User Guide

- Installation
  - Requirements
  - Install
- Example: Sea state contour
- Overall work flow and software architecture
- Define a joint distribution and calculate a contour
  - Create independent distribution
  - Create dependent distribution
  - Bundle distributions and dependencies in a multivariate distribution
  - Constructing the contour
- Fit a model structure to measured data
  - Comprehensive example

Installation

Requirements

Make sure you have installed Python *3.8* by typing

```
python --version
```

in your shell.

(Older version might work, but are not actively tested)

Consider using the python version management pyenv.

Install

Install the latest version of viroconcom from PyPI by typing

```
pip install viroconcom
```

Alternatively, you can install from viroconcom repository's Master branch by typing

```
pip install https://github.com/virocon-organization/viroconcom/archive/master.zip
```

Example: Sea state contour

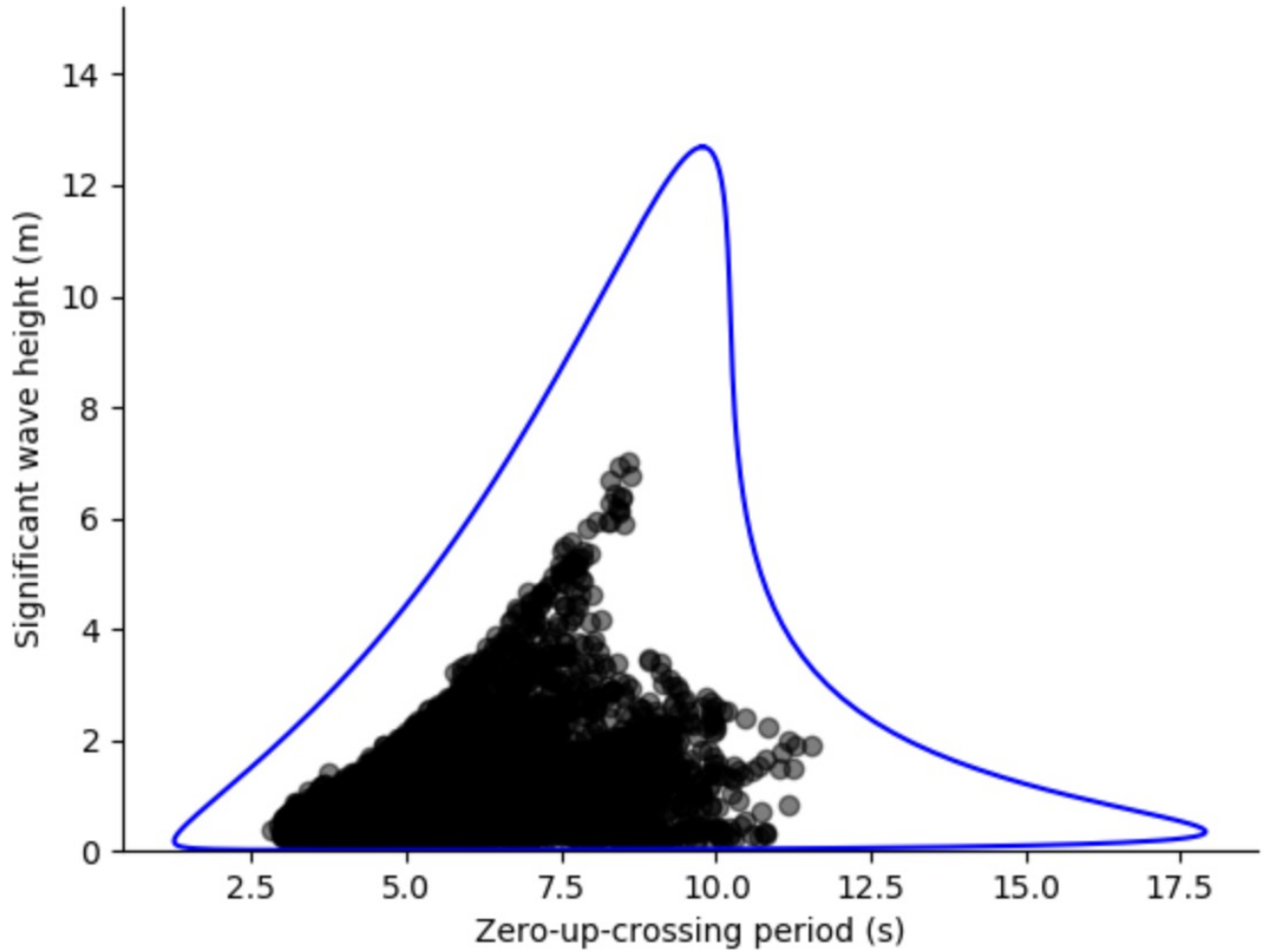
Let's start this user guide with a simple example.

Based on a dataset, the long-term joint distribution of sea states is estimated and this distribution will be used to construct an environmental contour with a return period of 50 years.

```
import matplotlib.pyplot as plt

from viroconcom.fitting import Fit
from viroconcom.contours import IFormContour
from viroconcom.read_write import read_ecbenchmark_dataset
from viroconcom.plot import plot_contour


# Load sea state measurements from the NDBC buoy 44007.
sample_hs, sample_tz, label_hs, label_tz = \
    read_ecbenchmark_dataset('datasets/1year_dataset_A.txt')

# Define the structure of the probabilistic model that will be fitted to the
# dataset. This model structure has been proposed in the paper "Global
# hierarchical models for wind and wave contours: Physical interpretations
# of the dependence functions" by Haselsteiner et al. (2020).
dist_description_hs = {'name': 'Weibull_Exp'}
dist_description_tz = {'name': 'Lognormal_SigmaMu',
                       'dependency': (0,  None, 0),
                       'functions': ('asymdecrease3', None, 'lnsquare2')}
model_structure = (dist_description_hs, dist_description_tz)
```

```
# Fit the model to the data.
fit = Fit((sample_hs, sample_tz), model_structure)
fitted_distribution = fit.mul_var_dist


# Compute an IFORM contour with a return period of 50 years.
tr = 50 # Return period in years.
ts = 1 # Sea state duration in hours.
contour = IFormContour(fitted_distribution, tr, ts)

# Plot the data and the contour.
fig, ax = plt.subplots(1, 1)
plt.scatter(sample_tz, sample_hs, c='black', alpha=0.5)
plot_contour(contour.coordinates[1], contour.coordinates[0],
             ax=ax, x_label=label_tz, y_label=label_hs)
plt.show()
```

The code, which is available as a Python file here, will create this plot:



Environmental contour with a return period of 50 years.

Overall work flow and software architecture

Figure 1 shows a flowchart that captures the overall functionality of viroconcom. A statistical model of the off-shore environment can be created by fitting a model structure to measured data. Then, this statistical model can be used to construct an environmental contour.

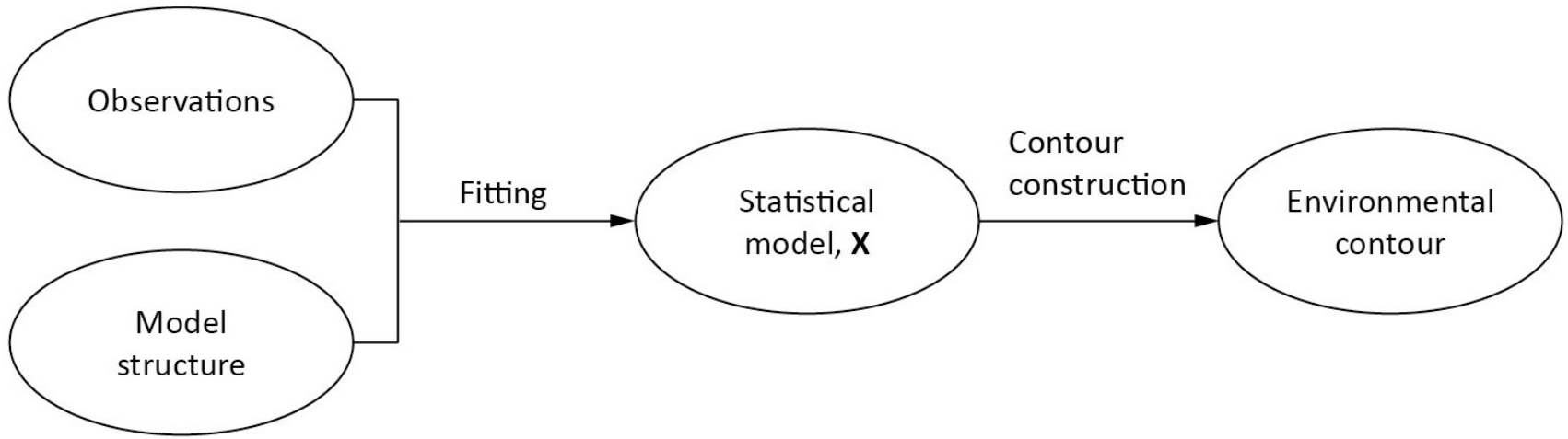


Figure 1. Flowchart showing how the process of fitting a model structure to measured data and constructing an environmental contour.

In viroconcom the class `Fit` handles the fitting, the class `MultivariateDistribution` represents the statistical model and the class `Contour` (and its child classes) handles the contour construction.

Figure 2 shows viroconcom's class diagram.

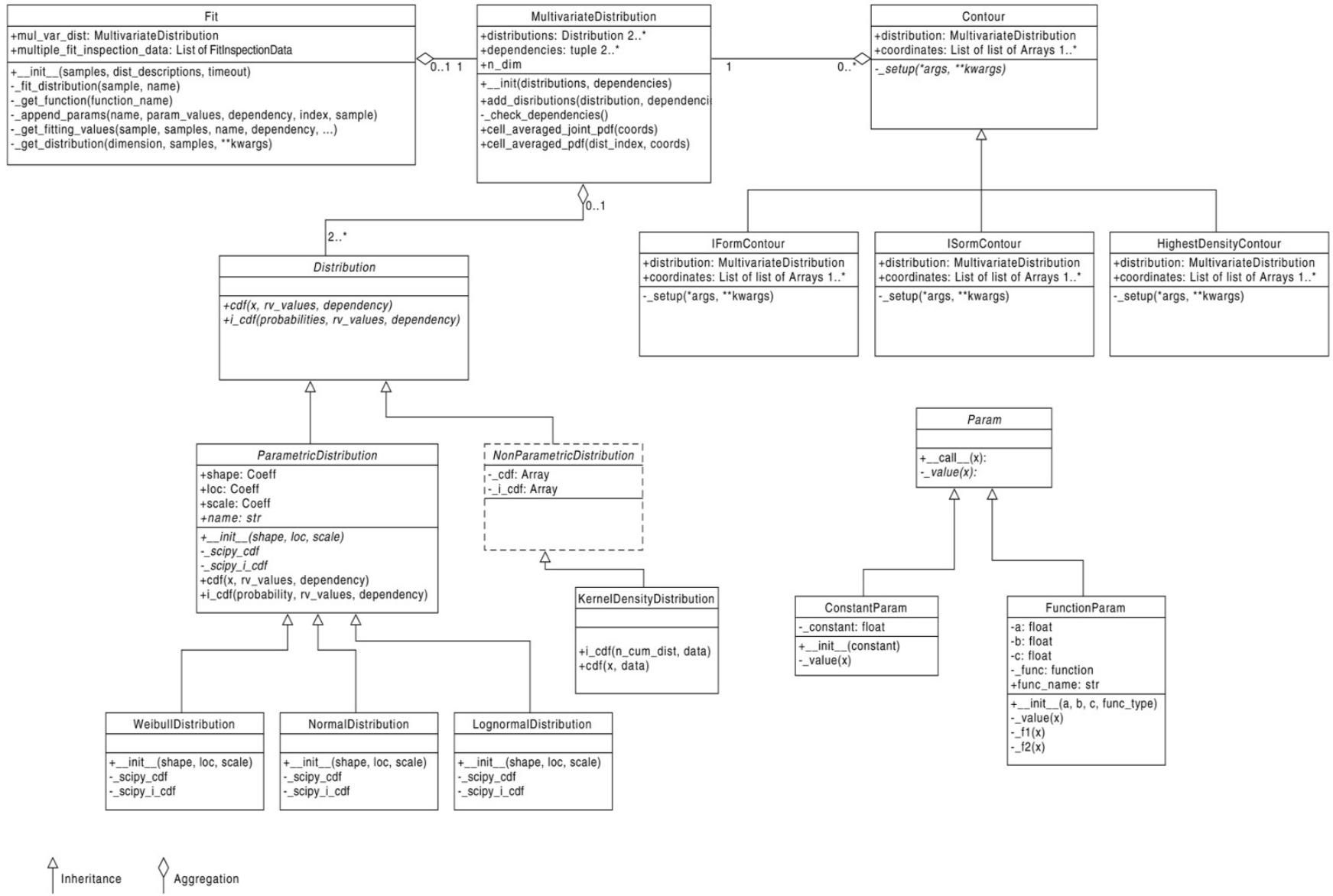


Figure 2. Class diagram showing among others the class Fit, which handles fitting a model structure to a dataset, and the class Contour, which handles contour construction. This class diagram was created for virocon-com version 1.1.0.

Define a joint distribution and calculate a contour

This chapter will explain how joint distributions and contours are handled in viroconcom. The process of estimating the parameter values of a joint distribution, the "fitting" is explained in the next chapter.

To create an environmental contour, first, we need to define a joint distribution. Then, we can choose a specific contour method and initiate the calculation. viroconcom uses so-called global hierarchical models to define the joint distribution and offers four common methods how an environmental contour can be defined based on a given joint distribution.

If the joint distribution is known, the procedure of calculating an environmental contour with viroconcom can be summarized as:

1. Create a first, independent univariate distribution.
2. Create another, usually dependent univariate distribution and define its dependency on the previous distributions.
3. Repeat step 2, until you have created a univariate distribution for each environmental variable.
4. Create a joint distribution by bundling the created univariate distributions.
5. Define the contour's return period and environmental state duration.
6. Choose a type of contour: `IFormContour`, `ISormContour`, `DirectSamplingContour` or `HighestDensityContour`.
7. Initiate the calculation.

These steps are explained in more detail in the following.

The file calculate_contours_similar_to_docs contains all the code that we will show on this page. We will use the sea state model that was proposed by Vanem and Bitner-Gregersen (2012; DOI: 10.1016/j.apor.2012.05.006) and compute environmental contours with return periods of 25 years.

Create independent distribution

Distributions are represented by the abstract class `Distribution`. This class is further subclassed by the abstract class `ParametricDistribution`. Distributions of this kind are described by three or four parameters: `shape`, `loc`, `scale` and possibly `shape2`. Though not all distributions need to make use of all parameters.

Currently there are five parametric distribution subclasses one can use to instantiate a distribution:

- `WeibullDistribution`
- `ExponentiatedWeibullDistribution`
- `LognormalDistribution`
- `NormalDistribution`
- `InverseGaussianDistribution`

This table shows, which variables of the probability density function are defined by specifying the scale, shape and location parameters:

distribution	probability density function	statistical parameter
normal distribution	$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$	scale: σ shape: - location: μ
Weibull distribution	$f(x) = \frac{\beta}{\alpha}\left(\frac{x-\gamma}{\alpha}\right)^{\beta-1} \exp\left[-\left(\frac{x-\gamma}{\alpha}\right)^{\beta}\right];\ x \geq \gamma.$	scale: α shape: β location: γ
log-normal distribution	$f(x) = \frac{1}{x\widetilde{\sigma}\sqrt{2\pi}} \exp\left[\frac{-(\ln x - \widetilde{\mu})^2}{2\widetilde{\sigma}^2}\right]$	scale: $e^{\widetilde{\mu}}$ shape: $\widetilde{\sigma}$ location: -

Distributions implemented in viroconcom and their parameters.

For the parameters there is the abstract class `Param`. As we want to create an independet distribution, we use the subclass `ConstantParam` to define `shape`, `loc`, and `scale`.

Say we want to create a Weibull distribution with `shape=1.471`, `loc=0.8888`, and `scale=2.776` (as in the model proposed by Vanem and Bitner-Gregersen).

We first create a WeibullDistribution object:

```
dist0 = WeibullDistribution(shape=1.471, loc=0.8888, scale=2.776)
```

We also need to create a dependency tuple for creating a `MultivariateDistribution` later on. This is a 3-element tuple with either `int` or `None` as entries. The first entry corresponds to `shape`, the second to `loc` and the third to `scale`. For an independent distribution all entries need to be set to `None`.

```
dep0 = (None, None, None)
```

Create dependent distribution

In a global hierarchical model, the dependency of a parametric distribution is described with dependence functions for the distribution's parameters. In Chapter Create independent distribution we used `ConstantParam` for the parameters. There is also `FunctionParam`, which can represent different dependence functions. It is callable and returns a parameter value depending on the value called with.

The following dependence functions, $f(x)$, are available under the given labels:

- **power3** : $f(x) = a + b * x^c$
- **exp3** : $f(x) = a + b * e^{c*x}$
- **lnsquare2** : $f(x) = \ln[a + b * \sqrt{(x/9.81)}]$
- **powerdecrease3** : $f(x) = a + 1/(x + b)^c$
- **asymdecrease3** : $f(x) = a + b/(1 + c * x)$
- **logistics4** : $f(x) = a + b/[1 + e^{-1*|c|*(x-d)}]$
- **alpha3** : $f(x) = (a + b * x^c)/2.0445^{1/logistics4(x,c_1,c_2,c_3,c_4)}$

Say we have a random variable X that is described by the distribution created in Create independent distribution. Now we want to create a distribution that describes the random variable Y, which is dependent on X (in common notation $Y|X$).

For this, we first need to define an order of the distributions, so that we can determine on which distributions another may depend. We define this order, as the order in which the univariate distribution are later on passed to the `MultivariateDistribution` constructor. For now we use the order of creation. The first distribution (that was described in Chapter Create independent distribution) has the index `0`. Thus, using our previously introduced random variables, $X_0 = X$ and $X_1 = Y$. In viroconcom, we need to use this order in the dependency tuples.

As already described in Chapter Create independent distribution the 3 entries in the tuple correspond to the **shape**, **loc**, and **scale** parameters and the entries are either **int** or **None**. If an entry is **None**, the corresponding parameter is independent. If an entry is an **int** the parameter depends on the distribution with that index, in the order defined above.

For example, a dependency tuple of **(0, None, 1)** means, that **shape** depends on the first distribution, **loc** is independent and **scale** depends on the second distribution.

We now want to create a dependent lognormal distribution that will represent the second variable, $X_1|X_0$. Opposed to, for example, a Weibull or normal distribution, a lognormal distribution is often not described by **shape**, **loc**, and **scale**, but by the mean **mu** and standard deviation **sigma** of the corresponding normal distribution. In this example, we want **mu** and **sigma** to depend on the prior created Weibull distribution. The **loc** parameter is ignored by the **LognormalDistribution**.

The conversion between **shape**, **scale**, **mu** and **sigma** is:

$$shape = \sigma$$

$$scale = e^{\mu}$$

The class **LognormalDistribution** has a constructor for **`shape** and **scale** as well as for **mu** and **sigma**.

Say we want to define the following dependence structure for $X_1|X_0$, where x_0 is a realization of X_0:

$$\sigma(x_0) = 0.04 + 0.1748 * e^{-0.2243 * x_0}$$

$$\mu(x_0) = 0.1 + 1.489^{x_0 * 0.1901}$$

In viroconcom, to define this dependence structure, first we create the parameters as `FunctionParam` using the keywords “exp3” and “power” to specify the wanted dependence functions

```
my_sigma = FunctionParam('exp3', 0.04, 0.1748, -0.2243)
my_mu = FunctionParam('power3', 0.1, 1.489, 0.1901)
```

Then we create the **`LognormalDistribution`** using the **`mu sigma`** constructor:

```
dist1 = LognormalDistribution(sigma=my_sigma, mu=my_mu)
```

And eventually define the dependency tuple:

```
dep1 = (0, None, 0)
```

Alternativly we could have defined the distribution as follows, using the wrapper argument of the `FunctionParam`:

```
shape = FunctionParam(0.04, 0.1748, -0.2243, "exp3")
scale = FunctionParam(0.1, 1.1489, 0.1901, "power3", wrapper=numpy.exp)
dist1 = LognormalDistribution(shape, None, scale)
dep1 = (0, None, 0)
```

Bundle distributions and dependencies in a multivariate distribution

To create a contour, we need a joint distribution. In viroconcom joint distributions can be represented by the `MultivariateDistribution` class.

To create a `MultivariateDistribution` we first have to bundle the distributions and dependencies in lists:

```
distributions = [dist0, dist1]
dependencies = [dep0, dep1]
```

The `MultivariateDistribution` can now simply be created by passing these lists to the constructor:

```
mul_dist = MultivariateDistribution(distributions, dependencies)
```

Constructing the contour

Next, we need to define the contour's exceedance probability, α, which is calculated using the return period, t_R, and the model's state duration, t_S:

$$\alpha = t_S/t_R$$

In viroconcom the return period is assumed to be given in years and the state duration is assumed to be given in hours.

Then we can select one of the four contour methods:

- Inverse first-order reliabilty method (IFORM)
- Inverse second-order reliablity method (ISORM)
- Direct sampling contour method
- Highest density contour method

Inverse first-order reliability method (IFORM)

With all contours, we need to specify the return period and the state duration. In addition, to create an IFORM contour we need to specify the number of points along the contour that shall be calculated.

Let us calculate 90 points along the contour such that we have a resolution of 2 degrees. With the previously created **`mul_dist`**, we can compute a contour with a **`return_period`** of **`25`** years and a **`state_duration`** of **`6`** hours like this:

```
iform_contour = IFormContour(mul_dist, 25, 6, 90)
```

Highest density contour method

To create a highest density contour, for the used numerical integration, we need to specify a grid in the variable space in addition to return period and state duration. This is done by passing the grid's **`limits`** and **`deltas`** to the constructor. **`limits`** has to be a list of tuples containing the min and max limits for the variable space, one tuple for each dimension. **`deltas`** specifies the grid cell size. It is either a list of length equal to the number of dimension, containing the step size per dimensions or a scalar. If it is a scalar it is used for all dimensions.

The grid includes the min and max values: **`x = [min, min + delta, ..., max - delta, max]`**

To create a highest density contour for the previously created **`mul_dist`** with a **`return_period`** of **`25`** years and a **`state_duration`** of **`6`**, we first define the variable space to be between 0 and 20 and set the step size to 0.5 in the first and 0.1 in the second dimension.:

```
limits = [(0, 20), (0, 20)]
deltas = [0.5, 0.1]
```

The contour can then be created as follows:

```
hdens_contour = HighestDensityContour(mul_dist, 25, 6, limits, deltas)
```

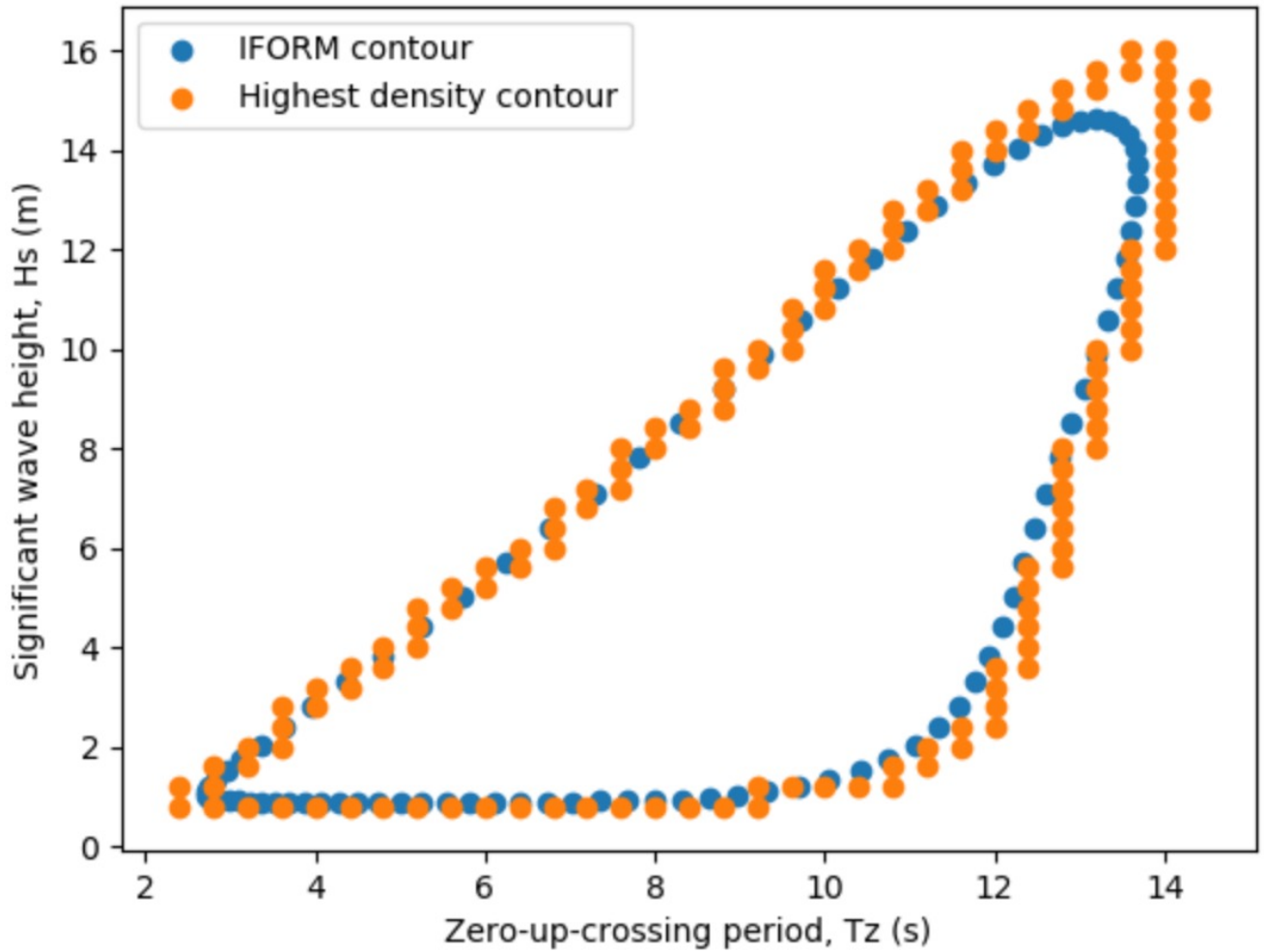
Plotting the contour

To plot the contour we need to access the **`coordinates`** attribute of the contour.

Using for example **`matplotlib`** the following code…

```
import matplotlib.pyplot as plt

plt.scatter(iform_contour.coordinates[1], iform_contour.coordinates[0],
            label='IFORM contour')
plt.scatter(hdens_contour.coordinates[1], hdens_contour.coordinates[0],
            label='Highest density contour')
plt.xlabel('Zero-up-crossing period, Tz (s)')
plt.ylabel('Significant wave height, Hs (m)')
plt.legend()
plt.show()
```

creates this plot:

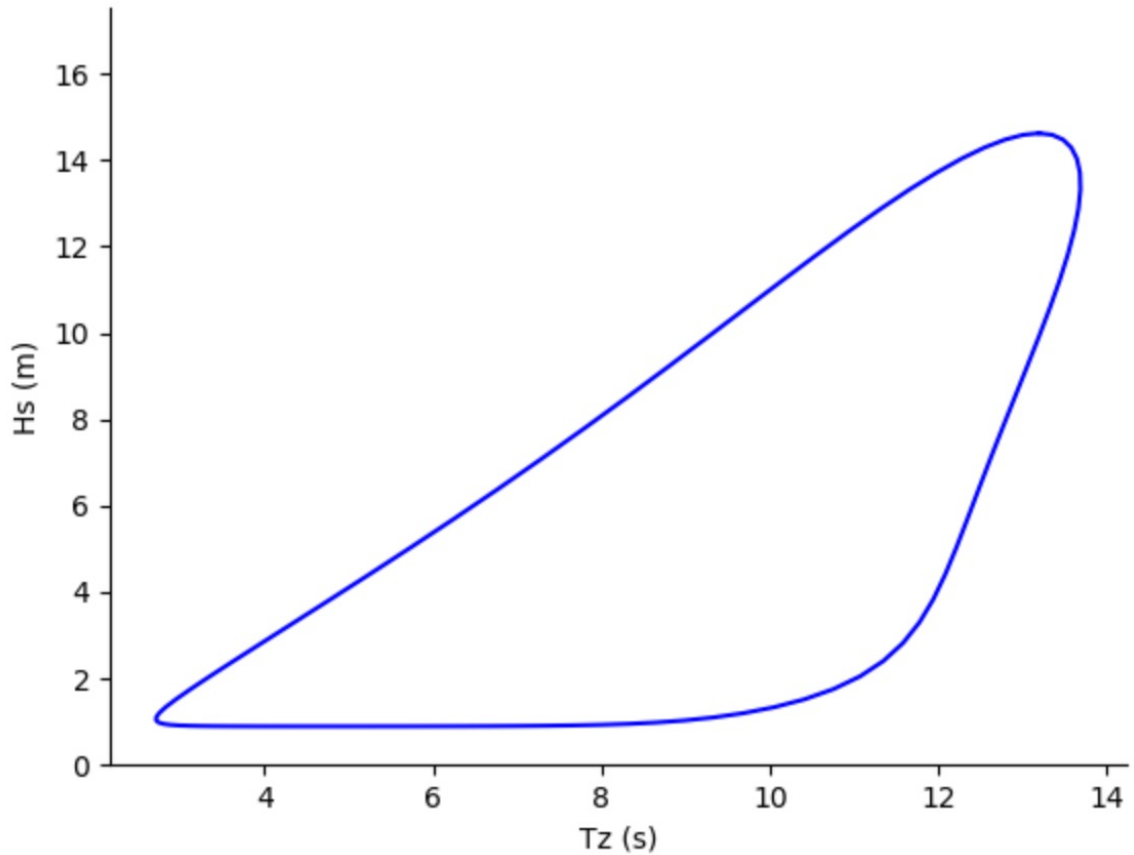


Plot of the calculated IFORM and highest density contours.

Alternatively, we could use viroconcom's standard plotting function…

```
from viroconcom.plot import plot_contour

plot_contour(iform_contour.coordinates[1], iform_contour.coordinates[0],
             x_label='Tp (s)', y_label='Hs (m)')
plt.show()
```

to create this plot:



Fit a model structure to measured data

The module fitting implements functionality to fit a model structure to measured environmental data. In other words, it can be used to estimate the parameter values of a joint distribution.

To fit a model structure to a dataset, we need to build an object of the class **`Fit`** in this module.

Exemplary call:

```
example_fit = Fit((sample_0, sample_1), (dist_description_0, dist_description_1))
```

It is important that the parameter **`samples`** is in the form (sample_0, sample_1, …). Each sample is a collection of data from type *list* and also all samples have the same length. The parameter **`dist_descriptions`** describes the structure of the probabilistic model that should be fitted to the sample. It should be from type *list* and should contain a dictionary for each dimension in the same sequence of the samples. It should accordingly have the same length as **`samples`**.

Each `dist_description` dictionary describes one dimension of the probabilistic model structure. It must contain the name of the current distribution under the key `name`, which could be, for example, `"Lognormal"`. If the distribution is conditional, it also must contain the keys `dependency` and `functions`. The `dependency` value must be of type *list*. In the sequence of `shape, loc, scale`, it contains integers for the dependency of the current parameter or *None* if it has no dependency. An entry of 0 means that the parameter depends upon the variable with index 0, for example $X_1|X_0$ if the second dimension (index 1) is specified. The `functions` value is of type *list* too, and is interpreted in the sequence `shape, loc, scale`. Its entries define which dependence functions are fitted. Additional keys such as `width_of_intervals` or `min_datapoints_for_fit` are optional and can be used to control the fitting procedure.

The following distributions are available under the given key values:

- **Weibull_2p** : 2-parameter Weibull distribution
- **Weibull_3p** : translated Weibull distribution (sometimes simply called 3-parameter Weibull distribution)
- **Weibull_Exp** : exponentiated Weibull distribution
- **Lognormal** : lognormal distribution parametrized with exp(mu) and sigma
- **Lognormal_SigmaMu** : lognormal distribution parametrized with mu and sigma
- **Normal** : normal distribution
- **InverseGaussian** : inverse gaussian distribution (**not** the inverse of the normal distribution)

The following dependence functions are available under the given key values:

- **power3** : $a + b * x^c$
- **exp3** : $a + b * e^{x*c}$
- **lnsquare2** : $\ln[a + b * \sqrt{(x/9.81)}]$
- **powerdecrease3** : $a + 1/(x + b)^c$
- **asymdecrease3** : $a + b/(1 + c * x)$
- **logistics4** : $a + b/[1 + e^{-1*|c|*(x-d)}]$
- **alpha3** : $(a + b * x^c)/2.0445^{1/logistics4(x,c_1,c_2,c_3,c_4)}$
- **poly1** : $a * x + b$
- **poly2** : $a * x^2 + b * x + c$, with $c - \frac{b^2}{4*a} \geq 0, c \leq 0$
- **None** : no dependency

The following optional keys and values are available:

- **number_of_intervals** : int. The sample of this variable will be divided into the given number of intervals. Intervals will be equally spaced.
- **width_of_intervals** : float. The sample of this variable will be divided into intervals with the given width.
- **points_per_interval** : int. The sample of this variable will be divided into intervals with the given number of points.
- **min_datapoints_for_fit** : int. A marginal distribution will only be fitted to an interval if the interval contains at least the given number of observations.
- **do_use_weights_for_dependence_function** : boolean. If true the dependence function is fitted with weights that normalize each parameter value.
- **fixed_parameters** : list with one entry for each parameter. *None* means that the parameter is free - it will be estimated. If a number is given, the parameter is fixed to that number and the value will not be estimated.

Example for a **dist_description** that could represent the marginal distribution of significant wave height:

```
dist_description_0 = {'name': 'Weibull_Exp',
                      'width_of_intervals': 1}
```

Example for a **dist_description** that could represent the conditonal distribution of zero-up-crossiong period:

```
dist_description_1 = {'name': 'Lognormal_SigmaMu',
                      'dependency': (0,  None, 0),
                      'functions': ('asymdecrease3', None, 'lnsquare2'),
                      'min_datapoints_for_fit': 50
                      }
```

In the given exemplary call (first code snipped), if Fit() is finished, the object **example_fit** will have the attribute **mul_var_dist** that is an object of **MultivariateDistribution**, holding the fitted joint distribution. Additionally, **example_fit** will have the attribute **multiple_fit_inspection_data**, which can be used to analyze the goodness of fit.

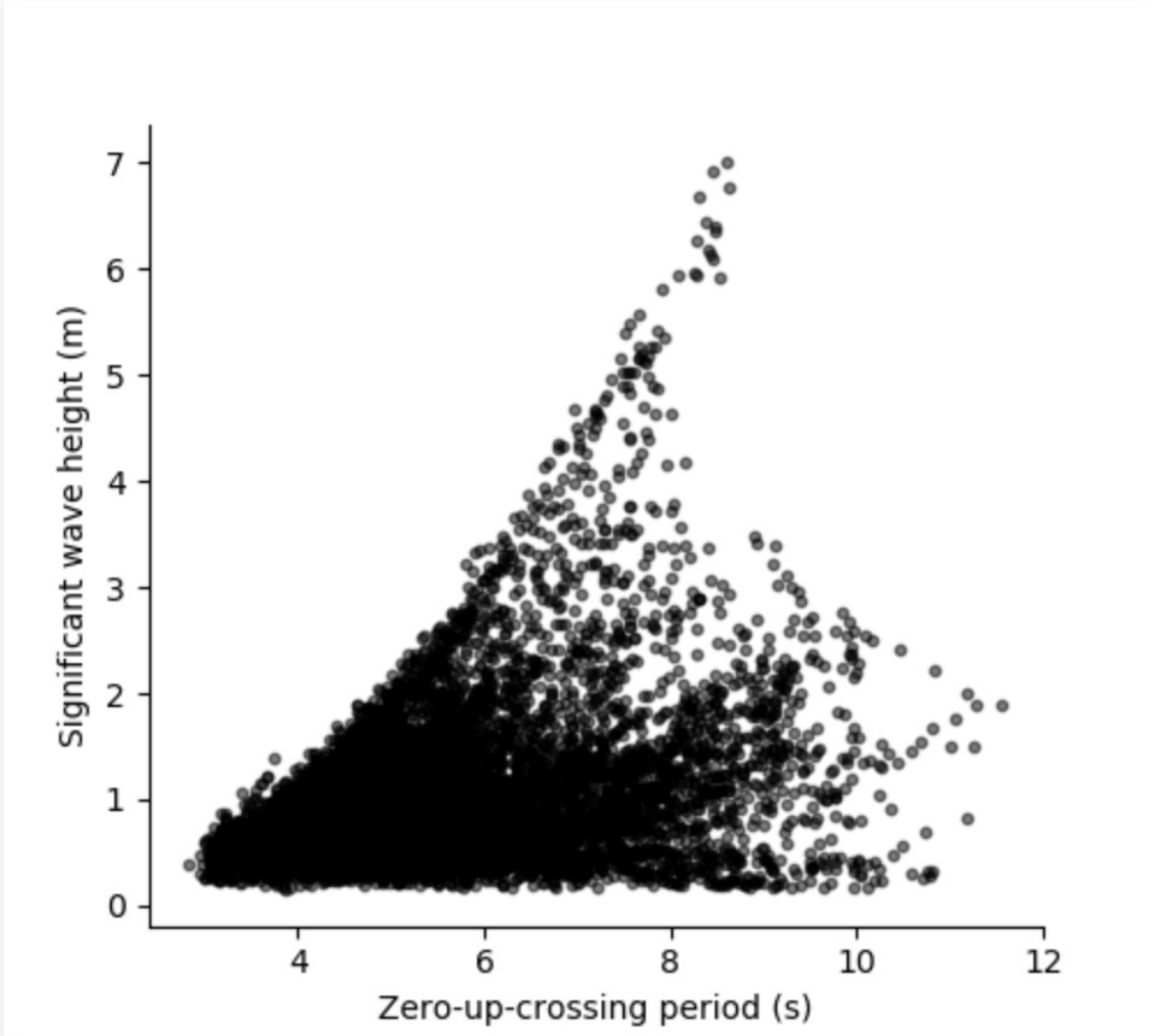
Comprehensive example

The following example is based on the file fit_distribution_similar_to_docs .

First, let us load a dataset that holds measurements of sea states. The first variable is significant wave height, H_s, and the second variable zero-up-crossing period, T_z

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

from viroconcom.read_write import read_ecbenchmark_dataset
from viroconcom.fitting import Fit
from viroconcom.contours import IFormContour
from viroconcom.plot import SamplePlotData, plot_sample, plot_marginal_fit, \
    plot_dependence_functions, plot_contour

sample_0, sample_1, label_hs, label_tz = \
    read_ecbenchmark_dataset('datasets/1year_dataset_A.txt')

fig, ax = plt.subplots(1, 1, figsize=(5, 4.5))
sample_plot_data = SamplePlotData(sample_1, sample_0)
plot_sample(sample_plot_data, ax)
plt.xlabel('Zero-up-crossing period (s)')
plt.ylabel('Significant wave height (m)')
plt.show()
```

The code snipped will create this plot:



1 year of measurements from NDBC's buoy 44007.

Now we describe the type of multivariate distribution that we want to fit to this data

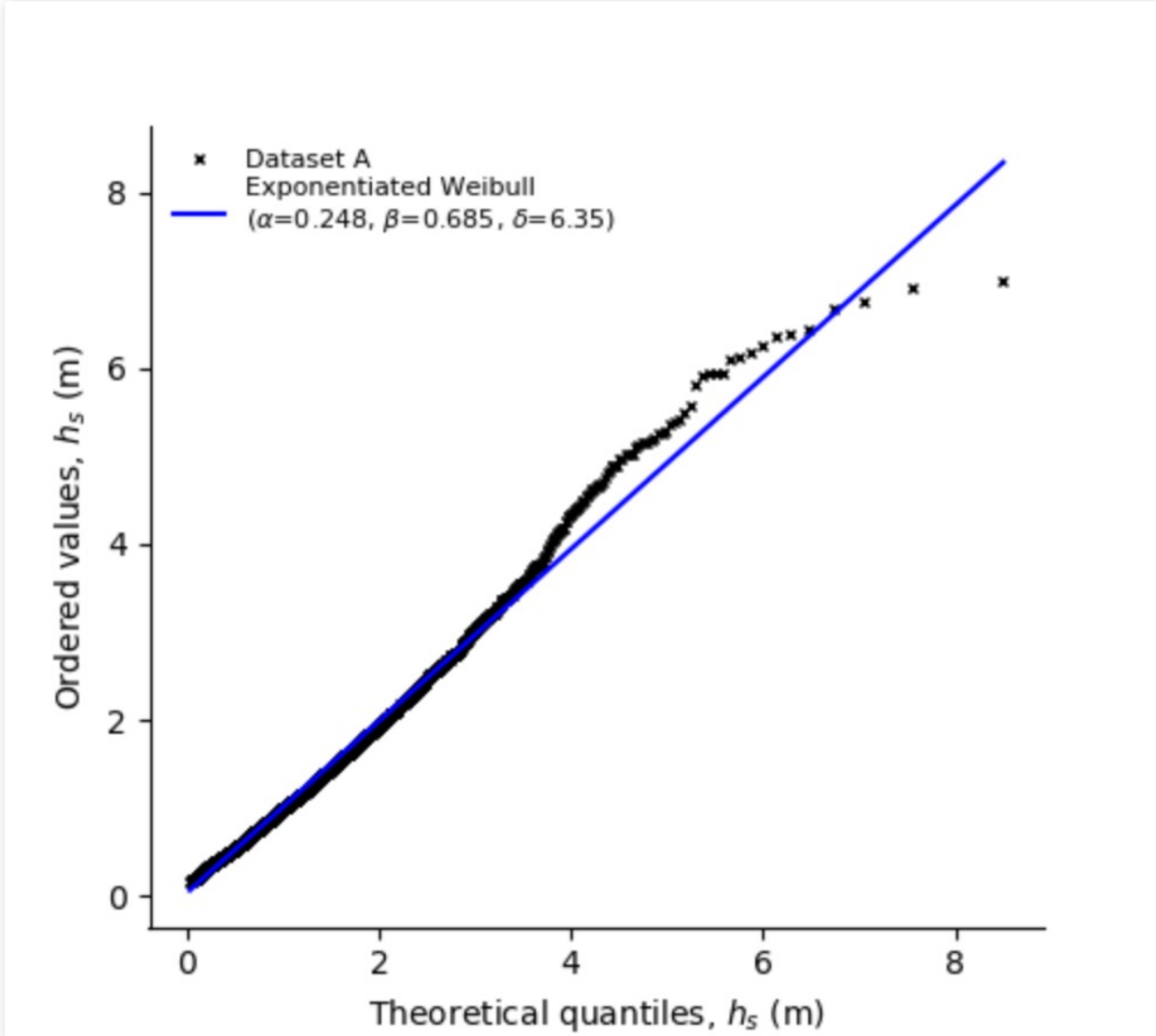
```
dist_description_0 = {'name': 'Weibull_Exp',
                      'width_of_intervals': 1}
dist_description_1 = {'name': 'Lognormal_SigmaMu',
                      'dependency': (0,  None, 0),
                      'functions': ('asymdecrease3', None, 'lnsquare2'),
                      'min_datapoints_for_fit': 50
                      }
```

Based on this description, we can compute the fit and save the two fitted distributions in dedicated variables

```
my_fit = Fit((sample_0, sample_1), (dist_description_0, dist_description_1))

fitted_hs_dist = my_fit.mul_var_dist.distributions[0]
fitted_tz_dist = my_fit.mul_var_dist.distributions[1]
```

Now, let us visualize the fit for the first variable using a QQ-plot

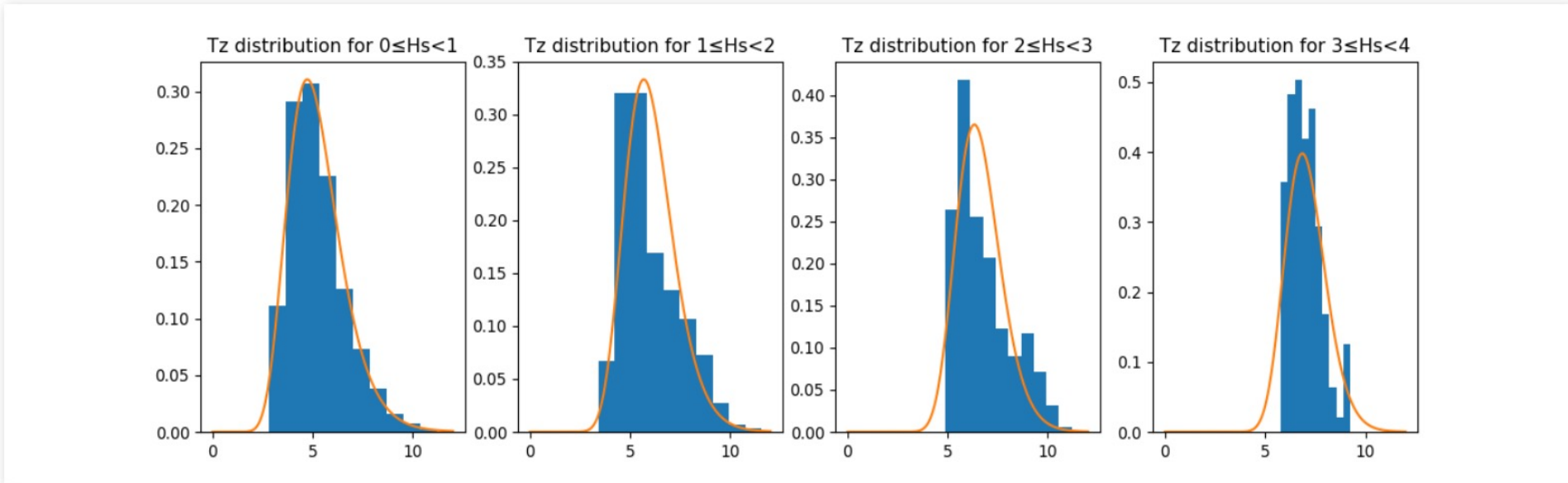
```
fig, ax = plt.subplots(1, 1, figsize=(5, 4.5))
plot_marginal_fit(sample_0, fitted_hs_dist, fig, ax, label='$h_s$ (m)',
dataset_char='A')
plt.show()
```



QQ-plot showing the fitted exponentiated Weibull distribution and the empirical wave height data..

For our second variable, we need some more plots to inspect it properly. Let us start with the marginal distributions that were fitted to Hs-intervals

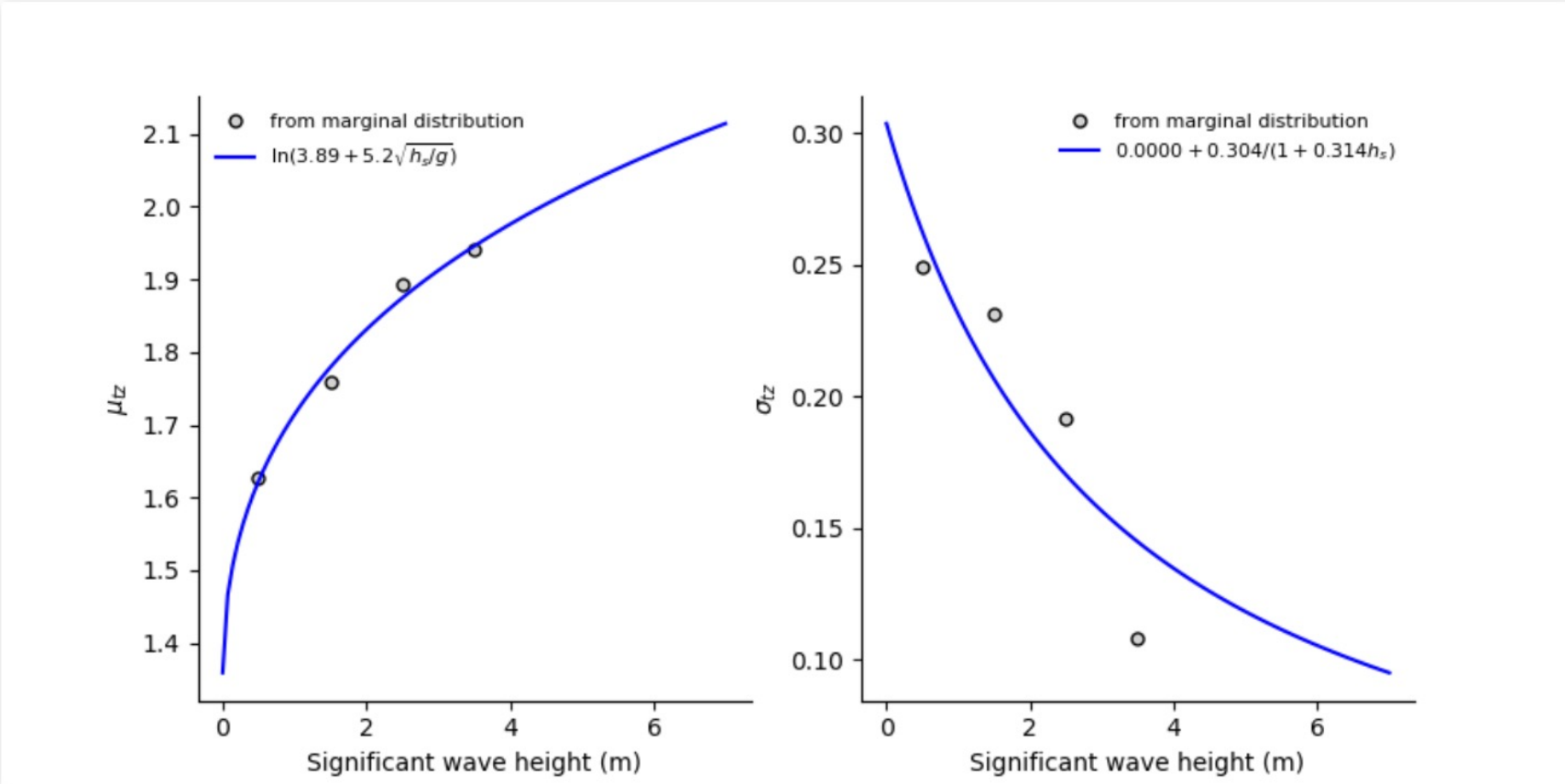
```
n_fits = len(my_fit.multiple_fit_inspection_data[1].scale_at)
fig, axs = plt.subplots(1, n_fits, figsize=(14, 4))
for i in range(n_fits):
        axs[i].set_title('Tz distribution for ' + str(i) + '≤Hs<' + str(i + 1))
        axs[i].hist(my_fit.multiple_fit_inspection_data[1].scale_samples[i], density=1)
        x = np.linspace(0, 12, 200)
        interval_center = my_fit.multiple_fit_inspection_data[1].scale_at[i]
        f = fitted_tz_dist.pdf(x, np.zeros(x.shape) + interval_center, (0, None, 0))
        axs[i].plot(x, f)
plt.show()
```



Fitted marginal distributions at different Hs intervals.

Now, let us analyze how well our dependence functions fit to the marginal distributions' four scale and shape values

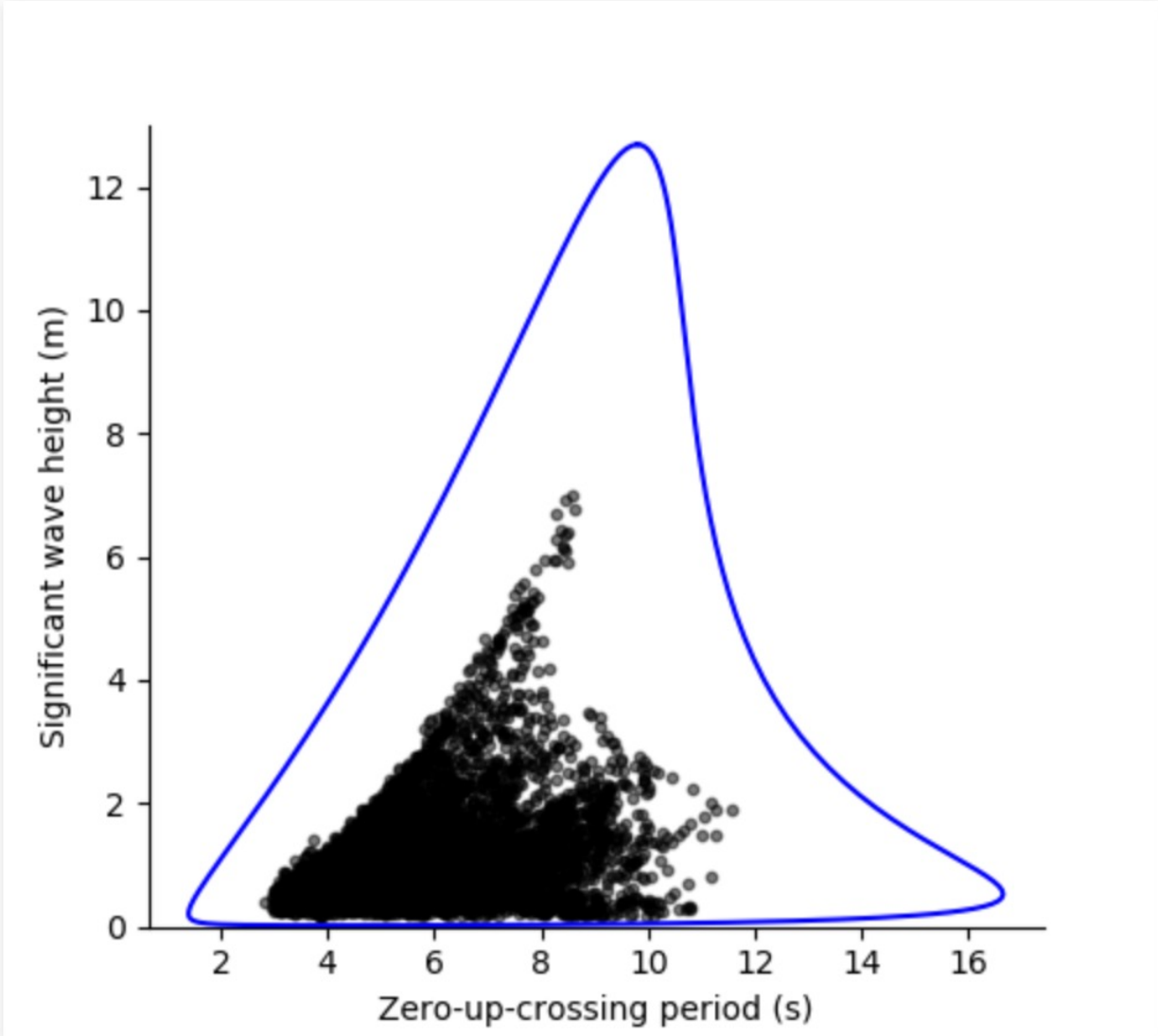
```
fig = plt.figure(figsize=(9, 4.5))
plot_dependence_functions(my_fit, fig, unconditonal_variable_label=label_hs,
                          factor_draw_longer=2)
plt.show()
```



Fitted dependence function.

Finally, let us use the fitted joint distribution to compute an environmental contour

```
iform_contour = IFormContour(my_fit.mul_var_dist, 50, 1)
fig, ax = plt.subplots(1, 1, figsize=(5, 4.5))
plot_contour(iform_contour.coordinates[1], iform_contour.coordinates[0],
             ax=ax, x_label=label_tz, y_label=label_hs,
             sample_plot_data=sample_plot_data, upper_ylim=13)
plt.show()
```



50-year environmental contour based on the fitted distribution.