dessn Documentation

Release 0.0.1

dessn

March 14, 2016

CONTENTS

CHAPTER

ONE

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.

`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 datae to the chain consumer, as the consume changes configuration values depending on the presupplied data.

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

The σ 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	α	β	γ
Model A	$0.1^{+0.9}_{-1.1}$	$5.1^{+2.9}_{-3.1}$	$-0.1^{+1.2}_{-1.0}$	0.0 ± 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='COLUMN'*, *parameters=None*, *extents=None*, *filename=None*, *display=False*, *truth=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.

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

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

Specifiy 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

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 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:

dessn.chain.demoTable module

class `dessn.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.

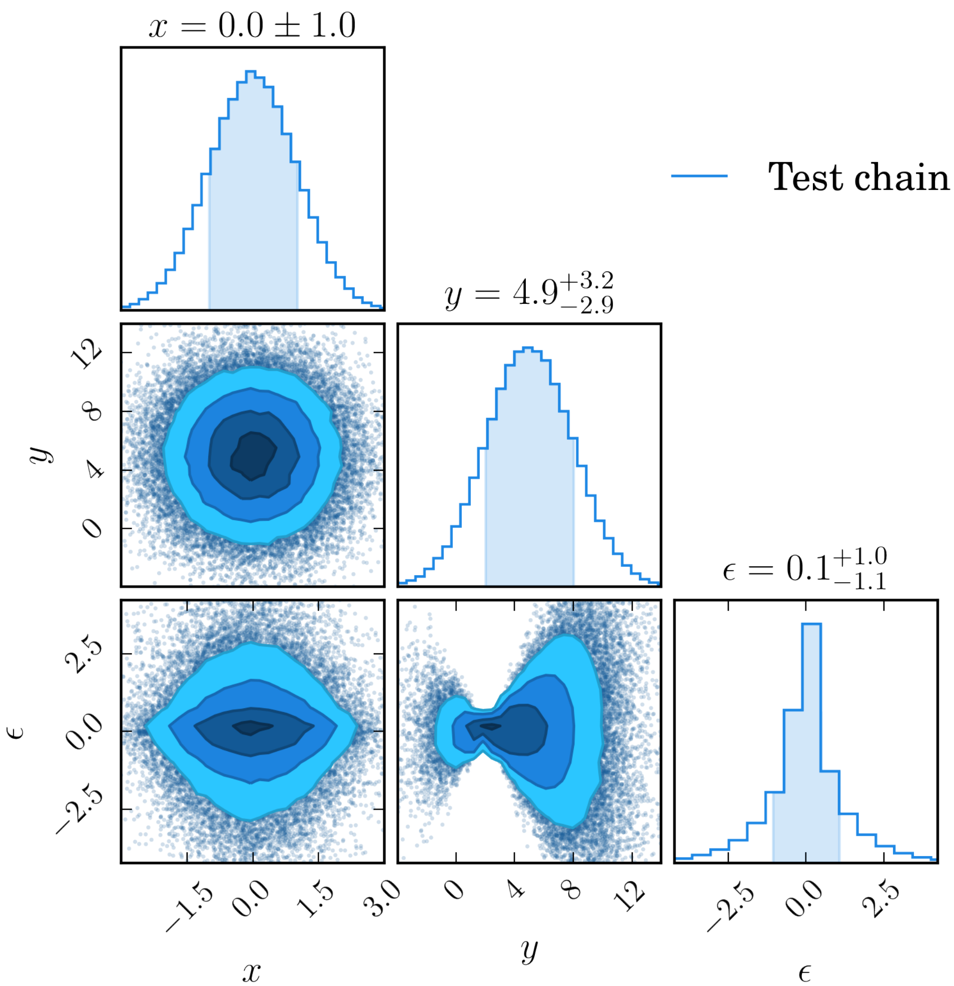


Table 1: The maximum likelihood results for the tested models

Model	x	y	α	β	γ
Model A	$0.1^{+0.9}_{-1.1}$	$5.1^{+2.9}_{-3.1}$	$-0.1^{+1.2}_{-1.0}$	0.0 ± 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

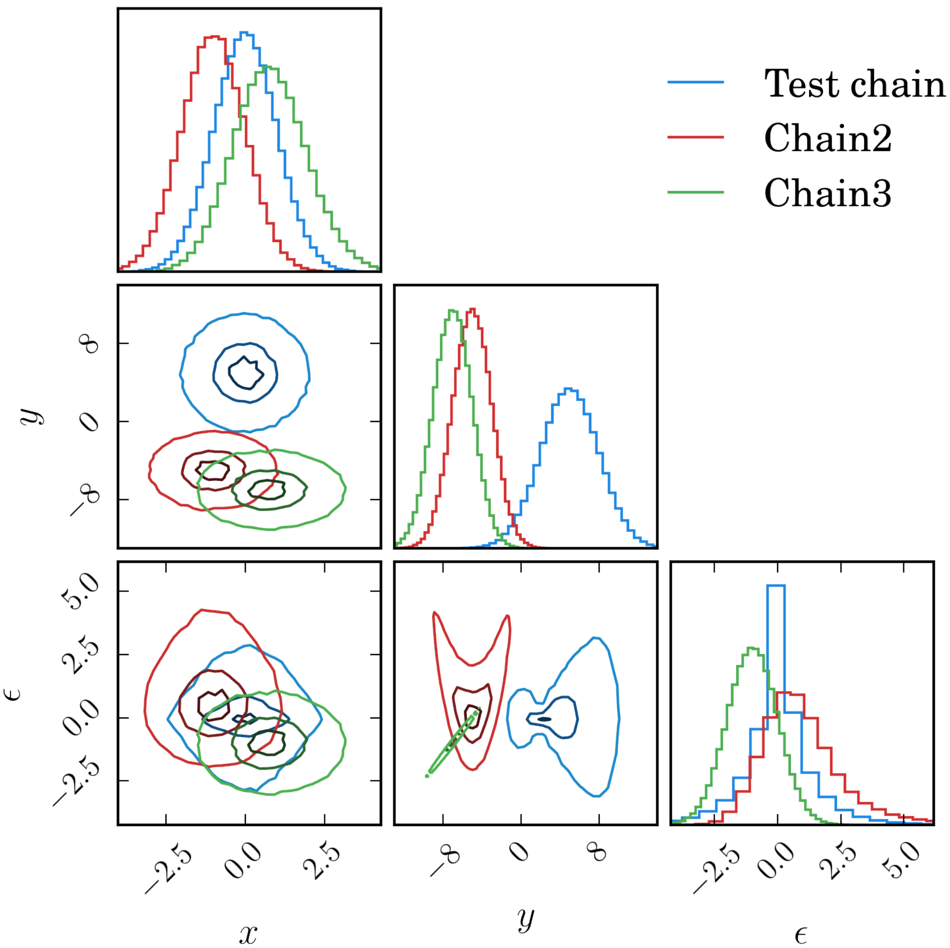
class `dessn.chain.demoThreeChains.`**`DemoThreeChains`**

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`**

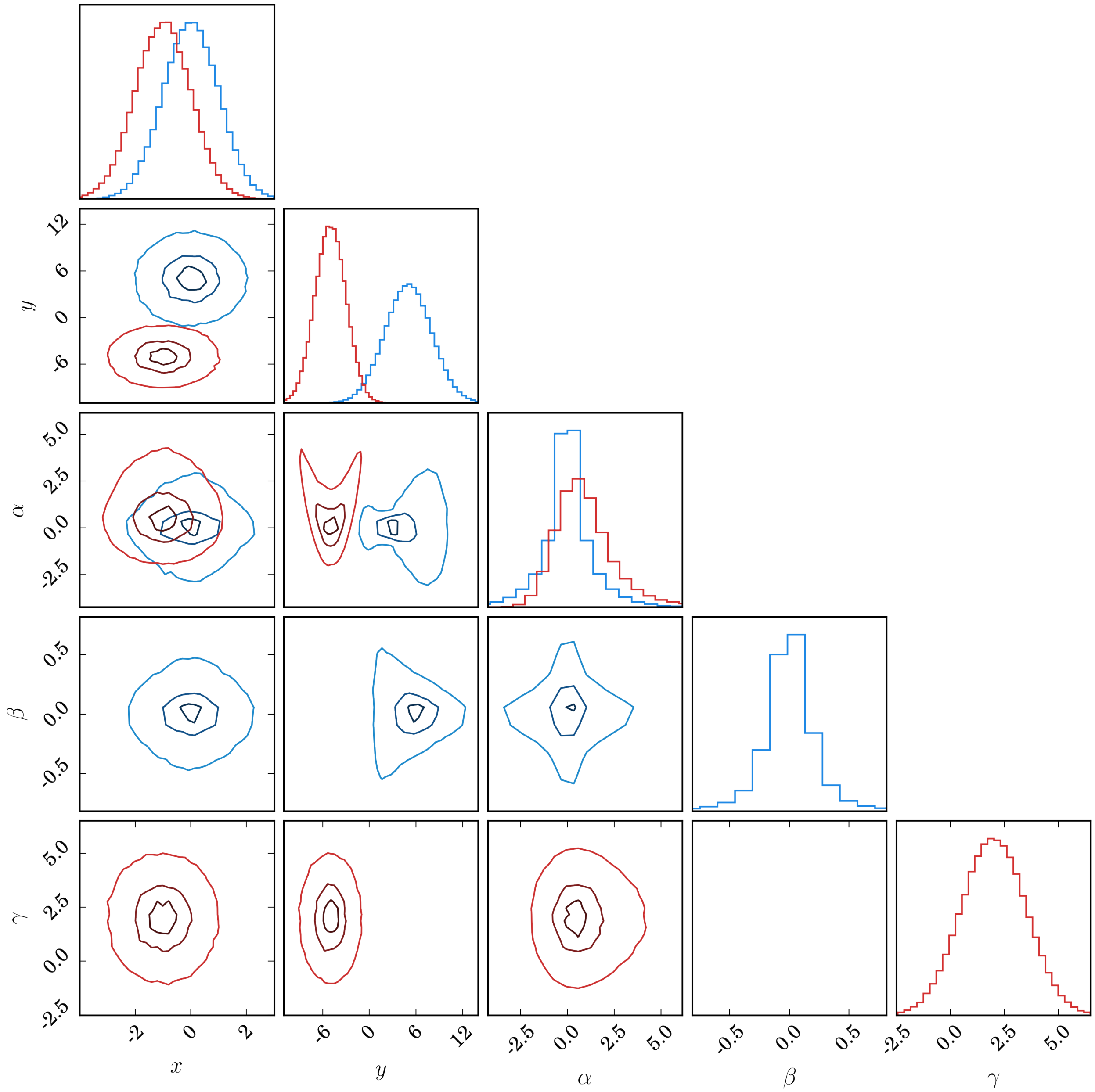
The multiple chain demo for Chain Consumer. Dummy class used to get documentation caught by `sphinx-apidoc`, it servers no other purpose.

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 Ω_m and Ω_Λ, where we assume

$w = 1$. Alternatively, we might assume flatness, and therefore fix Ω_Λ 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`**

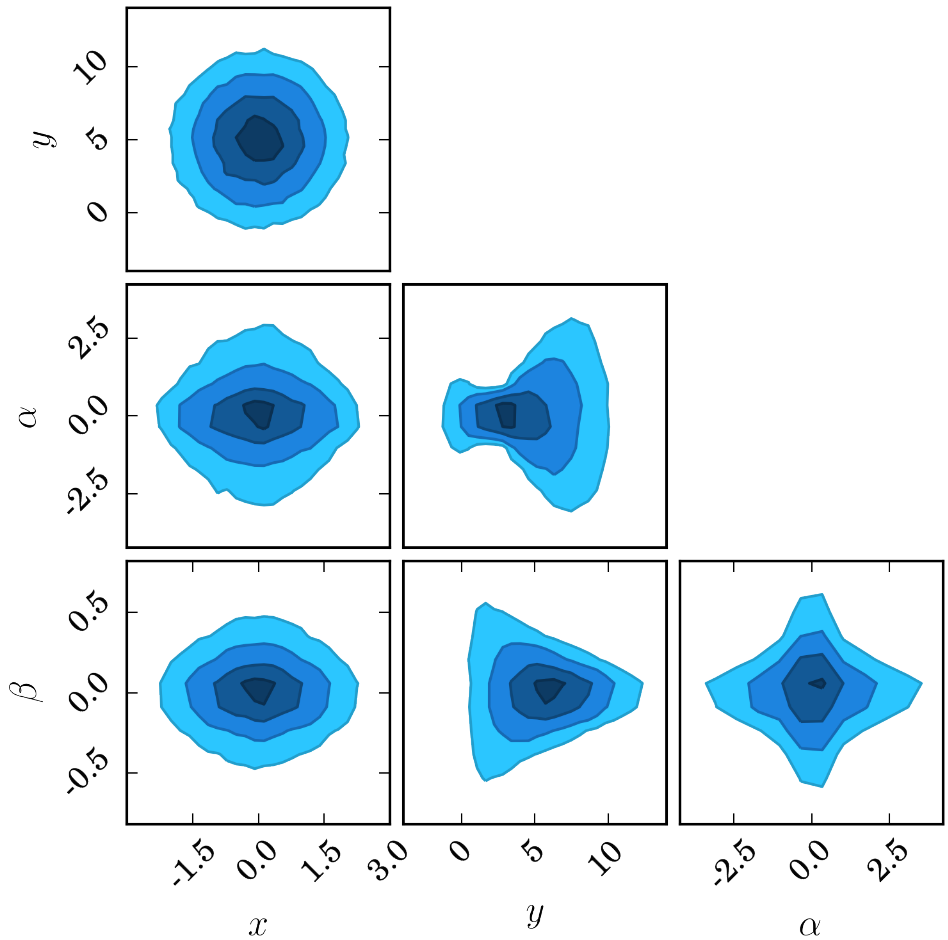
Bases: `object`

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 β parameter is not displayed.

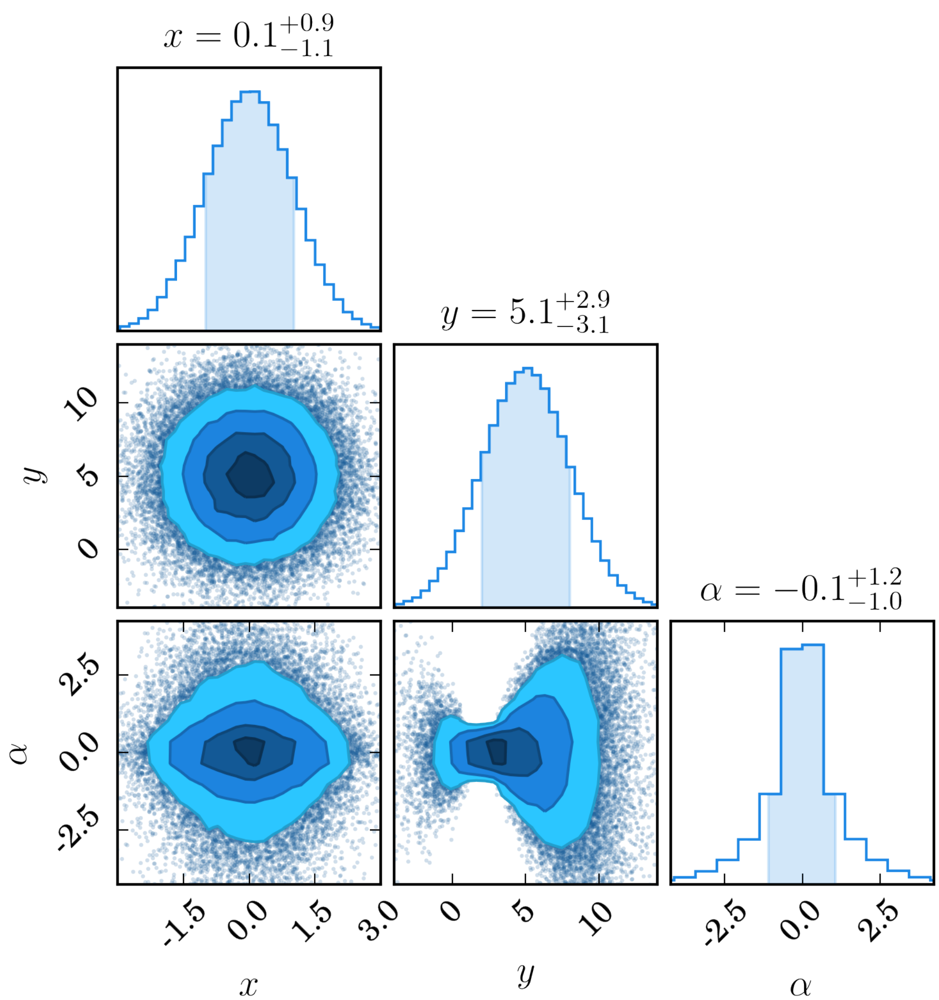
various3_flip_histogram()
When you only display two parameters and don't disable histograms, your plot will look different.

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.

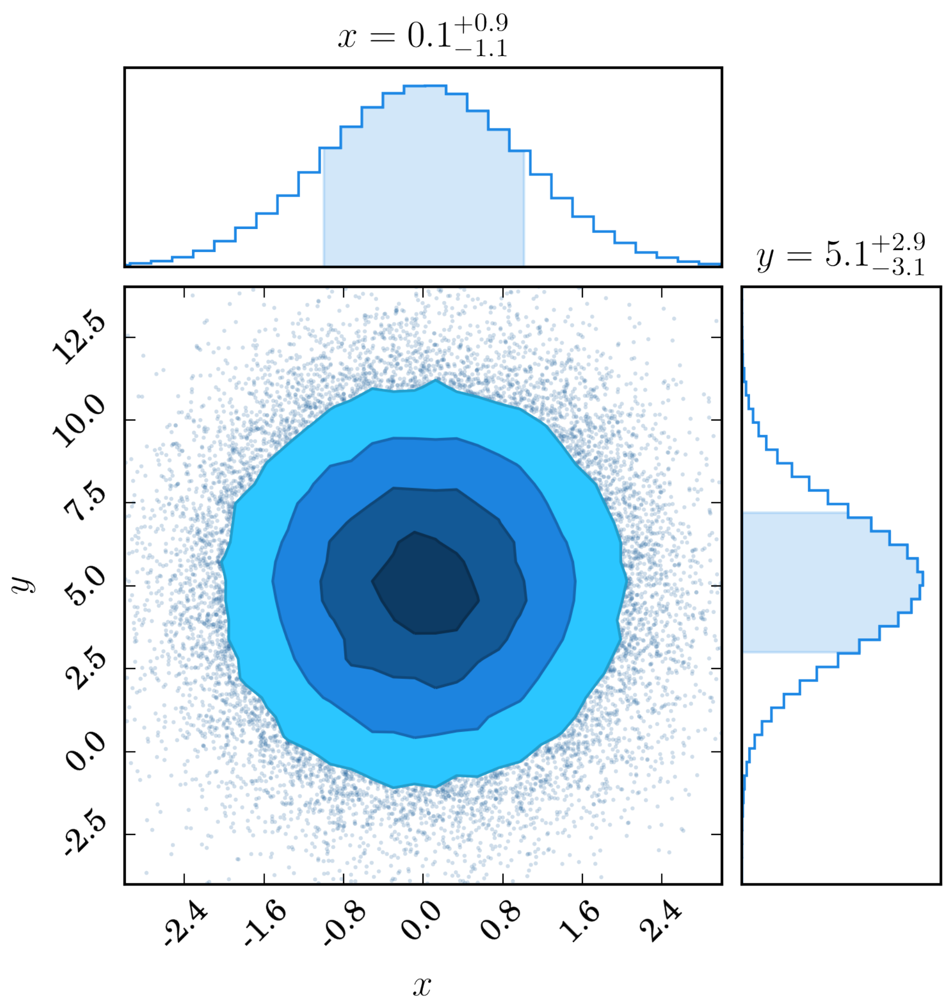
The max number of ticks is also modified in this example.

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 relevent axis. You can set this to always show or always not show by using the `force_summary` flag. Also, here we demonstrate more σ levels!

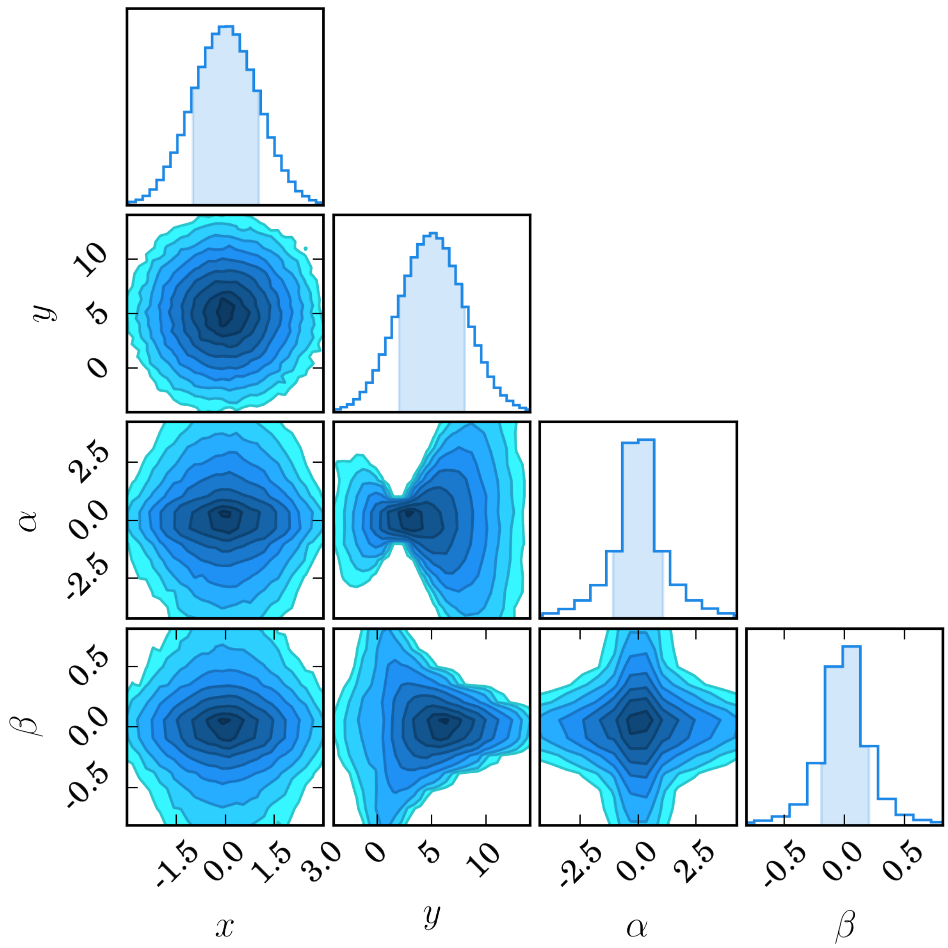
various5_custom_colours()
You can supply custom colours to the plotting. Be warned, if you have more chains than colours, you *will*



$x = 0.1^{+0.9}_{-1.1}$
$y = 5.1^{+2.9}_{-3.1}$
$\alpha = -0.1^{+1.2}_{-1.0}$
y
10
5
0
α
2.5
0.0
-2.5
-1.5
0.0
1.5
3.0
0
5
10
-2.5
0.0
2.5
x
y
α



$x = 0.1^{+0.9}_{-1.1}$
$y = 5.1^{+2.9}_{-3.1}$
12.5
10.0
7.5
5.0
2.5
0.0
-2.5
y
-2.4
-1.6
-0.8
0.0
0.8
1.6
2.4
x



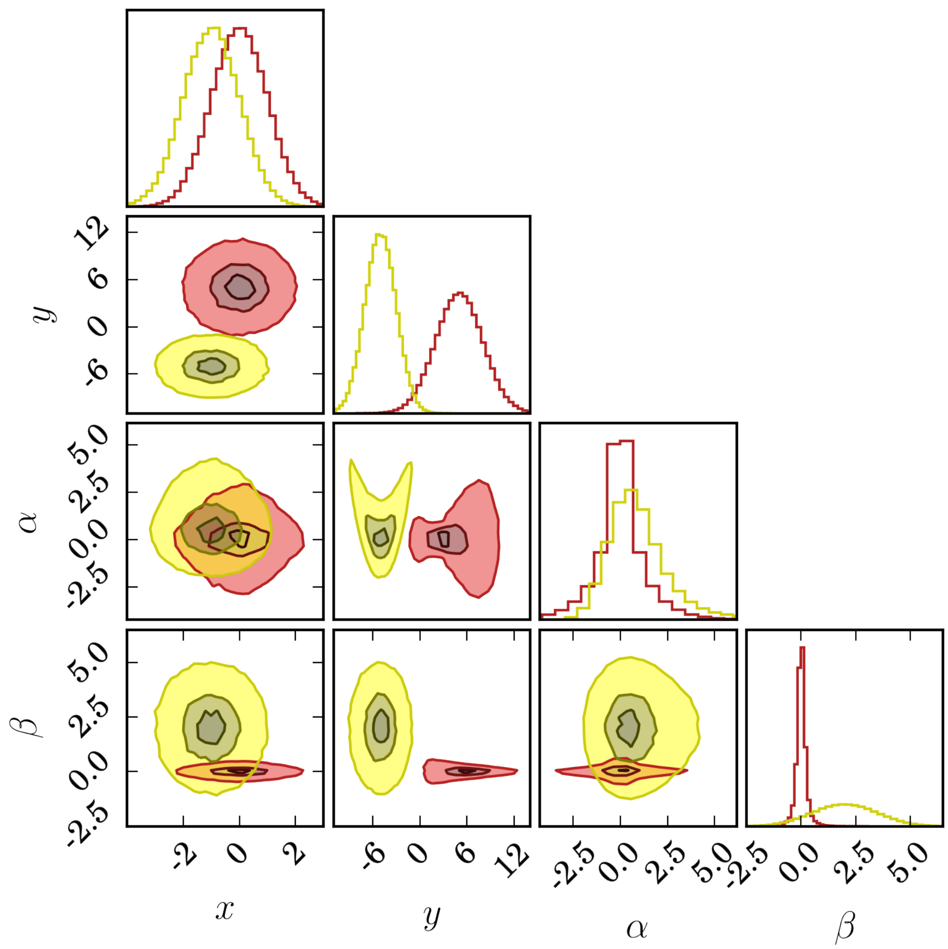
y
α
β
x
y
α
β

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 opactiy 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.

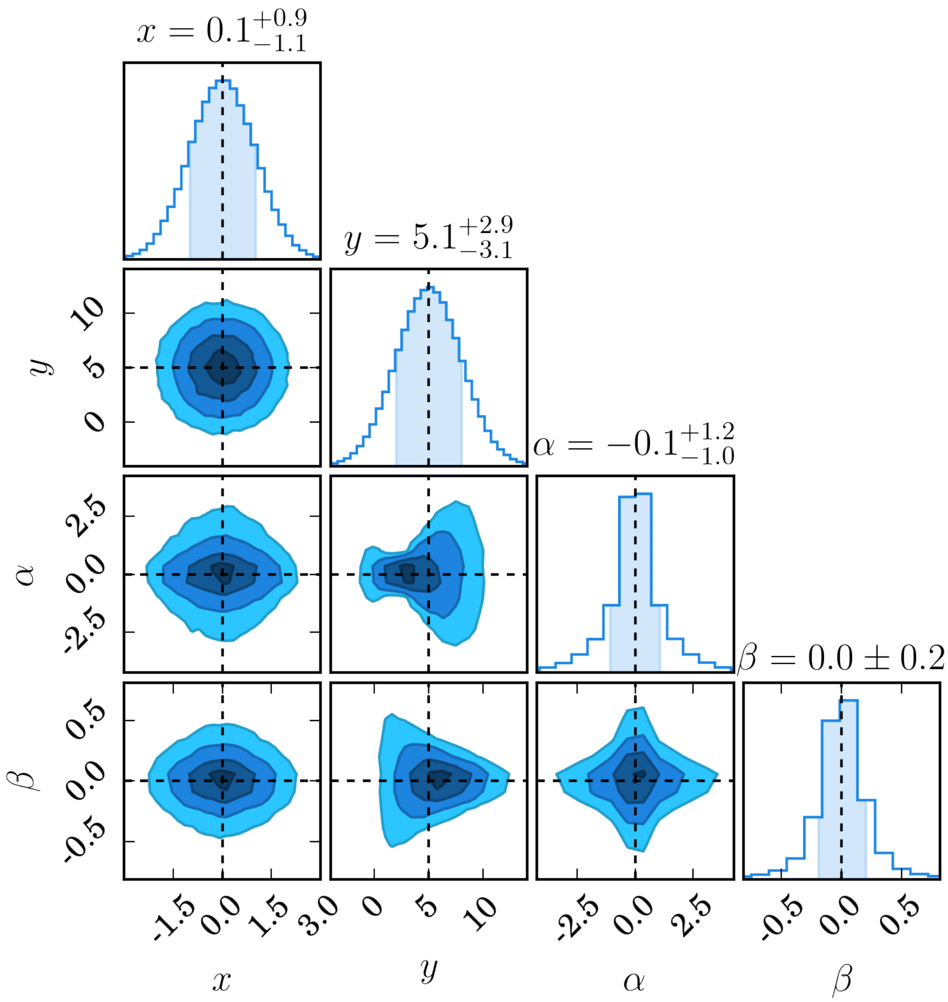
In the code there are two examples. The first, where a list is passed in, and the second, where an incomplete dictionary of truth values is passed in. In the second case, customised values for truth line plotting are used. The figures are respectively

various7_rainbow()

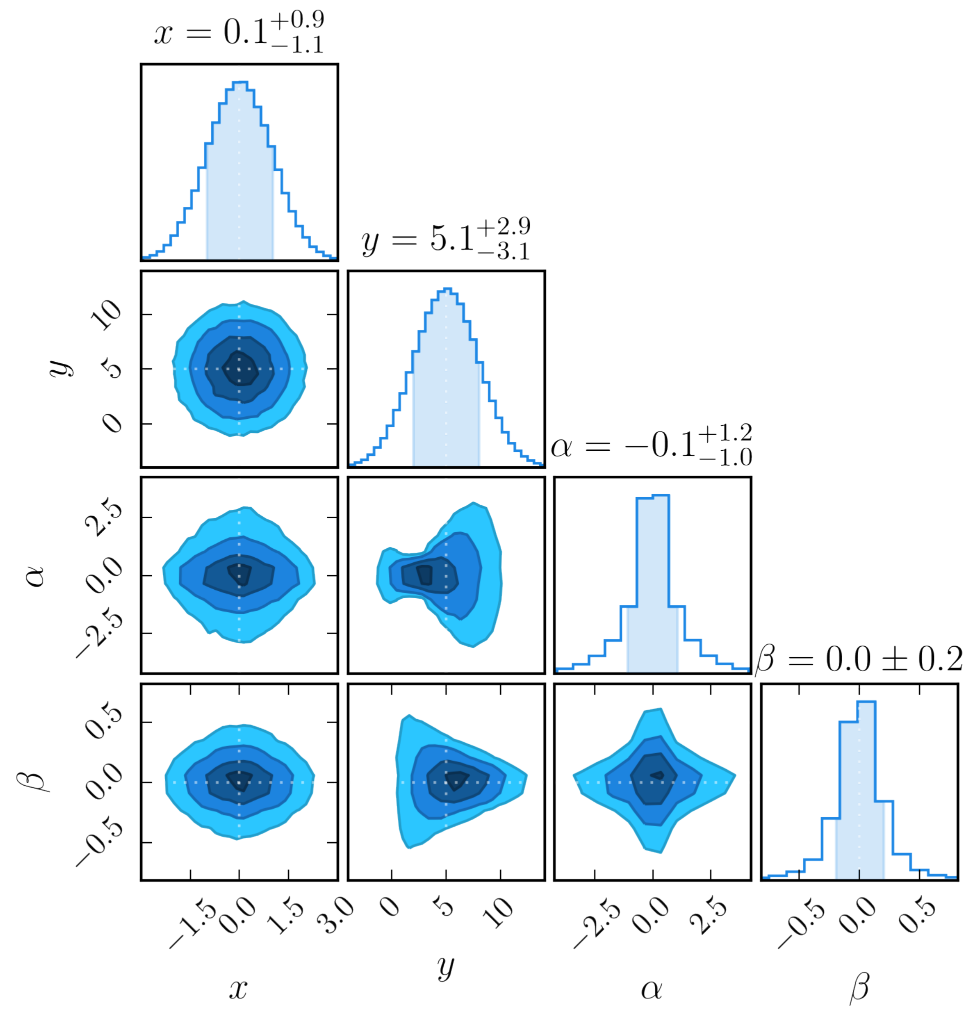
An example on using the rainbow with sans-serif fonts and too many bins!

various8_extents()

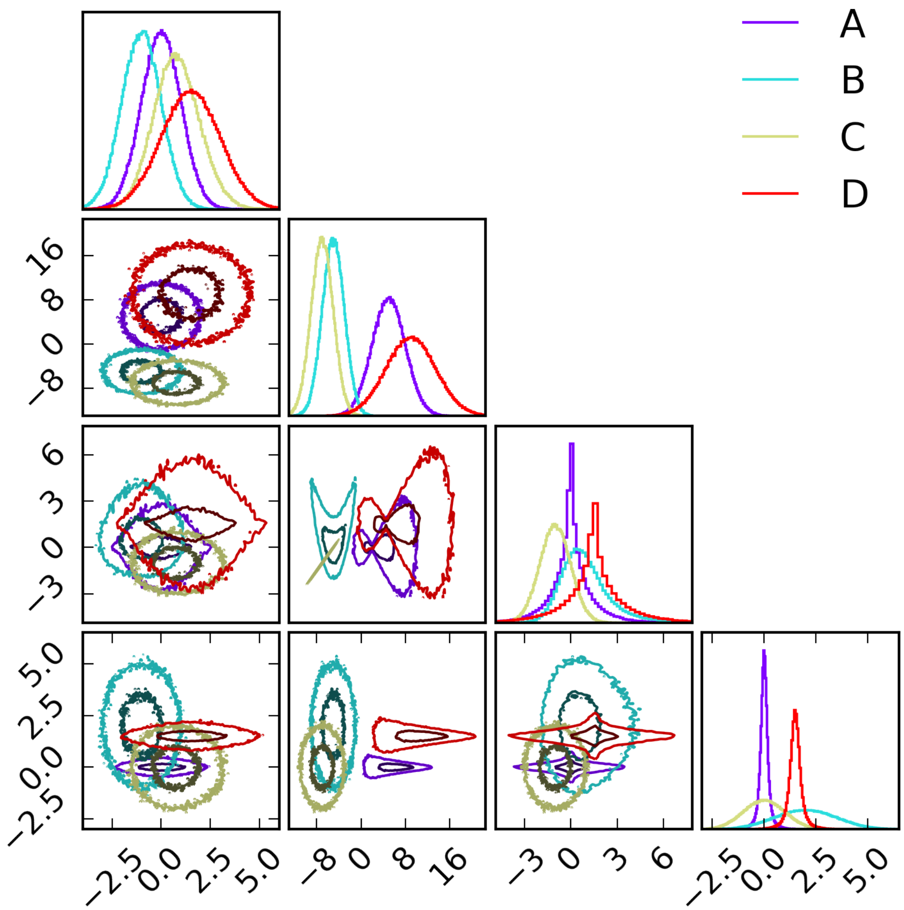
An example on using customised extents. Similarly to the example for truth values in `various7_truth_values()`, you can pass a list in, or a dictionary.



$x = 0.1^{+0.9}_{-1.1}$
$y = 5.1^{+2.9}_{-3.1}$
$\alpha = -0.1^{+1.2}_{-1.0}$
$\beta = 0.0 \pm 0.2$
x
y
α
β

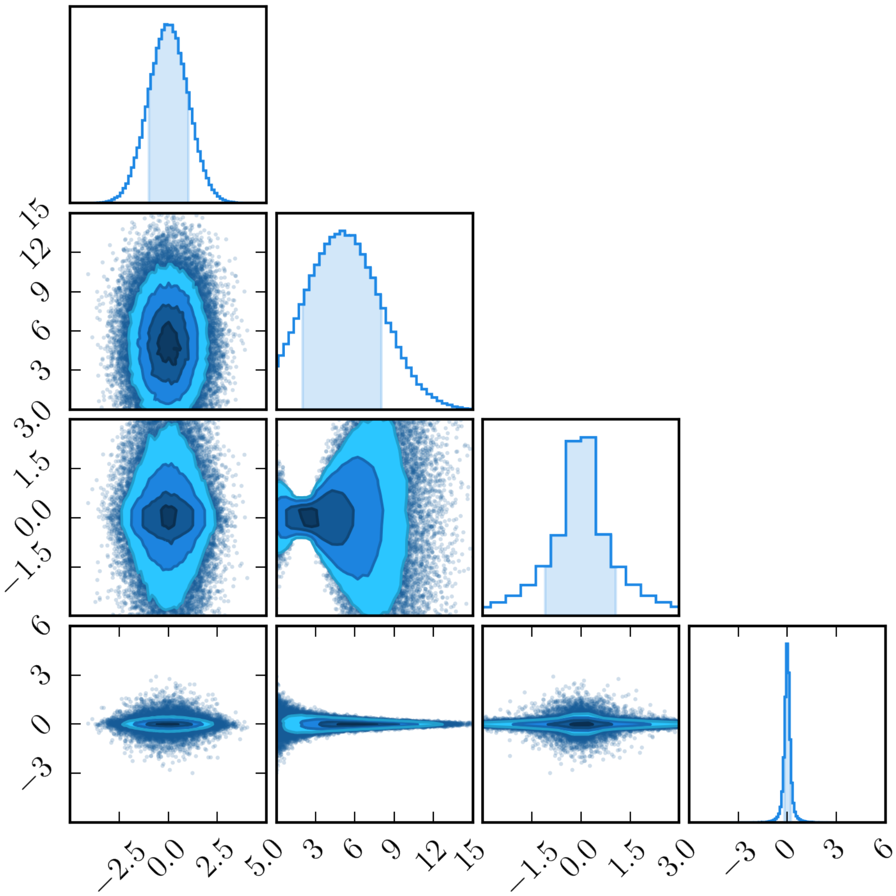


$x = 0.1^{+0.9}_{-1.1}$
$y = 5.1^{+2.9}_{-3.1}$
$\alpha = -0.1^{+1.2}_{-1.0}$
$\beta = 0.0 \pm 0.2$
x
y
α
β



A
B
C
D
16
8
0
−8
6
3
0
−3
5.0
2.5
0.0
−2.5
−2.5
0.0
2.5
5.0
−8
0
8
16
−3
0
3
6
−2.5
0.0
2.5
5.0

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`**

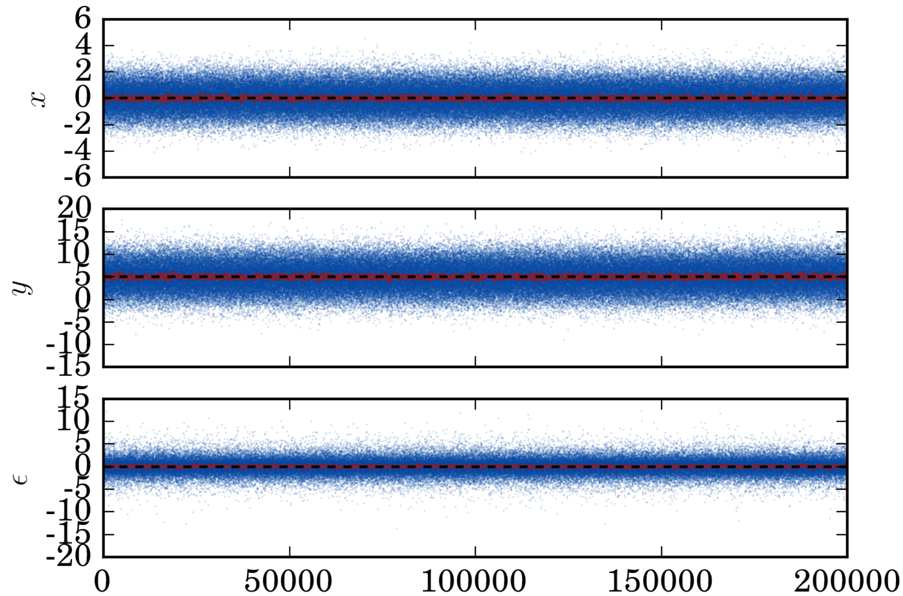
The single chain demo for Chain Consumer. Dummy class used to get documentation caught by `sphinx-apidoc`, it servers no other purpose.

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:



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 / kidden / 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 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 Ω_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:

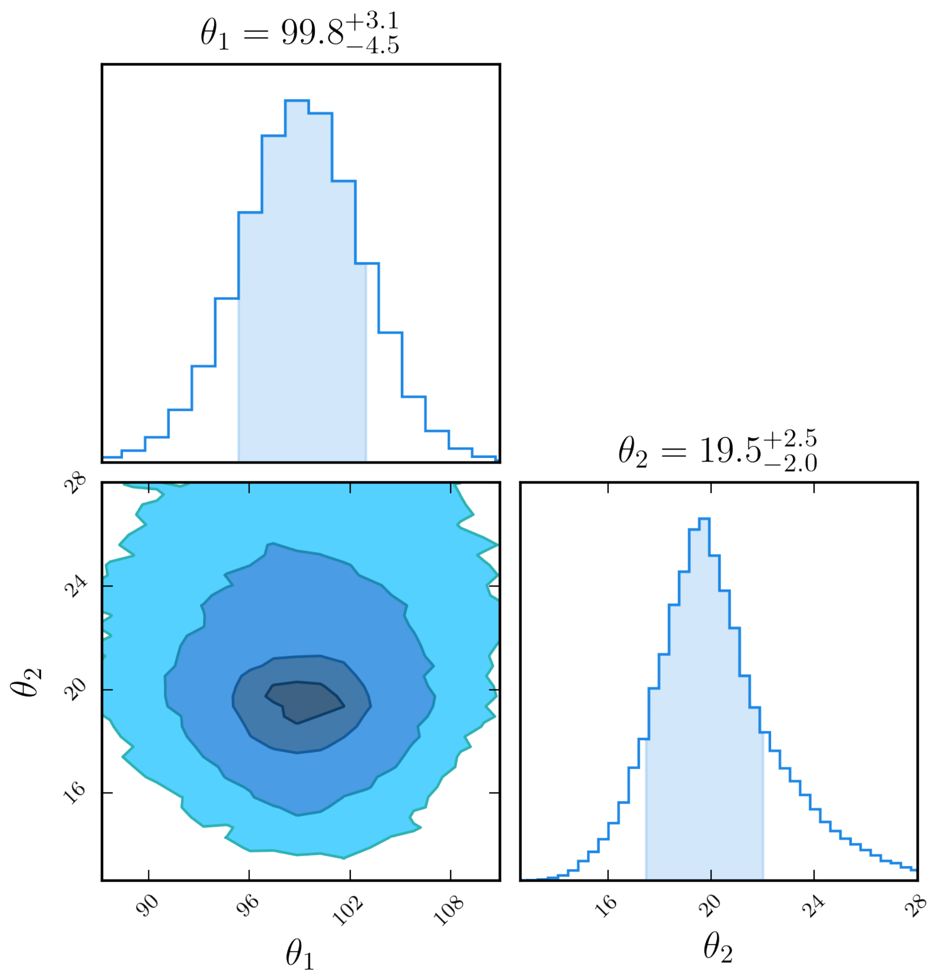
class `dessn.simple.modelbased.exampleModel.`**`FluxToLuminosity`**

Bases: *dessn.model.edge.Edge*

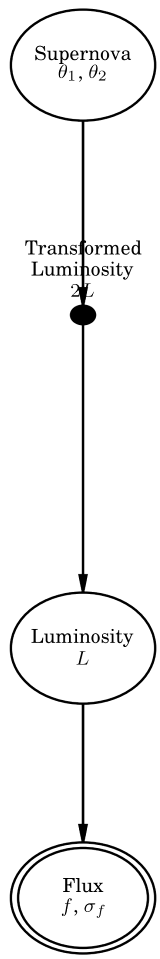
`get_log_likelihood`(*data*)

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

Bases: *dessn.model.node.NodeLatent*



$\theta_1 = 99.8^{+3.1}_{-4.5}$
$\theta_2 = 19.5^{+2.5}_{-2.0}$
θ_2
θ_1
θ_2
28
24
20
16
90
96
102
108
16
20
24
28


Supernova
θ_1, θ_2
Transformed
Luminosity
$2L$
Luminosity
L
Flux
f, σ_f

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 θ, 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 σ_i from the actual supernova luminosity, and the supernova luminosity is drawn from an underlying gaussian distribution parameterised by θ.

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

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

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

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.

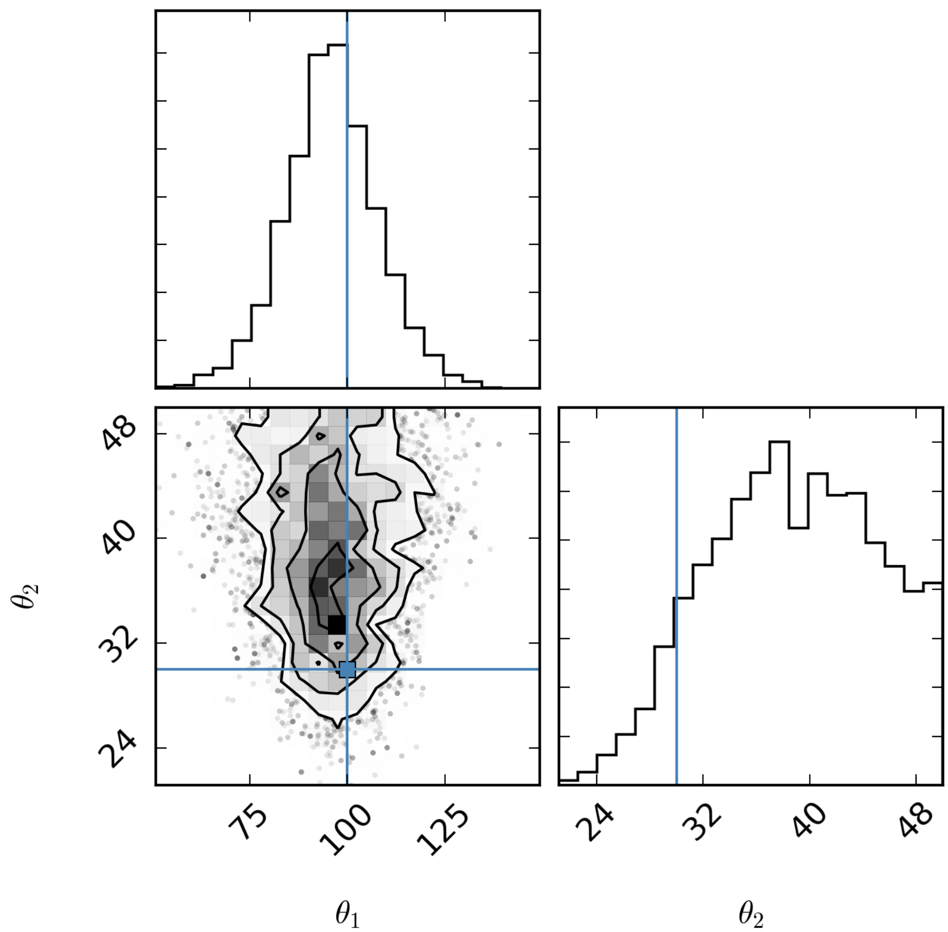
dessn.simple.exampleIntegral module

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 θ 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

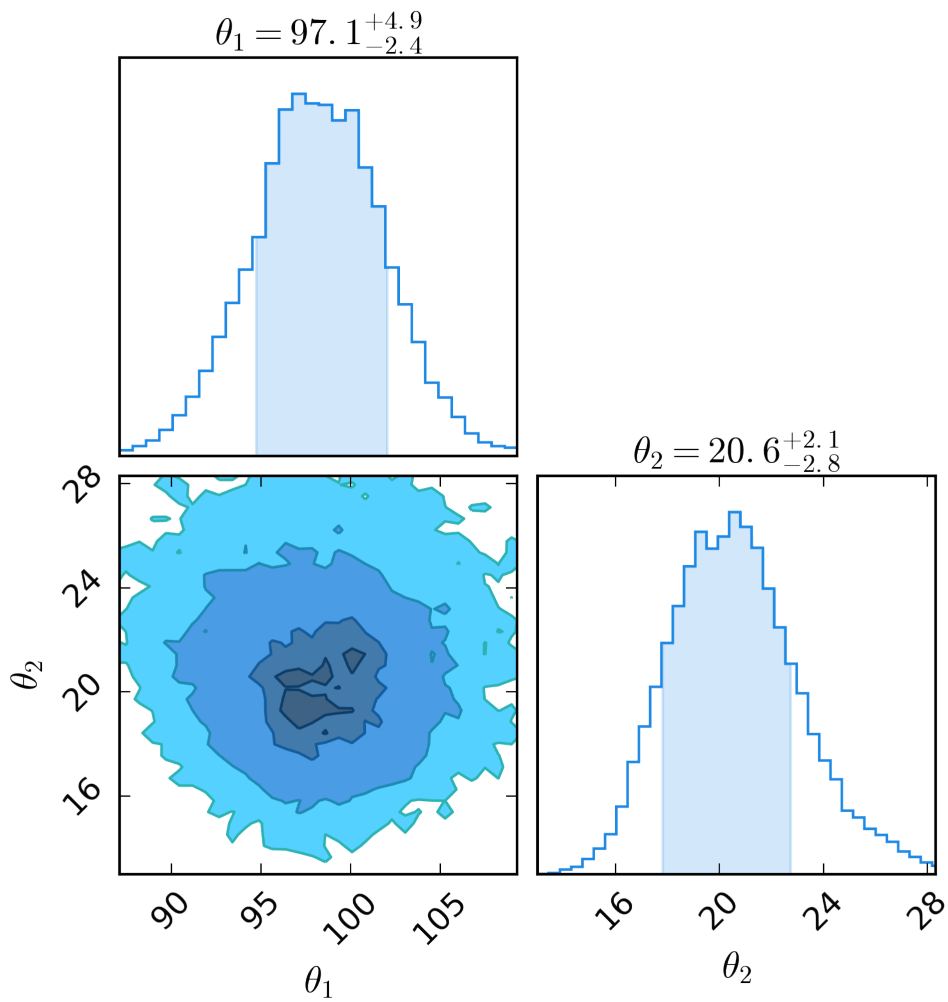
do_emcee(*nwalkers=500*, *nburn=500*, *nsteps=1000*)

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.



$\theta_1 = 97.1^{+4.9}_{-2.4}$
$\theta_2 = 20.6^{+2.1}_{-2.8}$
28
24
20
16
θ_2
90
95
100
105
θ_1
16
20
24
28
θ_2

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*)

Given CCD efficiency, convert from count to flux f and flux error σ_f.

$$f = \frac{\text{count}}{\text{conversion} \times \text{efficiency}}$$
$$\sigma_f = \frac{\sqrt{\text{count}}}{\text{conversion} \times \text{efficiency}}$$

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.

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

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

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

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

$$P(T|r) = \binom{N_{\rm Total}}{N_{\rm SnII}} r^{N_{\rm SnIa}} (1-r)^{N_{\rm SnII}}$$

In log space, this is

$$\log(P(T|r)) = \log \binom{N_{\rm Total}}{N_{\rm SnII}} + N_{\rm SnIa}\log(r) + N_{\rm SnII}\log(1-r)$$

In the code, I approximate the choose function using the log gamma functions.

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

`get_log_likelihood`(*data*)
Assume the redshift distribution follows a uniform distribution (for misidentification) with a tight Gaussian peak around the observed redshift.

Assumes the misidentifiation range is between $z = 0$ and $z = 2$. Also assumes the success rate is 99% for observed spectra

$$P(z_o|z) = \frac{0.01}{2} + \frac{0.99}{\sqrt{2\pi} z_{o,\rm err}} \exp\left(-\frac{(z-z_o)^2}{2z_{o,\rm err}^2} \right)$$

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.

$$P(T_o|T) = 0.1 + 0.8\delta_{T_o,T}$$

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*, *redshift_errors*)
Bases: *`dessn.model.node.NodeObserved`*

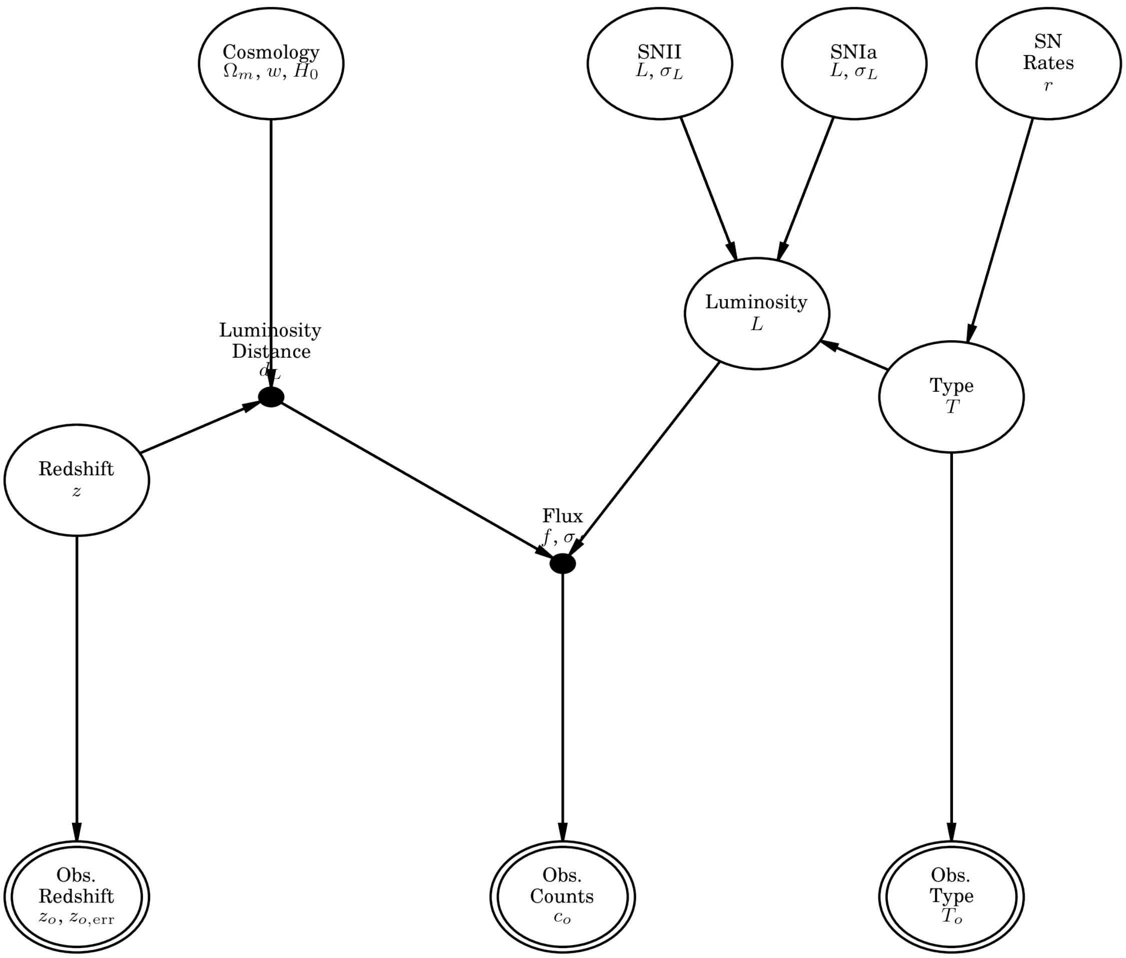
class `dessn.toy.observed.`**`ObservedType`**(*types*)
Bases: *`dessn.model.node.NodeObserved`*

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*)

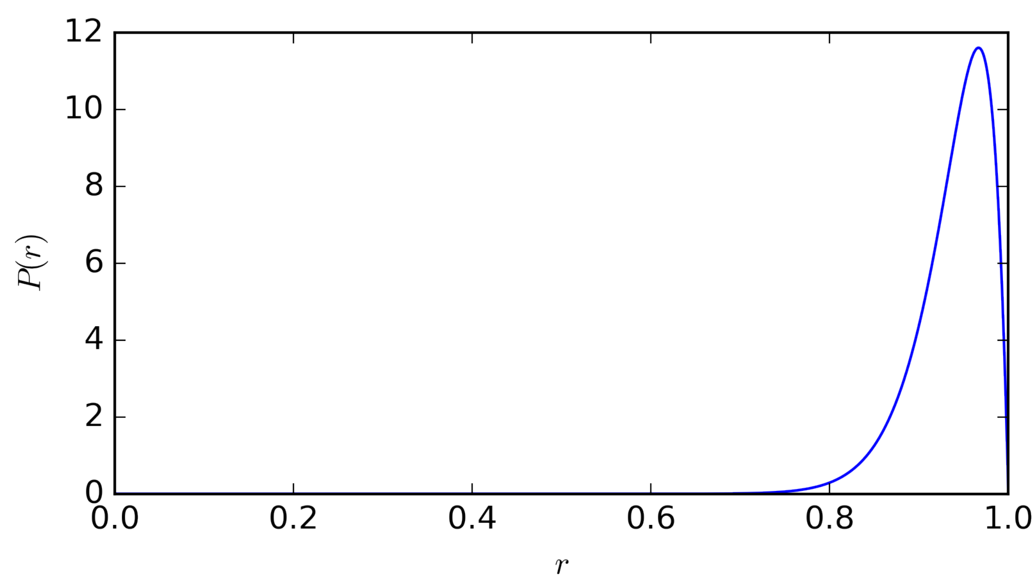
class dessn.toy.underlying.**SupernovaIaDist**
Bases: *dessn.model.node.NodeUnderlying*

get_log_prior(*data*)

class dessn.toy.underlying.**SupernovaRate**
Bases: *dessn.model.node.NodeUnderlying*

get_log_prior(*data*)
Here we model the prior on the supernova rate as a Beta function, parametrised as $\mathrm{Beta}(\alpha = 30, \beta = 2)$.

The probability density function is visualised below, where r represents the overall ratio of type Ia supernova over all supernova.



plot_dist()
Plots the distribution for easy visualisation.

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.math module

dessn.utility.math.**plus**(*loga*, *logb*)
Returns $log(a + b)$ when given $log(a)$ and $log(b)$.

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

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V