kalepy

Release 1.4.3

Luke Zoltan Kelley

May 22, 2023

CONTENTS:

Multidimensional kernel density estimation for distribution functions, resampling, and plotting.

kalepy on github

CHAPTER

ONE

INSTALLATION

```
pip install kalepy
```

or from source, for development:

```
git clone https://github.com/lzkelley/kalepy.git
pip install -e kalepy
```

CHAPTER

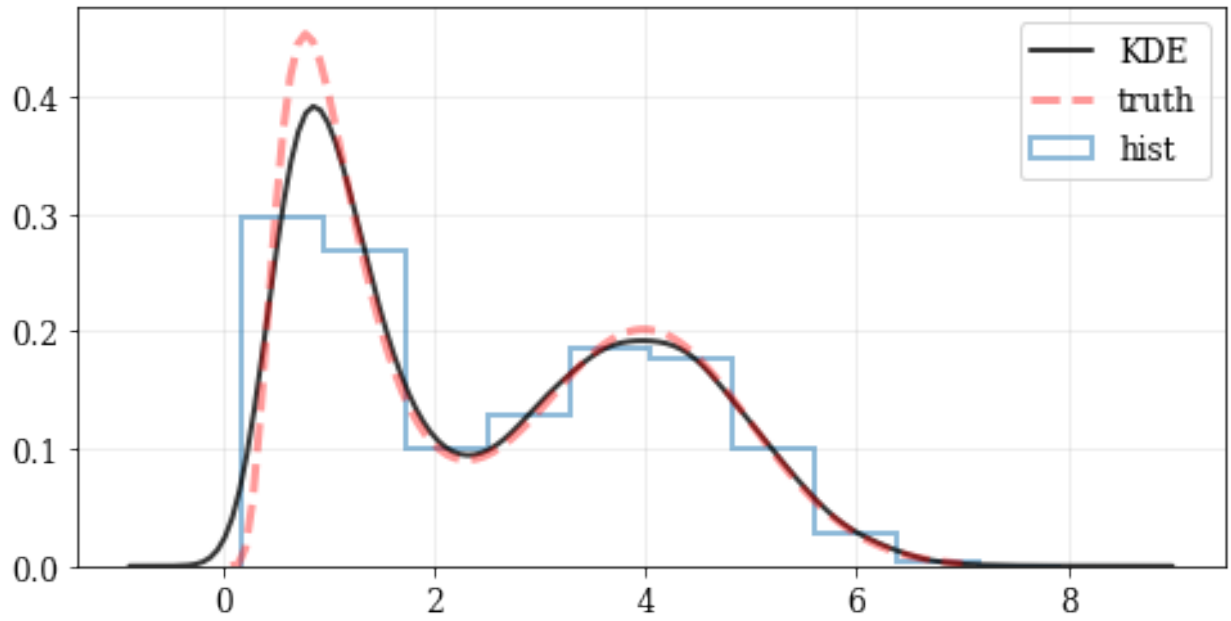
TWO

QUICKSTART

Basic examples are shown below.
The top-level API for KDE is here,
and for plotting is here,
The README file on github also includes installation and quickstart examples.

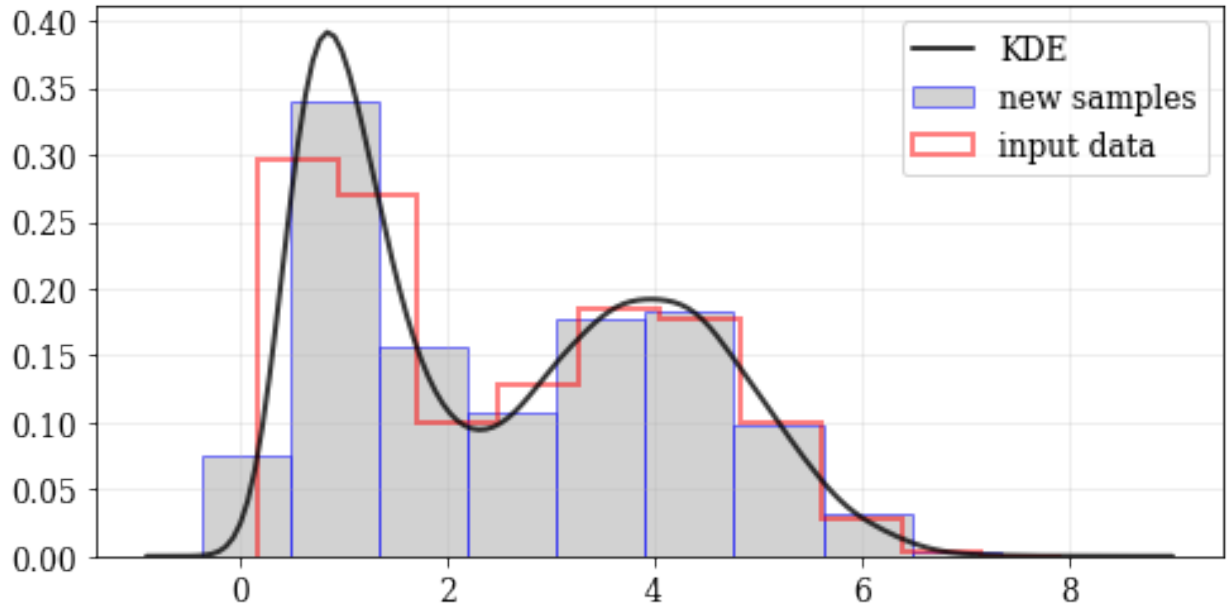
2.1 One dimensional kernel density estimation:

```
import kalepy as kale
import matplotlib.pyplot as plt
points, density = kale.density(data, points=None)
plt.plot(points, density, 'k-', lw=2.0, alpha=0.8, label='KDE')
```



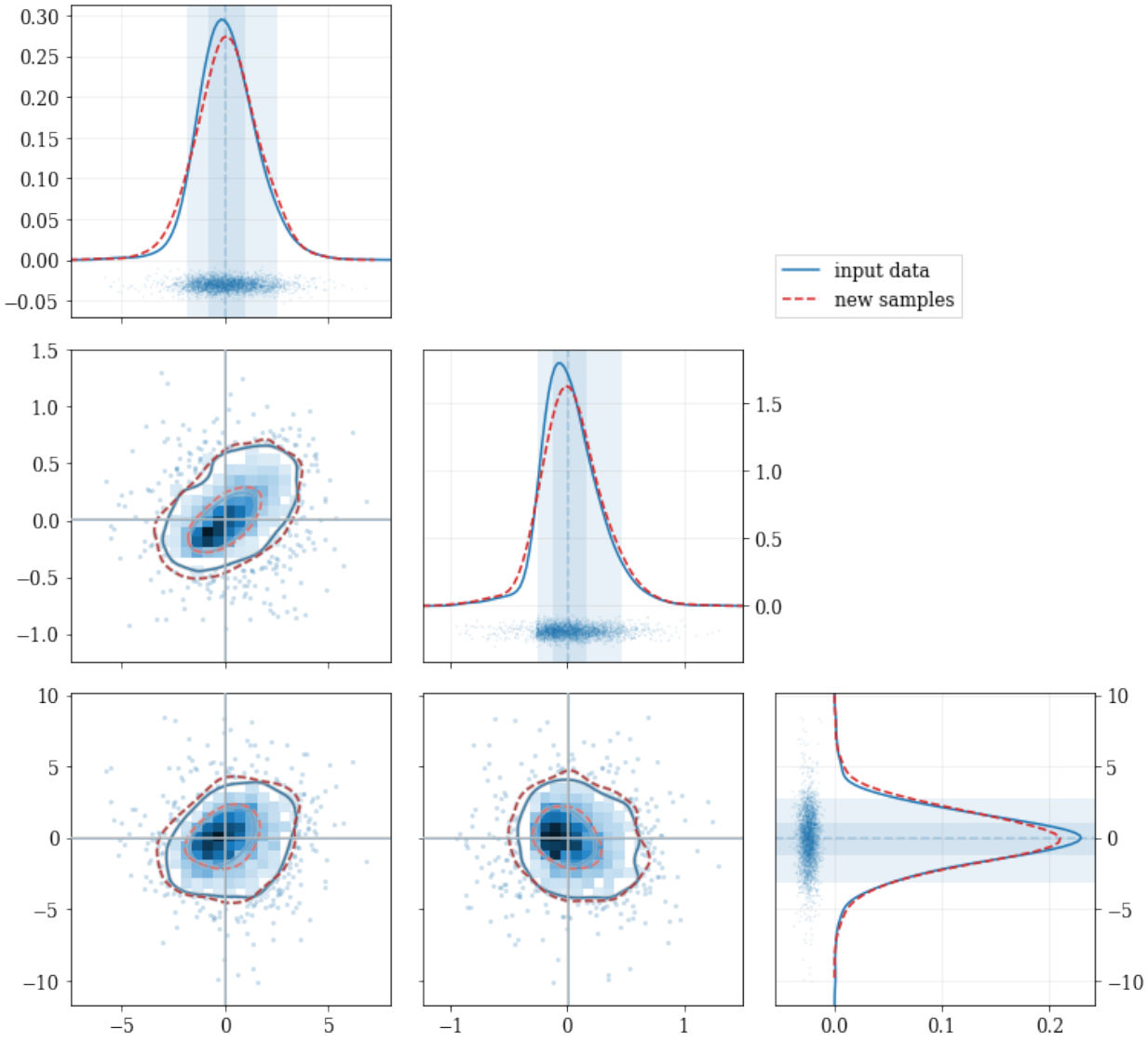
2.2 One dimensional resampling:

```
# Draw new samples from the KDE reconstructed PDF
samples = kale.resample(data)
plt.hist(samples, density=True, alpha=0.5, label='new samples', color='0.65', edgecolor=
↪'b')
```



2.3 Multi-dimensional kernel density estimation:

```
# Construct a KDE instance from data, shaped (N, 3) for `N` data points, and 3 dimensions
kde = kale.KDE(data)
# Build a corner plot using the `kalepy` plotting submodule
corner = kale.corner(kde)
```


input data
new samples

CHAPTER

THREE

DOCUMENTATION

A number of examples are included in the package notebooks, and the readme file. Some background information and references are included in the JOSS paper.

3.1 Kernel Density Estimation (KDE) API

The primary API is two functions in the top level package: *kalepy.density* and *kalepy.resample*. Additionally, *kalepy.pdf* is included which is a shorthand for *kalepy.density(..., probability=True)* — i.e. a normalized density distribution.

Each of these functions constructs a *KDE* (kalepy.kde.KDE) instance, calls the corresponding member function, and returns the results. If multiple operations will be done on the same data set, it will be more efficient to construct the *KDE* instance manually and call the methods on that. i.e.

```
kde = kalepy.KDE(data)              # construct `KDE` instance
points, density = kde.density()     # use `KDE` for density-estimation
new_samples = kde.resample()        # use same `KDE` for resampling
```

3.1.1 Basic Usage

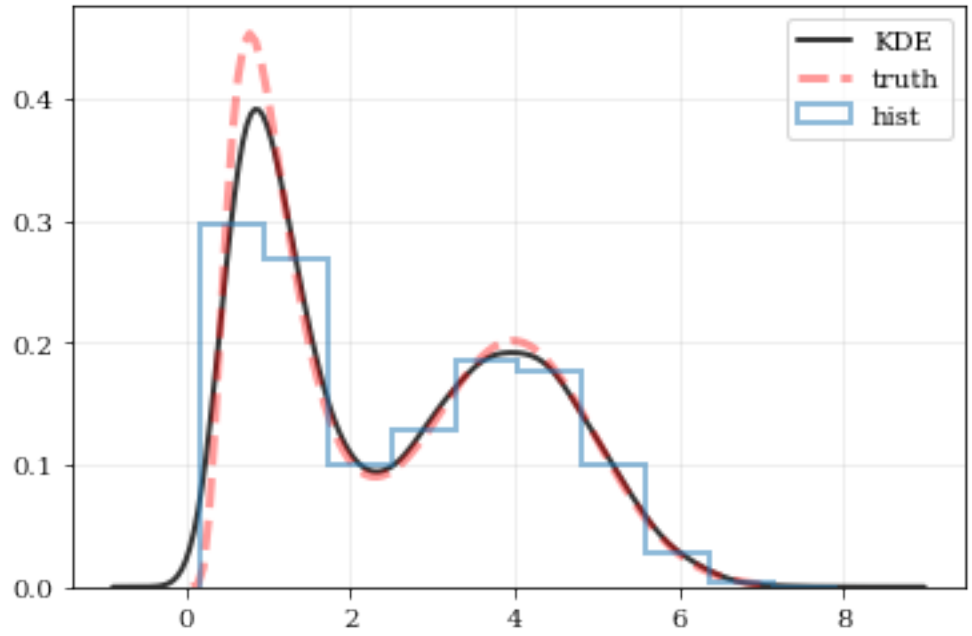
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

import kalepy as kale

from kalepy.plot import nbshow
```

Generate some random data, and its corresponding distribution function

```
NUM = int(1e4)
np.random.seed(12345)
# Combine data from two different PDFs
_d1 = np.random.normal(4.0, 1.0, NUM)
_d2 = np.random.lognormal(0, 0.5, size=NUM)
data = np.concatenate([_d1, _d2])

# Calculate the "true" distribution
xx = np.linspace(0.0, 7.0, 100)[1:]
yy = 0.5*np.exp(-(xx - 4.0)**2/2) / np.sqrt(2*np.pi)
yy += 0.5 * np.exp(-np.log(xx)**2/(2*0.5**2)) / (0.5*xx*np.sqrt(2*np.pi))
```

Plotting Smooth Distributions

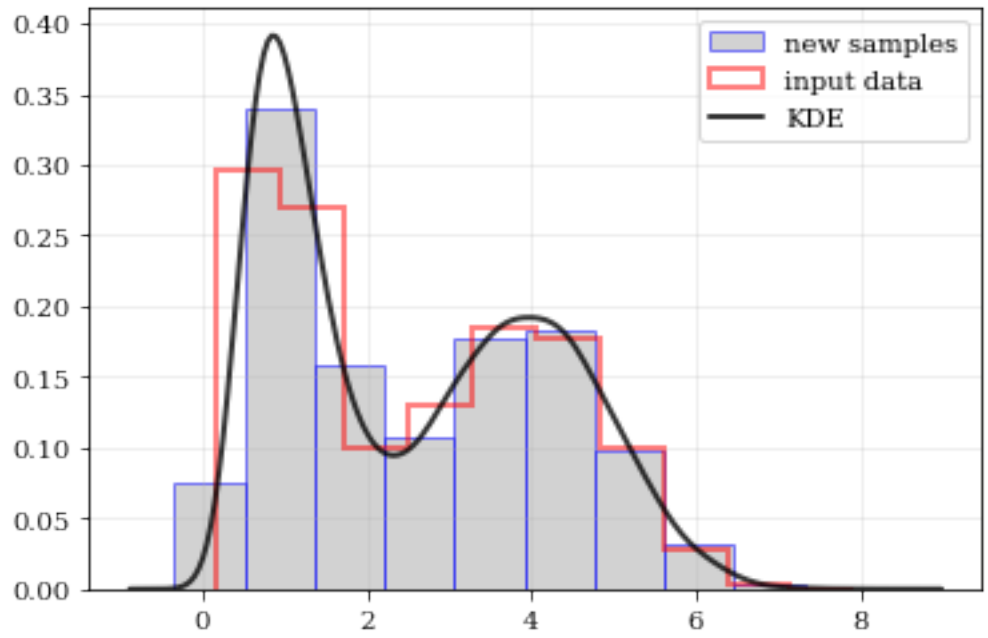
```
# Reconstruct the probability-density based on the given data points.
points, density = kale.density(data, probability=True)

# Plot the PDF
plt.plot(points, density, 'k-', lw=2.0, alpha=0.8, label='KDE')

# Plot the "true" PDF
plt.plot(xx, yy, 'r--', alpha=0.4, lw=3.0, label='truth')

# Plot the standard, histogram density estimate
plt.hist(data, density=True, histtype='step', lw=2.0, alpha=0.5, label='hist')

plt.legend()
nbshow()
```



resampling: constructing statistically similar values

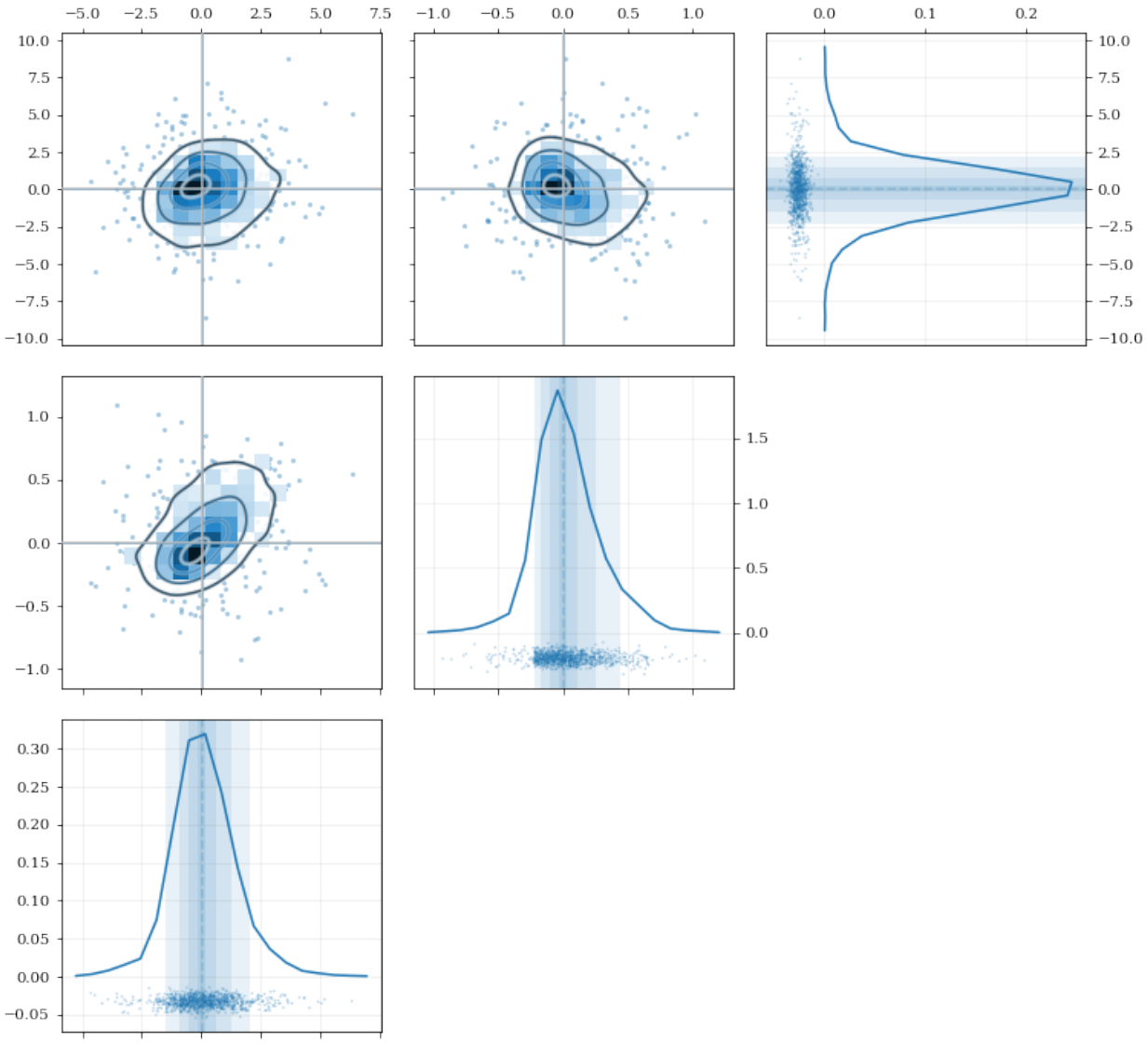
Draw a new sample of data-points from the KDE PDF

```
# Draw new samples from the KDE reconstructed PDF
samples = kale.resample(data)

# Plot new samples
plt.hist(samples, density=True, label='new samples', alpha=0.5, color='0.65', edgecolor=
↪'b')
# Plot the old samples
plt.hist(data, density=True, histtype='step', lw=2.0, alpha=0.5, color='r', label='input␣
↪data')

# Plot the KDE reconstructed PDF
plt.plot(points, density, 'k-', lw=2.0, alpha=0.8, label='KDE')

plt.legend()
nbshow()
```



Multivariate Distributions

```
# Load some random-ish three-dimensional data
data = kale.utils._random_data_3d_02()

# Construct a KDE
kde = kale.KDE(data)

# Plot the data and distributions using the builtin `kalepy.corner` plot
kale.corner(kde)

nbshow()
```

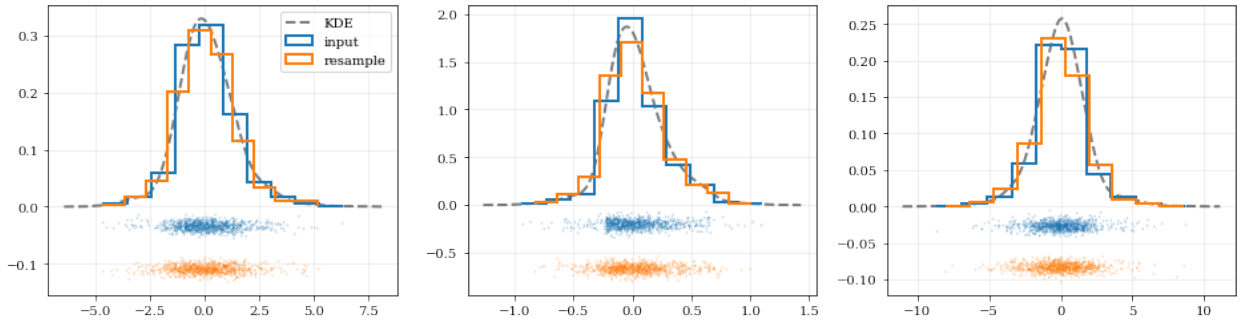



```
# Resample the data (default output is the same size as the input data)
samples = kde.resample()


# ---- Plot the input data compared to the resampled data ----

fig, axes = plt.subplots(figsize=[16, 4], ncols=kde.ndim)

for ii, ax in enumerate(axes):
    # Calculate and plot PDF for `ii`th parameter (i.e. data dimension `ii`)
    xx, yy = kde.density(params=ii, probability=True)
    ax.plot(xx, yy, 'k--', label='KDE', lw=2.0, alpha=0.5)
    # Draw histograms of original and newly resampled datasets
    *_, h1 = ax.hist(data[ii], histtype='step', density=True, lw=2.0, label='input')
    *_, h2 = ax.hist(samples[ii], histtype='step', density=True, lw=2.0, label='resample
↪')
    # Add 'kalepy.carpet' plots showing the data points themselves
```

(continues on next page)

(continued from previous page)

```
    kale.carpet(data[ii], ax=ax, color=h1[0].get_facecolor())
    kale.carpet(samples[ii], ax=ax, color=h2[0].get_facecolor(), shift=ax.get_ylim()[0])

axes[0].legend()
nbshow()
```

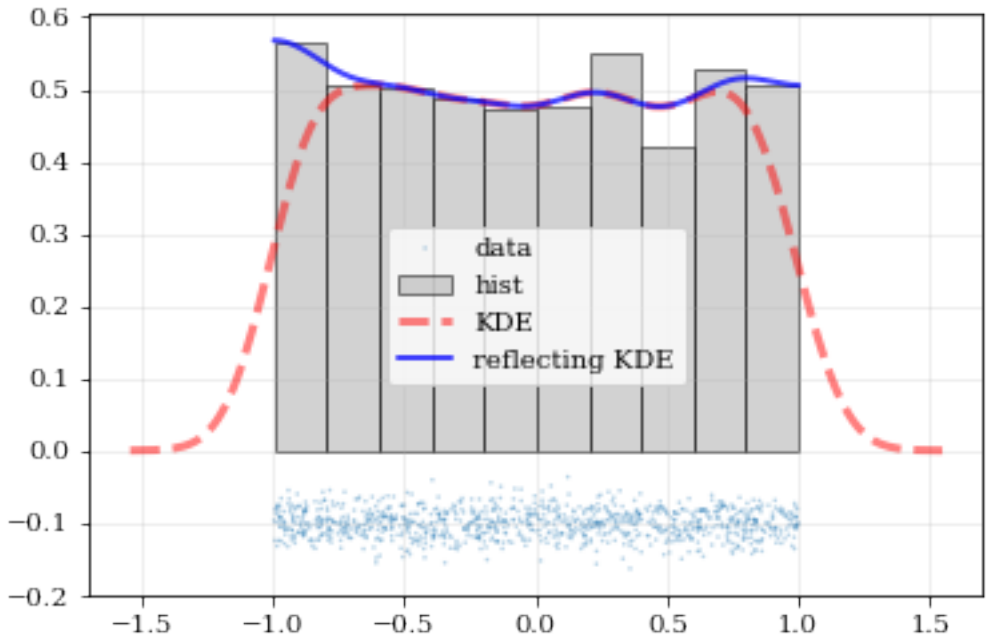


3.1.2 Fancy Usage

Reflecting Boundaries

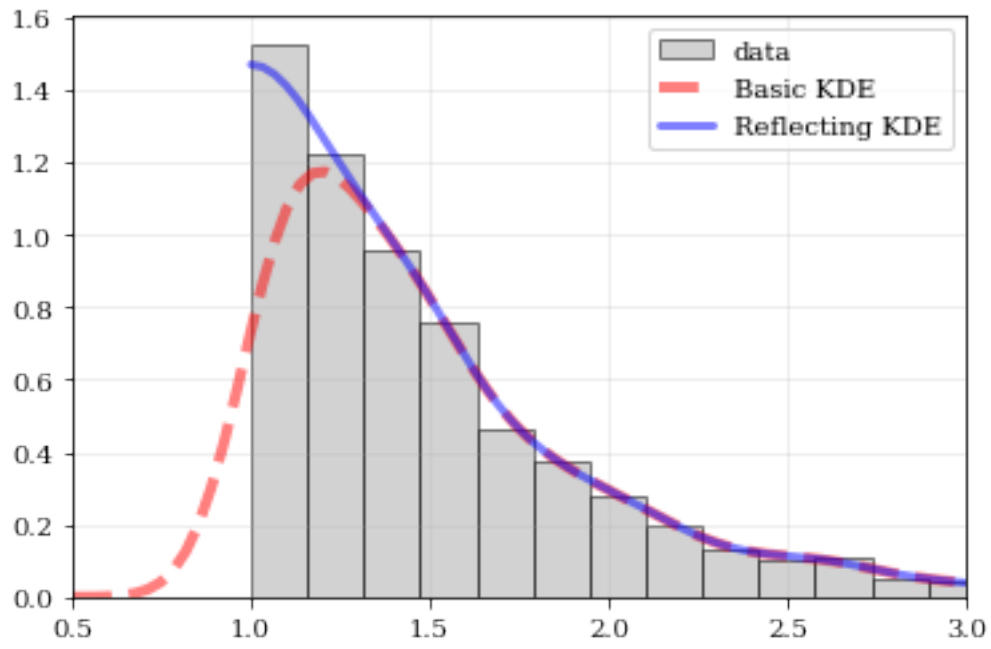
What if the distributions you're trying to capture have edges in them, like in a uniform distribution between two bounds? Here, the KDE chooses 'reflection' locations based on the extrema of the given data.

```
# Uniform data (edges at -1 and +1)
NDATA = 1e3
np.random.seed(54321)
data = np.random.uniform(-1.0, 1.0, int(NDATA))

# Create a 'carpet' plot of the data
kale.carpet(data, label='data')
# Histogram the data
plt.hist(data, density=True, alpha=0.5, label='hist', color='0.65', edgecolor='k')

# ---- Standard KDE will undershoot just-inside the edges and overshoot outside edges
points, pdf_basic = kale.density(data, probability=True)
plt.plot(points, pdf_basic, 'r--', lw=3.0, alpha=0.5, label='KDE')

# ---- Reflecting KDE keeps probability within the given bounds
# setting `reflect=True` lets the KDE guess the edge locations based on the data extrema
points, pdf_reflect = kale.density(data, reflect=True, probability=True)
plt.plot(points, pdf_reflect, 'b-', lw=2.0, alpha=0.75, label='reflecting KDE')

plt.legend()
nbshow()
```



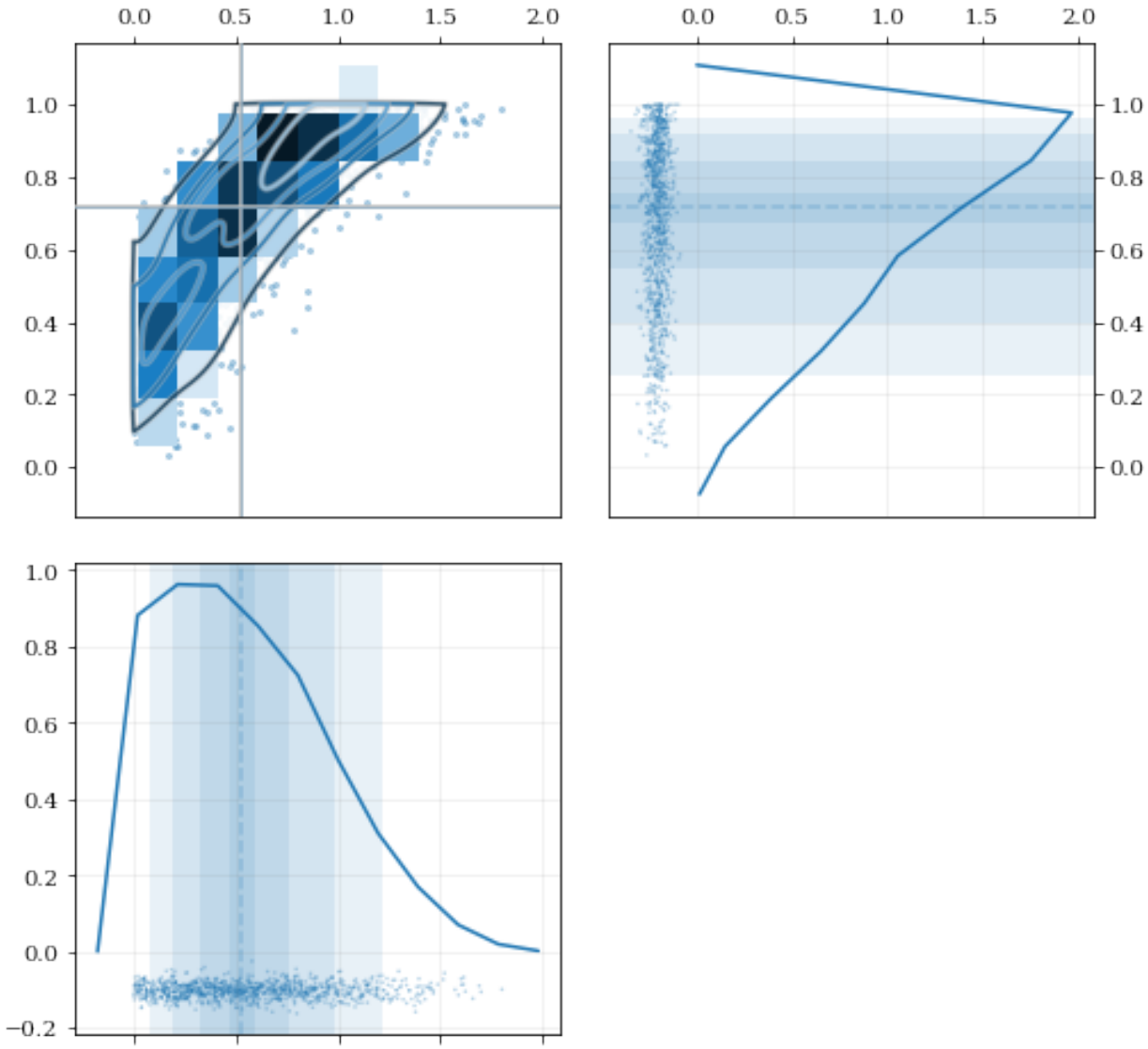
Explicit reflection locations can also be provided (in any number of dimensions).

```
# Construct random data, add an artificial 'edge'
np.random.seed(5142)
edge = 1.0
data = np.random.lognormal(sigma=0.5, size=int(3e3))
data = data[data >= edge]

# Histogram the data, use fixed bin-positions
edges = np.linspace(edge, 4, 20)
plt.hist(data, bins=edges, density=True, alpha=0.5, label='data', color='0.65',␣
↪edgecolor='k')

# Standard KDE with over & under estimates
points, pdf_basic = kale.density(data, probability=True)
plt.plot(points, pdf_basic, 'r--', lw=4.0, alpha=0.5, label='Basic KDE')

# Reflecting KDE setting the lower-boundary to the known value
#    There is no upper-boundary when `None` is given.
points, pdf_basic = kale.density(data, reflect=[edge, None], probability=True)
plt.plot(points, pdf_basic, 'b-', lw=3.0, alpha=0.5, label='Reflecting KDE')

plt.gca().set_xlim(edge - 0.5, 3)
plt.legend()
nbshow()
```



Multivariate Reflection

```
# Load a predefined dataset that has boundaries at:
#   x: 0.0 on the low-end
#   y: 1.0 on the high-end
data = kale.utils._random_data_2d_03()

# Construct a KDE with the given reflection boundaries given explicitly
kde = kale.KDE(data, reflect=[[0, None], [None, 1]])

# Plot using default settings
kale.corner(kde)

nbshow()
```



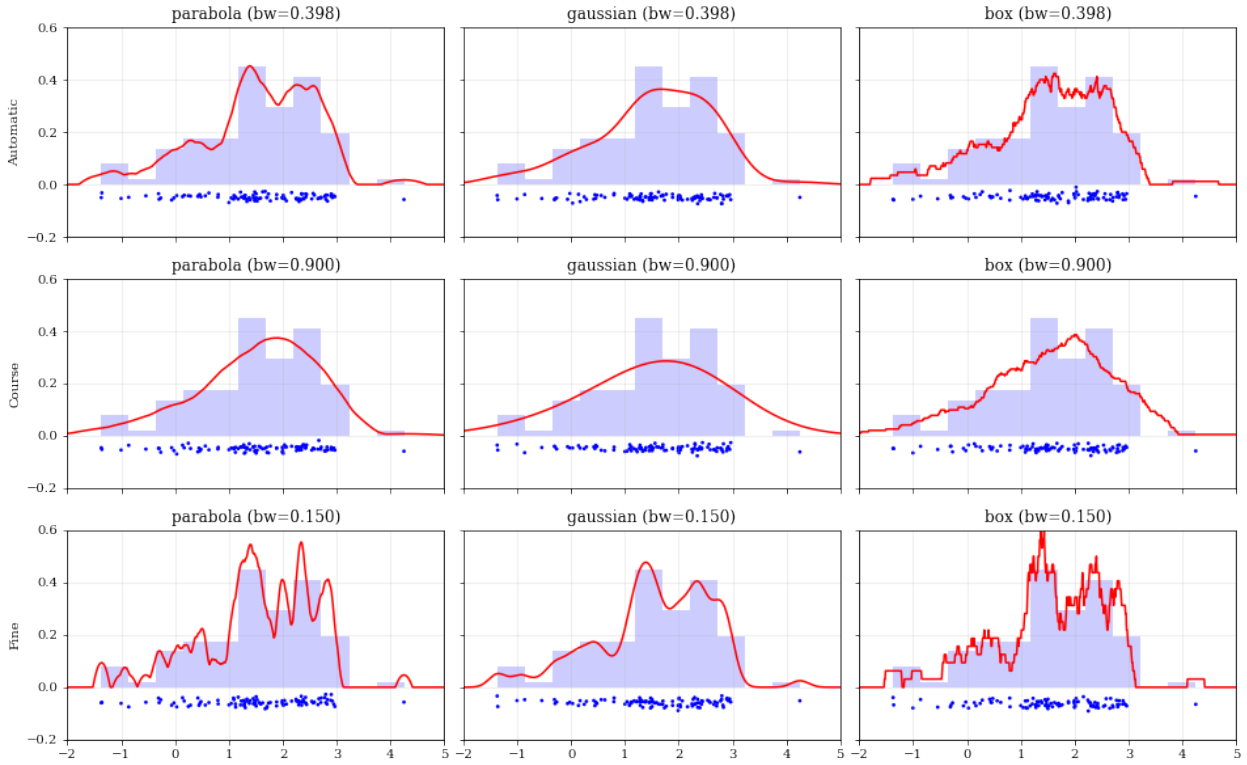
Specifying Bandwidths and Kernel Functions

```
# Load predefined 'random' data
data = kale.utils._random_data_1d_02(num=100)
# Choose a uniform x-spacing for drawing PDFs
xx = np.linspace(-2, 8, 1000)

# ------ Choose the kernel-functions and bandwidths to test -------  #
kernels = ['parabola', 'gaussian', 'box']                             #
bandwidths = [None, 0.9, 0.15]     # `None` means let kalepy choose  #
# ------------------------------------------------------------------  #

ylabels = ['Automatic', 'Course', 'Fine']
fig, axes = plt.subplots(figsize=[16, 10], ncols=len(kernels), nrows=len(bandwidths),␣
↪sharex=True, sharey=True)
```

(continues on next page)

(continued from previous page)

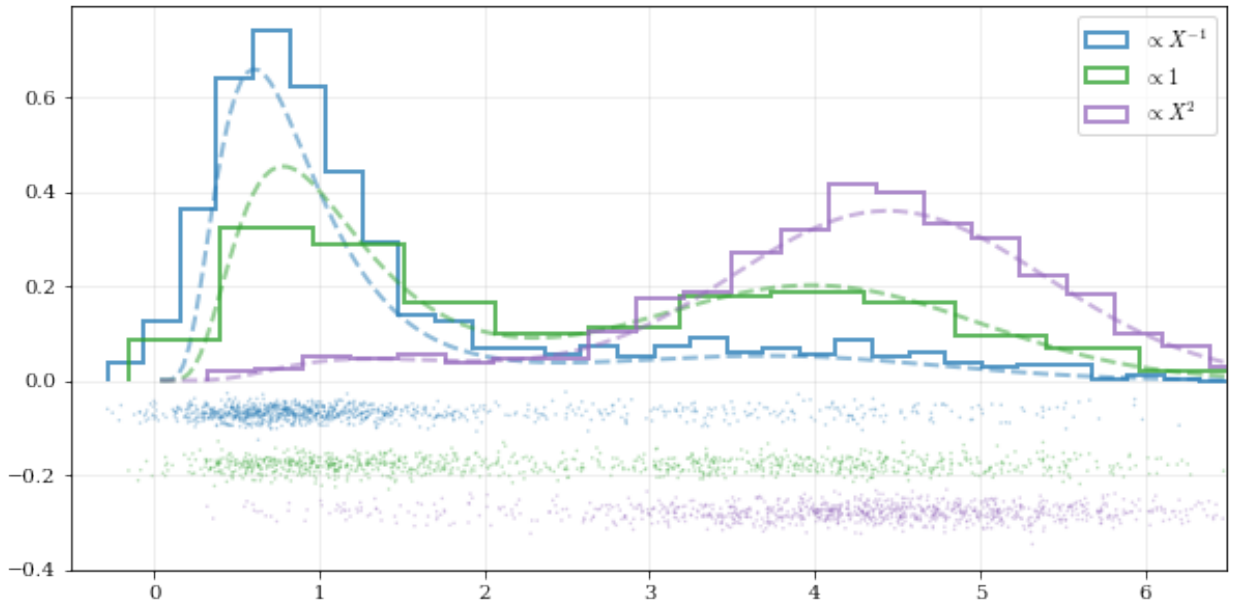
```
plt.subplots_adjust(hspace=0.2, wspace=0.05)
for (ii, jj), ax in np.ndenumerate(axes):

    # ---- Construct KDE using particular kernel-function and bandwidth ---- #
    kern = kernels[jj]                                                        #
    bw = bandwidths[ii]                                                       #
    kde = kale.KDE(data, kernel=kern, bandwidth=bw)                           #
    # ----------------------------------------------------------------------- #

    # If bandwidth was set to `None`, then the KDE will choose the 'optimal' value
    if bw is None:
        bw = kde.bandwidth[0, 0]

    ax.set_title('{} (bw={:.3f})'.format(kern, bw))
    if jj == 0:
        ax.set_ylabel(ylabels[ii])

    # plot the KDE
    ax.plot(*kde.pdf(points=xx), color='r')
    # plot histogram of the data (same for all panels)
    ax.hist(data, bins='auto', color='b', alpha=0.2, density=True)
    # plot  carpet  of the data (same for all panels)
    kale.carpet(data, ax=ax, color='b')

ax.set(xlim=[-2, 5], ylim=[-0.2, 0.6])
nbshow()
```



Resampling

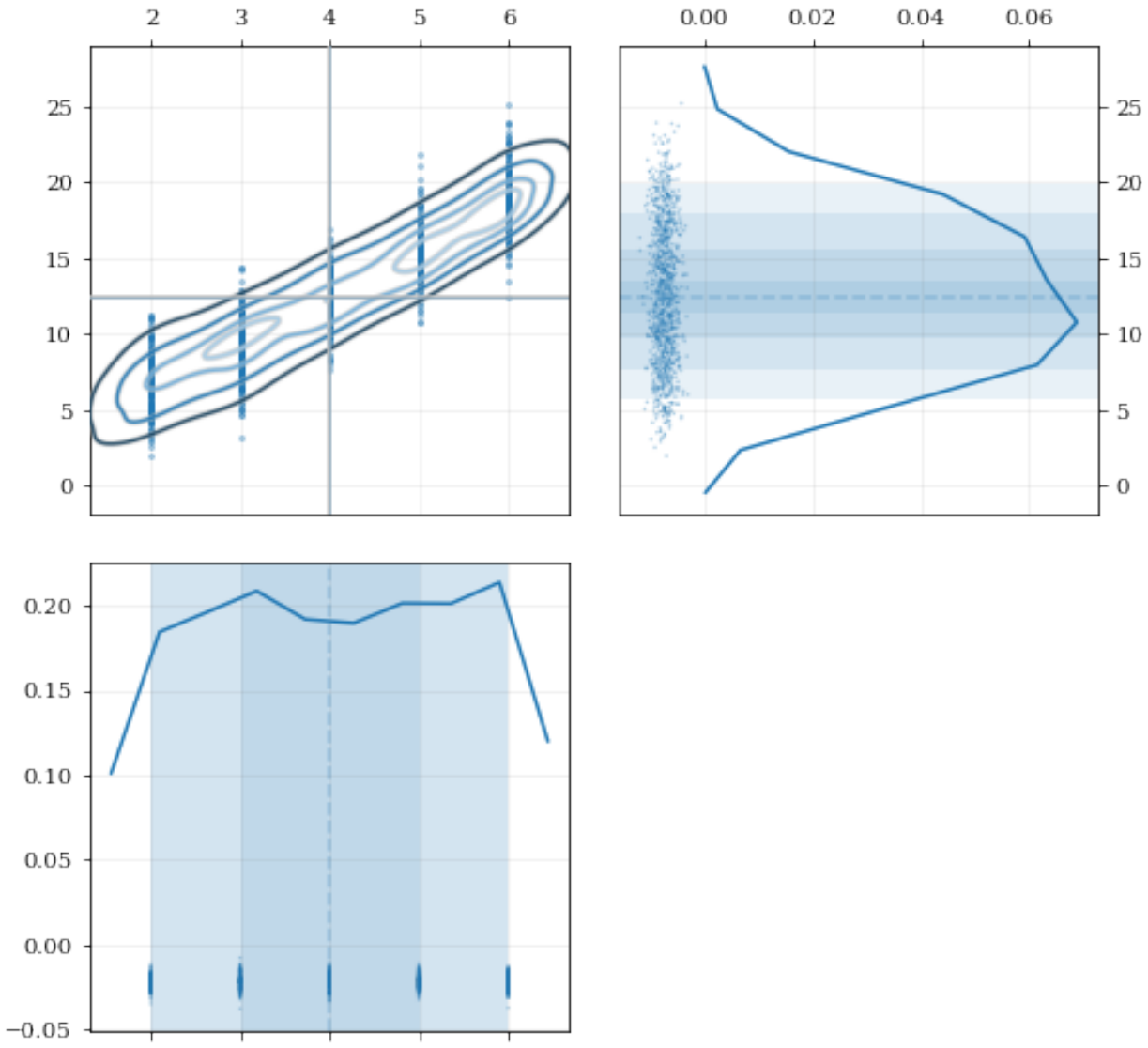
Using different data `weights`

```
# Load some random data (and the 'true' PDF, for comparison)
data, truth = kale.utils._random_data_1d_01()

# ---- Resample the same data, using different weightings ---- #
resamp_uni = kale.resample(data, size=1000)                      #
resamp_sqr  = kale.resample(data, weights=data**2, size=1000)     #
resamp_inv = kale.resample(data, weights=data**-1, size=1000)    #
# ------------------------------------------------------------ #


# ---- Plot different distributions ----

# Setup plotting parameters
kw = dict(density=True, histtype='step', lw=2.0, alpha=0.75, bins='auto')

xx, yy = truth
samples = [resamp_inv, resamp_uni, resamp_sqr]
yvals = [yy/xx, yy, yy*xx**2/10]
labels = [r'$\propto X^{-1}$', r'$\propto 1$', r'$\propto X^2$']

plt.figure(figsize=[10, 5])

for ii, (res, yy, lab) in enumerate(zip(samples, yvals, labels)):
    hh, = plt.plot(xx, yy, ls='--', alpha=0.5, lw=2.0)
    col = hh.get_color()
    kale.carpet(res, color=col, shift=-0.1*ii)
    plt.hist(res, color=col, label=lab, **kw)

plt.gca().set(xlim=[-0.5, 6.5])
# Add legend
plt.legend()
# display the figure if this is a notebook
nbshow()
```



Resampling while 'keeping' certain parameters/dimensions

```
# Construct covariant 2D dataset where the 0th parameter takes on discrete values
xx = np.random.randint(2, 7, 1000)
yy = np.random.normal(4, 2, xx.size) + xx**(3/2)
data = [xx, yy]

# 2D plotting settings: disable the 2D histogram & disable masking of dense scatter-
↪points
dist2d = dict(hist=False, mask_dense=False)

# Draw a corner plot
kale.corner(data, dist2d=dist2d)

nbshow()
```



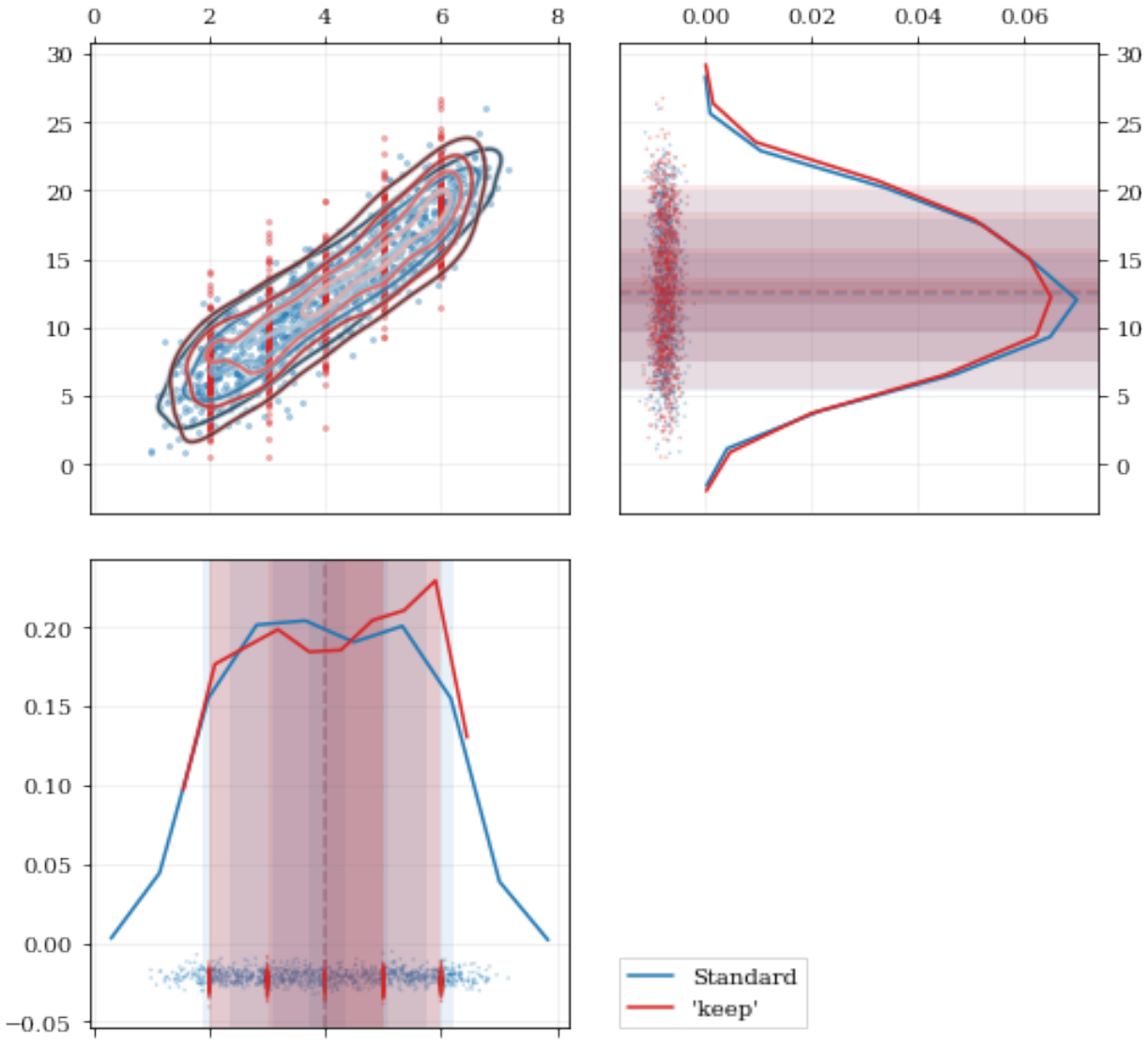
A standard KDE resampling will smooth out the discrete variables, creating a smooth(er) distribution. Using the `keep` parameter, we can choose to resample from the actual data values of that parameter instead of resampling with 'smoothing' based on the KDE.

```
kde = kale.KDE(data)

# ---- Resample the data both normally, and 'keep'ing the 0th parameter values ---- #
resamp_stnd = kde.resample()                                                        #
resamp_keep = kde.resample(keep=0)                                                  #
# ------------------------------------------------------------------------------- #

corner = kale.Corner(2)
dist2d['median'] = False    # disable median 'cross-hairs'
h1 = corner.plot(resamp_stnd, dist2d=dist2d)
h2 = corner.plot(resamp_keep, dist2d=dist2d)

corner.legend([h1, h2], ['Standard', "'keep'"])
```

(continues on next page)

(continued from previous page)

```
nbshow()
```



3.2 Plotting API

For more extended documentation, see the kalepy.plot submodule documentation.

```
import kalepy as kale
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
```

3.2.1 Top Level Functions

See the below.

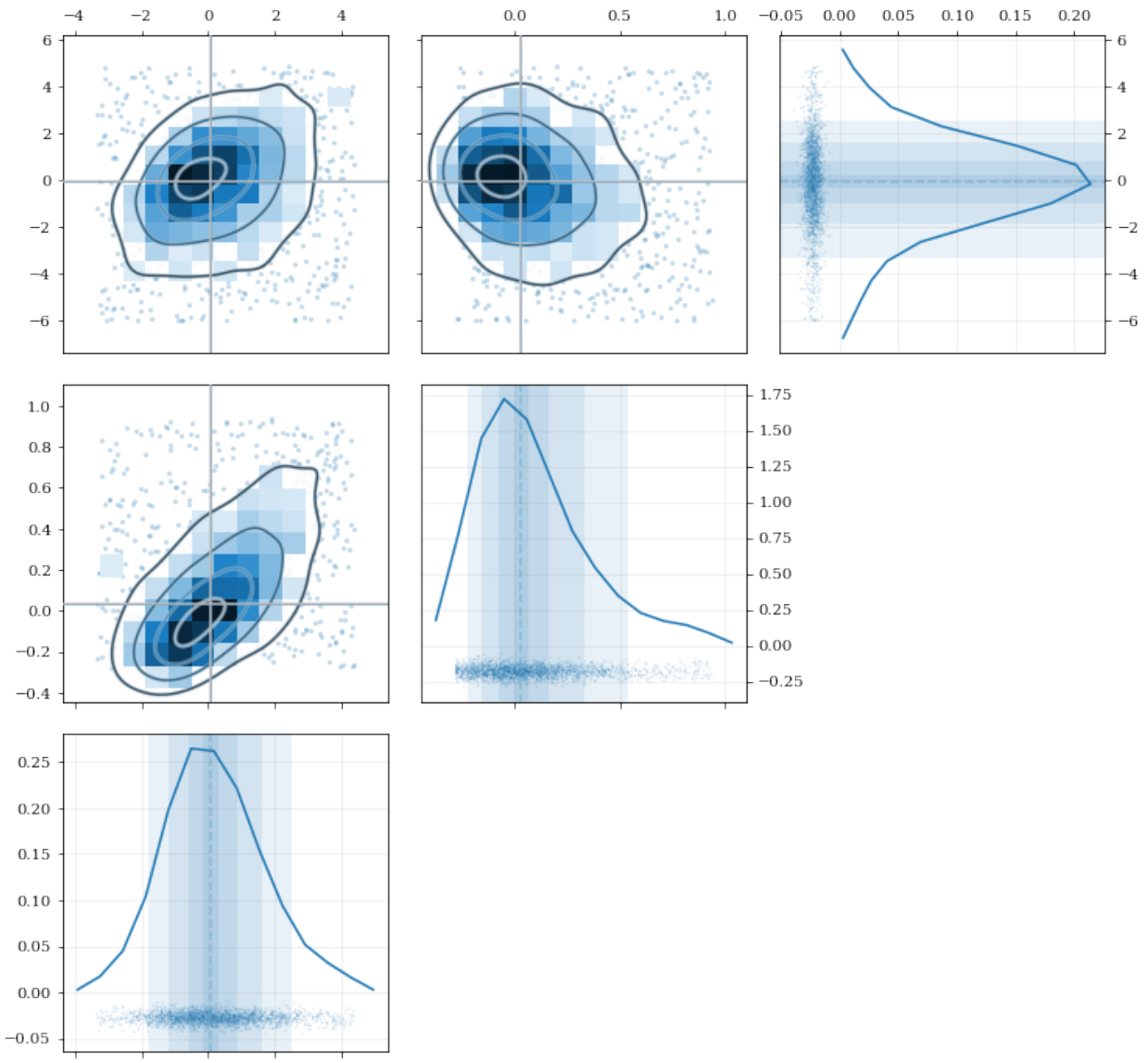
kalepy.corner() and the kalepy.Corner class

For the full documentation, see:

- kalepy.plot.corner
- kalepy.plot.Corner
- kalepy.plot.Corner.plot

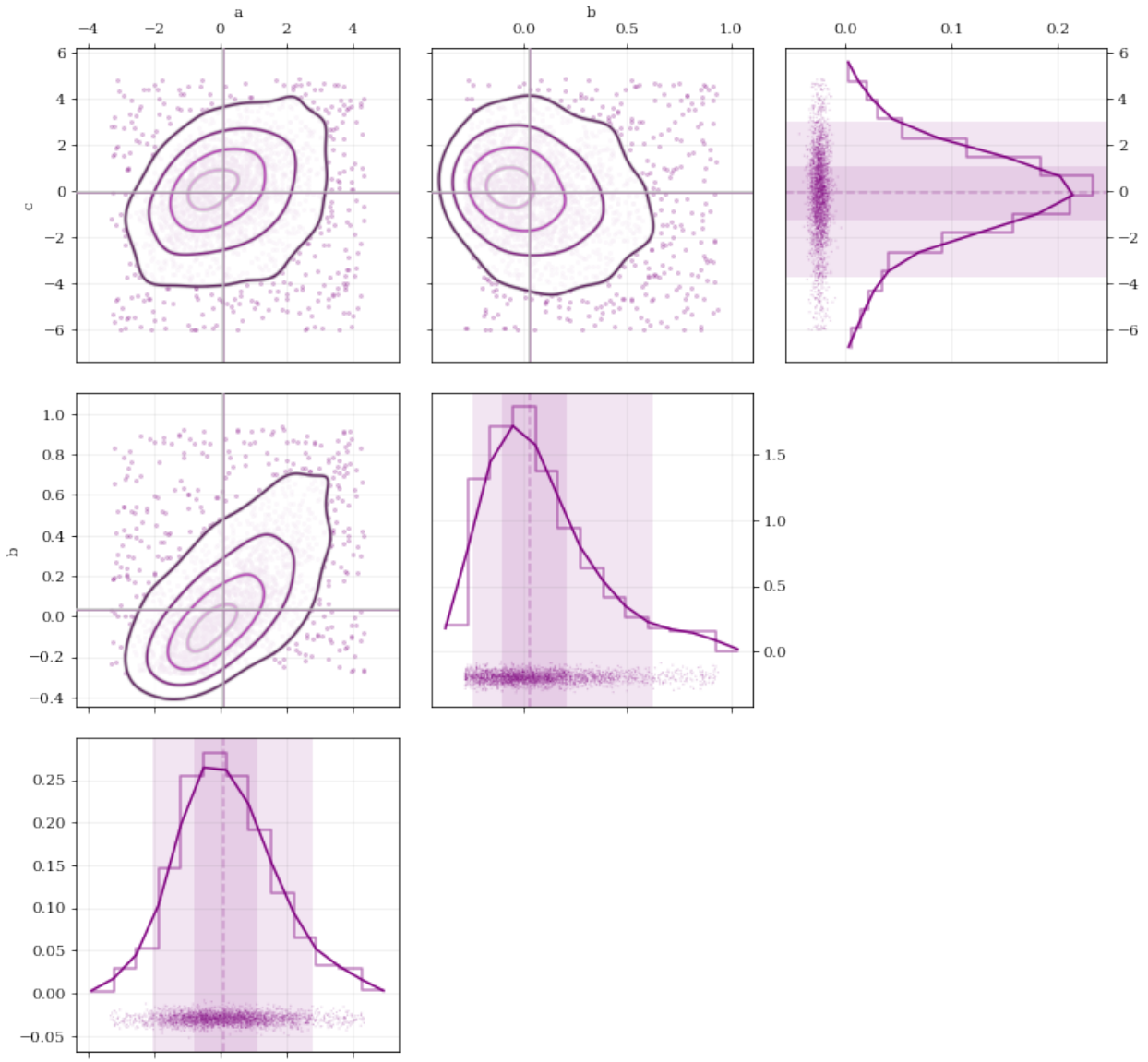
Plot some three-dimensional data called `data3` with shape (3, N) with `N` data points.

```
kale.corner(data3);
```



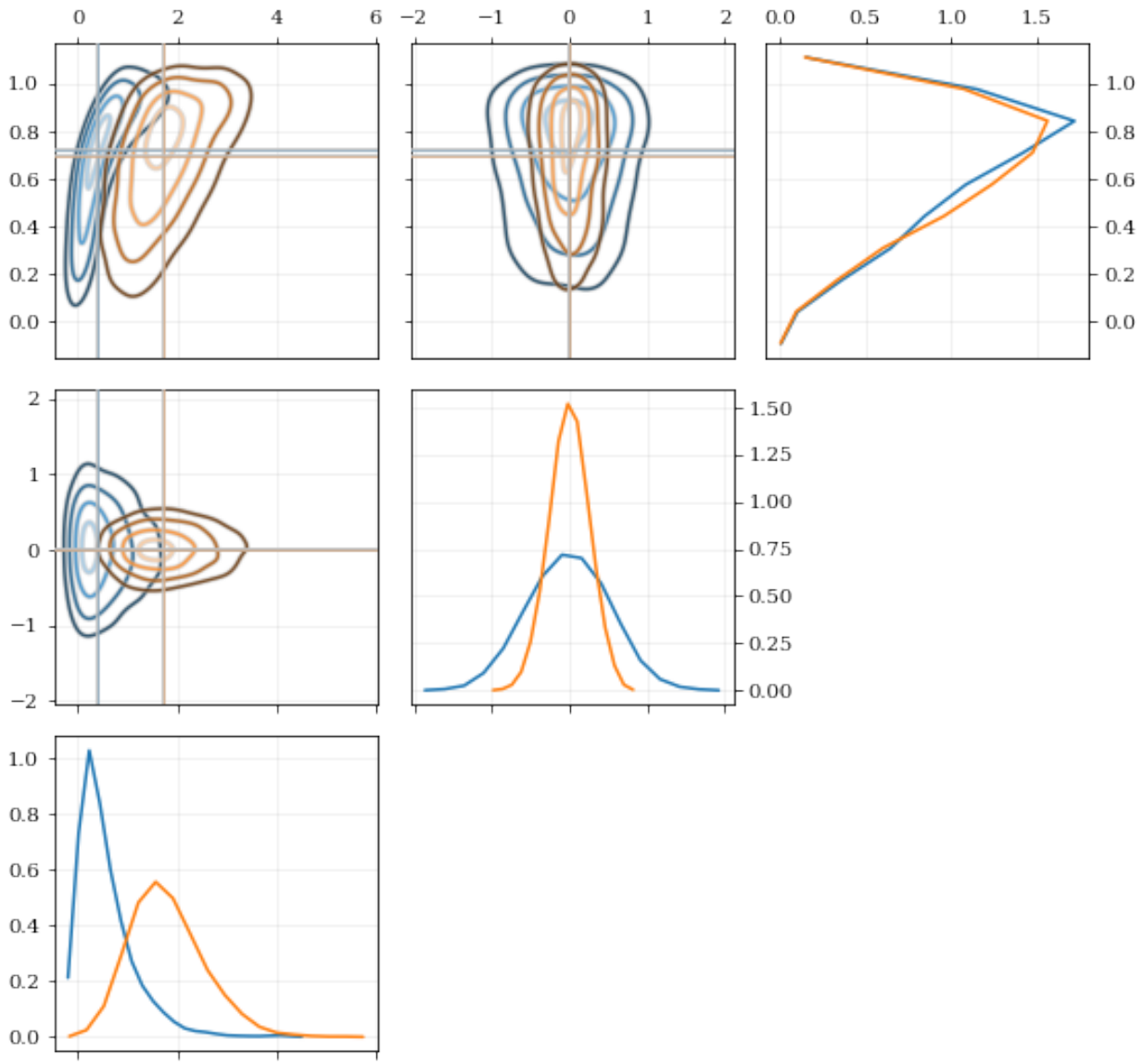
Extensive modifications are possible with passed arguments, for example:

```
# 1D plot settings: turn on histograms, and modify the confidence-interval quantiles
dist1d = dict(hist=True, quantiles=[0.5, 0.9])
# 2D plot settings: turn off the histograms, and turn on scatter
dist2d = dict(hist=False, scatter=True)

kale.corner(data3, labels=['a', 'b', 'c'], color='purple',
            dist1d=dist1d, dist2d=dist2d);
```



The `kalepy.corner` method is a wrapper that builds a `kalepy.Corner` instance, and then plots the given data. For additional flexibility, the `kalepy.Corner` class can be used directly. This is particularly useful for plotting multiple distributions, or using preconfigured plotting styles.

```
# Construct a `Corner` instance for 3 dimensional data, modify the figure size
corner = kale.Corner(3, figsize=[9, 9])

# Plot two different datasets using the `clean` plotting style
corner.clean(data3a)
corner.clean(data3b);
```



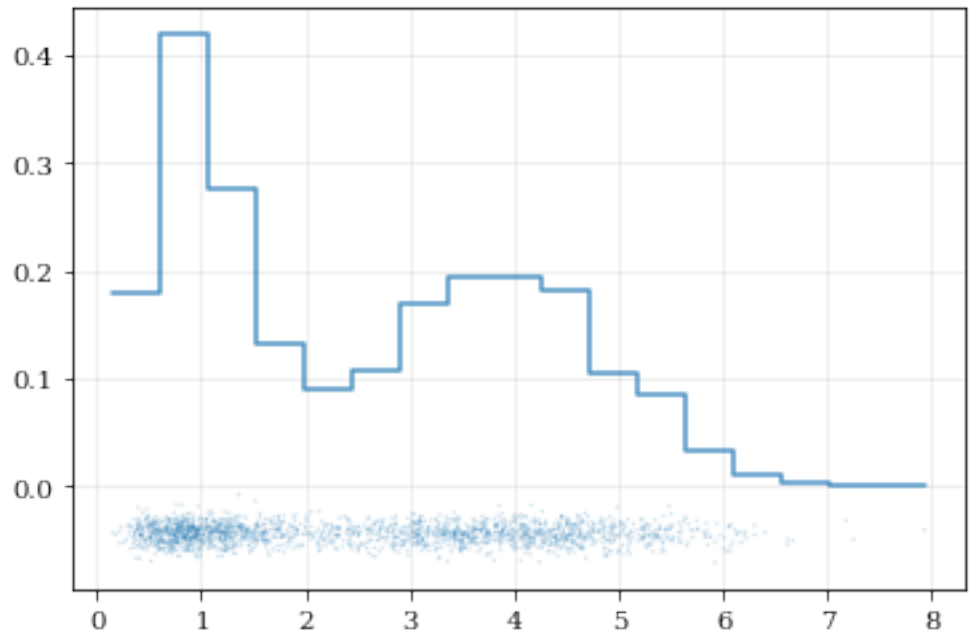
kalepy.dist1d and kalepy.dist2d

The `Corner` class ultimately calls the functions `dist1d` and `dist2d` to do the actual plotting of each figure panel. These functions can also be used directly.

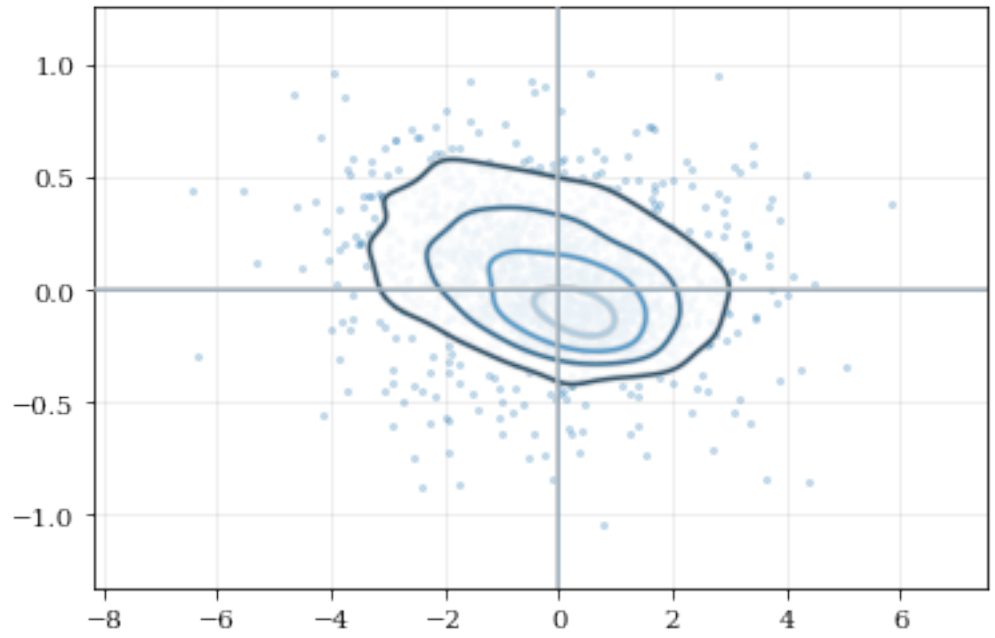
For the full documentation, see:

- kalepy.plot.dist1d
- kalepy.plot.dist2d

```
# Plot a 1D dataset, shape: (N,) for `N` data points
kale.dist1d(data1);
```




```
# Plot a 2D dataset, shape: (2, N) for `N` data points
kale.dist2d(data2, hist=False);
```



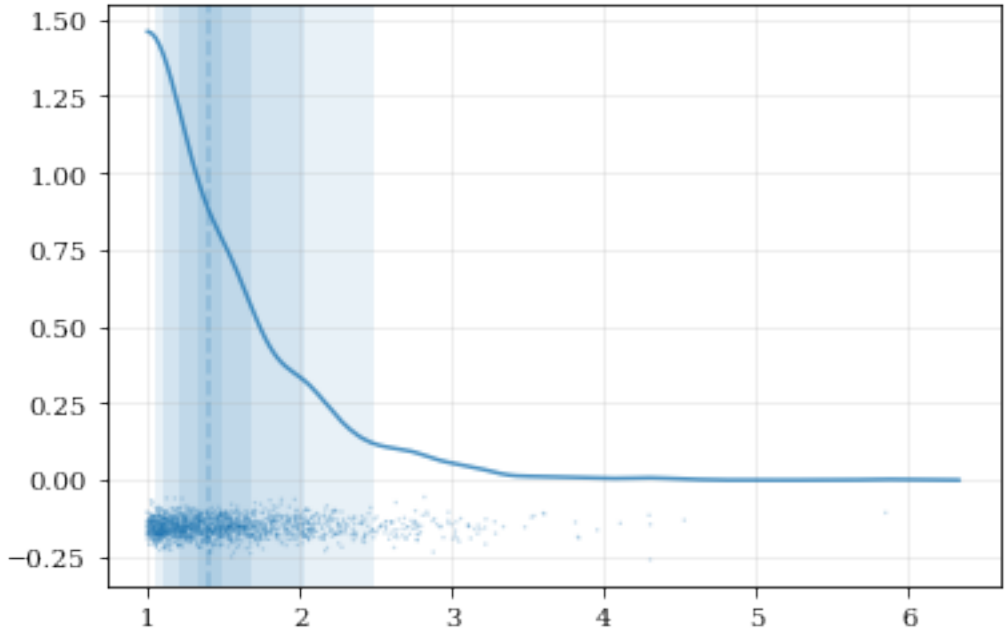
These functions can also be called on a `kalepy.KDE` instance, which is particularly useful for utilizing the advanced KDE functionality like reflection.

```
# Construct a random dataset, and truncate it on the left at 1.0
import numpy as np
data = np.random.lognormal(sigma=0.5, size=int(3e3))
data = data[data >= 1.0]
```

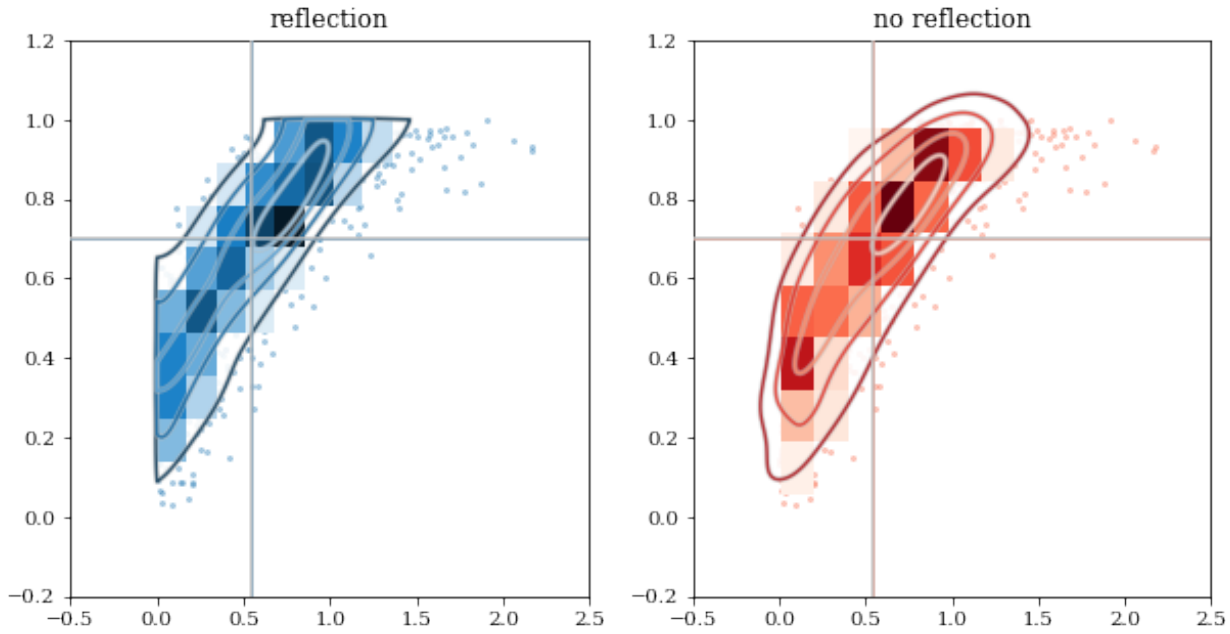
(continues on next page)

(continued from previous page)

```
# Construct a KDE, and include reflection (only on the lower/left side)
kde_reflect = kale.KDE(data, reflect=[1.0, None])
# plot, and include confidence intervals
hr = kale.dist1d(kde_reflect, confidence=True);
```




```
# Load a predefined 2D, 'random' dataset that includes boundaries on both dimensions
data = kale.utils._random_data_2d_03(num=1e3)
# Initialize figure
fig, axes = plt.subplots(figsize=[10, 5], ncols=2)

# Construct a KDE included reflection
kde = kale.KDE(data, reflect=[[0, None], [None, 1]])

# plot using KDE's included reflection parameters
kale.dist2d(kde, ax=axes[0]);

# plot data without reflection
kale.dist2d(data, ax=axes[1], cmap='Reds')

titles = ['reflection', 'no reflection']
for ax, title in zip(axes, titles):
    ax.set(xlim=[-0.5, 2.5], ylim=[-0.2, 1.2], title=title)
```



3.3 kalepy package

3.3.1 Subpackages

kalepy.tests package

Submodules

kalepy.tests.test_distributions module

class kalepy.tests.test_distributions.**Test_Distribution**

Bases: *Test_Distribution_Generic*

test_evaluate()

test_evaluate_nd()

class kalepy.tests.test_distributions.**Test_Distribution_Generic**

Bases: object

kalepy.tests.test_distributions.**test_get_distribution_class**()

kalepy.tests.test_kde module

class kalepy.tests.test_kde.**Test_KDE_Construct_From_Hist**

Bases: object

classmethod setup_class()

test_compare_1d()

test_compare_2d()

class kalepy.tests.test_kde.**Test_KDE_PDF**

Bases: object

compare_scipy_1d(*kernel*)

compare_scipy_2d(*kernel*)

pdf_params_fixed_bandwidth(*kernel*)

reflect_1d(*kernel*)

reflect_2d(*kernel*)

classmethod setup_class()

class kalepy.tests.test_kde.**Test_KDE_PDF_Box**

Bases: *Test_KDE_PDF*

test_pdf_params_fixed_bandwidth()

test_reflect_1d()

test_reflect_2d()

class kalepy.tests.test_kde.**Test_KDE_PDF_Gaussian**

Bases: *Test_KDE_PDF*

test_compare_scipy_1d()

test_compare_scipy_2d()

test_pdf_params_fixed_bandwidth()

test_reflect_1d()

test_reflect_2d()

class kalepy.tests.test_kde.**Test_KDE_Resample**

Bases: object

classmethod setup_class()

test_different_bws()

test_reflect_1d()

test_reflect_2d()

test_resample_keep_params_1()

kalepy.tests.test_kernels module

class kalepy.tests.test_kernels.**Test_Kernels_Generic**

Bases: object

classmethod kernel_at_dim(*kern*, *ndim*, *num=1000000.0*)

classmethod setup_class()

kalepy.tests.test_kernels.**test_check_reflect**()

kalepy.tests.test_kernels.**test_check_reflect_boolean**()

kalepy.tests.test_kernels.**test_kernels_evaluate**()

kalepy.tests.test_kernels.**test_kernels_evaluate_nd**()

kalepy.tests.test_sample module

class kalepy.tests.test_sample.**Test_Sample_Outliers**

Bases: object

classmethod setup_class()

class kalepy.tests.test_sample.**Test_Sampler_Grid**

Bases: object

classmethod setup_class()

test_shapes_invalid()

test_shapes_valid()

kalepy.tests.test_utils module

class kalepy.tests.test_utils.**Test_Bound_Indices**

Bases: object

test_1d()

class kalepy.tests.test_utils.**Test_Centroids**

Bases: object

test_general_1d()

Test a few 1D grids, randomly generated

test_general_2d()

Test a few 2D grids, randomly generated

test_general_3d()

Test a few 3D grids, randomly generated

test_single_1d()

Test a few 1D single-bins

test_single_2d()

Test a few 2D single-bins

class kalepy.tests.test_utils.Test_Cumsum

Bases: object

test_axis()

With axis given, this just needs to match *numpy.cumsum* results.

test_no_axis()

class kalepy.tests.test_utils.Test_Histogram

Bases: object

classmethod setup_class()

test_hist()

test_hist_dens()

test_hist_dens_prob()

test_hist_prob()

class kalepy.tests.test_utils.Test_IsIntegral

Bases: object

test_array()

test_int()

class kalepy.tests.test_utils.Test_Midpoints

Bases: object

classmethod setup_class()

test_midpoints_axes()

test_midpoints_lin()

test_midpoints_log()

test_midpoints_off_center()

class kalepy.tests.test_utils.Test_Really1D

Bases: object

test_0d_false()

test_1d_true()

test_2d_false()

class kalepy.tests.test_utils.Test_Spacing

Bases: object

classmethod setup_class()

test_lin()

test_log()

class kalepy.tests.test_utils.**Test_Trapz**

Bases: object

test_1d()

test_2d()

test_nd()

class kalepy.tests.test_utils.**Test_Trapz_Dens_To_Mass**

Bases: object

test_ndim()

test_ndim_a1()

test_ndim_a2()

Module contents

3.3.2 Submodules

kalepy.kde module

kalepy's top-level KDE class which provides all direct KDE functionality.

Contents:

- *KDE* : class for interfacing with KDEs and derived functionality.

class kalepy.kde.**KDE**(*dataset*, *bandwidth=None*, *weights=None*, *kernel=None*, *extrema=None*, *points=None*, *reflect=None*, *covariance=None*, *neff=None*, *diagonal=False*, *helper=True*, *bw_rescale=None*, ***kwargs*)

Bases: object

Core class and primary API for using *kalepy*, by constructin a KDE based on given data.

The *KDE* class acts as an API to the underlying *kernel* structures and methods. From the passed data, a 'bandwidth' is calculated and/or set (using optional specifications using the *bandwidth* argument). A *kernel* is constructed (using optional specifications in the *kernel* argument) which performs the calculations of the kernel density estimation.

Notes

Reflection

Reflective boundary conditions can be used to better reconstruct a PDF that is known to have finite support (i.e. boundaries outside of which the PDF should be zero).

The *pdf* and *resample* methods accept the keyword-argument (kwarg) *reflect* to specify that a reflecting boundary should be used.

reflect
[(D,) array_like, None] Locations at which reflecting boundary conditions should be imposed. For each dimension *D*, a pair of boundary locations (for: lower, upper) must be specified, or *None*. *None* can also be given to specify no boundary at that location.

If a pair of boundaries are given, then the first value corresponds to the lower boundary, and the second value to the upper boundary, in that dimension. If there should only be a single lower or upper boundary, then *None* should be passed as the other boundary value.

For example, *reflect=[None, [-1.0, 1.0], [0.0, None]]*, specifies that the 0th dimension has no boundaries, the 1st dimension has boundaries at both -1.0 and 1.0, and the 2nd dimension has a lower boundary at 0.0, and no upper boundary.

Projection / Marginalization

The PDF can be calculated for only particular parameters/dimensions. The *pdf* method accepts the keyword-argument (kwarg) *params* to specify particular parameters over which to calculate the PDF (i.e. the other parameters are projected over).

params
[int, array_like of int, None (default)] Only calculate the PDF for certain parameters (dimensions).

If *None*, then calculate PDF along all dimensions. If *params* is specified, then the target evaluation points *pnts*, must only contain the corresponding dimensions.

For example, if the *dataset* has shape (4, 100), but *pdf* is called with *params=(1, 2)*, then the *pnts* array should have shape *(2, M)* where the two provides dimensions correspond to the 1st and 2nd variables of the *dataset*.

TO-DO: add notes on *keep* parameter

Dynamic Range

When the elements of the covariace matrix between data variables differs by numerous orders of magnitude, the KDE values (especially marginalized values) can become spurious. One solution is to use a diagonal covariance matrix by initializing the KDE instance with *diagonal=True*. An alternative is to transform the input data in such a way that each variable's dynamic range becomes similar (e.g. taking the log of the values). A warning is given if the covariance matrix has a large dynamic very-large dynamic range, but no error is raised.

Examples

Construct semi-random data:

```
>>> import numpy as np
>>> np.random.seed(1234)
>>> data = np.random.normal(0.0, 1.0, 1000)
```

Construct *KDE* instance using this data, and the default bandwidth and kernels.

```
>>> import kalepy as kale
>>> kde = kale.KDE(data)
```

Compare original PDF and the data to the reconstructed PDF from the KDE:

```
>>> xx = np.linspace(-3, 3, 400)
>>> pdf_tru = np.exp(-xx*xx/2) / np.sqrt(2*np.pi)
>>> xx, pdf_kde = kde.density(xx, probability=True)
```

```
>>> import matplotlib.pyplot as plt
>>> ll = plt.plot(xx, pdf_tru, 'k--', label='Normal PDF')
>>> _, bins, _ = plt.hist(data, bins=14, density=True,
↪ color='0.5', rwidth=0.9, alpha=0.5, label='Data')
>>> ll = plt.plot(xx, pdf_kde, 'r-', label='KDE')
>>> ll = plt.legend()
```

Compare the KDE reconstructed PDF to the true PDF, make sure the chi-squared is consistent:

```
>>> dof = xx.size - 1
>>> x2 = np.sum(np.square(pdf_kde - pdf_tru)/pdf_tru**2)
>>> x2 = x2 / dof
>>> x2 < 0.1
True
>>> print("Chi-Squared: {:.1e}".format(x2))
Chi-Squared: 1.7e-02
```

Draw new samples from the data and make sure they are consistent with the original data:

```
>>> import scipy as sp
>>> samp = kde.resample()
>>> ll = plt.hist(samp, bins=bins, density=True, color='r', alpha=0.5, rwidth=0.5,
↪                     label='Samples')
>>> ks, pv = sp.stats.ks_2samp(data, samp)
>>> pv > 0.05
True
```

Initialize the *KDE* class with the given dataset and optional specifications.

Parameters

dataset
[array_like (N,) or (D,N,)] Dataset from which to construct the kernel-density-estimate.

bandwidth
[str, float, array of float, None [optional]] Specification for the bandwidth, or the method by which the bandwidth should be determined. If a *str* is given, it must match one of the standard bandwidth determination methods. If a *float* is given, it is used as the bandwidth in each dimension. If an array of *float* are given, then each value will be used as the bandwidth for the corresponding data dimension.

weights
[array_like (N,), None [optional]] Weights corresponding to each *dataset* point. Must match the number of points *N* in the *dataset*. If *None*, weights are uniformly set to 1.0 for each value.

kernel
[str, Distribution, None [optional]] The distribution function that should be used for the kernel. This can be a *str* specification that must match one of the existing distribution functions, or this can be a *Distribution* subclass itself that overrides the *_evaluate* method.

neff
[int, None [optional]] An effective number of datapoints. This is used in the plugin bandwidth

determination methods. If *None*, *neff* is calculated from the *weights* array. If *weights* are all uniform, then *neff* equals the number of datapoints *N*.

diagonal
[bool,] Whether the bandwidth/covariance matrix should be set as a diagonal matrix (i.e. without covariances between parameters). NOTE: see *KDE* docstrings, 'Dynamic Range'.

`__init__`(*dataset*, *bandwidth=None*, *weights=None*, *kernel=None*, *extrema=None*, *points=None*, *reflect=None*, *covariance=None*, *neff=None*, *diagonal=False*, *helper=True*, *bw_rescale=None*, ***kwargs*)

Initialize the *KDE* class with the given dataset and optional specifications.

Parameters

dataset
[array_like (N,) or (D,N,)] Dataset from which to construct the kernel-density-estimate.

bandwidth
[str, float, array of float, None [optional]] Specification for the bandwidth, or the method by which the bandwidth should be determined. If a *str* is given, it must match one of the standard bandwidth determination methods. If a *float* is given, it is used as the bandwidth in each dimension. If an array of *float* are given, then each value will be used as the bandwidth for the corresponding data dimension.

weights
[array_like (N,), None [optional]] Weights corresponding to each *dataset* point. Must match the number of points *N* in the *dataset*. If *None*, weights are uniformly set to 1.0 for each value.

kernel
[str, Distribution, None [optional]] The distribution function that should be used for the kernel. This can be a *str* specification that must match one of the existing distribution functions, or this can be a *Distribution* subclass itself that overrides the *_evaluate* method.

neff
[int, None [optional]] An effective number of datapoints. This is used in the plugin bandwidth determination methods. If *None*, *neff* is calculated from the *weights* array. If *weights* are all uniform, then *neff* equals the number of datapoints *N*.

diagonal
[bool,] Whether the bandwidth/covariance matrix should be set as a diagonal matrix (i.e. without covariances between parameters). NOTE: see *KDE* docstrings, 'Dynamic Range'.

`property bandwidth`

`cdf`(*pnts*, *params=None*, *reflect=None*)

Cumulative Distribution Function based on KDE smoothed data.

Parameters

pnts
[([D,]N,) array_like of scalar] Target evaluation points

Returns

cdf
[(N,) ndarray of scalar] CDF Values at the target points

`property covariance`

property dataset

density(*points=None*, *reflect=None*, *params=None*, *grid=False*, *probability=False*)

Evaluate the KDE distribution at the given data-points.

This method acts as an API to the *Kernel.pdf* method for this instance's *kernel*.

Parameters

points
[([D,]M,) array_like of float, or (D,) set of array_like point specifications] The locations at which the PDF should be evaluated. The number of dimensions *D* must match that of the *dataset* that initialized this class' instance. NOTE: If the *params* kwarg (see below) is given, then only those dimensions of the target parameters should be specified in *points*. The meaning of *points* depends on the value of the *grid* argument:

- *grid=True* : *points* must be a set of (D,) array_like objects which each give the evaluation points for the corresponding dimension to produce a grid of values. For example, for a 2D dataset, *points=([0.1, 0.2, 0.3], [1, 2])*, would produce a grid of points with shape (3, 2): *[[0.1, 1], [0.1, 2]], [[0.2, 1], [0.2, 2]], [[0.3, 1], [0.3, 2]]*, and the returned values would be an array of the same shape (3, 2).
- *grid=False* : *points* must be an array_like (D,M) describing the position of *M* sample points in each of *D* dimensions. For example, for a 3D dataset: *points=([0.1, 0.2], [1.0, 2.0], [10, 20])*, describes 2 sample points at the 3D locations, *(0.1, 1.0, 10)* and *(0.2, 2.0, 20)*, and the returned values would be an array of shape (2,).

reflect
[(D,) array_like, None] Locations at which reflecting boundary conditions should be imposed. For each dimension *D* (matching the input data), a pair of boundary locations (lower, upper) must be specified, or *None*. *None* can also be given as one of the two locations, to specify no boundary at that location. If the data is one-dimensional (D=1), then *reflect* may be shaped as (2,). See class docstrings:*Reflection* for more information.

params
[int, array_like of int, None] Only calculate the PDF for certain parameters (dimensions). See class docstrings:*Projection* for more information.

grid
[bool,] Evaluate the KDE distribution at a grid of points specified by *points*. See *points* argument description above.

probability
[bool, normalize the results to sum to unity]

Returns

points
[array_like of scalar] Locations at which the PDF is evaluated.

vals
[array_like of scalar] PDF evaluated at the given points

property extrema

classmethod from_hist(*bins*, *hist*, *bandwidth='bin width'*, **args*, ***kwargs*)

Alternative constructor using a histogram as input instead of individual data points.

Parameters

bins
[([D,]N,) array_like of scalar] Histogram bins. If using multiple dimensions N can be different for different dimensions.

hist
[(N,[N,...]) array_like of scalar] Histogram to construct KDE from. If in multiple dimensions dimensions can have different N.

bandwidth
[str or float] Bandwidth. Defaults to width of bin in each dimension. Accepts all arguments passed to bandwidth when constructed using __init__.

***args, **kwargs**
[tuple, dict] Arguments passed to __init__ constructor.

Returns

kde
[instance of KDE] Initialized KDE instance.

property kernel

property ndata

property ndim

property neff

pdf(**args*, ***kwargs*)

property points

property reflect

resample(*size=None*, *keep=None*, *reflect=None*, *squeeze=True*)

Draw new values from the kernel-density-estimate calculated PDF.

The KDE calculates a PDF from the given dataset. This method draws new, semi-random data points from that PDF.

Parameters

size
[int, None (default)] The number of new data points to draw. If *None*, then the number of *datapoints* is used.

keep
[int, array_like of int, None (default)] Parameters/dimensions where the original data-values should be drawn from, instead of from the reconstructed PDF. TODO: add more information.

reflect
[(D,) array_like, None (default)] Locations at which reflecting boundary conditions should be imposed. For each dimension *D*, a pair of boundary locations (for: lower, upper) must be specified, or *None*. *None* can also be given to specify no boundary at that location.

squeeze
[bool, (default: True)] If the number of dimensions *D* is one, then return an array of shape (L,) instead of (1, L).

Returns

samples
[([D,]L) ndarray of float] Newly drawn samples from the PDF, where the number of points *L* is determined by the *size* argument. If *squeeze* is True (default), and the number of dimensions in the original dataset *D* is one, then the returned array will have shape (L,).

property weights

kalepy.kernels module

Kernal basis functions for KDE calculations, used by *kalepy.kde.KDE* class.

class kalepy.kernels.**Kernel**(*distribution*, *bandwidth*, *covariance*, *helper=False*, *chunk=100000.0*)

Bases: object

property FINITE

__init__(*distribution*, *bandwidth*, *covariance*, *helper=False*, *chunk=100000.0*)

property bandwidth

property covariance

density(*points*, *data*, *weights=None*, *reflect=None*, *params=None*)

Calculate the Density Function using this Kernel.

Parameters

points
[(D, N), 2darray of float,] *N* points at which to evaluate the density function over *D* parameters (dimensions). Locations must be specified for each dimension of the data, or for each of target *params* dimensions of the data.

property distribution

property matrix

property matrix_inv

property norm

resample(*data*, *weights=None*, *size=None*, *keep=None*, *reflect=None*, *squeeze=True*)

kalepy.kernels.**get_distribution_class**(*arg=None*)

kalepy.plot module

kalepy's plotting submodule

This submodule containts the *Corner* class, and all plotting methods. The *Corner* class, and additional API functions are imported into the base package namespace of *kalepy*, e.g. *kalepy.Corner* and *kalepy.carpet* access the *kalepy.plot.Corner* and *kalepy.plot.carpet* methods respectively.

Additional options and customization:

The core plotting routines, such as *draw_hist1d*, *draw_hist2d*, *draw_contour2d*, etc include a fairly large number of keyword arguments for customization. The top level API methods, such as *corner()* or *Corner.plot_data()* often do not provide access to all of those arguments, but additional customization is possible by using the drawing methods directly, and optionally subclassing the *Corner* class to provide additional or different functionality.

Plotting API

- *Corner* : class for corner/triangle/pair plots.
- *corner* : method which constructs a *Corner* instance and plots 1D and 2D distributions.
- *dist1d* : plot a 1D distribution with numerous possible elements (e.g. histogram, carpet, etc)
- *dist2d* : plot a 2D distribution with numerous possible elements (e.g. histogram, contours, etc)
- *carpet* : draw a 1D scatter-like plot to semi-quantitatively depict a distribution.
- *contour* : draw a 2D contour plot. A wrapper of additional functionality around *plt.contour*
- *confidence* : draw 1D confidence intervals using shaded bands.
- *hist1d* : draw a 1D histogram
- *hist2d* : draw a 2D histogram. A wrapper of additional functionality around *plt.pcolormesh*

class kalepy.plot.**Corner**(*kde_data*, *weights=None*, *origin='tl'*, *rotate=True*, *axes=None*, *labels=None*, *limits=None*, *ticks=None*, ***kwfig*)

Bases: object

Class for creating 'corner' / 'pair' plots of multidimensional covariances.

The *Corner* class acts as a constructor for a *matplotlib* figure and axes, and coordinates the plotting of 1D and 2D distributions. The *kalepy.plot.dist1d()* and *kalepy.plot.dist2d()* methods are used for plotting the distributions. The class methods provide wrappers, and default setting for those methods. The *Corner.plot* method is the standard plotting method with default parameters chosen for plotting a single, multidimensional dataset. For overplotting numerous datasets, the *Corner.clean* or *Corner.plot_data* methods are better.

Examples

Load some predefined 3D data, and generate a default corner plot:

```
>>> import kalepy as kale
>>> data = kale.utils._random_data_3d_03()
>>> corner = kale.corner(data)
```

Load two different datasets, and overplot them using a *kalepy.Corner* instance.

```
>>> data1 = kale.utils._random_data_3d_03(par=[0.0, 0.5], cov=0.05)
>>> data2 = kale.utils._random_data_3d_03(par=[1.0, 0.25], cov=0.5)
>>> corner = kale.Corner(3)   # construct '3' dimensional corner-plot (i.e. 3x3 axes)
>>> _ = corner.clean(data1)
>>> _ = corner.clean(data2)
```

Methods

- `plot`	(the standard plotting method which, by default, includes both KDE and data elements.)
- `clean`	(minimal plots with only the KDE generated PDF in 1D and contours in 2D, by default.)
- `hist`	(minimal plots with only the data based 1D and 2D histograms, by default.)
- `plot_kde`	(plot elements with only KDE based info: the *clean* settings with a little more.)
- `plot_data`	(plot elements without using KDE info.)

Initialize Corner instance and construct figure and axes based on the given arguments.

Parameters

kde_data
[object, one of the following]

- int D, the number of parameters/dimensions to construct a DxD corner plot.
- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.

weights
[array_like scalar (N,) or None] The weights for each data point. NOTE: only applicable when *kde_data* is a (D,N) dataset.

labels
[array_like str (N,) of names for each parameters.]

limits
[None, or (D,2) of scalar] Specification for the limits of each axes (for each of *D* parameters): * None : the limits are determined automatically, * (D,2) : limits for each axis

rotate
[bool,] Whether or not the bottom-right-most axes should be rotated.

****kwfig**
[keyword-arguments passed to *_figax()* for constructing figure and axes.] See *kalepy.plot._figax()* for specifications.

`__init__`(*kde_data*, *weights=None*, *origin='tl'*, *rotate=True*, *axes=None*, *labels=None*, *limits=None*, *ticks=None*, ***kwfig*)

Initialize Corner instance and construct figure and axes based on the given arguments.

Parameters

kde_data
[object, one of the following]

- int D, the number of parameters/dimensions to construct a DxD corner plot.
- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.

weights
[array_like scalar (N,) or None] The weights for each data point. NOTE: only applicable when *kde_data* is a (D,N) dataset.

labels
[array_like str (N,) of names for each parameters.]

limits
[None, or (D,2) of scalar] Specification for the limits of each axes (for each of *D* parameters): * None : the limits are determined automatically, * (D,2) : limits for each axis

rotate
[bool,] Whether or not the bottom-right-most axes should be rotated.

****kwfig**
[keyword-arguments passed to *_figax()* for constructing figure and axes.] See *kalepy.plot._figax()* for specifications.

`clean`(*kde_data=None*, *weights=None*, *dist1d={}*, *dist2d={}*, ***kwargs*)

Wrapper for *plot_kde* that sets parameters for minimalism: PDF and contours only.

Parameters

kde_data
[*kalepy.KDE* instance, (D,N) array_like of scalars, or None]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.
- *None* : use the KDE/data stored during class initialization. raises *ValueError* if no KDE/data was provided

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

dist1d
[dict of keyword-arguments passed to the *kale.plot.dist1d* method.]

dist2d
[dict of keyword-arguments passed to the *kale.plot.dist2d* method.]

****kwargs**
[additiona keyword-arguments passed directly to *Corner.plot_kde*.]

`hist`(*kde_data=None*, *weights=None*, *dist1d={}*, *dist2d={}*, ***kwargs*)

Wrapper for *plot_data* that sets parameters to only plot 1D and 2D histograms of data.

Parameters

kde_data
[*kalepy.KDE* instance, (D,N) array_like of scalars, or None]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.
- *None* : use the KDE/data stored during class initialization. raises *ValueError* if no KDE/data was provided

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

dist1d
[dict of keyword-arguments passed to the *kale.plot.dist1d* method.]

dist2d
[dict of keyword-arguments passed to the *kale.plot.dist2d* method.]

****kwargs**
[additiona keyword-arguments passed directly to *Corner.plot_kde*.]

legend(*handles*, *labels*, *index=None*, *loc=None*, *fancybox=False*, *borderaxespad=0*, ***kwargs*)

plot(*kde_data=None*, *edges=None*, *weights=None*, *quantiles=None*, *limit=None*, *color=None*, *cmap=None*, *dist1d={}*, *dist2d={}*)

Plot with standard settings for plotting a single, multidimensional dataset or KDE.

This function coordinates the drawing of a corner plot that ultimately uses the *kalepy.plot.dist1d* and *kalepy.plot.dist2d* methods to draw parameter distributions using an instance of *kalepy.kde.KDE*.

Parameters

kde_data
[*kalepy.KDE* instance, (D,N) array_like of scalars, or *None*]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.
- *None* : use the KDE/data stored during class initialization. raises *ValueError* if no KDE/data was provided

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins for all dimensions, locations calculated automatically
- (D,) array_like of int : the number of bins for each of *D* dimensions
- (D,) of array_like : the bin-edge locations for each of *D* dimensions, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension.
- (X,) array_like of scalar : the bin-edge locations to be used for all dimensions
- *None* : the number and locations of bins are calculated automatically for each dim

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

quantiles
[*None* or array_like of scalar values in [0.0, 1.0] denoting the fractions of] data to demarkate with contours and confidence bands.

limit
[bool or *None*, whether the axes limits should be reset based on the plotted data.] If *None*, then the limits will be readjusted unless *limits* were provided on class initialization.

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.]

If *None*:

- *cmap* is given, then the color will be set to the *cmap* midpoint.
- *cmap* is not given, then the color will be determined by the next value of the default matplotlib color-cycle, and *cmap* will be set to a matching colormap.

This parameter effects the color of 1D: histograms, confidence intervals, and carpet; 2D: scatter points.

cmap
[matplotlib colormap specification, or *None*]

- All valid matplotlib specifications can be used, e.g. named value (like 'Reds' or 'viridis') or a *matplotlib.colors.Colormap* instance.
- If *None* then a colormap is constructed based on the value of *color* (see above).

dist1d
[dict of keyword-arguments passed to the *kale.plot.dist1d* method.]

dist2d
[dict of keyword-arguments passed to the *kale.plot.dist2d* method.]

`plot_data`(*data=None*, *edges=None*, *weights=None*, *quantiles=None*, *limit=None*, *color=None*, *cmap=None*, *dist1d={}*, *dist2d={}*)

Plot with default settings to emphasize the given data (not KDE derived properties).

This function coordinates the drawing of a corner plot that ultimately uses the *kalepy.plot.dist1d* and *kalepy.plot.dist2d* methods to draw parameter distributions using an instance of *kalepy.kde.KDE*.

Parameters

data
[(D,N) array_like of scalars, *kalepy.KDE* instance, or None]

- array_like scalar (D,N) of data with *D* parameters and *N* data points.
- *None* : use the KDE/data stored during class initialization. raises *ValueError* if no KDE/data was provided
- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.

NOTE: if a *KDE* instance is given, or one was stored during initilization, then the
dataset is extracted from the instance.

edges
[object specifying historgam edge locations; or None]

- int : the number of bins for all dimensions, locations calculated automatically
- (D,) array_like of int : the number of bins for each of *D* dimensions
- (D,) of array_like : the bin-edge locations for each of *D* dimensions, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension.
- (X,) array_like of scalar : the bin-edge locations to be used for all dimensions
- *None* : the number and locations of bins are calculated automatically for each dim

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

quantiles
[*None* or array_like of scalar values in [0.0, 1.0] denoting the fractions of] data to demarkate with contours and confidence bands.

limit
[bool or *None*, whether the axes limits should be reset based on the plotted data.] If

None, then the limits will be readjusted unless *limits* were provided on class initialization.

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.]

If *None*:

- *cmap* is given, then the color will be set to the *cmap* midpoint.
- *cmap* is not given, then the color will be determined by the next value of the default matplotlib color-cycle, and *cmap* will be set to a matching colormap.

This parameter effects the color of 1D: histograms, confidence intervals, and carpet; 2D: scatter points.

cmap
[matplotlib colormap specification, or *None*]

- All valid matplotlib specifications can be used, e.g. named value (like 'Reds' or 'viridis') or a *matplotlib.colors.Colormap* instance.
- If *None* then a colormap is constructed based on the value of *color* (see above).

dist1d
[dict of keyword-arguments passed to the *kale.plot.dist1d* method.]

dist2d
[dict of keyword-arguments passed to the *kale.plot.dist2d* method.]

`plot_kde`(*kde=None*, *edges=None*, *weights=None*, *quantiles=None*, *limit=None*, *ls='-'*, *color=None*, *cmap=None*, *dist1d={}*, *dist2d={}*)

Plot with default settings to emphasize the KDE derived distributions.

This function coordinates the drawing of a corner plot that ultimately uses the *kalepy.plot.dist1d* and *kalepy.plot.dist2d* methods to draw parameter distributions using an instance of *kalepy.kde.KDE*.

Parameters

kde
[*kalepy.KDE* instance, (D,N) array_like of scalars, or None]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.
- *None* : use the KDE/data stored during class initialization. raises *ValueError* if no KDE/data was provided

edges
[object specifying historgam edge locations; or None]

- int : the number of bins for all dimensions, locations calculated automatically
- (D,) array_like of int : the number of bins for each of *D* dimensions
- (D,) of array_like : the bin-edge locations for each of *D* dimensions, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension.
- (X,) array_like of scalar : the bin-edge locations to be used for all dimensions
- *None* : the number and locations of bins are calculated automatically for each dim

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde* argument is a (D,N) array_like of scalar data from which a *KDE* instance is created.

quantiles
[*None* or array_like of scalar values in [0.0, 1.0] denoting the fractions of] data to demarkate with contours and confidence bands.

limit
[bool or *None*, whether the axes limits should be reset based on the plotted data.] If *None*, then the limits will be readjusted unless *limits* were provided on class initialization.

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.]

If *None*:

- *cmap* is given, then the color will be set to the *cmap* midpoint.
- *cmap* is not given, then the color will be determined by the next value of the default matplotlib color-cycle, and *cmap* will be set to a matching colormap.

This parameter effects the color of 1D: histograms, confidence intervals, and carpet; 2D: scatter points.

cmap
[matplotlib colormap specification, or *None*]

- All valid matplotlib specifications can be used, e.g. named value (like 'Reds' or 'viridis') or a *matplotlib.colors.Colormap* instance.
- If *None* then a colormap is constructed based on the value of *color* (see above).

dist1d
[dict of keyword-arguments passed to the *kale.plot.dist1d* method.]

dist2d
[dict of keyword-arguments passed to the *kale.plot.dist2d* method.]

target(*targets*, *upper_limits=None*, *lower_limits=None*, *lw=1.0*, *fill_alpha=0.1*, ***kwargs*)

kalepy.plot.**carpet**(*xx*, *weights=None*, *ax=None*, *ystd=None*, *yave=None*, *shift=0.0*, *limit=None*, *fancy=False*, *random='normal'*, *rotate=False*, ***kwargs*)

Draw a 'carpet plot' that shows semi-quantitatively the distribution of points.

The given data (*xx*) is plotted as scatter points, where the abscissa (typically x-values) are the actual locations of the data and the ordinate are generated randomly. The size and transparency of points are chosen based on the number of points. If *weights* are given, it the size of the data points are chosen proportionally.

NOTE: the *shift* argument determines the reference ordinate-value of the distribution, this is particularly useful when numerous datasets are being overplotted.

Parameters

xx
[(N,) array_like of scalar, the data values to be plotted]

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

ax
[*None* or *matplotlib.axis.Axis*, if *None* the *plt.gca()* is used]

ystd
[scalar or *None*, a measure of the dispersion in the ordinate scatter of values] If *None* then an appropriate value is guessed based *yave* or the axis limits

yave
[scalar or *None*, the baseline at which the ordinate values are generated,] This is very similar to the *shift* argument, determining the ordinate-offset, but in the case that *ystd* is not given but *yave* is given, then the *yave* value determines *ystd*.

shift
[scalar,] A systematic ordinate shift of all data-points, particularly useful when multiple datasets are being plotted, such that one carpet plot can be offset from the other(s).

limit
[int or *None*,] Maximum number of points to draw. If more data points are provided, a *limit* subset of them are chosen and plotted.

fancy
[bool,] *Experimental* resizing of data-points to visually emphasize outliers.

random
[str, one of ['normal', 'uniform'],] How the ordinate values are randomly generated: either a uniform or normal (i.e. Gaussian).

rotate
[bool, if True switch the x and y values such that x becomes the ordinate.]

kwargs
[additional keyword-arguments passed to *matplotlib.axes.Axes.scatter()*]

kalepy.plot.**confidence**(*data*, *ax=None*, *weights=None*, *quantiles=[0.5, 0.9]*, *median=True*, *rotate=False*, ***kwargs*)

Plot 1D Confidence intervals at the given quantiles.

For each quantile *q*, a shaded range is plotted that includes a fration *q* of data values around the median. Ultimately either *plt.axhspan* or *plt.axvspan* is used for drawing.

Parameters

data
[(N,) array_like of scalar, the data values around which to calculate confidence intervals]

ax
[*None* or *matplotlib.axes.Axes* instance, if *None* then *plt.gca()* is used.]

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

quantiles
[array_like of scalar values in [0.0, 1.0] denoting the fractions of data to mark.]

median
[bool, mark the location of the median value.]

rotate
[bool, if true switch the x and y coordinates (i.e. rotate plot 90deg clockwise).]

****kwargs**
[additional keyword-arguments passed to *plt.axhspan* or *plt.axvspan*.]

kalepy.plot.**contour**(*data*, *edges=None*, *ax=None*, *weights=None*, *color=None*, *cmap=None*, *quantiles=None*, *smooth=1.0*, *upsample=2*, *pad=1*, ***kwargs*)

Calculate and draw 2D contours.

This is a wrapper for *draw_contour*, which in turn wraps *plt.contour*. This function constructs bin-edges and calculates the histogram from which the contours are calculated.

Parameters

data
[(2, N) array_like of scalars,] The data from which contours should be calculated.

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins for both dimensions, locations calculated automatically
- (2,) array_like of int : the number of bins for each dimension.
- (2,) of array_like : the bin-edge locations for each dimension, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension: i.e. 6 total.
- (X,) array_like of scalar : the bin-edge locations to be used for both dimensions.
- *None* : the number and locations of bins are calculated automatically.

ax
[*matplotlib.axes.Axes* instance, or *None*; if *None* then *plt.gca()* is used.]

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.]

If *None*:

- *cmap* is given, then the color will be set to the *cmap* midpoint.
- *cmap* is not given, then the color will be determined by the next value of the default matplotlib color-cycle, and *cmap* will be set to a matching colormap.

This parameter effects the color of 1D: histograms, confidence intervals, and carpet; 2D: scatter points.

cmap
[matplotlib colormap specification, color or list of colors, or *None*]

- All valid matplotlib specifications can be used, e.g. named value (like 'Reds' or 'viridis') or a *matplotlib.colors.Colormap* instance.
- color or list of colors: *matplotlib.colors.ListedColormap* is constructed.
- If *None* then a colormap is constructed based on the value of *color* (see above).

quantiles
[*None* or array_like of scalar values in [0.0, 1.0] denoting the fractions of] data to demarkate with contours and confidence bands.

smooth
[scalar or *None*/*False*,] if scalar: The width, in histogram bins, of a gaussian smoothing filter if *None* or *False*: no smoothing.

upsample
[int or *None*/*False*,] if int: the factor by which to upsample the histogram by interpolation. if *None* or *False*: no upsampling

pad
[int, True, or *None*/*False*,] if int: the number of edge bins added to the histogram to close contours hitting the edges if true: the default padding size is used if *None* or *False*: no padding is used.

****kwargs**
[additiona keyword-arguments passed to *kalepy.plot.draw_contour2d()*.]

kalepy.plot.**corner**(*kde_data*, *labels=None*, *kwcorner={}*, ***kwplot*)

Simple wrapper function to construct a *Corner* instance and plot the given data.

See *kalepy.plot.Corner* and *kalepy.plot.Corner.plot* for more information.

Parameters

kde_data
[*kalepy.KDE* instance, or (D,N) array_like of scalars]

- **instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted.** In this case the *param* argument selects which dimension/parameter is plotted if numerous are included in the *KDE*.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.

labels
[*None* or (D,) array_like of str, names of each parameter being plotted.]

kwcorner
[dict, keyword-arguments passed to *Corner* constructor.]

****kwplot**
[additional keyword-arguments passed to *Corner.plot* method.]

kalepy.plot.**dist1d**(*kde_data*, *ax=None*, *edges=None*, *weights=None*, *probability=True*, *param=0*, *rotate=False*, *density=None*, *confidence=False*, *hist=None*, *carpet=True*, *color=None*, *quantiles=None*, *ls=None*, *alpha=None*, ***kwargs*)

Draw 1D data distributions with numerous possible components.

The components of the plot are controlled by the arguments: * *density* : a KDE distribution curve, * *confidence* : 1D confidence bands calculated from a KDE, * *hist* : 1D histogram from the provided data, * *carpet* : 'carpet plot' (see *kalepy.plot.carpet()*) showing the data as a scatter-like plot.

Parameters

kde_data
[*kalepy.KDE* instance, (D,N) array_like of scalars, or *None*]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted. In this case the *param* argument selects which dimension/parameter is plotted if numerous are included in the *KDE*.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.

ax
[*matplotlib.axes.Axes* instance, or *None*; if *None* then *plt.gca()* is used.]

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins, locations calculated automatically
- array_like : the bin-edge locations
- *None* : the number and locations of bins are calculated automatically

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

probability
[bool,] Whether distributions (*hist* and *density*) are normalized such that the sum is unity.

param
[int,] If a *KDE* instance is provided as the *kde_data* argument, and it includes multiple dimensions/parameters of data, then this argument determines which parameter is plotted.

rotate
[bool, if true switch the x and y coordinates (i.e. rotate plot 90deg clockwise).]

density
[bool or *None*, whether the density KDE distribution is plotted or not.] If *None* then this is set based on what is passed as the *kde_data*.

confidence
[bool, whether confidence intervals are plotted based on the KDE distribution,] intervals are placed according to the *quantiles* argument.

hist
[bool or *None*, whether a histogram is plotted from the given data.] If *None*, then the value is chosen based on the given *kde_data* argument.

carpet
[bool or number, whether or not a 'carpet plot' is shown from the given data.] If *carpet* is a number, it is the maximum number of points that are plotted.

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.] If *None* then the color will be determined by the next value of the default matplotlib color-cycle.

quantiles
[array_like of scalar values in [0.0, 1.0] denoting the fractions of data to mark.]

ls
[str or None, matplotlib linestyle specification]

****kwargs**
[additional keyword-arguments passed to *plt.plot* command when plotting 'density'] and 'hist' components.

kalepy.plot.**dist2d**(*kde_data*, *ax=None*, *edges=None*, *weights=None*, *params=[0, 1]*, *quantiles=None*, *sigmas=None*, *color=None*, *cmap=None*, *smooth=None*, *upsample=None*, *pad=True*, *ls='-'*, *outline=True*, *median=True*, *scatter=True*, *contour=True*, *hist=True*, *mask_dense=None*, *mask_below=True*, *mask_alpha=0.9*)

Draw 2D data distributions with numerous possible components.

The components of the plot are controlled by the arguments: * *median* : the median values of each coordinate in a 'cross-hairs' style, * *scatter* : 2D scatter points of the raw data, * *contour* : 2D contour plot from the KDE, * *hist* : 2D histogram of the raw data.

These components are modified by: * *mask_dense* : mask over scatter points within the outer-most contour interval, * *mask_below* : mask out (ignore) histogram bins below a certain value.

Parameters

kde_data
[*kalepy.KDE* instance, or (D,N) array_like of scalars]

- instance of *kalepy.kde.KDE*, providing the data and KDE to be plotted. In this case the *param* argument selects which dimension/parameter is plotted if numerous are included in the *KDE*.
- array_like scalar (D,N) of data with *D* parameters and *N* data points.

ax
[*matplotlib.axes.Axes* instance, or *None*; if *None* then *plt.gca()* is used.]

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins for both dimensions, locations calculated automatically
- (2,) array_like of int : the number of bins for each dimension.
- (2,) of array_like : the bin-edge locations for each dimension, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension: i.e. 6 total.
- (X,) array_like of scalar : the bin-edge locations to be used for both dimensions.
- *None* : the number and locations of bins are calculated automatically.

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

param
[(2,) array_like of int,] If a *KDE* instance is provided as the *kde_data* argument, and it includes multiple dimensions/parameters of data, then this argument determines which parameters are plotted.

quantiles
[array_like of scalar values in [0.0, 1.0] denoting the fractions of data to contour.]

sigmas
[array_like of positive scalar values denoting contour levels.]

color
[matplotlib color specification (i.e. named color, hex or rgb) or *None*.]

If *None*:

- *cmap* is given, then the color will be set to the *cmap* midpoint.
- *cmap* is not given, then the color will be determined by the next value of the default matplotlib color-cycle, and *cmap* will be set to a matching colormap.

This parameter effects the color of 1D: histograms, confidence intervals, and carpet; 2D: scatter points.

cmap
[matplotlib colormap specification, or *None*]

- All valid matplotlib specifications can be used, e.g. named value (like 'Reds' or 'viridis') or a *matplotlib.colors.Colormap* instance.

- If *None* then a colormap is constructed based on the value of *color* (see above).

smooth
[scalar or *None*/*False*, smoothing of plotted contours (*only*)] if scalar: The width, in histogram bins, of a gaussian smoothing filter if *None* or *False*: no smoothing.

upsample
[int or *None*/*False*, upsampling of plotted contours (*only*)] if int: the factor by which to upsample the histogram by interpolation. if *None* or *False*: no upsampling

pad
[int, True, or *None*/*False*,] if int: the number of edge bins added to the histogram to close contours hitting the edges if true: the default padding size is used if *None* or *False*: no padding is used.

ls
[str or *None*, matplotlib linestyle specification for 'contour' and 'mdedian' components.]

outline
[bool, add outline path effects to median and contour lines]

median
[bool, mark the location of the median values in both dimensions (cross-hairs style).]

scatter
[bool or number, whether to plot a 2D scatter of the data points.] If *scatter* is a number, it is the maximum number of scatter points plotted (including those that may be masked over). The *mask_dense* parameter determines if some of these points are masked over.

contour
[bool, whether or not contours are plotted at the given *quantiles*.]

hist
[bool, whether a 2D histogram is plotted from the given data.]

mask_dense
[bool, whether to mask over high-density scatter points (within the lowest contour).]

mask_below
[bool or scalar; whether, or the value below which, hist bins should be excluded.]

- If True : exclude histogram bins with less than the average weight of a data point. If *weights* are not given, this means exclude empty histogram bins.
- If False : do not exclude any bins (i.e. include all bins).
- If scalar : exclude histogram bins with values below the given value.

Notes

- There is no *probability* argument because the normalization of the 2D distributions currently has no effect.

kalepy.plot.**hist1d**(*data*, *edges=None*, *ax=None*, *weights=None*, *density=False*, *probability=False*, *renormalize=False*, *joints=True*, *positive=True*, *rotate=False*, ***kwargs*)

Calculate and draw a 1D histogram.

This is a thin wrapper around the *kalepy.plot.draw_hist1d()* method which draws a histogram that has already been computed (e.g. with *kalepy.utils.histogram* or *numpy.histogram*).

Parameters

data
[(N,) array_like of scalar, data to be histogrammed.]

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins, locations calculated automatically
- array_like : the bin-edge locations
- *None* : the number and locations of bins are calculated automatically

ax
[*matplotlib.axes.Axes* instance, or *None*; if *None* then *plt.gca()* is used.]

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

density
[bool or *None*, whether the density KDE distribution is plotted or not.] If *None* then this is set based on what is passed as the *kde_data*.

probability
[bool,] Whether distributions (*hist* and *density*) are normalized such that the sum is unity. NOTE: this can be overridden by the *renormalize* argument.

renormalize
[bool or scalar, whether or to what value to renormalize the histrogram maximum.] If True : renormalize the maximum histogram value to unity. If False : do not renormalize. If scalar : renormalize the histogram maximum to this value.

joints
[bool, plot the vertical connectors ('joints') between histogram bins; if False, only] horizontal lines are plotted for each bin.

positive
[bool, only plot bins with positive values.]

rotate
[bool, if true switch the x and y coordinates (i.e. rotate plot 90deg clockwise).]

****kwargs**
[additional keyword-arguments passed to *kalepy.plot.draw_hist1d()*.] Any arguments not caught by *draw_hist1d()* are eventually passed to *plt.plot()* method.

Notes

- TO-DO: Add *scipy.binned_statistic* functionality for arbitrary statistics beyond histgramming.

`kalepy.plot.`**`hist2d`**`(`*data*, *edges=None*, *ax=None*, *weights=None*, *mask_below=False*, ***kwargs*`)`

Calculate and draw a 2D histogram.

This is a thin wrapper around the *kalepy.plot.draw_hist2d()* method which draws a 2D histogram that has already been computed (e.g. with *numpy.histogram2d*).

Parameters

data
[(2, N) array_like of scalar, data to be histogrammed.]

edges
[object specifying historgam edge locations; or *None*]

- int : the number of bins for both dimensions, locations calculated automatically
- (2,) array_like of int : the number of bins for each dimension.
- (2,) of array_like : the bin-edge locations for each dimension, e.g. ([0, 1, 2], [0.0, 0.1, 0.2, 0.3],) would describe two bins for the 0th dimension, and 3 bins for the 1st dimension: i.e. 6 total.
- (X,) array_like of scalar : the bin-edge locations to be used for both dimensions.
- *None* : the number and locations of bins are calculated automatically.

ax
[*matplotlib.axes.Axes* instance, or *None*; if *None* then *plt.gca()* is used.]

weights
[*None* or (N,) array_like of scalar, the weighting of each data-point if and] only-if the given *kde_data* argument is a (D,N) array_like of scalar data.

mask_below
[bool or scalar; whether, or the value below which, hist bins should be excluded.]

- If True : exclude histogram bins with less than the average weight of a data point. If *weights* are not given, this means exclude empty histogram bins.
- If False : do not exclude any bins (i.e. include all bins).
- If scalar : exclude histogram bins with values below the given value.

****kwargs**
[additional keyword-arguments passed to *kalepy.plot.draw_hist2d()*.] Any arguments not caught by *draw_hist1d()* are eventually passed to *plt.pcolormesh()*.

Notes

- TO-DO: Add *scipy.binned_statistic* functionality for arbitrary statistics beyond histgramming.

kalepy.sample module

Perform sampling of distributions and functions.

class kalepy.sample.**Sample_Grid**(*edges*, *dens*, *mass=None*, *scalar_dens=None*, *scalar_mass=None*)

Bases: `object`

Sample from a given probability distribution evaluated on a regular grid.

The grid has probability densities (*dens*) evaluated at the grid edges, and probability masses (*mass*) corresponding to the centroid of each bin. The centroids are calculated from the edge positions, weighted by probability density. If *mass* is not given, it is calculated by integrating the densities over each bin (using the trapezoid rule).

Process for drawing 'N' samples from the distributon:

1) Using the masses of each bin, the CDF is calculated.
2) N random values are chosen, and the CDF is inverted to find which bin they correspond to. The CDF is flattened into 1D to accomodate any dimensionality of grid, and then the chosen bins are re-mapped to ND space.

3) Within each bin, the position of each drawn sample is chosen proportionally to the probability density, based on the density-gradient within each cell.

Initialize *Sample_Grid* with the given grid edges and probability distribution.

Parameters

edges
[array_like] Bin edges along each dimension.

dens
[array_like] Probability density evaluated at grid edges.

mass
[array_like or None] Probability mass (i.e. number of samples) for each bin. Evaluated at bin centers or centroids. If no *mass* is given, it is calculated by integrating *dens* over each bin using the trapezoid rule. See: *_init_data()*.

`__init__`(*edges*, *dens*, *mass=None*, *scalar_dens=None*, *scalar_mass=None*)

Initialize *Sample_Grid* with the given grid edges and probability distribution.

Parameters

edges
[array_like] Bin edges along each dimension.

dens
[array_like] Probability density evaluated at grid edges.

mass
[array_like or None] Probability mass (i.e. number of samples) for each bin. Evaluated at bin centers or centroids. If no *mass* is given, it is calculated by integrating *dens* over each bin using the trapezoid rule. See: *_init_data()*.

`property grid`

`sample`(*nsamp=None*, *interpolate=True*, *return_scalar=None*)

Sample from the probability distribution.

Parameters

nsamp
[scalar or None]

interpolate
[bool]

return_scalar
[bool]

Returns

vals
[(D, N) ndarray of scalar]

`class` kalepy.sample.`Sample_Outliers`(*edges*, *dens*, *threshold=10.0*, ***kwargs*)

Bases: *Sample_Grid*

Sample outliers from a given probability distribution evaluated on a regular grid.

"Outliers" are points in areas of low probability mass, which are drawn randomly. "Inliers" are bins with high probability mass, which are assumed to be well represented by the centroid of those bins. The *threshold* parameter determines the dividing point between low and high probability masses.

The grid has probability densities (*dens*) evaluated at the grid edges, and probability masses (*mass*) corresponding to the centroid of each bin. The centroids are calculated from the edge positions, weighted by probability density. If *mass* is not given, it is calculated by integrating the densities over each bin (using the trapezoid rule).

Process for drawing 'N' samples from the distributon: 1) Bins with 'low' probability density (i.e. *mass* < *threshold*) are sampled in the same way as the super-class *Sample_Grid*. These values are given a *weight* of 1.0. 2) Bins with 'high' probability density (*mass* > *threshold*), are all used (i.e. with no stochasticity), where the location of sample points is the bin centroid (i.e. grid points weighted by probability density), and the *weight* is the total bin mass.

Initialize *Sample_Grid* with the given grid edges and probability distribution.

Parameters

edges
[array_like] Bin edges along each dimension.

dens
[array_like] Probability density evaluated at grid edges.

mass
[array_like or None] Probability mass (i.e. number of samples) for each bin. Evaluated at bin centers or centroids. If no *mass* is given, it is calculated by integrating *dens* over each bin using the trapezoid rule. See: *_init_data()*.

__init__(*edges*, *dens*, *threshold=10.0*, ***kwargs*)

Initialize *Sample_Grid* with the given grid edges and probability distribution.

Parameters

edges
[array_like] Bin edges along each dimension.

dens
[array_like] Probability density evaluated at grid edges.

mass
[array_like or None] Probability mass (i.e. number of samples) for each bin. Evaluated at bin centers or centroids. If no *mass* is given, it is calculated by integrating *dens* over each bin using the trapezoid rule. See: *_init_data()*.

sample(*poisson_inside=False*, *poisson_outside=False*, ***kwargs*)

Outlier sample the distribution.

Parameters

poisson_inside
[bool,]

Returns

nsamp
[int]

vals
[(D, N) ndarray] Sampled values with *N* samples, and values for *D* dimensions.

weights
[(N,) ndarray] Weights of samples values.

kalepy.sample.**sample_grid**(*edges*, *dens*, *nsamp=None*, *mass=None*, *scalar_dens=None*, *scalar_mass=None*, *squeeze=None*, ***sample_kwargs*)

Draw samples following the given distribution.

Parameters

edges
[(D,) list/tuple of array_like,] Edges of the (parameter space) grid. For *D* dimensions, this is a list/tuple of D entries, where each entry is an array_like of scalars giving the grid-points along that dimension. For example, *edges=([x, y], [a, b, c])* is a (2x3) dim array with coordinates: *[(x,a), (x,b), (x,c)], [(y,a), (y,b), (y,c)]*.

dist
[(N1,…,ND) array_like of scalar,] Distribution values specified at either the grid edges, or grid centers. e.g. for the (2x3) example above, *dist* should be either (2,3) or (1, 2)

nsamp
[int or None] Number of samples to draw (floats are cast to integers).

scalar
[None, or array_like of scalar] Scalar values to associate with the given distribution. Can be specified at either grid-centers or grid-edges, but the latter will be averaged down to grid-center values.

sample_kwargs
[additional keyword-arguments, optional] Additional arguments passed to the *Sample_Grid.sample()* method.

Returns

vals
[(D, N) array of sample points,] Sample points drawn from the given distribution in *D*, number of points *N* is that specified by *nsamp* param.

[weights]
[(N,) array of weights, returned if *scalar* is given] Scalar factors for each sample point.

kalepy.sample.**sample_grid_proportional**(*edges*, *dens*, *portion*, *nsamp*, *mass=None*, ***sample_kwargs*)

kalepy.sample.**sample_outliers**(*edges*, *data*, *threshold*, *nsamp=None*, *mass=None*, ***sample_kwargs*)

Sample a PDF randomly in low-density regions, and with weighted points at high-densities.

Selects (semi-)random samples from the given PDF. In high-density regions, bin centroids are used as representative points and recieve a corresponding (large) weight. Low-density regions are sampled proportionally with actual (weight = one) points.

Parameters

edges
[list/tuple of array_like] An iterable containing the grid edges for each dimension of the space.

data
[ndarray] Array giving the PDF to sample.

threshold
[float] Threshold mass below which true-samples should be drawn. Representative (centroid) values will be chosen for bins above this threshold.

nsamp
[int, optional] Number of samples to draw.

mass
[ndarray, optional] Probability mass function determining the number of samples to draw in each bin.

Returns

vals
weights

kalepy.speed_test module

DEVELOPMENT: submodule for running speed-tests for optimization checking.

kalepy.speed_test.**main**()

kalepy.utils module

kalepy's internal, utility functions.

kalepy.utils.**add_cov**(*data*, *cov*)

kalepy.utils.**array_str**(*data*, *num=3*, *format=':.2e'*)

kalepy.utils.**bins**(**args*, ***kwargs*)

Calculate *np.linspace(*args)* and return also centers and widths.

Returns

xe
[(N,) bin edges]

xc
[(N-1,) bin centers]

dx
[(N-1,) bin widths]

kalepy.utils.**bound_indices**(*data*, *bounds*, *outside=False*)

Find the indices of the *data* array that are bounded by the given *bounds*.

If *outside* is True, then indices for values *outside* of the bounds are returned.

kalepy.utils.**centroids**(*edges*, *data*)

Calculate the centroids (centers of mass) of each cell in the given grid.

Parameters

edges
[(D,) array_like of array_like]

data
[(…) ndarray of scalar]

Returns

coms
[ndarray (D, …)]

kalepy.utils.**cov_keep_vars**(*matrix*, *keep*, *reflect=None*)

kalepy.utils.**cumsum**(*vals*, *axis=None*)

Perform a cumulative sum without flattening the input array.

See: https://stackoverflow.com/a/60647166/230468

Parameters

vals
[array_like of scalar] Input values to sum over.

axis
[None or int] Axis over which to perform the cumulative sum.

Returns

res
[ndarray of scalar] Same shape as input *vals*

kalepy.utils.**cumtrapz**(*pdf*, *edges*, *prepend=True*, *axis=None*)

Perform a cumulative integration using the trapezoid rule.

Parameters

pdf
[array_like of scalar] Input values (e.g. a PDF) to be integrated.

edges
[[D,] list of (array_like of scalar)] Edges defining bins along each dimension. This should be an array/list of edges for each of *D* dimensions.

prepend
[bool] Whether or not to prepend zero values along the integrated dimensions.

axis
[None or int] Axis/Dimension over which to integrate.

Returns

cdf
[ndarray of scalar] Values integrated over the desired axes. Shape: * If *prepend* is False, the shape of *cdf* will be one smaller than the input *pdf* * in all dimensions integrated over. * If *prepend* is True, the shape of *cdf* will match that of the input *pdf*.

kalepy.utils.**flatlen**(*arr*)

kalepy.utils.**flatten**(*arr*)

Flatten a ND array, whether jagged or not, into a 1D array.

kalepy.utils.**histogram**(*data*, *bins=None*, *weights=None*, *density=False*, *probability=False*)

kalepy.utils.**iqrange**(*data*, *log=False*, *weights=None*)

Calculate inter-quartile range of the given data.

kalepy.utils.**isinteger**(*val*, *iterable=True*)

Test whether the given variable is an integer (i.e. *numbers.integral* subclass).

Parameters

val
[object,] Variable to test.

iterable
[bool,] Allow argument to be an iterable.

Returns

bool
[whether or not the input is an integer, or integer iterable]

kalepy.utils.**isjagged**(*arr*)

Test if the given array is jagged.

kalepy.utils.**jshape**(*arr*, *level=0*, *printout=False*, *prepend=''*, *indent=' '*)

Print the complete shape (even if jagged) of the given array.

kalepy.utils.**matrix_invert**(*matrix*, *helper=True*)

kalepy.utils.**meshgrid**(**args*, *indexing='ij'*, ***kwargs*)

kalepy.utils.**midpoints**(*arr*, *log=False*, *axis=-1*, *squeeze=False*)

Return the midpoints between values in the given array.

If the given array is N-dimensional, midpoints are calculated from the last dimension.

Parameters

arr
[ndarray of scalars,] Input array.

log
[bool or None,] Find midpoints in log-space

axis
[int, sequence, or *None*,] The axis about which to find the midpoints. If *None*, find the midpoints along all axes. If a sequence (tuple, list, or array), take the midpoints along each specified axis.

Returns

mids
[ndarray of floats,] The midpoints of the input array. The resulting shape will be the same as the input array *arr*, except that *mids.shape[axis] == arr.shape[axis]-1*.

kalepy.utils.**minmax**(*data*, *positive=False*, *prev=None*, *stretch=None*, *log_stretch=None*, *limit=None*)

kalepy.utils.**parse_edges**(*data*, *edges=None*, *extrema=None*, *weights=None*, *params=None*, *nmin=5*, *nmax=1000*, *pad=None*, *refine=1.0*, *bw=None*)

kalepy.utils.**quantiles**(*values*, *percs=None*, *sigmas=None*, *weights=None*, *axis=None*, *values_sorted=False*)

Compute weighted quartiles.

Taken from *zcode.math.statistics* Based on @Alleo answer: http://stackoverflow.com/a/29677616/230468

Parameters

values: (N,)
input data

percs: (M,) scalar
Desired percentiles of the data, within range [0.0, 1.0].

weights: (N,) or None
Weighted for each input data point in *values*.

values_sorted: bool
If True, then input values are assumed to already be sorted.

Returns

percs
[(M,) float] Array of percentiles of the weighted input data.

kalepy.utils.**really1d**(*arr*)

Test whether an array_like is really 1D (i.e. not a jagged ND array).

Test whether the input array is uniformly one-dimensional, as apposed to (e.g.) a `ndim == 1` list or array of irregularly shaped sub-lists/sub-arrays. True for an empty list *[]*.

Parameters

arr
[array_like] Array to be tested.

Returns

bool
Whether *arr* is purely 1D.

kalepy.utils.**rem_cov**(*data*, *cov=None*)

kalepy.utils.**run_if**(*func*, *target*, **args*, *otherwise=None*, ***kwargs*)

kalepy.utils.**run_if_notebook**(*func*, **args*, *otherwise=None*, ***kwargs*)

kalepy.utils.**run_if_script**(*func*, **args*, *otherwise=None*, ***kwargs*)

kalepy.utils.**spacing**(*data*, *scale='log'*, *num=None*, *dex=10*, ***kwargs*)

kalepy.utils.**stats**(*data*, *shape=True*, *sample=3*, *stats=True*)

kalepy.utils.**stats_str**(*data*, *percs=[0.0, 0.16, 0.5, 0.84, 1.0]*, *ave=False*, *std=False*, *weights=None*, *format=None*, *log=False*, *label_log=True*)

Return a string with the statistics of the given array.

Parameters

data
[ndarray of scalar] Input data from which to calculate statistics.

percs
[array_like of scalars in {0, 100}] Which percentiles to calculate.

ave
[bool] Include average value in output.

std
[bool] Include standard-deviation in output.

format
[str] Formatting for all numerical output, (e.g. *":.2f"*).

log
[bool] Convert values to log10 before printing.

Returns

out
[str] Single-line string of the desired statistics.

kalepy.utils.**subdivide**(*xx*, *num=1*, *log=False*)

Subdivide the giving array (e.g. bin edges) by the given factor.

Parameters

xx
[(X,) array_like of scalar,] Input array to be subdivided.

num
[int,] Subdivide each bin by this factor. Subdividing "once" (i.e. num=1) produces 2x number of bins. In general the number of output bins is `X * (num + 1)`.

log
[bool,] Subdivide evenly in log-space, instead of linear space (e.g. [0, 10.0] ==> [0.0, 3.16, 10.0])

Returns

div
[(X * num+1,) ndarray of float] Subdivided array with a number of elements equal to the length of the input array 'X' times one plus the subdivision factor *num*.

`kalepy.utils.`**`trapz_dens_to_mass`**(*pdf*, *edges*, *axis=None*)

Convert from density to mass, for values on the corner of a grid, using the trapezoid rule.

Parameters

pdf
[array_like] Density values, computed at the grid edges specified by the *edges* list-of-lists.

edges
[array_like of array_like] List of edge-locations along each dimension specifying the grid of values at which *pdf* are located. e.g. *[[x0, x1, … xn], [y0, y1, … ym], …]* The length of each sub-list in *edges*, must match the shape of *pdf*. e.g. if *edges* is a (3,) list, composed of sub-lists with lengths: *[N, M, L,]* then the shape of *pdf* must be *(N, M, L,)*.

axis
[int, array_like int, or None] Along which axes to convert from density to mass. If *None*, apply to all axes.

Returns

mass
[array_like] The *mass* array has as many dimensions as *pdf*, with each dimension one element shorter. e.g. if the shape of *pdf* is (N, M, …), then the shape of *mass* is (N-1, M-1, …).

`kalepy.utils.`**`trapz_nd`**(*data*, *edges*, *axis=None*)

3.3.3 Module contents

Multidimensional kernel density estimation for distribution functions, resampling, and plotting.

`kalepy.`**`density`**(*data*, *points=None*, *weights=None*, *reflect=None*, *probability=False*, *grid=False*, ***kwargs*)

Use a KDE to calculate the density of the given data.

This function (1) constructs a kernel-density estimate of the distribution function from which the given *data* were sampled; then (2) returns the values of the distribution function at *points* (which are automatically generated if they are not given).

Parameters

dataset
[array_like (N,) or (D,N,)] Dataset from which to construct the kernel-density-estimate. For multivariate data with *D* variables and *N* values, the data must be shaped (D,N). For univariate (D=1) data, this can be a single array with shape (N,).

points
[([D,]M,) array_like of float, or (D,) set of array_like point specifications] The locations at which the PDF should be evaluated. The number of dimensions *D* must match that of the *dataset* that initialized this class' instance. NOTE: If the *params* kwarg (see below) is given, then only those dimensions of the target parameters should be specified in *points*.

The meaning of *points* depends on the value of the *grid* argument:

- *grid=True* : *points* must be a set of (D,) array_like objects which each give the evaluation points for the corresponding dimension to produce a grid of values. For example, for a 2D dataset, *points=([0.1, 0.2, 0.3], [1, 2])*, would produce a grid of points with shape (3, 2): *[[0.1, 1], [0.1, 2]], [[0.2, 1], [0.2, 2]], [[0.3, 1], [0.3, 2]]*, and the returned values would be an array of the same shape (3, 2).
- *grid=False* : *points* must be an array_like (D,M) describing the position of *M* sample points in each of *D* dimensions. For example, for a 3D dataset: *points=([0.1, 0.2], [1.0, 2.0], [10, 20])*, describes 2 sample points at the 3D locations, *(0.1, 1.0, 10)* and *(0.2, 2.0, 20)*, and the returned values would be an array of shape (2,).

weights
[array_like (N,), None] Weights corresponding to each *dataset* point. Must match the number of points *N* in the *dataset*. If *None*, weights are uniformly set to 1.0 for each value.

reflect
[(D,) array_like, None] Locations at which reflecting boundary conditions should be imposed. For each dimension *D*, a pair of boundary locations (for: lower, upper) must be specified, or *None*. *None* can also be given to specify no boundary at that location. See class docstrings:*Reflection* for more information.

params
[int, array_like of int, None] Only calculate the PDF for certain parameters (dimensions). See class docstrings:*Projection* for more information.

grid
[bool,] Evaluate the KDE distribution at a grid of points specified by *points*. See *points* argument description above.

probability
[bool, normalize the results to sum to unity]

Returns

points
[array_like of scalar] Locations at which the PDF is evaluated.

vals
[array_like of scalar] PDF evaluated at the given points

`kalepy.pdf`(*data*, *points=None*, *weights=None*, *reflect=None*, *params=None*, *grid=False*, ***kwargs*)

Use a KDE to calculate the probability-density of the given data.

Wrapper for *kalepy.density(..., probability=True)*.

Parameters

dataset
[array_like (N,) or (D,N,)] Dataset from which to construct the kernel-density-estimate. For multivariate data with *D* variables and *N* values, the data must be shaped (D,N). For univariate (D=1) data, this can be a single array with shape (N,).

points
[([D,]M,) array_like of float, or (D,) set of array_like point specifications] The locations at which the PDF should be evaluated. The number of dimensions *D* must match that of the *dataset* that initialized this class' instance. NOTE: If the *params* kwarg (see below) is given, then only those dimensions of the target parameters should be specified in *points*.

The meaning of *points* depends on the value of the *grid* argument:

- *grid=True* : *points* must be a set of (D,) array_like objects which each give the evaluation points for the corresponding dimension to produce a grid of values. For example, for a 2D dataset, *points=([0.1, 0.2, 0.3], [1, 2])*, would produce a grid of points with shape (3, 2): *[[0.1, 1], [0.1, 2]], [[0.2, 1], [0.2, 2]], [[0.3, 1], [0.3, 2]]*, and the returned values would be an array of the same shape (3, 2).
- *grid=False* : *points* must be an array_like (D,M) describing the position of *M* sample points in each of *D* dimensions. For example, for a 3D dataset: *points=([0.1, 0.2], [1.0, 2.0], [10, 20])*, describes 2 sample points at the 3D locations, *(0.1, 1.0, 10)* and *(0.2, 2.0, 20)*, and the returned values would be an array of shape (2,).

weights
[array_like (N,), None] Weights corresponding to each *dataset* point. Must match the number of points *N* in the *dataset*. If *None*, weights are uniformly set to 1.0 for each value.

reflect
[(D,) array_like, None] Locations at which reflecting boundary conditions should be imposed. For each dimension *D*, a pair of boundary locations (for: lower, upper) must be specified, or *None*. *None* can also be given to specify no boundary at that location. See class docstrings:*Reflection* for more information.

params
[int, array_like of int, None] Only calculate the PDF for certain parameters (dimensions). See class docstrings:*Projection* for more information.

grid
[bool,] Evaluate the KDE distribution at a grid of points specified by *points*. See *points* argument description above.

Returns

points
[array_like of scalar] Locations at which the PDF is evaluated.

vals
[array_like of scalar] PDF evaluated at the given points

kalepy.**resample**(*data*, *size=None*, *weights=None*, *reflect=None*, *keep=None*, ***kwargs*)

Use a KDE to resample from a reconstructed density function of the given data.

This function (1) constructs a kernel-density estimate of the distribution function from which the given *data* were sampled; then, (2) resamples *size* data points from that function. If *size* is not given, then the same number of points are returned as in the input *data*.

Parameters

data
[([D,]N,) array_like of scalar, the data to be resampled,] The input data can be either: * 1D array_like (N,) with *N* data points, or * 2D array_like (D,N) with *D* parameters, and *N* data points

size
[int, None (default)] The number of new data points to draw. If *None*, then the number of *datapoints* is used.

weights
[(N,) array_like or *None*,] The weights of which each data point in *data*.

reflect
[(D,) array_like, None (default)] Locations at which reflecting boundary conditions should be imposed. For each dimension *D*, a pair of boundary locations (for: lower, upper) must be specified, or *None*. *None* can also be given to specify no boundary at that location.

keep
[int, array_like of int, None (default)] Parameters/dimensions where the original data-values should be drawn from, instead of from the reconstructed PDF. TODO: add more information.

Returns

samples
[([D,]L) ndarray of float] Newly drawn samples from the PDF, where the number of points *L* is determined by the *size* argument. If *squeeze* is True (default), and the number of dimensions in the original dataset *D* is one, then the returned array will have shape (L,).

3.4 kalepy

CHAPTER

FOUR

DEVELOPMENT & CONTRIBUTIONS

Please visit the github page to make contributions to the package. Particularly if you encounter any difficulties or bugs in the code, please submit an issue, which can also be used to ask questions about usage, or to submit general suggestions and feature requests. Direct additions, fixes, or other contributions are very welcome which can be done by submitting pull requests. If you are considering making a contribution / pull-request, please open an issue first to make sure it won't clash with other changes in development or planned for the future. Some known issues and indended future-updates are noted in the change-log file. If you are looking for ideas of where to contribute, this would be a good place to start.

Updates and changes to the newest version of *kalepy* will not always be backwards compatible. The package is consistently versioned, however, to ensure that functionality and compatibility can be maintained for dependencies. Please consult the change-log for summaries of recent changes.

4.1 Test Suite

If you are making, or considering making, changes to the *kalepy* source code, the are a large number of built in continuous-integration tests, both in the kalepy/tests directory, and in the kalepy notebooks. Many of the notebooks are automatically converted into test scripts, and run during continuous integration. If you are working on a local copy of *kalepy*, you can run the tests using the tester.sh script (i.e. '$ bash tester.sh'), which will include the test notebooks.

CHAPTER

FIVE

ATTRIBUTION

A JOSS paper has been published on the *kalepy* package. If you have found this package useful in your research, please add a reference to the code paper:

```
@article{Kelley2021,
  doi = {10.21105/joss.02784},
  url = {https://doi.org/10.21105/joss.02784},
  year = {2021},
  publisher = {The Open Journal},
  volume = {6},
  number = {57},
  pages = {2784},
  author = {Luke Zoltan Kelley},
  title = {kalepy: a Python package for kernel density estimation, sampling and plotting}
↪,
  journal = {Journal of Open Source Software}
}
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

Q

R

S

T

W