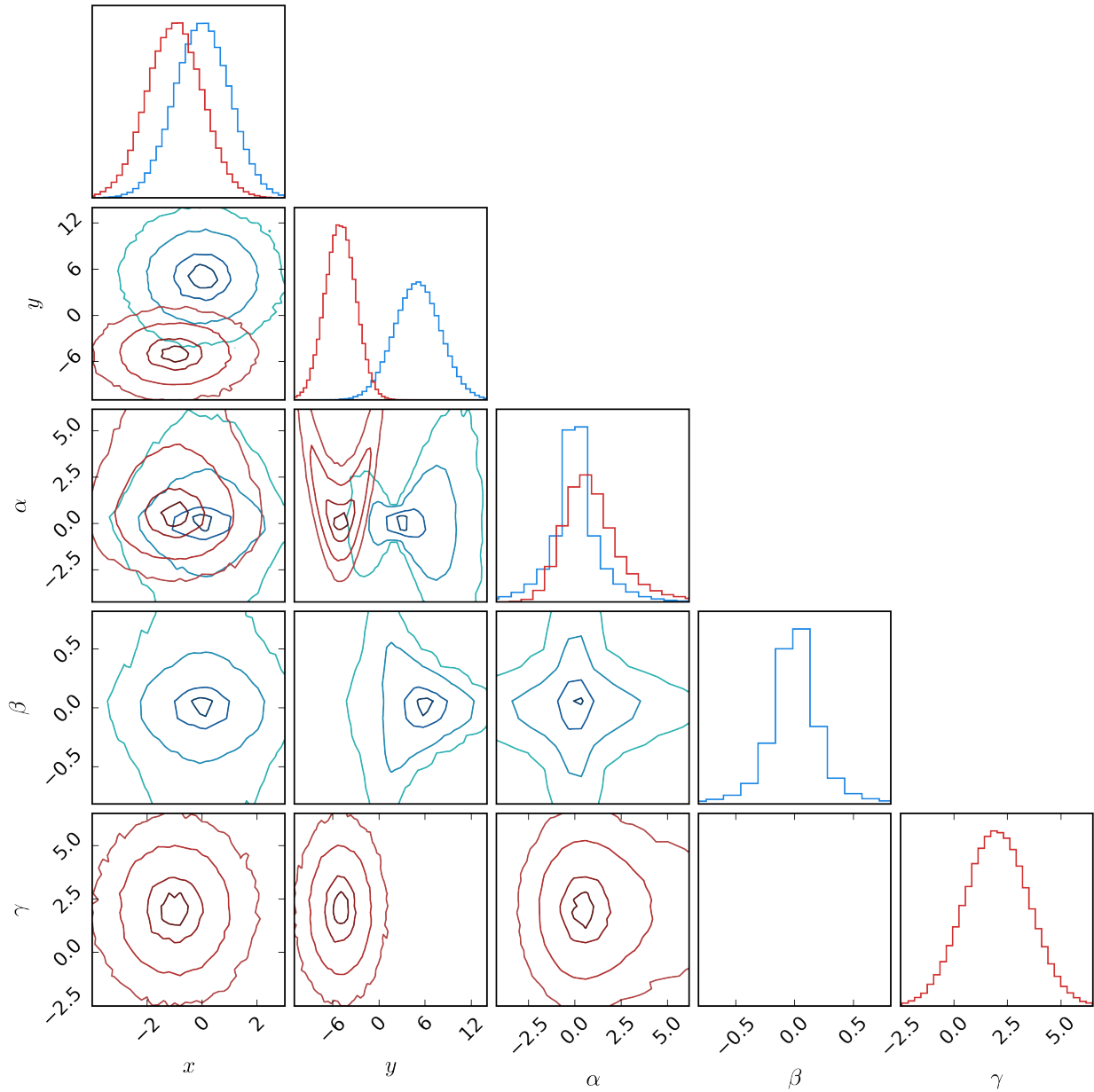
**class** `dessn.chain.demoTwoDisjointChains.DemoTwoDisjointChains`

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 dependent 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:



## 1.1.2 dessn.entry package

### Submodules

`dessn.entry.sim` module

## 1.1.3 dessn.model package

### Submodules

`dessn.model.edge` module

**class** `dessn.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

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

Adds a node into the models collection of nodes.

**chain\_plot** (*\*\*kwargs*)

Creates a chain plot of the model's chain.

This is my own implementation of a corner plot.

**Parameters** `kwargs` : dict

Arguments to pass to the `plot()` method. See the method link for more details.

**Returns** figure

a matplotlib figure of the chain plot

**chain\_summary** ()

Gets a summary of fit parameters through `ChainConsumer` and the `get_summary()` method. See the method link for more details

**corner** (*filename=None, display=True*)

Creates a corner plot from the model's chain.

**Parameters** `filename` : str, optional

If set, saves the figure to this filename

**display** : bool, optional

If true, shows the plot. If false, simply return the figure

**Returns** figure

a matplotlib figure of the corner plot

**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\_walkers=None*, *num\_steps=5000*, *num\_burn=3000*, *temp\_dir=None*,  
*save\_interval=300*)

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\_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\_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.node module

**class** `dessn.model.node.Node` (*node\_name, names, labels, parameter\_type*)

Bases: `object`

A node represented on a PGM model. Normally encapsulated by a single parameter, or several related parameters.

The Node class can essentially be thought of as a wrapper around a parameter or variable in your model. However, as some parameters are highly related (for example, flux and flux error), Nodes allow you to declare multiple parameters.

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

**Parameters** `node_name` : str

The node name, only used when plotting on a PGM

`names` : str or list[str]

The model parameter encapsulated by the node, or list of model parameters

`labels` : str or list[str]

Latex ready labels for the given names. Used in the PGM and corner plots.

`parameter_type` : *NodeType*

The type of subclass. Informs the model how to utilise the node.

**class** `dessn.model.node.NodeLatent` (*node\_name, names, labels*)

Bases: *dessn.model.node.Node*

A node 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 simply 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** `node_name` : str

The node name, only used when plotting on a PGM

`names` : str or list[str]

The model parameter encapsulated by the node, or list of model parameters

`labels` : str or list[str]

Latex ready labels for the given names. Used in the PGM and corner plots.

**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.node.NodeObserved` (*node\_name, names, labels, datas*)

Bases: `dessn.model.node.Node`

A node representing one or more observed variables

This node is used for all observables in the model. In addition to a normal node, it also contains data, which can be in arbitrary format. This data is what is given to the incoming and outgoing node edges to calculate likelihoods.

**Parameters** `node_name` : str

The node name, only used when plotting on a PGM

`names` : str or list[str]

The model parameter encapsulated by the node, or list of model parameters

`labels` : str or list[str]

Latex ready labels for the given names. Used in the PGM and corner plots.

`datas` : object or list[obj]

One data object for each supplied parameter name. **Must** be the same length as names if names is a list.

`get_data()`

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

**class** `dessn.model.node.NodeTransformation` (*node\_name, names, labels*)

Bases: `dessn.model.node.Node`

A node representing a variable transformation.

This node 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 node would be represented by a point, not an ellipse.

Note that this node declares all associated parameters to be transformation parameters, although the transformation functions themselves are defined by the edges into and out of this node.

**Parameters** `node_name` : str

The node name, only used when plotting on a PGM

`names` : str or list[str]

The model parameter encapsulated by the node, or list of model parameters

`labels` : str or list[str]

Latex ready labels for the given names. Used in the PGM and corner plots.

**class** `dessn.model.node.NodeType`

Bases: `enum.Enum`

**LATENT** = <NodeType.LATENT: 3>

**OBSERVED** = <NodeType.OBSERVED: 2>

**TRANSFORMATION** = <NodeType.TRANSFORMATION: 4>

**UNDERLYING** = <NodeType.UNDERLYING: 1>

**class** `dessn.model.node.NodeUnderlying` (*node\_name*, *names*, *labels*)

Bases: `dessn.model.node.Node`

A node 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`

**Parameters** `node_name` : str

The node name, only used when plotting on a PGM

`names` : str or list[str]

The model parameter encapsulated by the node, or list of model parameters

`labels` : str or list[str]

Latex ready labels for the given names. Used in the PGM and corner plots.

**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

### 1.1.4 dessn.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

### dessn.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.simple.modelbased.exampleModel` module

**class** `dessn.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 `Node` or from `Edge`.

I leave the documentation for `Node` 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.simple.modelbased.exampleModel.FluxToLuminosity`

Bases: `dessn.model.edge.Edge`

`get_log_likelihood(data)`

**class** `dessn.simple.modelbased.exampleModel.LatentLuminosity(n=100)`

Bases: `dessn.model.node.NodeLatent`

`get_num_latent()`

**class** `dessn.simple.modelbased.exampleModel.LuminosityToAdjusted`

Bases: `dessn.model.edge.EdgeTransformation`

`get_transformation(data)`

**class** `dessn.simple.modelbased.exampleModel.LuminosityToSupernovaDistribution`

Bases: `dessn.model.edge.Edge`

`get_log_likelihood(data)`

**class** `dessn.simple.modelbased.exampleModel.ObservedFlux(n=100)`

Bases: `dessn.model.node.NodeObserved`

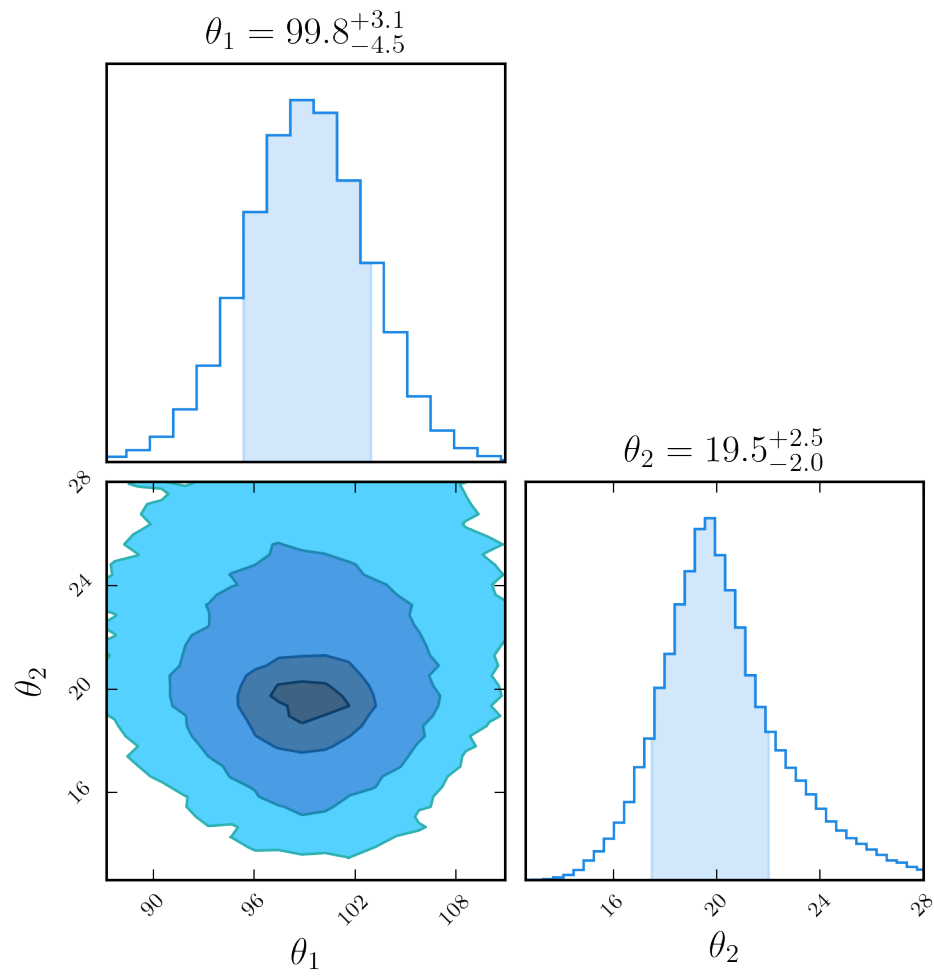
**class** `dessn.simple.modelbased.exampleModel.UnderlyingSupernovaDistribution`

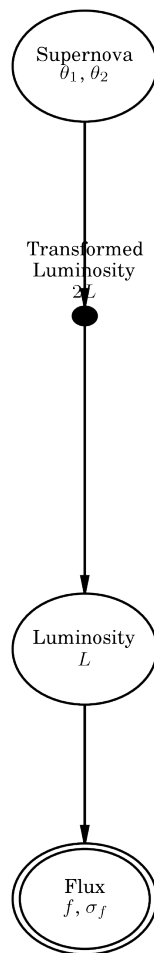
Bases: `dessn.model.node.NodeUnderlying`

`get_log_prior(data)`

**class** `dessn.simple.modelbased.exampleModel.UselessTransformation`

Bases: `dessn.model.node.NodeTransformation`





## Submodules

### dessn.simple.example module

**class** `dessn.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  $\theta$ .

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

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.

## dessn.simple.exampleIntegral module

**class** `dessn.simple.exampleIntegral.ExampleIntegral` (*n=10*, *theta\_1=100.0*, *theta\_2=30.0*)

Bases: `dessn.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

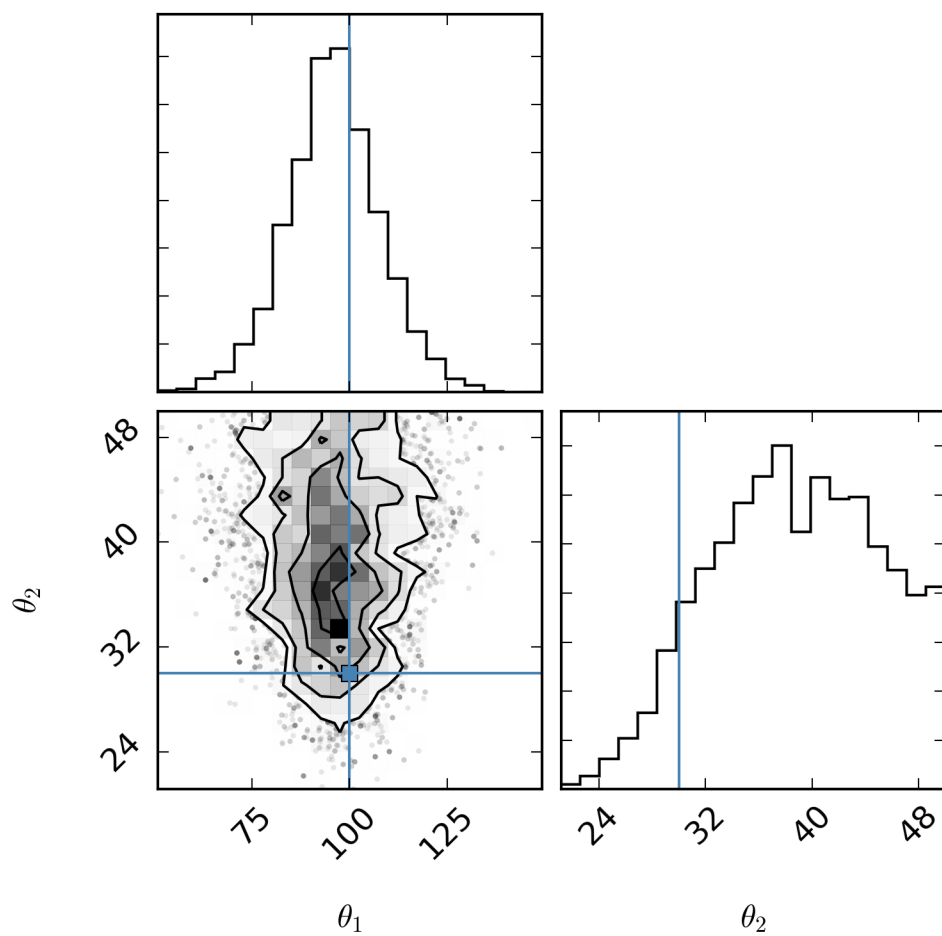
**theta\_2** : float, optional

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.simple.exampleLatent module

**class** `dessn.simple.exampleLatent.ExampleLatent` ( $n=30$ ,  $theta\_1=100.0$ ,  $theta\_2=20.0$ )

Bases: `dessn.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

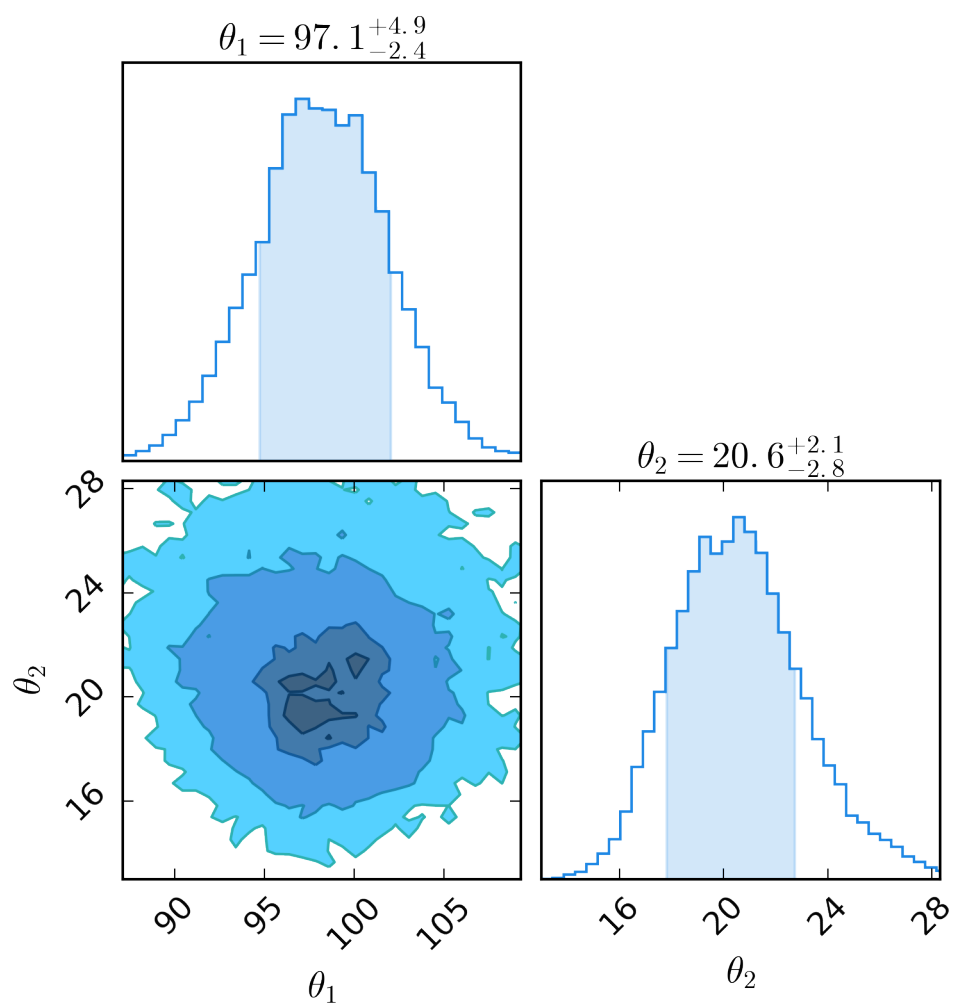
**theta\_2** : float, optional

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

## 1.1.5 dessn.simulation package

### Submodules

#### dessn.simulation.observationFactory module

```
class dessn.simulation.observationFactory.ObservationFactory (**kwargs)
```

Bases: object

**check\_kwargs** ()

**get\_observations** (*num*)

Still needs massive refactoring

#### dessn.simulation.simulation module

```
class dessn.simulation.simulation.Simulation
```

Bases: object

**get\_simulation** (*num\_trans*=30)

## 1.1.6 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.EdgeTransformation

    get_transformation(data)

class dessn.toy.edges.ToFlux
    Bases: dessn.model.edge.EdgeTransformation

    get_transformation(data)

class dessn.toy.edges.ToLuminosity
    Bases: dessn.model.edge.Edge

    get_log_likelihood(data)

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)

class dessn.toy.edges.ToRedshift
    Bases: dessn.model.edge.Edge

    get_log_likelihood(data)

class dessn.toy.edges.ToType
    Bases: dessn.model.edge.Edge

    get_log_likelihood(data)
```

### dessn.toy.latent module

```
class dessn.toy.latent.Luminosity(n)
    Bases: dessn.model.node.NodeLatent

    get_num_latent()

class dessn.toy.latent.Redshift(n)
    Bases: dessn.model.node.NodeLatent

    get_num_latent()

class dessn.toy.latent.Type(n)
    Bases: dessn.model.node.NodeLatent

    get_num_latent()
```

### dessn.toy.observed module

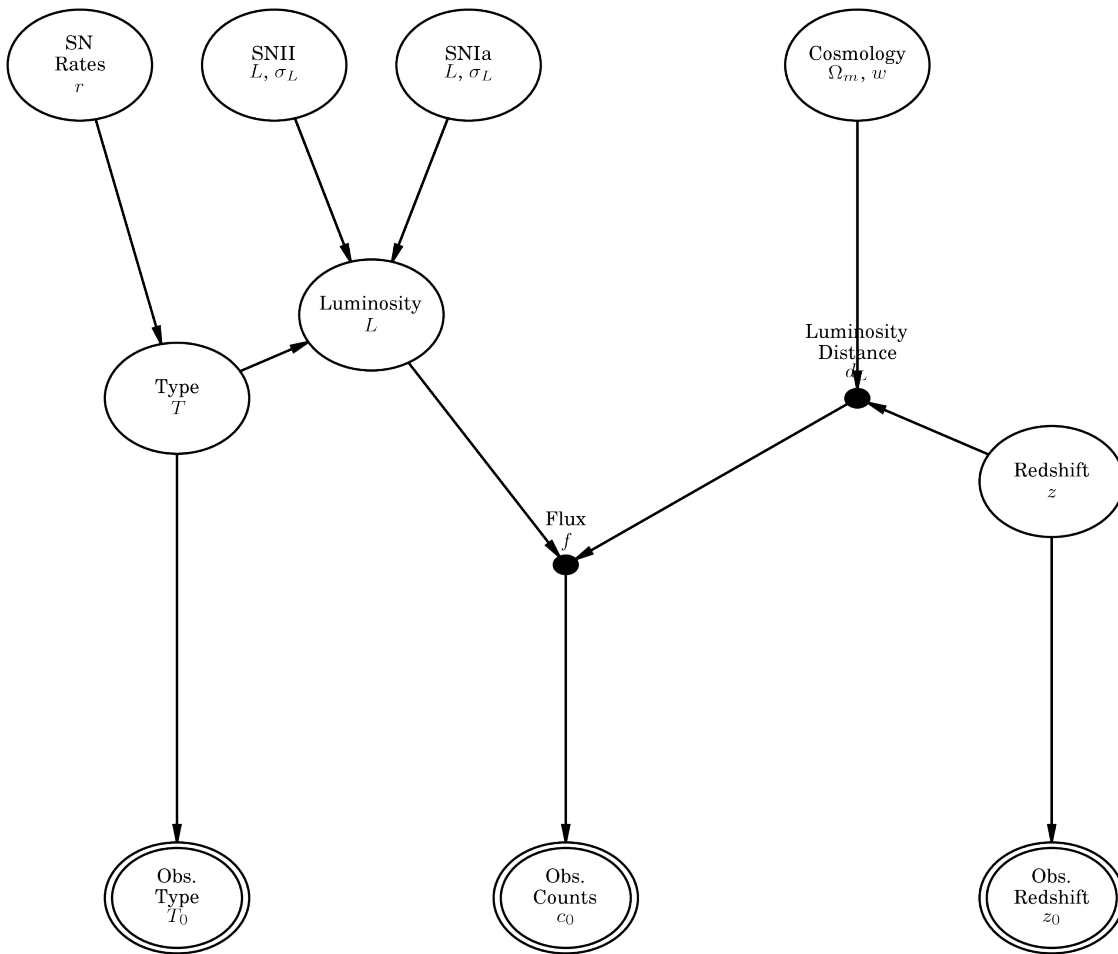
```
class dessn.toy.observed.ObservedCounts(counts)
    Bases: dessn.model.node.NodeObserved
```

**class** `dessn.toy.observed.ObservedRedshift` (*redshifts*)  
 Bases: `dessn.model.node.NodeObserved`

**class** `dessn.toy.observed.ObservedType` (*types*)  
 Bases: `dessn.model.node.NodeObserved`

### dessn.toy.toyModel module

**class** `dessn.toy.toyModel.ToyModel`  
 Bases: `dessn.model.model.Model`  
 A modified toy model.



### dessn.toy.transformations module

**class** `dessn.toy.transformations.Flux`  
 Bases: `dessn.model.node.NodeTransformation`

**class** `dessn.toy.transformations.LuminosityDistance`  
Bases: `dessn.model.node.NodeTransformation`

### dessn.toy.underlying module

**class** `dessn.toy.underlying.Cosmology`  
Bases: `dessn.model.node.NodeUnderlying`  
**get\_log\_prior** (*data*)

**class** `dessn.toy.underlying.SupernovaIIDist`  
Bases: `dessn.model.node.NodeUnderlying`  
**get\_log\_prior** (*data*)

**class** `dessn.toy.underlying.SupernovaIaDist`  
Bases: `dessn.model.node.NodeUnderlying`  
**get\_log\_prior** (*data*)

**class** `dessn.toy.underlying.SupernovaRate`  
Bases: `dessn.model.node.NodeUnderlying`  
**get\_log\_prior** (*data*)

## 1.1.7 dessn.utility package

### Submodules

#### dessn.utility.hdemcee module

**class** `dessn.utility.hdemcee.EmceeWrapper` (*sampler*)  
Bases: `object`  
**get\_results** ()  
**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, 1](#)
- [dessn.chain, 1](#)
- [dessn.chain.chain, 1](#)
- [dessn.chain.demoOneChain, 3](#)
- [dessn.chain.demoThreeChains, 5](#)
- [dessn.chain.demoTwoDisjointChains, 5](#)
- [dessn.entry, 7](#)
- [dessn.entry.sim, 7](#)
- [dessn.model, 7](#)
- [dessn.model.edge, 7](#)
- [dessn.model.model, 8](#)
- [dessn.model.node, 10](#)
- [dessn.simple, 12](#)
- [dessn.simple.example, 16](#)
- [dessn.simple.exampleIntegral, 17](#)
- [dessn.simple.exampleLatent, 19](#)
- [dessn.simple.modelbased, 12](#)
- [dessn.simple.modelbased.exampleModel, 13](#)
- [dessn.simulation, 21](#)
- [dessn.simulation.observationFactory, 21](#)
- [dessn.simulation.simulation, 21](#)
- [dessn.toy, 21](#)
- [dessn.toy.edges, 22](#)
- [dessn.toy.latent, 22](#)
- [dessn.toy.observed, 22](#)
- [dessn.toy.toyModel, 23](#)
- [dessn.toy.transformations, 23](#)
- [dessn.toy.underlying, 24](#)
- [dessn.utility, 24](#)
- [dessn.utility.hdemcee, 24](#)
- [dessn.utility.newtonian, 24](#)



## A

`add_chain()` (dessn.chain.chain.ChainConsumer method), 1  
`add_edge()` (dessn.model.model.Model method), 8  
`add_node()` (dessn.model.model.Model method), 8

## C

`chain_plot()` (dessn.model.model.Model method), 8  
`chain_summary()` (dessn.model.model.Model method), 8  
ChainConsumer (class in dessn.chain.chain), 1  
`check_kwargs()` (dessn.simulation.observationFactory.ObservationFactory method), 21  
`corner()` (dessn.model.model.Model method), 8  
Cosmology (class in dessn.toy.underlying), 24

## D

DemoOneChain (class in dessn.chain.demoOneChain), 3  
DemoThreeChains (class in dessn.chain.demoThreeChains), 5  
DemoTwoDisjointChains (class in dessn.chain.demoTwoDisjointChains), 5  
dessn (module), 1  
dessn.chain (module), 1  
dessn.chain.chain (module), 1  
dessn.chain.demoOneChain (module), 3  
dessn.chain.demoThreeChains (module), 5  
dessn.chain.demoTwoDisjointChains (module), 5  
dessn.entry (module), 7  
dessn.entry.sim (module), 7  
dessn.model (module), 7  
dessn.model.edge (module), 7  
dessn.model.model (module), 8  
dessn.model.node (module), 10  
dessn.simple (module), 12  
dessn.simple.example (module), 16  
dessn.simple.exampleIntegral (module), 17  
dessn.simple.exampleLatent (module), 19  
dessn.simple.modelbased (module), 12  
dessn.simple.modelbased.exampleModel (module), 13  
dessn.simulation (module), 21  
dessn.simulation.observationFactory (module), 21  
dessn.simulation.simulation (module), 21

dessn.toy (module), 21  
dessn.toy.edges (module), 22  
dessn.toy.latent (module), 22  
dessn.toy.observed (module), 22  
dessn.toy.toyModel (module), 23  
dessn.toy.transformations (module), 23  
dessn.toy.underlying (module), 24  
dessn.utility (module), 24  
dessn.utility.hdemcee (module), 24  
dessn.utility.newtonian (module), 24  
`do_emcee()` (dessn.simple.example.Example method), 16  
`do_emcee()` (dessn.simple.exampleIntegral.ExampleIntegral method), 17  
`do_emcee()` (dessn.simple.exampleLatent.ExampleLatent method), 19

## E

Edge (class in dessn.model.edge), 7  
EdgeTransformation (class in dessn.model.edge), 7  
EmceeWrapper (class in dessn.utility.hdemcee), 24  
Example (class in dessn.simple.example), 16  
ExampleIntegral (class in dessn.simple.exampleIntegral), 17  
ExampleLatent (class in dessn.simple.exampleLatent), 19  
ExampleModel (class in dessn.simple.modelbased.exampleModel), 13

## F

`finalise()` (dessn.model.model.Model method), 9  
`fit()` (dessn.utility.newtonian.NewtonianPosition method), 24  
`fit_model()` (dessn.model.model.Model method), 9  
Flux (class in dessn.toy.transformations), 23  
FluxToLuminosity (class in dessn.simple.modelbased.exampleModel), 13

## G

`get_data()` (dessn.model.node.NodeObserved method), 11  
`get_data()` (dessn.simple.example.Example static method), 16

[get\\_likelihood\(\)](#) (dessn.simple.example.Example method), 16  
[get\\_likelihood\(\)](#) (dessn.simple.exampleIntegral.ExampleIntegral method), 19  
[get\\_likelihood\(\)](#) (dessn.simple.exampleLatent.ExampleLatent method), 21  
[get\\_log\\_likelihood\(\)](#) (dessn.model.edge.Edge method), 7  
[get\\_log\\_likelihood\(\)](#) (dessn.model.edge.EdgeTransformation method), 8  
[get\\_log\\_likelihood\(\)](#) (dessn.simple.modelbased.exampleModel.FluxToLuminosity method), 13  
[get\\_log\\_likelihood\(\)](#) (dessn.simple.modelbased.exampleModel.LuminosityToSupernovaDistribution method), 13  
[get\\_log\\_likelihood\(\)](#) (dessn.toy.edges.ToLuminosity method), 22  
[get\\_log\\_likelihood\(\)](#) (dessn.toy.edges.ToRate method), 22  
[get\\_log\\_likelihood\(\)](#) (dessn.toy.edges.ToRedshift method), 22  
[get\\_log\\_likelihood\(\)](#) (dessn.toy.edges.ToType method), 22  
[get\\_log\\_prior\(\)](#) (dessn.model.node.NodeUnderlying method), 12  
[get\\_log\\_prior\(\)](#) (dessn.simple.modelbased.exampleModel.UnderlyingSupernovaDistribution method), 13  
[get\\_log\\_prior\(\)](#) (dessn.toy.underlying.Cosmology method), 24  
[get\\_log\\_prior\(\)](#) (dessn.toy.underlying.SupernovaIaDist method), 24  
[get\\_log\\_prior\(\)](#) (dessn.toy.underlying.SupernovaIIDist method), 24  
[get\\_log\\_prior\(\)](#) (dessn.toy.underlying.SupernovaRate method), 24  
[get\\_num\\_latent\(\)](#) (dessn.model.node.NodeLatent method), 10  
[get\\_num\\_latent\(\)](#) (dessn.simple.modelbased.exampleModel.LatentLuminosity method), 13  
[get\\_num\\_latent\(\)](#) (dessn.toy.latent.Luminosity method), 22  
[get\\_num\\_latent\(\)](#) (dessn.toy.latent.Redshift method), 22  
[get\\_num\\_latent\(\)](#) (dessn.toy.latent.Type method), 22  
[get\\_observations\(\)](#) (dessn.simulation.observationFactory.ObservationFactory method), 21  
[get\\_parameter\\_text\(\)](#) (dessn.chain.chain.ChainConsumer method), 1  
[get\\_pgm\(\)](#) (dessn.model.model.Model method), 9  
[get\\_posterior\(\)](#) (dessn.simple.example.Example method), 17  
[get\\_prior\(\)](#) (dessn.simple.example.Example method), 17  
[get\\_results\(\)](#) (dessn.utility.hdemcee.EmceeWrapper method), 24  
[get\\_simulation\(\)](#) (dessn.simulation.simulation.Simulation method), 21  
[get\\_summary\(\)](#) (dessn.chain.chain.ChainConsumer method), 2  
[get\\_transformation\(\)](#) (dessn.model.edge.EdgeTransformation method), 8  
[get\\_transformation\(\)](#) (dessn.simple.modelbased.exampleModel.LuminosityToSupernovaDistribution method), 13  
[get\\_transformation\(\)](#) (dessn.toy.edges.ToCount method), 22  
[get\\_transformation\(\)](#) (dessn.toy.edges.ToFlux method), 22  
[get\\_transformation\(\)](#) (dessn.toy.edges.ToLuminosityDistance method), 22  
[iterate\(\)](#) (dessn.utility.newtonian.NewtonianPosition method), 24

## L

[LATENT](#) (dessn.model.node.NodeType attribute), 11  
[LatentLuminosity](#) (class in dessn.simple.modelbased.exampleModel), 13  
[Luminosity](#) (class in dessn.toy.latent), 22  
[LuminosityDistance](#) (class in dessn.toy.transformations), 23  
[LuminosityToAdjusted](#) (class in dessn.simple.modelbased.exampleModel), 13  
[LuminosityToSupernovaDistribution](#) (class in dessn.simple.modelbased.exampleModel), 13

## M

[Model](#) (class in dessn.model.model), 8

## N

[NewtonianPosition](#) (class in dessn.utility.newtonian), 24  
[Node](#) (class in dessn.model.node), 10  
[NodeLatent](#) (class in dessn.model.node), 10  
[NodeObserved](#) (class in dessn.model.node), 11  
[NodeTransformation](#) (class in dessn.model.node), 11  
[NodeType](#) (class in dessn.model.node), 11  
[NodeUnderlying](#) (class in dessn.model.node), 11

## O

[ObservationFactory](#) (class in dessn.simulation.observationFactory), 21  
[OBSERVED](#) (dessn.model.node.NodeType attribute), 11  
[ObservedCounts](#) (class in dessn.toy.observed), 22  
[ObservedFlux](#) (class in dessn.simple.modelbased.exampleModel), 13  
[ObservedRedshift](#) (class in dessn.toy.observed), 22



ObservedType (class in `dessn.toy.observations`), 23

## P

`plot()` (`dessn.chain.chain.ChainConsumer` method), 2

`plot_bars()` (`dessn.chain.chain.ChainConsumer` method), 2

`plot_contour()` (`dessn.chain.chain.ChainConsumer` method), 2

`plot_observations()` (`dessn.simple.example.Example` method), 17

## R

Redshift (class in `dessn.toy.latent`), 22

`run_chain()` (`dessn.utility.hdemcee.EmceeWrapper` method), 24

## S

Simulation (class in `dessn.simulation.simulation`), 21

SupernovaIaDist (class in `dessn.toy.underlying`), 24

SupernovaIIDist (class in `dessn.toy.underlying`), 24

SupernovaRate (class in `dessn.toy.underlying`), 24

## T

ToCount (class in `dessn.toy.edges`), 22

ToFlux (class in `dessn.toy.edges`), 22

ToLuminosity (class in `dessn.toy.edges`), 22

ToLuminosityDistance (class in `dessn.toy.edges`), 22

ToRate (class in `dessn.toy.edges`), 22

ToRedshift (class in `dessn.toy.edges`), 22

ToType (class in `dessn.toy.edges`), 22

ToyModel (class in `dessn.toy.toyModel`), 23

TRANSFORMATION (`dessn.model.node.NodeType` attribute), 11

Type (class in `dessn.toy.latent`), 22

## U

UNDERLYING (`dessn.model.node.NodeType` attribute), 11

UnderlyingSupernovaDistribution (class in `dessn.simple.modelbased.exampleModel`), 13

UselessTransformation (class in `dessn.simple.modelbased.exampleModel`), 13