Uncertainpy Documentation

Release 1.2.3

Simen TennÃ¸e

Nov 02, 2020

Content:

Uncertainpy

Uncertainpy is a python toolbox for uncertainty quantification and sensitivity analysis tailored towards computational neuroscience.

Uncertainpy is model independent and treats the model as a black box where the model can be left unchanged. Uncertainpy implements both quasi-Monte Carlo methods and polynomial chaos expansions using either point collocation or the pseudo-spectral method. Both of the polynomial chaos expansion methods have support for the rosenblatt transformation to handle dependent input parameters.

Uncertainpy is feature based, i.e., if applicable, it recognizes and calculates the uncertainty in features of the model, as well as the model itself. Examples of features in neuroscience can be spike timing and the action potential shape.

Uncertainpy is tailored towards neuroscience models, and comes with several common neuroscience models and features built in, but new models and features can easily be implemented. It should be noted that while Uncertainpy is tailored towards neuroscience, the implemented methods are general, and Uncertainpy can be used for many other types of models and features within other fields.

CHAPTER 1

Installation

Uncertainpy works with with Python 3. Uncertainpy can easily be installed using pip. The minimum install is:

> pip install uncertainpy

To install all requirements you can write:

> pip install uncertainpy[all]

Specific optional requirements can also be installed, see below for an explanation. Uncertainpy can also be installed by cloning the Github repository:

```
$ git clone https://github.com/simetenn/uncertainpy
$ cd /path/to/uncertainpy
$ python setup.py install
```

`setup.py` are able to install different set of dependencies. For all options run:

```
$ python setup.py --help
```

Alternatively, Uncertainpy can be easily installed (minimum install) with conda using conda-forge channel:

```
$ conda install -c conda-forge uncertainpy
```

The above installation, within a conda environment, is only compatible with Python 3.x.

1.1 Dependencies

Uncertainpy has the following dependencies:

- `chaospy`
- `tqdm`
- `h5py`
- `multiprocess`

- `numpy`
- `scipy`
- `seaborn`
- `matplotlib`
- `xvfbwrapper`
- `six`
- `SALib`
- `exdir`

These are installed with the minimum install.

`xvfbwrapper` requires `xvfb`, which can be installed with:

```
sudo apt-get install xvfb
```

Additionally Uncertainpy has a few optional dependencies for specific classes of models and for features of the models.

1.1.1 EfelFeatures

`uncertainpy.EfelFeatures` requires the Python package

- `efel`

which can be installed with:

```
pip install uncertainpy[efel_features]
```

or:

```
pip install efel
```

or through:

```
python setup.py install --efel_features
```

1.1.2 NetworkFeatures

`uncertainpy.NetworkFeatures` requires the Python packages

- `elephant`
- `neo`
- `quantities`

which can be installed with:

```
pip install uncertainpy[network_features]
```

or:

```
pip install elephant, neo, quantities
```

or through:

```
python setup.py install --network_features
```

1.1.3 NeuronModel

`uncertainpy.NeuronModel` requires the external simulator NEURON (with Python), a simulator for neurons. NEURON must be installed by the user.

1.1.4 NestModel

`uncertainpy.NestModel` requires the external simulator NEST (with Python), a simulator for network of neurons. NEST must be installed by the user.

1.2 Test suite

Uncertainpy comes with an extensive test suite that can be run with the `test.py` script. For how to use `test.py` run:

```
$ python test.py --help
```

`test.py` has all the above dependencies in addition to:

- `click`

These dependencies can be installed with:

```
pip install uncertainpy[tests]
```

or:

```
pip install click
```

or through:

```
python setup.py install --tests
```

1.3 Documentation

The documentation is generated through `sphinx`, and has the following dependencies:

- `sphinx`
- `sphinx_rtd_theme`

These dependencies can be installed with:

```
pip install uncertainpy[docs]
```

or:

```
pip install sphinx, sphinx_rtd_theme
```

or through:

```
python setup.py install --docs
```

The documentation is build by:

```
cd docs
make html
```

CHAPTER 2

Quickstart

This section gives a brief overview of what you need to know to perform an uncertainty quantification and sensitivity analysis with Uncertainpy. It only gives the most basic way of getting started, many more options than shown here are available.

The uncertainty quantification and sensitivity analysis includes three main components:

- The **model** we want to examine.
- The **parameters** of the model.
- Specifications of **features** in the model output.

The model and the parameters are required, while the feature specification is optional. The above components are brought together in the *UncertaintyQuantification* class. This class is the main class to interact with, and is a wrapper for the uncertainty calculations.

2.1 Uncertainty quantification

The *UncertaintyQuantification* class is used to define the problem, perform the uncertainty quantification, and to save and visualize the results. Among others, `UncertaintyQuantification` takes the following arguments:

```
UQ = un.UncertaintyQuantification(
        model=...,                        # Required
        parameters=...,                   # Required
        features=...,                     # Optional
    )
```

The arguments are given as instances of their corresponding Uncertainpy classes (*Models*, *Parameters*, and *Features*). These classes are briefly described below. After the problem is defined, an uncertainty quantification and sensitivity analysis can be performed using the `UncertaintyQuantification.quantify` method. Among others, `quantify` takes the following arguments:

```
data = UQ.quantify(
    method="pc"``"mc",
    pc_method="collocation"``"spectral",
    rosenblatt=False``True
)
```

The *method* argument allows the user to choose whether Uncertainpy should use polynomial chaos (`"pc"`) or quasi-Monte carlo (`"mc"`) methods to calculate the relevant statistical metrics. If polynomial chaos are chosen, *pc_method* further specifies whether point collocation (`"collocation"`) or spectral projection (`"spectral"`) methods is used to calculate the expansion coefficients. Finally, *rosenblatt* (`False` or `True`) determines if the Rosenblatt transformation should be used. The Rosenblatt is required if the uncertain parameters are dependent. If nothing is specified, Uncertainpy by default uses polynomial chaos based on point collocation without the Rosenblatt transformation. The results from the uncertainty quantification are automatically saved and plotted. Additionally, the results from the uncertainty quantification are returned in `data`, as a `Data` object (see *Data*).

2.2 Models

The easiest way to create a model is to use a Python function. We need a Python function that runs a simulation on a specified model for a given set of model parameters, and returns the simulation output. An example outline of a model function is:

```
def example_model(parameter_1, parameter_2):
    # An algorithm for the model, or a script that runs
    # an external model, using the given input parameters.

    # Returns the model output and model time
    # along with the optional info object.
    return time, values, info
```

Such a model function can be given as the *model* argument to the `UncertaintyQuantification` class. Note that sometimes an features or the preprocessing requires that additional info object is required to be returned from the model.

For more on models see *Models*.

2.3 Parameters

The parameters of a model are defined by two properties, they must have (i) a name and (ii) either a fixed value or a distribution. It is important that the name of the parameter is the same as the name given as the input argument in the model function. A parameter is considered uncertain if it has a probability distribution, and the distributions are given as Chaospy distributions. 64 different univariate distributions are defined in Chaospy. For a list of available distributions and detailed instructions on how to create probability distributions with Chaospy, see Section 3.3 in the Chaospy paper.

parameters is a dictionary with the above information, the names of the parameters are the keys, and the fixed values or distributions of the parameters are the values. As an example, if we have two parameters, where the first is named `name_1` and has a uniform probability distributions in the interval $[8, 16]$, and the second is named `name_2` and has a fixed value 42, the list become:

```
import chaospy as cp
parameters = {"name_1": cp.Uniform(8, 16), "name_2": 42}
```

The *parameter* argument in `UncertaintyQuantification` is such a dictionary.

For more on parameters see *Parameters*.

2.4 Features

Features are specific traits of the model output, and Uncertainpy has support for performing uncertainty quantification and sensitivity analysis of features of the model output, in addition to the model output itself. Features are defined by creating a Python function to calculate a specific feature from the model output. The feature function take the items returned by the model as as input arguments, calculates a specific feature of this model output and returns the results. quantification on.

The outline for a feature function is:

```
def example_feature(time, values, info):
    # Calculate the feature using time, values and info.

    # Return the feature times and values.
    return time_feature, values_feature
```

The *features* argument to `UncertaintyQuantification` can be given as a list of feature functions we want to examine.

For more on features see *Features*.

CHAPTER 3

Examples

This is a collection of examples that shows the use of Uncertainpy for a few different case studies.

3.1 A cooling coffee cup model

Here we show an example (found in `examples/coffee_cup`) where we examine the changes in temperature of a cooling coffee cup that follows Newton's law of cooling:

$$\frac{dT(t)}{dt} = -\kappa(T(t) - T_{env})$$

This equation tells how the temperature T of the coffee cup changes with time t, when it is in an environment with temperature T_{env}. κ is a proportionality constant that is characteristic of the system and regulates how fast the coffee cup radiates heat to the environment. For simplicity we set the initial temperature to a fixed value, $T_0 = 95°\text{C}$, and let κ and T_{env} be uncertain parameters. We give the uncertain parameters in the following distributions:

$$\begin{aligned}\kappa &= \text{Uniform}(0.025, 0.075),\\ T_{env} &= \text{Uniform}(15, 25).\end{aligned}$$

3.1.1 Using a function

There are two approaches to creating the model, using a function or a class. The function method is easiest so we start with that approach. The complete for this example can be found in `examples/coffee_cup/uq_coffee_function.py`. We start by importing the packages we use:

```
import uncertainpy as un
import chaospy as cp                    # To create distributions
import numpy as np                      # For the time array
from scipy.integrate import odeint      # To integrate our equation
```

To create the model we define a Python function `coffee_cup` that takes the uncertain parameters `kappa` and `T_env` as input arguments. Inside this function we solve our equation by integrating it using `scipy.integrate.odeint`, before we return the results. The implementation of the model function is:

```
# Create the coffee cup model function
def coffee_cup(kappa, T_env):
    # Initial temperature and time array
    time = np.linspace(0, 200, 150)            # Minutes
    T_0 = 95                                   # Celsius

    # The equation describing the model
    def f(T, time, kappa, T_env):
        return -kappa*(T - T_env)

    # Solving the equation by integration
    temperature = odeint(f, T_0, time, args=(kappa, T_env))[:, 0]

    # Return time and model output
    return time, temperature
```

We could use this function directly in `UncertaintyQuantification`, but we would like to have labels on the axes when plotting. So we create a *Model* with the above run function and labels:

```
# Create a model from the coffee_cup function and add labels
model = un.Model(run=coffee_cup, labels=["Time (min)", "Temperature (C)"])
```

The next step is to define the uncertain parameters. We use Chaospy to create the distributions, and create a parameter dictionary:

```
# Create the distributions
kappa_dist = cp.Uniform(0.025, 0.075)
T_env_dist = cp.Uniform(15, 25)

# Define the parameter dictionary
parameters = {"kappa": kappa_dist, "T_env": T_env_dist}
```

We can now calculate the uncertainty and sensitivity using polynomial chaos expansions with point collocation, which is the default option of `quantify`. We set the seed to easier be able to reproduce the result.

```
# Set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)

# Perform the uncertainty quantification using
# polynomial chaos with point collocation (by default)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

The complete code becomes:

```
import uncertainpy as un
import chaospy as cp                       # To create distributions
import numpy as np                         # For the time array
from scipy.integrate import odeint         # To integrate our equation


# Create the coffee cup model function
def coffee_cup(kappa, T_env):
    # Initial temperature and time array
    time = np.linspace(0, 200, 150)            # Minutes
    T_0 = 95                                   # Celsius

```

(continues on next page)

(continued from previous page)

```
    # The equation describing the model
    def f(T, time, kappa, T_env):
        return -kappa*(T - T_env)

    # Solving the equation by integration
    temperature = odeint(f, T_0, time, args=(kappa, T_env))[:, 0]

    # Return time and model output
    return time, temperature


# Create a model from the coffee_cup function and add labels
model = un.Model(run=coffee_cup, labels=["Time (min)", "Temperature (C)"])

# Create the distributions
kappa_dist = cp.Uniform(0.025, 0.075)
T_env_dist = cp.Uniform(15, 25)

# Define the parameter dictionary
parameters = {"kappa": kappa_dist, "T_env": T_env_dist}

# Set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)

# Perform the uncertainty quantification using
# polynomial chaos with point collocation (by default)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

3.1.2 Using a class

The model can also be created as a class instead of using a function. Most of the code is unchanged. The complete for this example is in `examples/coffee_cup/uq_coffee_class.py`. We create a class that inherits from *Model*. To add the labels we call on the constructor of the parent class and give it the labels.

```
# Create the coffee cup model
class CoffeeCup(un.Model):
    # Add labels to the model by calling the constructor of the parent un.Model
    def __init__(self):
        super(CoffeeCup, self).__init__(labels=["Time (s)", "Temperature (C)"])
```

We can then implement the run method:

```
    # Define the run method
    def run(self, kappa, T_env):
        # Initial temperature and time array
        time = np.linspace(0, 200, 150)            # Minutes
        T_0 = 95                                   # Celsius

        # The equation describing the model
        def f(T, time, kappa, T_env):
            return -kappa*(T - T_env)

        # Solving the equation by integration
        temperature = odeint(f, T_0, time, args=(kappa, T_env))[:, 0]
```

(continues on next page)

(continued from previous page)

```
        # Return time and model output
        return time, temperature
```

Now, instead of creating a model from a model function, we initialize our `CoffeeCup` model:

```
# Initialize the model
model = CoffeeCup()
```

While the rest is unchanged:

```
# Create the distributions
kappa_dist = cp.Uniform(0.025, 0.075)
T_env_dist = cp.Uniform(15, 25)

# Define the parameters dictionary
parameters = {"kappa": kappa_dist, "T_env": T_env_dist}

# Set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)

# Perform the uncertainty quantification using
# polynomial chaos with point collocation (by default)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

3.2 A cooling coffee cup model with dependent parameters

Here we show an example (found in `examples/coffee_cup_dependent/uq_coffee_dependent_function.py`) where we examine a cooling coffee cup model with dependent parameters. We modify the *simple cooling coffee cup* model by introducing two auxillary variables α and $\hat{\kappa}$:

$$\kappa = \alpha\hat{\kappa}$$

to get:

$$\frac{dT(t)}{dt} = -\alpha\hat{\kappa}\left(T(t) - T_{env}\right).$$

The auxillary variables are made dependent by requiring that the model should be identical to the original model. We assume that α is an uncertain scaling factor:

$$\alpha = \mathrm{Uniform}(0.5, 1.5),$$

and set:

$$\hat{\kappa} = \frac{\kappa}{\alpha}.$$

Which gives us the following distributions:

$$\begin{aligned} \alpha &= \mathrm{Uniform}(0.5, 1.5) \\ \hat{\kappa} &= \frac{\mathrm{Uniform}(0.025, 0.075)}{\alpha} \\ T_{env} &= \mathrm{Uniform}(15, 25). \end{aligned}$$

With Chaospy we can create these dependencies using arithmetic operators:

```
# Create the distributions
T_env_dist = cp.Uniform(15, 25)
alpha_dist = cp.Uniform(0.5, 1.5)
kappa_hat_dist = cp.Uniform(0.025, 0.075)/alpha_dist

# Define the parameters dictionary
parameters = {"alpha": alpha_dist,
              "kappa_hat": kappa_hat_dist,
              "T_env": T_env_dist}
```

We can use this `parameters` dictionary directly when we set up the uncertainty quantification

```
# We can use the parameters dictionary directly
# when we set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)
```

The Rosenblatt transformation is by default automatically used we have the parameters that are dependent. We also set the seed to easier be able to reproduce the result.

```
# We can use the parameters dictionary directly
# when we set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)

# Perform the uncertainty quantification,
# which automatically use the Rosenblatt transformation
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

The complete code example become:

```
import uncertainpy as un
import chaospy as cp
import numpy as np
from scipy.integrate import odeint


# Create the coffee cup model function
def coffee_cup_dependent(kappa_hat, T_env, alpha):
    # Initial temperature and time
    time = np.linspace(0, 200, 150)            # Minutes
    T_0 = 95                                   # Celsius

    # The equation describing the model
    def f(T, time, alpha, kappa_hat, T_env):
        return -alpha*kappa_hat*(T - T_env)

    # Solving the equation by integration.
    temperature = odeint(f, T_0, time, args=(alpha, kappa_hat, T_env))[:, 0]

    # Return time and model results
    return time, temperature


# Create a model from the coffee_cup_dependent function and add labels
model = un.Model(coffee_cup_dependent, labels=["Time (s)", "Temperature (C)"])

# Create the distributions
```

(continues on next page)

(continued from previous page)

```
T_env_dist = cp.Uniform(15, 25)
alpha_dist = cp.Uniform(0.5, 1.5)
kappa_hat_dist = cp.Uniform(0.025, 0.075)/alpha_dist

# Define the parameters dictionary
parameters = {"alpha": alpha_dist,
              "kappa_hat": kappa_hat_dist,
              "T_env": T_env_dist}

# We can use the parameters dictionary directly
# when we set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model, parameters=parameters)

# Perform the uncertainty quantification,
# which automatically use the Rosenblatt transformation
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

In this case, the distribution we assign to α does not matter for the end result, as the distribution for $\hat{\kappa}$ will be scaled accordingly. Using the Rosenblatt transformation, an uncertainty quantification and sensitivity analysis of the dependent coffee cup model therefore returns the same results as seen in the simple coffee cup model, where the role of the original κ is taken over by $\hat{\kappa}$, while the sensitivity to the additional parameter α becomes strictly zero.

3.3 The Hodgkin-Huxley model

Here we examine the canonical Hodgkin-Huxley model (Hodgkin and Huxley, 1952). An uncertainty analysis of this model has been performed previously (Valderrama et al., 2015), and we here we repeat a part of that study using Uncertainpy.

The here used version of the Hodgkin-Huxley model has 11 parameters:

Parameter	Value	Unit	Meaning
V_0	-10	mV	Initial voltage
C_m	1	F/cm^2	Membrane capacitance
$\bar{g}_\mathrm{Na}$	120	mS/cm^2	Sodium (Na) conductance
$\bar{g}_\mathrm{K}$	36	mS/cm^2	Potassium (K) conductance
$\bar{g}_\mathrm{I}$	0.3	mS/cm^2	Leak current conductance
E_Na	112	mV	Sodium equilibrium potential
E_K	-12	mV	Potassium equilibrium potential
E_I	10.613	mV	Leak current equilibrium potential
n_0	0.0011		Initial potassium activation gating variable
m_0	0.0003		Initial sodium activation gating variable
h_0	0.9998		Initial sodium inactivation gating variable

As in the previous study, we assume each of these parameters have a uniform distribution in the range $\pm 10\%$ around their original value.

We use uncertainty quantification and sensitivity analysis to explore how this parameter uncertainty affect the model output, i.e., the action potential response of the neural membrane potential V_m to an external current injection. The model was exposed to a continuous external stimulus of $140\mu\mathrm{A/cm}^2$ starting at $t = 0$, and we examined the membrane potential in the time window between t = 5 and 15 ms

As in the *cooling coffee cup example*, we implement the Hodgkin-Huxley model as a Python function (found in /examples/valderrama/valderrama.py):

```
import uncertainpy as un

import numpy as np
from scipy.integrate import odeint


# External stimulus
def I(time):
    return 140 # micro A/cm**2


def valderrama(V_0=-10,
               C_m=1,
               gbar_Na=120,
               gbar_K=36,
               gbar_L=0.3,
               E_Na=112,
               E_K=-12,
               E_l=10.613,
               m_0=0.0011,
               n_0=0.0003,
               h_0=0.9998):

    # Setup time
    end_time = 15          # ms
    dt = 0.025             # ms
    time = np.arange(0, end_time + dt, dt)

    # K channel
    def alpha_n(V):
        return 0.01*(10 - V)/(np.exp((10 - V)/10.) - 1)


    def beta_n(V):
        return 0.125*np.exp(-V/80.)

    def n_f(n, V):
        return alpha_n(V)*(1 - n) - beta_n(V)*n

    def n_inf(V):
        return alpha_n(V)/(alpha_n(V) + beta_n(V))


    # Na channel (activating)
    def alpha_m(V):
        return 0.1*(25 - V)/(np.exp((25 - V)/10.) - 1)

    def beta_m(V):
        return 4*np.exp(-V/18.)

    def m_f(m, V):
        return alpha_m(V)*(1 - m) - beta_m(V)*m

    def m_inf(V):
        return alpha_m(V)/(alpha_m(V) + beta_m(V))
```

(continues on next page)

(continued from previous page)

```
    # Na channel (inactivating)
    def alpha_h(V):
        return 0.07*np.exp(-V/20.)

    def beta_h(V):
        return 1/(np.exp((30 - V)/10.) + 1)

    def h_f(h, V):
        return alpha_h(V)*(1 - h) - beta_h(V)*h

    def h_inf(V):
        return alpha_h(V)/(alpha_h(V) + beta_h(V))


    def dXdt(X, t):
        V, h, m, n = X

        g_Na = gbar_Na*(m**3)*h
        g_K = gbar_K*(n**4)
        g_l = gbar_L

        dmdt = m_f(m, V)
        dhdt = h_f(h, V)
        dndt = n_f(n, V)

        dVdt = (I(t) - g_Na*(V - E_Na) - g_K*(V - E_K) - g_l*(V - E_l))/C_m

        return [dVdt, dhdt, dmdt, dndt]


    initial_conditions = [V_0, h_0, m_0, n_0]

    X = odeint(dXdt, initial_conditions, time)
    values = X[:, 0]

    # Only return from 5 seconds onwards, as in the Valderrama paper
    values = values[time > 5]
    time = time[time > 5]

    # Add info needed by certain spiking features and efel features
    info = {"stimulus_start": time[0], "stimulus_end": time[-1]}

    return time, values, info
```

We use this function when we perform the uncertainty quantification and sensitivity analysis (found in `/examples/valderrama/uq_valderrama.py`). We first initialize our model:

```
# Initialize the model
model = un.Model(run=valderrama,
                 labels=["Time (ms)", "Membrane potential (mV)"])
```

Then we create the set of parameters:

```
# Define a parameter dictionary
```

(continues on next page)

(continued from previous page)

```
parameters = {"V_0": -10,
              "C_m": 1,
              "gbar_Na": 120,
              "gbar_K": 36,
              "gbar_L": 0.3,
              "m_0": 0.0011,
              "n_0": 0.0003,
              "h_0": 0.9998,
              "E_Na": 112,
              "E_K": -12,
              "E_l": 10.613}

# Create the parameters
parameters = un.Parameters(parameters)
```

We use `set_all_distributions()` and `uniform()` to give all parameters a uniform distribution in the range $\pm 10\%$ around their fixed value.

```
# Set all parameters to have a uniform distribution
# within a 20% interval around their fixed value
parameters.set_all_distributions(un.uniform(0.2))
```

`set_all_distributions` sets the distribution of all parameters. If it receives a function as input, it gives that function the fixed value of each parameter, and expects to receive Chaospy functions. `uniform` is a closure. It takes *interval* as input and returns a function which takes the *fixed_value* of each parameter as input and returns a Chaospy distribution with this *interval* around the *fixed_value*. Ultimately the distribution of each parameter is set to *interval* around their *fixed_value*:

```
cp.Uniform(fixed_value - abs(interval/2.*fixed_value),
           fixed_value + abs(interval/2.*fixed_value)).
```

We can now use polynomial chaos expansions with point collocation to calculate the uncertainty and sensitivity of the model. We also set the seed to easier be able to reproduce the result.

```
# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

The complete code for the uncertainty quantification and sensitivity becomes:

```
import uncertainpy as un
import chaospy as cp

from valderrama import valderrama

# Initialize the model
model = un.Model(run=valderrama,
                 labels=["Time (ms)", "Membrane potential (mV)"])

# Define a parameter dictionary
parameters = {"V_0": -10,
              "C_m": 1,
              "gbar_Na": 120,
              "gbar_K": 36,
```

(continues on next page)

(continued from previous page)

```
              "gbar_L": 0.3,
              "m_0": 0.0011,
              "n_0": 0.0003,
              "h_0": 0.9998,
              "E_Na": 112,
              "E_K": -12,
              "E_l": 10.613}

# Create the parameters
parameters = un.Parameters(parameters)

# Set all parameters to have a uniform distribution
# within a 20% interval around their fixed value
parameters.set_all_distributions(un.uniform(0.2))

# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

3.4 A multi-compartment model of a thalamic interneuron implemented in NEURON

In this example we illustrate how Uncertainpy can be used on models implemented in NEURON. For this example, we select a previously published model of an interneuron in the dorsal lateral geniculate nucleus Halnes et al., 2011. Since the model is in implemented in NEURON, the original model can be used directly with Uncertainpy with the use of *NeuronModel*. The code for this case study is found in `/examples/interneuron/uq_interneuron.py`. To be able to run this example you require both the NEURON simulator, as well as the interneuron model saved in the folder `/interneuron_model/`.

In the original modeling study, a set of 7 parameters were tuned manually through a series of trials and errors until the interneuron model obtained the desired response characteristics. The final parameter set is:

Parameter	Value	Unit	Neuron variable	Meaning
g_{Na}	0.09	S/cm^2	`gna`	Max Na$^+$-conductance in soma
g_{Kdr}	0.37	S/cm^2	`gkdr`	Max direct rectifying K$^+$-conductance in soma
g_{CaT}	1.17e-5	S/cm^2	`gcat`	Max T-type Ca^{2+}-conductance in soma
g_{CaL}	9e-4	S/cm^2	`gcal`	Max L-type Ca^{2+}-conductance in soma
g_{h}	1.1e-4	S/cm^2	`ghbar`	Max conductance of a non-specific hyperpolarization activated cation channel in soma
g_{AHP}	6.4e-5	S/cm^2	`gahp`	Max afterhyperpolarizing K$^+$-conductance in soma
g_{CAN}	2e-8	S/cm^2	`gcanbar`	Max conductance of a Ca^{2+}-activated non-specific cation channel in soma

To perform an uncertainty quantification and sensitivity analysis of this model, we assume each of these 7 parameters have a uniform uncertainty distribution in the interval $\pm 10\%$ around their original value. We create these parameters similar to how we did in the *Hodgkin-Huxley example*:

```
# Define a parameter list
parameters= {"gna": 0.09,
             "gkdr": 0.37,
             "gcat": 1.17e-5,
             "gcal": 0.0009,
             "ghbar": 0.00011,
             "gahp": 6.4e-5,
             "gcanbar": 2e-8}

# Create the parameters
parameters = un.Parameters(parameters)

# Set all parameters to have a uniform distribution
# within a 20% interval around their fixed value
parameters.set_all_distributions(un.uniform(0.2))
```

A point-to-point comparison of voltage traces is often uninformative, and we therefore want to perform a feature based analysis of the model. Since we examine a spiking neuron model, we choose the features in *SpikingFeatures*:

```
# Initialize the features
features = un.SpikingFeatures(features_to_run="all")
```

We study the response of the interneuron to a somatic current injection between 1000 ms $< t <$ 1900 ms. `SpikingFeatures` needs to know the start and end time of this stimulus to be able to calculate certain features. They are specified through the `stimulus_start` and `stimulus_end` arguments when initializing `NeuronModel`. Additionally, the interneuron model uses adaptive time steps, meaning we have to set `interpolate=True`. In this way we tell Uncertainpy to perform an interpolation to get the output on a regular form before performing the analysis: We also give the path to the folder where the neuron model is stored with `path="interneuron_model/"`. `NeuronModel` loads the NEURON model from `mosinit.hoc`, sets the parameters of the model, evaluates the model and returns the somatic membrane potential of the neuron, (the voltage of the section named `"soma"`). `NeuronModel` therefore does not require a model function.

```
# Initialize the model with the start and end time of the stimulus
model = un.NeuronModel(path="interneuron_model/", interpolate=True,
                       stimulus_start=1000, stimulus_end=1900)
```

We set up the problem, adding our features before we use polynomial chaos expansion with point collocation to compute the statistical metrics for the model output and all features. We also set the seed to easier be able to reproduce the result.

```
# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters,
                                  features=features)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

The complete code becomes:

```
import uncertainpy as un

# Define a parameter list
parameters= {"gna": 0.09,
             "gkdr": 0.37,
             "gcat": 1.17e-5,
             "gcal": 0.0009,
```

(continues on next page)

(continued from previous page)

```
              "ghbar": 0.00011,
              "gahp": 6.4e-5,
              "gcanbar": 2e-8}

# Create the parameters
parameters = un.Parameters(parameters)

# Set all parameters to have a uniform distribution
# within a 20% interval around their fixed value
parameters.set_all_distributions(un.uniform(0.2))

# Initialize the features
features = un.SpikingFeatures(features_to_run="all")

# Initialize the model with the start and end time of the stimulus
model = un.NeuronModel(path="interneuron_model/", interpolate=True,
                       stimulus_start=1000, stimulus_end=1900)

# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters,
                                  features=features)
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(seed=10)
```

3.5 A sparsely connected recurrent network using Nest

In the last case study, we use Uncertainpy to perform a feature based analysis of the sparsely connected recurrent network by Brunel (2000). We implement the Brunel network using NEST inside a Python function, and create 10000 inhibitory and 2500 excitatory neurons. We record the output from 20 of the excitatory neurons, and simulate the network for 1000 ms. This is the values used to create the results in the Uncertainpy paper. If you want to just test the network, we recommend reducing the model to 2000 inhibitory and 500 excitatory neurons, and only simulate the network for 100 ms. To be able to run this example you require NEST to be anle to run the model and `elephant`, `neo`, and `quantities` to be able to use the network features.

We want to use *NestModel* to create our model. `NestModel` requires the model function to be specified through the `run` argument, unlike `NeuronModel`. The NEST model function has the same requirements as a regular model function, except it is restricted to return only two objects: the final simulation time (`simulation_end`), and a list of spike times for each neuron in the network (`spiketrains`). `NestModel` then postproccess this result for us to a regular result. The final uncertainty quantification of a NEST network therefore predicts the probability for a spike to occur at any specific time point in the simulation. We implement the Brunel network as such a function (found in `/examples/brunel/brunel.py`):

```
import nest

def brunel_network(eta=2, g=2, delay=1.5, J=0.1):
    """
    A sparsely connected recurrent network (Brunel).

    Brunel N, Dynamics of Sparsely Connected Networks of Excitatory and
    Inhibitory Spiking Neurons, Journal of Computational Neuroscience 8,
    183-208 (2000).

```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    eta : {int, float}, optional
        External rate relative to threshold rate. Default is 2.
    g : {int, float}, optional
        Ratio inhibitory weight/excitatory weight. Default is 5.
    delay : {int, float}, optional
        Synaptic delay in ms. Default is 1.5.
    J : {int, float}, optional
        Amplitude of excitatory postsynaptic current. Default is 0.1

    Notes
    -----
    Brunel N, Dynamics of Sparsely Connected Networks of Excitatory and
    Inhibitory Spiking Neurons, Journal of Computational Neuroscience 8,
    183-208 (2000).
    """
    # Network parameters
    N_rec = 20             # Record from 20 neurons
    simulation_end = 1000  # Simulation time

    tau_m = 20.0           # Time constant of membrane potential in ms
    V_th = 20.0
    N_E = 10000            # Number of excitatory neurons
    N_I = 2500             # Number of inhibitory neurons
    N_neurons = N_E + N_I  # Number of neurons in total
    C_E = int(N_E/10)      # Number of excitatory synapses per neuron
    C_I = int(N_I/10)      # Number of inhibitory synapses per neuron
    J_I = -g*J             # Amplitude of inhibitory postsynaptic current
    cutoff = 100           # Cutoff to avoid transient effects, in ms

    nu_ex = eta*V_th/(J*C_E*tau_m)
    p_rate = 1000.0*nu_ex*C_E

    nest.ResetKernel()

    # Configure kernel
    nest.SetKernelStatus({"grng_seed": 10})

    nest.SetDefaults('iaf_psc_delta',
                     {'C_m': 1.0,
                      'tau_m': tau_m,
                      't_ref': 2.0,
                      'E_L': 0.0,
                      'V_th': V_th,
                      'V_reset': 10.0})

    # Create neurons
    nodes   = nest.Create('iaf_psc_delta', N_neurons)
    nodes_E = nodes[:N_E]
    nodes_I = nodes[N_E:]

    noise = nest.Create('poisson_generator',1,{'rate': p_rate})

    spikes = nest.Create('spike_detector',2,
                         [{'label': 'brunel-py-ex'},
                          {'label': 'brunel-py-in'}])
```

(continues on next page)

(continued from previous page)

```
    spikes_E = spikes[:1]
    spikes_I = spikes[1:]


    # Connect neurons to each other
    nest.CopyModel('static_synapse_hom_w', 'excitatory',
                   {'weight':J, 'delay':delay})
    nest.Connect(nodes_E, nodes,
                 {'rule': 'fixed_indegree', 'indegree': C_E},
                 'excitatory')

    nest.CopyModel('static_synapse_hom_w', 'inhibitory',
                   {'weight': J_I, 'delay': delay})
    nest.Connect(nodes_I, nodes,
                {'rule': 'fixed_indegree', 'indegree': C_I},
                 'inhibitory')



    # Connect poisson generator to all nodes
    nest.Connect(noise, nodes, syn_spec='excitatory')

    nest.Connect(nodes_E[:N_rec], spikes_E)
    nest.Connect(nodes_I[:N_rec], spikes_I)


    # Run the simulation
    nest.Simulate(simulation_end)


    events_E = nest.GetStatus(spikes_E, 'events')[0]
    events_I = nest.GetStatus(spikes_I, 'events')[0]


    # Excitatory spike trains
    # Makes sure the spiketrain is added even if there are no results
    # to get a regular result
    spiketrains = []
    for sender in nodes_E[:N_rec]:
        spiketrain = events_E["times"][events_E["senders"] == sender]
        spiketrain = spiketrain[spiketrain > cutoff] - cutoff
        spiketrains.append(spiketrain)

    simulation_end -= cutoff

    return simulation_end, spiketrains
```

And use it to create our model (example found in `/examples/brunel/uq_brunel.py`): We set `ignore=True` since we are not interested in the model result itself. This is recommended for NEST models as long as you do not need the model results, since the uncertainty calculations for the for the model results require much time and memory.

```
# Create a Nest model from the brunel network function
# We set ``ignore=True`` since we are not interested in
```

(continues on next page)

(continued from previous page)

```
# the model result itself.
# This is recommended for NEST models as long as you do not
# need the model results, since the uncertainty calculations for the
# for the model results require much time and memory.
model = un.NestModel(run=brunel_network, ignore=True)
```

The Brunel model has four uncertain parameters:

1. the external rate (ν_{ext}) relative to threshold rate (ν_{thr}) given as $\eta = \nu_{\text{ext}}/\nu_{\text{thr}}$,
2. the relative strength of the inhibitory synapses g,
3. the synaptic delay D, and
4. the amplitude of excitatory postsynaptic current J_e.

Depending on the parameterizations of the model, the Brunel network may be in several different activity states. For the current example, we limit our analysis to two of these states. We create two sets of parameters, one for each of two states, and assume the parameter uncertainties are characterized by uniform probability distributions within the ranges below:

Parameter	Range SR	Range AI	Variable	Meaning
η	$[1.5, 3.5]$	$[1.5, 3.5]$	`eta`	External rate relative to threshold rate
g	$[1, 3]$	$[5, 8]$	`g`	Relative strength of inhibitory synapses
D	$[1.5, 3]$	$[1.5, 3]$	`delay`	Synaptic delay (ms)
J_e	$[0.05, 0.15]$	$[0.05, 0.15]$	`J_e`	Amplitude excitatory postsynaptic current (mV)

These ranges correspond to the synchronous regular (SR) state, where the neurons are almost completely synchronized, and the asynchronous irregular (AI) state, where the neurons fire individually at low rates. We create two sets of parameters, one for each state:

```
# Parametes for the synchronous regular (SR) state
parameters = {"eta": cp.Uniform(1.5, 3.5),
              "g": cp.Uniform(1, 3),
              "delay": cp.Uniform(1.5, 3)}
parameters_SR = un.Parameters(parameters)

# Parameter for the asynchronous irregular (AI) state
parameters = {"eta": cp.Uniform(1.5, 2.2),
              "g": cp.Uniform(5, 8),
              "delay": cp.Uniform(1.5, 3)}
parameters_AI = un.Parameters(parameters)
```

We use the features in *NetworkFeatures* to examine features of the Brunel network.

```
features = un.NetworkFeatures()
```

We set up the problems with the SR parameter set and use polynomial chaos with point collocation to perform the uncertainty quantification and sensitivity analysis. We specify a filename for the data, and a folder where to save the figures, to keep the results from the AI and SR state separated. We also set the seed to easier be able to reproduce the result.

```
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters_SR,
                                  features=features)

# Perform uncertainty quantification
# and save the data and plots under their own name
# We set the seed to easier be able to reproduce the result
UQ.quantify(figure_folder="figures_brunel_SR",
            filename="brunel_SR",
            seed=10)
```

We then change the parameters, and perform the uncertainty quantification and sensitivity analysis for the new set of parameters, again specifying a filename and figure folder.

```
# Change the set of parameters
UQ.parameters = parameters_AI

# Perform uncertainty quantification on the new parameter set
# and save the data and plots under their own name
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(figure_folder="figures_brunel_AI",
                   filename="brunel_AI",
                   seed=10)
```

The complete code is:

```
import uncertainpy as un
import chaospy as cp

from brunel import brunel_network

# Create a Nest model from the brunel network function
# We set ``ignore=True`` since we are not interested in
# the model result itself.
# This is recommended for NEST models as long as you do not
# need the model results, since the uncertainty calculations for the
# for the model results require much time and memory.
model = un.NestModel(run=brunel_network, ignore=True)


# Parametes for the synchronous regular (SR) state
parameters = {"eta": cp.Uniform(1.5, 3.5),
              "g": cp.Uniform(1, 3),
              "delay": cp.Uniform(1.5, 3)}
parameters_SR = un.Parameters(parameters)

# Parameter for the asynchronous irregular (AI) state
parameters = {"eta": cp.Uniform(1.5, 2.2),
              "g": cp.Uniform(5, 8),
              "delay": cp.Uniform(1.5, 3)}
parameters_AI = un.Parameters(parameters)

# Initialize network features
features = un.NetworkFeatures()

# Set up the problem
```

(continues on next page)

(continued from previous page)

```
UQ = un.UncertaintyQuantification(model,
                                  parameters=parameters_SR,
                                  features=features)

# Perform uncertainty quantification
# and save the data and plots under their own name
# We set the seed to easier be able to reproduce the result
UQ.quantify(figure_folder="figures_brunel_SR",
            filename="brunel_SR",
            seed=10)


# Change the set of parameters
UQ.parameters = parameters_AI

# Perform uncertainty quantification on the new parameter set
# and save the data and plots under their own name
# We set the seed to easier be able to reproduce the result
data = UQ.quantify(figure_folder="figures_brunel_AI",
                   filename="brunel_AI",
                   seed=10)
```

3.6 A layer 5 pyramidal neuron implemented with NEURON

In this example we illustrate how we can subclass a *NeuronModel* to customize the methods. We select a set of reduced models of layer 5 pyramidal neurons (Bahl et al., 2012). The code for this example is found in `/examples/bahl/uq_bahl.py`. To be able to run this example you require both the NEURON simulator, as well as the layer 5 pyramidal neuron model saved in the folder `/bahl_model/`.

Since the model is implemented in NEURON, we use the *NeuronModel*. The problem is that this model require us to recalculate certain properties of the model after the parameters have been set. We therefore have to make change to the `NeuronModel` class so we recalculate these properties. The standard `run()` method implemented in `NeuronModel` calls `set_parameters()` to set the parameters. We therefore only need to change this method in the `NeuronModel`. First we subclass `NeuronModel`. For ease of use, we hardcode in the path to the Bahl model.

```
# Subclassing NeuronModel
class NeuronModelBahl(un.NeuronModel):
    def __init__(self, stimulus_start=None, stimulus_end=None):
        # Hardcode the path of the Bahl neuron model
        super(NeuronModelBahl, self).__init__(interpolate=True,
                                              path="bahl_model",
                                              stimulus_start=stimulus_start,
                                              stimulus_end=stimulus_end)
```

We then implement a new `set_parameters` method, that recalculates the required properties after the parameters have been set.

```
    # Reimplement the set_parameters method used by run
    def set_parameters(self, parameters):
        for parameter in parameters:
            self.h(parameter + " = " + str(parameters[parameter]))

        # These commands must be added for this specific
        # model to recalculate the parameters after they have been set
```

(continues on next page)

(continued from previous page)

```
        self.h("recalculate_passive_properties()")
        self.h("recalculate_channel_densities()")
```

Now we can initialize our new model.

```
# Initialize the model with the start and end time of the stimulus
model = NeuronModelBahl(stimulus_start=100, stimulus_end=600)
```

We can then create the uncertain parameters, which we here set to be `"e_pas"` and `"apical Ra"`. Here we do not create a *Parameter* object, but use the parameter list directly, to show that this option exists.

```
# Define a parameter list and use it directly
parameters = {"e_pas": cp.Uniform(-60, -85),
              "apical Ra": cp.Uniform(150, 300)}
```

The we use *SpikingFeatures*.

```
# Initialize the features
features = un.SpikingFeatures()
```

Lastly we set up and perform the uncertainty quantification and sensitivity analysis.

```
# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model,
                                  parameters=parameters,
                                  features=features)
data = UQ.quantify()
```

The complete code becomes:

```
import uncertainpy as un
import chaospy as cp

# Subclassing NeuronModel
class NeuronModelBahl(un.NeuronModel):
    def __init__(self, stimulus_start=None, stimulus_end=None):
        # Hardcode the path of the Bahl neuron model
        super(NeuronModelBahl, self).__init__(interpolate=True,
                                              path="bahl_model",
                                              stimulus_start=stimulus_start,
                                              stimulus_end=stimulus_end)

    # Reimplement the set_parameters method used by run
    def set_parameters(self, parameters):
        for parameter in parameters:
            self.h(parameter + " = " + str(parameters[parameter]))

        # These commands must be added for this specific
        # model to recalculate the parameters after they have been set
        self.h("recalculate_passive_properties()")
        self.h("recalculate_channel_densities()")


# Initialize the model with the start and end time of the stimulus
model = NeuronModelBahl(stimulus_start=100, stimulus_end=600)

```

(continues on next page)

(continued from previous page)

```
# Define a parameter list and use it directly
parameters = {"e_pas": cp.Uniform(-60, -85),
              "apical Ra": cp.Uniform(150, 300)}

# Initialize the features
features = un.SpikingFeatures()

# Perform the uncertainty quantification
UQ = un.UncertaintyQuantification(model=model,
                                  parameters=parameters,
                                  features=features)
data = UQ.quantify()
```

CHAPTER 4

Frequently asked questions

Here is a collection of frequently asked questions.

4.1 Is Uncertainpy usable with multiple model outputs?

Yes, however it does unfortunately not have direct support for this. Uncertainpy by default only performs an uncertainty quantification of the first model output returned. But you can return the additional model outputs in the info dictionary, and then define new features that extract each model output from the info dictionary, see the code example in *Multiple model outputs*.

CHAPTER 5

UncertaintyQuantification

The *uncertainpy.UncertaintyQuantification* class is used to define the problem, perform the uncertainty quantification and sensitivity analysis, and save and visualize the results. `UncertaintyQuantification` combines the three main components required to perform an uncertainty quantification and sensitivity analysis:

- The **model** we want to examine.
- The **parameters** of the model.
- Specifications of **features** in the model output.

The model and parameters are required components, while the feature specifications are optional.

Among others, UncertaintyQuantification takes the arguments:

```
UQ = un.UncertaintyQuantification(
        model=Model(...),                       # Required
        parameters=Parameters(...),             # Required
        features=Features(...)                  # Optional
)
```

The arguments are given as instances of their corresponding Uncertainpy classes (*Models*, *Parameters*, and *Features*).

After the problem is set up, an uncertainty quantification and sensitivity analysis can be performed by using the *uncertainpy.UncertaintyQuantification.quantify()* method. Among others, `quantify` takes the optional arguments:

```
data = UQ.quantify(
    method="pc"|"mc",
    pc_method="collocation"|"spectral",
    rosenblatt=False|True
)
```

The *method* argument allows the user to choose whether Uncertainpy should use polynomial chaos expansions (`"pc"`) or quasi-Monte Carlo (`"mc"`) methods to calculate the relevant statistical metrics. If polynomial chaos expansions are chosen, *pc_method* further specifies whether point collocation (`"collocation"`) or spectral projection (`"spectral"`) methods are used to calculate the expansion coefficients. Finally, *rosenblatt* (`False` or `True`) determines if the Rosenblatt transformation should be used. If nothing is specified, Uncertainpy by default uses polynomial chaos expansions based on point collocation without the Rosenblatt transformation.

The results from the uncertainty quantification are returned in `data`, as a `Data` object(see *Data*). The results are also automatically saved in a folder named `data`, and figures automatically plotted and saved in a folder named `figures`, both in the current directory. The returned `data` object is therefore not necessary to use.

Polynomial chaos expansions are recommended as long as the number of uncertain parameters is small (typically > 20), as polynomial chaos expansions in these cases are much faster than quasi-Monte Carlo methods. Additionally, sensitivity analysis is currently not yet available for studies based on the quasi-Monte Carlo method. Which of the polynomial chaos expansions methods to choose is problem dependent, but in general the pseudo-spectral method is faster than point collocation, but has lower stability. We therefore generally recommend the point collocation method.

We note that there is no guarantee each set of sampled parameters produces a valid model or feature output. For example, a feature such as the spike width will not be defined in a model evaluation that produces no spikes. In such cases, Uncertainpy gives a warning which includes the number of runs that failed to return a valid output, and performs the uncertainty quantification and sensitivity analysis using the reduced set of valid runs. Point collocation (as well as the quasi-Monte Carlo method) are robust towards missing values as long as the number of results remaining is high enough, another reason the point collocation method is recommend. However, if a large fraction of the simulations fail, the user could consider redefining the problem (e.g., by using narrower parameter distributions).

5.1 API Reference

class `uncertainpy.`**`UncertaintyQuantification`**(*model*, *parameters*, *features=None*, *uncertainty_calculations=None*, *create_PCE_custom=None*, *custom_uncertainty_quantification=None*, *CPUs=u'max'*, *logger_level=u'info'*, *logger_filename=u'uncertainpy.log'*, *backend=u'auto'*)

Perform an uncertainty quantification and sensitivity analysis of a model and features of the model.

It implements both quasi-Monte Carlo methods and polynomial chaos expansions using either point collocation or the pseudo-spectral method. Both of the polynomial chaos expansion methods have support for the rosenblatt transformation to handle dependent input parameters.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **parameters** (*{None, Parameters instance, list of Parameter instances, list with [[name, value, distribution], . . .]}*) – Either None, a Parameters instance or a list of the parameters that should be created. The two lists are similar to the arguments sent to Parameters. Default is None.
- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated. Default is None.
- **uncertainty_calculations** (*UncertaintyCalculations or UncertaintyCalculations subclass instance, optional*) – An UncertaintyCalculations class or subclass that implements (custom) uncertainty quantification and sensitivity analysis methods.
- **create_PCE_custom** (*callable, optional*) – A custom method for calculating the polynomial chaos approximation. For the requirements of the function see

`UncertaintyCalculations.create_PCE_custom`. Overwrites existing `create_PCE_custom` method. Default is None.

- **custom_uncertainty_quantification** (*callable, optional*) – A custom method for calculating uncertainties. For the requirements of the function see `UncertaintyCalculations.custom_uncertainty_quantification`. Overwrites existing `custom_uncertainty_quantification` method. Default is None.
- **CPUs** (*{int, None, "max"}, optional*) – The number of CPUs to use when calculating the model and features. If None, no multiprocessing is used. If "max", the maximum number of CPUs on the computer (multiprocess.cpu_count()) is used. Default is "max".
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed Default logger level is "info".
- **logger_filename** (*str*) – Name of the logfile. If None, no logging to file is performed. Default is "uncertainpy.log".
- **backend** (*{"auto", "hdf5", "exdir"}, optional*) – The fileformat used to save and load data to/from file. "auto" assumes the filenames ends with either ".h5" for HDF5 files or ".exdir" for Exdir files. If unknown fileextension defaults to saving data as HDF5 files. "hdf5" saves and loads files from HDF5 files. "exdir" saves and loads files from Exdir files. Default is "auto".

Variables

- **`model`** (*`Model or Model subclass`*) – The model to perform uncertainty quantification on.
- **`parameters`** (*`Parameters`*) – The uncertain parameters.
- **`features`** (*`Features or Features subclass`*) – The features of the model to perform uncertainty quantification on.
- **`uncertainty_calculations`** (*`UncertaintyCalculations or UncertaintyCalculations subclass`*) – UncertaintyCalculations object responsible for performing the uncertainty quantification calculations.
- **`data`** (*`Data`*) – A data object that contains the results from the uncertainty quantification. Contains all model and feature evaluations, as well as all calculated statistical metrics.

Raises `ValueError` – If unsupported backend is chosen.

See also:

`uncertainpy.features`, *`uncertainpy.Parameter`*, *`uncertainpy.Parameters`*, `uncertainpy.models`, *`uncertainpy.core.UncertaintyCalculations`*

`uncertainpy.core.UncertaintyCalculations.create_PCE_custom` Requirements for create_PCE_custom

`uncertainpy.models.Model.run` Requirements for the model run function.

`custom_uncertainty_quantification`(*plot=u'condensed_first'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *save=True*, *data_folder=u'data'*, *filename=None*, ***custom_kwargs*)

Perform a custom uncertainty quantification and sensitivity analysis, implemented by the user.

Parameters

- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".

- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.
- ****custom_kwargs** – Any number of arguments for the custom uncertainty quantification.

Raises `NotImplementedError` – If the custom uncertainty quantification method have not been implemented.

Notes

For details on how to implement the custom uncertainty quantification method see UncertaintyCalculations.custom_uncertainty_quantification.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

See also:

`uncertainpy.plotting.PlotUncertainty()`, *`uncertainpy.Parameters()`*

`uncertainpy.core.UncertaintyCalculations.custom_uncertainty_quantification()`
Requirements for custom_uncertainty_quantification

`features`
Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

`uncertainpy.features.Features`, *`uncertainpy.features.GeneralSpikingFeatures`*, *`uncertainpy.features.SpikingFeatures`*, *`uncertainpy.features.GeneralNetworkFeatures`*, *`uncertainpy.features.NetworkFeatures`*

`load`(*filename*)
Load data from disk.

Parameters **filename** (*str*) – Name of the stored data file.

See also:

uncertainpy.Data() Data class

model

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

uncertainpy.models.Model, *uncertainpy.models.Model.run*, *uncertainpy.models.NestModel*, *uncertainpy.models.NeuronModel*

monte_carlo(*uncertain_parameters=None*, *nr_samples=10000*, *seed=None*, *plot=u'condensed_first'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *save=True*, *data_folder=u'data'*, *filename=None*)

Perform an uncertainty quantification using the quasi-Monte Carlo method.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when performing the uncertainty quantification. If None, all uncertain parameters are used. Default is None.
- **nr_samples** (*int, optional*) – Number of samples for the quasi-Monte Carlo sampling. *nr_samples* is used for the uncertainty quantification and `(nr_samples/2)*(nr_uncertain_parameters + 2)` samples is used for the sensitivity analysis. Default *nr_samples* is 10**4.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.
- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.

Returns **data** – A data object that contains the results from the uncertainty quantification. Contains all model and feature evaluations, as well as all calculated statistical metrics.

Return type Data

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

Which method to choose is problem dependent, but as long as the number of uncertain parameters is low (less than around 20 uncertain parameters) polynomial chaos methods are much faster than Monte Carlo methods. Above this Monte Carlo methods are the best.

In the quasi-Monte Carlo method we quasi-randomly draw `(nr_samples/2)*(nr_uncertain_parameters + 2)` (nr_samples=10**4 by default) parameter samples using Saltelli's sampling scheme. We require this number of samples to be able to calculate the Sobol indices. We evaluate the model for each of these parameter samples and calculate the features from each of the model results. This step is performed in parallel to speed up the calculations. Then we use *nr_samples* of the model and feature results to calculate the mean, variance, and 5th and 95th percentile for the model and each feature. Lastly, we use all calculated model and each feature results to calculate the Sobol indices using Saltellie's approach.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

Sensitivity analysis is currently not yet available for the quasi-Monte Carlo method.

See also:

`uncertainpy.Data()`, *`uncertainpy.Parameters()`*, *`uncertainpy.plotting.PlotUncertainty()`*

`uncertainpy.core.UncertaintyCalculations.monte_carlo()` Uncertainty quantification using quasi-Monte Carlo methods

`monte_carlo_single`(*uncertain_parameters=None*, *nr_samples=10000*, *seed=None*, *plot=u'condensed_first'*, *save=True*, *data_folder=u'data'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *filename=None*)

Perform an uncertainty quantification for a single parameter at the time using the quasi-Monte Carlo method.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when performing the uncertainty quantification. If None, all uncertain parameters are used. Default is None.
- **nr_samples** (*int, optional*) – Number of samples for the quasi-Monte Carlo sampling. *nr_samples* is used for the uncertainty quantification and `(nr_samples/2)*(nr_uncertain_parameters + 2)` samples is used for the sensitivity analysis. Default *nr_samples* is 10**4.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.
- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".

- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.

Returns **data_dict** – A dictionary that contains the data objects for each single parameter calculation.

Return type dict

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

Which method to choose is problem dependent, but as long as the number of uncertain parameters is low (less than around 20 uncertain parameters) polynomial chaos methods are much faster than Monte Carlo methods. Above this Monte Carlo methods are the best.

In the quasi-Monte Carlo method we quasi-randomly draw `(nr_samples/2)*(nr_uncertain_parameters + 2)` (nr_samples=10**4 by default) parameter samples using Saltelli's sampling scheme. We require this number of samples to be able to calculate the Sobol indices. We evaluate the model for each of these parameter samples and calculate the features from each of the model results. This step is performed in parallel to speed up the calculations. Then we use *nr_samples* of the model and feature results to calculate the mean, variance, and 5th and 95th percentile for the model and each feature. Lastly, we use all calculated model and each feature results to calculate the Sobol indices using Saltellie's approach.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

Sensitivity analysis is currently not yet available for the quasi-Monte Carlo method.

See also:

`uncertainpy.Data()`, *`uncertainpy.plotting.PlotUncertainty()`*, *`uncertainpy.Parameters()`*

`uncertainpy.core.UncertaintyCalculations.monte_carlo()` Uncertainty quantification using quasi-Monte Carlo methods

`parameters`

Model parameters.

Parameters **new_parameters** (*{None, Parameters instance, list of Parameter instances, list [[name, value, distribution], . . .]}*) – Either None, a Parameters instance or a list of the parameters that should be created. The two lists are similar to the arguments sent to Parameters. Default is None.

Returns **parameters** – Parameters of the model. If None, no parameters have been set.

Return type {None, Parameters}

See also:

`uncertainpy.Parameter`, *`uncertainpy.Parameters`*

plot (*type=u'condensed_first'*, *folder=u'figures'*, *figureformat=u'.png'*)

Create plots for the results of the uncertainty quantification and sensitivity analysis. `self.data` must exist and contain the results.

Parameters

- **data** (*Data*) – A data object that contains the results from the uncertainty quantification.
- **type** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **folder** (*str*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".

Notes

These plots are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

See also:

`uncertainpy.Data()`, `uncertainpy.plotting.PlotUncertainty()`

polynomial_chaos (*method=u'collocation'*, *rosenblatt=u'auto'*, *uncertain_parameters=None*, *polynomial_order=4*, *nr_collocation_nodes=None*, *quadrature_order=None*, *nr_pc_mc_samples=10000*, *allow_incomplete=True*, *seed=None*, *plot=u'condensed_first'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *save=True*, *data_folder=u'data'*, *filename=None*, ***custom_kwargs*)

Perform an uncertainty quantification and sensitivity analysis using polynomial chaos expansions.

Parameters

- **method** (*{"collocation", "spectral", "custom"}, optional*) – The method to use when creating the polynomial chaos approximation, if the polynomial chaos method is chosen. "collocation" is the point collocation method "spectral" is pseudo-spectral projection, and "custom" is the custom polynomial method. Default is "collocation".
- **rosenblatt** (*{"auto", bool}, optional*) – If the Rosenblatt transformation should be used. The Rosenblatt transformation must be used if the uncertain parameters have dependent variables. If "auto" the Rosenblatt transformation is used if there are dependent parameters, and it is not used of the parameters have independent distributions. Default is "auto".
- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when performing the uncertainty quantification. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose, if polynomial chaos with point collocation is used. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.

- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method, if polynomial chaos with pseudo-spectral projection is used. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **nr_pc_mc_samples** (*int, optional*) – Number of samples for the Monte Carlo sampling of the polynomial chaos approximation, if the polynomial chaos method is chosen.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.
- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.
- ****custom_kwargs** – Any number of arguments for the custom polynomial chaos method, `create_PCE_custom`.

Returns **data** – A data object that contains the results from the uncertainty quantification. Contains all model and feature evaluations, as well as all calculated statistical metrics.

Return type Data

Raises

- `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.
- `ValueError` – If *method* not one of "collocation", "spectral" or "custom".
- `NotImplementedError` – If custom pc method is chosen and have not been implemented.

Notes

Which method to choose is problem dependent, but as long as the number of uncertain parameters is low (less than around 20 uncertain parameters) polynomial chaos methods are much faster than Monte Carlo methods. Above this Monte Carlo methods are the best.

For polynomial chaos, the pseudo-spectral method is faster than point collocation, but has lower stability. We therefore generally recommend the point collocation method.

The model and feature do not necessarily give results for each node. The collocation method are robust towards missing values as long as the number of results that remain is high enough. The pseudo-spectral method on the other hand, is sensitive to missing values, so *allow_incomplete* should be used with care in that case.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

Changing the parameters of the polynomial chaos methods should be done with care, and implementing custom methods is only recommended for experts.

See also:

`uncertainpy.Data()`, `uncertainpy.Parameters()`, `uncertainpy.plotting.PlotUncertainty()`

`uncertainpy.core.UncertaintyCalculations.polynomial_chaos()` Uncertainty quantification using polynomial chaos expansions

`uncertainpy.core.UncertaintyCalculations.create_PCE_custom()` Requirements for create_PCE_custom

`polynomial_chaos_single`(*method=u'collocation'*, *rosenblatt=u'auto'*, *polynomial_order=4*, *uncertain_parameters=None*, *nr_collocation_nodes=None*, *quadrature_order=None*, *nr_pc_mc_samples=10000*, *allow_incomplete=True*, *seed=None*, *plot=u'condensed_first'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *save=True*, *data_folder=u'data'*, *filename=None*)

Perform an uncertainty quantification and sensitivity analysis for a single parameter at the time using polynomial chaos expansions.

Parameters

- **method** (*{"collocation", "spectral", "custom"}, optional*) – The method to use when creating the polynomial chaos approximation, if the polynomial chaos method is chosen. "collocation" is the point collocation method "spectral" is pseudo-spectral projection, and "custom" is the custom polynomial method. Default is "collocation".
- **rosenblatt** (*{"auto", bool}, optional*) – If the Rosenblatt transformation should be used. The Rosenblatt transformation must be used if the uncertain parameters have dependent variables. If "auto" the Rosenblatt transformation is used if there are dependent parameters, and it is not used of the parameters have independent distributions. Default is "auto".
- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to performing the uncertainty quantification for. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose, if polynomial chaos with point collocation is used. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.
- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method, if polynomial chaos with pseudo-spectral projection is used. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **nr_pc_mc_samples** (*int, optional*) – Number of samples for the Monte Carlo sampling of the polynomial chaos approximation, if the polynomial chaos method is chosen.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.

- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.
- ****custom_kwargs** – Any number of arguments for the custom polynomial chaos method, `create_PCE_custom`.

Returns **data_dict** – A dictionary that contains the data for each single parameter calculation.

Return type dict

Raises

- `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.
- `ValueError` – If *method* not one of "collocation", "spectral" or "custom".
- `NotImplementedError` – If custom pc method is chosen and have not been implemented.

Notes

Which method to choose is problem dependent, but as long as the number of uncertain parameters is low (less than around 20 uncertain parameters) polynomial chaos methods are much faster than Monte Carlo methods. Above this Monte Carlo methods are the best.

For polynomial chaos, the pseudo-spectral method is faster than point collocation, but has lower stability. We therefore generally recommend the point collocation method.

The model and feature do not necessarily give results for each node. The collocation method are robust towards missing values as long as the number of results that remain is high enough. The pseudo-spectral method on the other hand, is sensitive to missing values, so *allow_incomplete* should be used with care in that case.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

Changing the parameters of the polynomial chaos methods should be done with care, and implementing custom methods is only recommended for experts.

See also:

`uncertainpy.Data()`, *`uncertainpy.Parameters()`*, *`uncertainpy.plotting.PlotUncertainty()`*

`uncertainpy.core.UncertaintyCalculations.polynomial_chaos()` Uncertainty quantification using polynomial chaos expansions

`uncertainpy.core.UncertaintyCalculations.create_PCE_custom()` Requirements for create_PCE_custom

quantify(*method=u'pc'*, *pc_method=u'collocation'*, *rosenblatt=u'auto'*, *uncertain_parameters=None*, *polynomial_order=4*, *nr_collocation_nodes=None*, *quadrature_order=None*, *nr_pc_mc_samples=10000*, *nr_mc_samples=10000*, *allow_incomplete=True*, *seed=None*, *single=False*, *plot=u'condensed_first'*, *figure_folder=u'figures'*, *figureformat=u'.png'*, *save=True*, *data_folder=u'data'*, *filename=None*, ***custom_kwargs*)

Perform an uncertainty quantification and sensitivity analysis using polynomial chaos expansions or quasi-Monte Carlo methods.

Parameters

- **method** (*{"pc", "mc", "custom"}, optional*) – The method to use when performing the uncertainty quantification and sensitivity analysis. "pc" is polynomial chaos method, "mc" is the quasi-Monte Carlo method and "custom" are custom uncertainty quantification methods. Default is "pc".
- **pc_method** (*{"collocation", "spectral", "custom"}, optional*) – The method to use when creating the polynomial chaos approximation, if the polynomial chaos method is chosen. "collocation" is the point collocation method "spectral" is pseudo-spectral projection, and "custom" is the custom polynomial method. Default is "collocation".
- **rosenblatt** (*{"auto", bool}, optional*) – If the Rosenblatt transformation should be used. The Rosenblatt transformation must be used if the uncertain parameters have dependent variables. If "auto" the Rosenblatt transformation is used if there are dependent parameters, and it is not used of the parameters have independent distributions. Default is "auto".
- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when performing the uncertainty quantification. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose, if polynomial chaos with point collocation is used. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.
- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method, if polynomial chaos with pseudo-spectral projection is used. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **nr_pc_mc_samples** (*int, optional*) – Number of samples for the Monte Carlo sampling of the polynomial chaos approximation, if the polynomial chaos method is chosen. Default is 10**4.
- **nr_mc_samples** (*int, optional*) – Number of samples for the quasi-Monte Carlo sampling, if the quasi-Monte Carlo method is chosen. *nr_mc_samples* is used for the uncertainty quantification and `(nr_mc_samples/2)*(nr_uncertain_parameters + 2)` samples is used for the sensitivity analysis. Default *nr_mc_samples* is 10**4.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.

- **single** (*bool*) – If an uncertainty quantification should be performed with only one uncertain parameter at the time. Requires that the values of each parameter is set. Default is False.
- **plot** (*{"condensed_first", "condensed_total", "condensed_no_sensitivity", "all", "evaluations", None}, optional*) – Type of plots to be created. "condensed_first" is a subset of the most important plots and only plots each result once, and contains plots of the first order Sobol indices. "condensed_total" is similar, but with the total order Sobol indices, and "condensed_no_sensitivity" is the same without any Sobol indices plotted. "all" creates every plot. "evaluations" plots the model and feature evaluations. None plots nothing. Default is "condensed_first".
- **figure_folder** (*str, optional*) – Name of the folder where to save all figures. Default is "figures".
- **figureformat** (*str*) – The figure format to save the plots in. Supports all formats in matplolib. Default is ".png".
- **save** (*bool, optional*) – If the data should be saved. Default is True.
- **data_folder** (*str, optional*) – Name of the folder where to save the data. Default is "data".
- **filename** (*{None, str}, optional*) – Name of the data file. If None the model name is used. Default is None.
- ****custom_kwargs** – Any number of arguments for either the custom polynomial chaos method, `create_PCE_custom`, or the custom uncertainty quantification, `custom_uncertainty_quantification`.

Returns **data** – A data object that contains the results from the uncertainty quantification. Contains all model and feature evaluations, as well as all calculated statistical metrics. If *single* = True, then returns a dictionary that contains the data objects for each single parameter calculation.

Return type Data, dict containing data objects

Raises

- `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.
- `ValueError` – If *method* not one of "pc", "mc" or "custom".
- `ValueError` – If *pc_method* not one of "collocation", "spectral" or "custom".
- `NotImplementedError` – If custom method or custom pc method is chosen and have not been implemented.

Notes

Which method to choose is problem dependent, but as long as the number of uncertain parameters is low (less than around 20 uncertain parameters) polynomial chaos methods are much faster than Monte Carlo methods. Above this Monte Carlo methods are the best.

For polynomial chaos, the pseudo-spectral method is faster than point collocation, but has lower stability. We therefore generally recommend the point collocation method.

The model and feature do not necessarily give results for each node. The collocation method and quasi-Monte Carlo methods are robust towards missing values as long as the number of results that remain is high enough. The pseudo-spectral method on the other hand, is sensitive to missing values, so *allow_incomplete* should be used with care in that case.

In the quasi-Monte Carlo method we quasi-randomly draw `(nr_mc_samples/2)*(nr_uncertain_parameters + 2)` (nr_mc_samples=10**4 by default) parameter samples using Saltelli's sampling scheme. We require this number of samples to be able to calculate the Sobol indices. We evaluate the model for each of these parameter samples and calculate the features from each of the model results. This step is performed in parallel to speed up the calculations. Then we use *nr_mc_samples* of the model and feature results to calculate the mean, variance, and 5th and 95th percentile for the model and each feature. Lastly, we use all calculated model and each feature results to calculate the Sobol indices using Saltellie's approach.

The plots created are intended as quick way to get an overview of the results, and not to create publication ready plots. Custom plots of the data can easily be created by retrieving the data from the Data class.

Changing the parameters of the polynomial chaos methods should be done with care, and implementing custom methods is only recommended for experts.

See also:

`uncertainpy.Parameters()`, *`uncertainpy.Data()`*, *`uncertainpy.plotting.PlotUncertainty()`*

`uncertainpy.core.UncertaintyCalculations.polynomial_chaos()` Uncertainty quantification using polynomial chaos expansions

`uncertainpy.core.UncertaintyCalculations.monte_carlo()` Uncertainty quantification using quasi-Monte Carlo methods

`uncertainpy.core.UncertaintyCalculations.create_PCE_custom()` Requirements for create_PCE_custom

`uncertainpy.core.UncertaintyCalculations.custom_uncertainty_quantification()` Requirements for custom_uncertainty_quantification

`save` (*filename*, *folder=u'data'*)

Save `data` to disk.

Parameters

- **filename** (*str*) – Name of the data file.
- **folder** (*str, optional*) – The folder to store the data in. Creates the folder if it does not exist. Default is "/data".

See also:

`uncertainpy.Data()` Data class

`uncertainty_calculations`

The class for performing the calculations for the uncertainty quantification and sensitivity analysis.

Parameters **new_uncertainty_calculations** (*UncertaintyCalculations or UncertaintyCalculations subclass instance*) – New UncertaintyCalculations object responsible for performing the uncertainty quantification calculations.

Returns **uncertainty_calculations** – UncertaintyCalculations object responsible for performing the uncertainty quantification calculations.

Return type UncertaintyCalculations or UncertaintyCalculations subclass instance

See also:

`uncertainpy.core.UncertaintyCalculations`

CHAPTER 6

Models

In order to perform the uncertainty quantification and sensitivity analysis of a model, Uncertainpy needs to set the parameters of the model, run the model using those parameters, and receive the model output. The main class for models is *Model*, which is used to create custom models. Uncertainpy has built-in support for NEURON and NEST models, found in the *NeuronModel* and *NestModel* classes respectively. Uncertainpy also has support for multiple model outputs through the use of additional features. It should be noted that while Uncertainpy is tailored towards neuroscience, it is not restricted to only neuroscience models. Uncertainpy can be used on any model that meets the criteria in this section.

6.1 Model

Generally, models are created through the `Model` class. `Model` takes the argument `run` and the optional arguments `postprocess`, `adaptive` and `labels`.

```
model = un.Model(run=example_model,
                 postprocess=example_postprocess,
                 interpolate=True,
                 labels=["xlabel", "ylabel"])
```

The `run` argument must be a Python function that runs a simulation on a specific model for a given set of model parameters, and returns the simulation output. We call such a function for a model function. The `postprocess` argument is a Python function used to postprocess the model output if required. We go into details on the requirements of the `postprocess` and model functions below. `interpolate` specifies whether the model should be interpolated to a regular form. This is required for for example models with adaptive time steps. For adaptive models, Uncertainpy automatically interpolates the output to a regular form (the same number of points for each model evaluation). Finally, `labels` allows the user to specify a list of labels to be used on the axes when plotting the results.

6.1.1 Defining a model function

As explained above, the `run` argument is a Python function that runs a simulation on a specific model for a given set of model parameters, and returns the simulation output. An example outline of a model function is:

```python
def example_model(parameter_1, parameter_2):
    # An algorithm for the model, or a script that runs
    # an external model, using the given input parameters.

    # Returns the model output and model time
    # along with the optional info object.
    return time, values, info
```

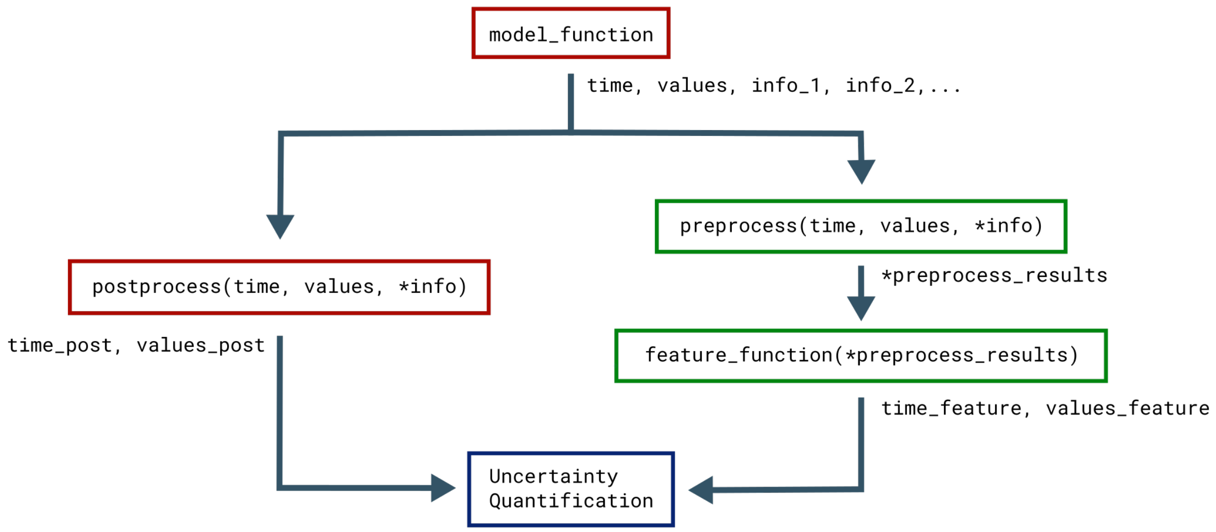
Such a model function has the following requirements:

1. **Input.** The model function takes a number of arguments which define the uncertain parameters of the model.
2. **Run the model.** The model must then be run using the parameters given as arguments.
3. **Output.** The model function must return at least two objects, the model time (or equivalent, if applicable) and model output. Additionally, any number of optional info objects can be returned. In Uncertainpy, we refer to the time object as `time`, the model output object as `values`, and the remaining objects as `info`.
    1. **Time** (`time`). The `time` can be interpreted as the x-axis of the model. It is used when interpolating (see below), and when certain features are calculated. We can return `None` if the model has no time associated with it.
    2. **Model output** (`values`). The model output must either be regular, or it must be possible to interpolate or postprocess the output (see *Features*) to a regular form.
    3. **Additional info** (`info`). Some of the methods provided by Uncertainpy, such as the later defined model postprocessing, feature preprocessing, and feature calculations, require additional information from the model (e.g., the time a neuron receives an external stimulus). We recommend to use a single dictionary as info object, with key-value pairs for the information, to make debugging easier. Uncertainpy always uses a single dictionary as the `info` object. Certain features require that specific keys are present in this dictionary.

The model itself does not need to be implemented in Python. Any simulator can be used, as long as we can control the model parameters and retrieve the simulation output via Python. We can as a shortcut pass a model function to the `model` argument in *UncertaintyQuantification*, instead of first having to create a `Model` instance.

6.1.2 Defining a postprocess function

The `postprocess` function is used to postprocess the model output before it is used in the uncertainty quantification. Postprocessing does not change the model output sent to the feature calculations. This is useful if we need to transform the model output This is useful if we need to transform the model output to a regular result for the uncertainty quantification, but still need to preserve the original model output to reliably detect the model features.


model_function
time, values, info_1, info_2,...
preprocess(time, values, *info)
*preprocess_results
postprocess(time, values, *info)
time_post, values_post
feature_function(*preprocess_results)
time_feature, values_feature
Uncertainty Quantification

This figure illustrates how the objects returned by the model function are sent to both model `postprocess`, and feature `preprocess` (see *Features*). Functions associated with the model are in red while functions associated with features are in green.

An example outline of the `postprocess` function is:

```python
def example_postprocess(time, values, info):
    # Postprocess the result to a regular form using time,
    # values, and info returned by the model function.

    # Return the postprocessed model output and time.
    return time_postprocessed, values_postprocessed
```

The only time postprocessing is required for Uncertainpy to work, is when the model produces output that can not be interpolated to a regular form by Uncertainpy. Postprocessing is for example required for network models that give output in the form of spike trains, i.e. time values indicating when a given neuron fires. It should be noted that postprocessing of spike trains is already implemented in Uncertainpy, in the *NestModel*. For most purposes user defined postprocessing will not be necessary.

The requirements for the `postprocess` function are:

1. **Input.** `postprocess` must take the objects returned by the model function as input arguments.
2. **Postprocessing.** The model time (`time`) and output (`values`) must be postprocessed to a regular form, or to a form that can be interpolated to a regular form by Uncertainpy. If additional information is needed from the model, it can be passed along in the `info` object.
3. **Output.** The `postprocess` function must return two objects:
    1. **Model time** (`time_postprocessed`). The first object is the postprocessed time (or equivalent) of the model. We can return `None` if the model has no time. Note that the automatic interpolation of the postprocessed time can only be performed if a postprocessed time is returned (if an interpolation is required).
    2. **Model output** (`values_postprocessed`). The second object is the postprocessed model output.

6.1.3 API Reference

class `uncertainpy.models.`**`Model`**(*run=None*, *interpolate=False*, *labels=[]*, *postprocess=None*, *ignore=False*, *suppress_graphics=False*, *logger_level=u'info'*, ***model_kwargs*)

Class for storing the model to perform uncertainty quantification and sensitivity analysis on.

The `run` method must either be implemented or set to a function, and is responsible for running the model. If you want to calculate features directly from the original model results, but still need to postprocess the model results to perform the uncertainty quantification, you can implement the postprocessing in the `postprocess` method.

Parameters

- **run** (*{None, callable}, optional*) – A function that implements the model. See the `run` method for requirements of the function. Default is None.
- **interpolate** (*bool, optional*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **labels** (*list, optional*) – A list of label names for the axes when plotting the model. On the form `["x-axis", "y-axis", "z-axis"]`, with the number of axes that is correct for the model output. Default is an empty list.
- **postprocess** (*{None, callable}, optional*) – A function that implements the postprocessing of the model. See the `postprocess` method for requirements of the function. Default is None.
- **ignore** (*bool, optional*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. Default is False.
- **suppress_graphics** (*bool, optional*) – Suppress all graphics created by the model. Default is False.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed. Default logger level is "info".
- ****model_kwargs** – Any number of arguments passed to the model function when it is run.

Variables

- **`labels`** (*`list`*) – A list of label names for the axes when plotting the model. On the form `["x-axis", "y-axis", "z-axis"]`, with the number of axes that is correct for the model output.
- **`interpolate`** (*`bool`*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **`name`** (*`str`*) – Name of the model. Either the name of the class or the name of the function set as run.
- **`suppress_graphics`** (*`bool`*) – Suppress all graphics created by the model.
- **`ignore`** (*`bool`*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. The model results are still postprocessed if a postprocessing is implemented. Default is False.

See also:

`uncertainpy.models.Model.run`, *`uncertainpy.models.Model.postprocess`*

`evaluate`(***parameters*)

Run the model with parameters and default model_kwargs options, and validate the result.

Parameters **parameters (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the model, either setting them with Python, or assigning them to the simulator.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns None or numpy.nan.
- **values** (*array_like*) – Result of the model. Note that *values* myst either be regular (have the same number of points for different paramaters) or be able to be interpolated.
- *info, optional* – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`postprocess`

Postprocessing of the time and results from the model.

No postprocessing is performed, and the direct model results are currently returned. If postprocessing is needed it should follow the below format.

Parameters

- ***model_result** – Variable length argument list. Is the values that `run` returns. It contains *time* and *values*, and then any number of optional *info* values.
- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values the model should return `None` or `numpy.nan`.
- **values** (*array_like*) – Result of the model.
- **info, optional** – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns `None` or `numpy.nan`.
- **values** (*array_like*) – The postprocessed model results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Notes

Perform a postprocessing of the model results before they are sent to the uncertainty quantification. The model results must either be regular or be able to be interpolated. This is because the uncertainty quantification methods needs results with the same number of points for each set of parameters to be able to perform the uncertainty quantification.

`postprocess` is implemented to make the model results regular, or on a form that can be interpolated. The results from the postprocessing is not used to calculate features, and is therefore used if you want to calculate features directly from the original model results, but still need to postprocess the model results to perform the uncertainty quantification.

The requirements for a `postprocess` function are:

1. **Input.** `postprocess` must take the objects returned by the model function as input arguments.
2. **Postprocessing.** The model time (`time`) and output (`values`) must be postprocessed to a regular form, or to a form that can be interpolated to a regular form by Uncertainpy. If additional information is needed from the model, it can be passed along in the `info` object.
3. **Output.** The `postprocess` function must return two objects:
   1. **Model time** (`time_postprocessed`). The first object is the postprocessed time (or equivalent) of the model. We can return `None` if the model has no time. Note that the automatic interpolation of the postprocessed time can only be performed if a postprocessed time is returned (if an interpolation is required).
   2. **Model output** (`values_postprocessed`). The second object is the postprocessed model output.

`run`

Run the model and return time and model result.

This method must either be implemented or set to a function and is responsible for running the model. See Notes for requirements.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the model, either setting them with Python, or assigning them to the simulator.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns None or numpy.nan.
- **values** (*array_like*) – Result of the model. Note that *values* myst either be regular (have the same number of points for different paramaters) or be able to be interpolated.
- *info, optional* – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

Raises `NotImplementedError` – If no run method have been implemented or set to a function.

Notes

The `run` method must either be implemented or set to a function. Both options have the following requirements:

1. **Input.** The model function takes a number of arguments which define the uncertain parameters of the model.
2. **Run the model.** The model must then be run using the parameters given as arguments.
3. **Output.** The model function must return at least two objects, the model time (or equivalent, if applicable) and model output. Additionally, any number of optional info objects can be returned. In

Uncertainpy, we refer to the time object as `time`, the model output object as `values`, and the remaining objects as `info`. Note that while we refer to these objects as `time`, `values` and `info` in Uncertainpy, it does not matter what you call the objects returned by the run function.

1. **Time** (`time`). The `time` can be interpreted as the x-axis of the model. It is used when interpolating (see below), and when certain features are calculated. We can return `None` if the model has no time associated with it.
2. **Model output** (`values`). The model output must either be regular, or it must be possible to interpolate or postprocess the output to a regular form.
3. **Additional info** (`info`). Some of the methods provided by Uncertainpy, such as the later defined model postprocessing, feature preprocessing, and feature calculations, require additional information from the model (e.g., the time a neuron receives an external stimulus). We recommend to use a single dictionary as info object, with key-value pairs for the information, to make debugging easier. Uncertainpy always uses a single dictionary as the `info` object. Certain features require that specific keys are present in this dictionary.

The model does not need to be implemented in Python, you can use any model/simulator as long as you are able to set the model parameters of the model from the run method Python and return the results from the model into the run method.

If you want to calculate features directly from the original model results, but still need to postprocess the model results to perform the uncertainty quantification, you can implement the postprocessing in the `postprocess` method.

See also:

`uncertainpy.features`

`uncertainpy.features.Features.preprocess` Preprocessing of model results before feature calculation

`uncertainpy.model.Model.postprocess` Postprocessing of model result.

`set_parameters`(***parameters*)
Set all named arguments as attributes of the model class.

Parameters **parameters (*A number of named arguments (name=value).*) – All set as attributes of the class.

`validate_postprocess`(*postprocess_result*)
Validate the results from `postprocess`.

This method ensures that `postprocess` returns *time* and *values*.

Parameters model_results – Any type of postprocessed model results returned by `postprocess`.

Raises

- `ValueError` – If the postprocessed model result does not fit the requirements.
- `TypeError` – If the postprocessed model result does not fit the requirements.

Notes

Tries to verify that *time* and *values* are returned from `postprocess`. `postprocess` must return two objects on the format: `return time, values`, where:

- **`time_postprocessed`** [`{None, numpy.nan, array_like}`.] The first object is the postprocessed time (or equivalent) of the model. We can return `None` if the model has no time. Note that the automatic interpolation of the postprocessed time can only be performed if a postprocessed time is returned (if an interpolation is required).

- **values_postprocessed** [array_like.] The second object is the postprocessed model output.

Both of these must be regular or on a form that can be interpolated.

See also:

uncertainpy.models.Model.postprocess()

validate_run(*model_result*)

Validate the results from `run`.

This method ensures `run` returns *time*, *values*, and optional info objects.

Parameters model_results – Any type of model results returned by `run`.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ....` Where:

- `time` : `{None, numpy.nan, array_like}`. Time values of the model. If no time values it should return None or numpy.nan.
- `values` : `array_like` Result of the model.
- `info`, optional. Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

uncertainpy.models.Model.run()

6.2 NeuronModel

NEURON is a widely used simulator for multi-compartmental neural models. Uncertainpy has support for NEURON models through the `NeuronModel` class, a subclass of *Model*. Among others, `NeuronModel` takes the arguments:

```
model = un.NeuronModel(path="path/to/neuron_model",
                       interpolate=True,
                       stimulus_start=1000,               # ms
                       stimulus_end=1900)                 # ms
```

`path` is the path to the folder where the NEURON model is saved (the location of the `mosinit.hoc` file). `interpolate` indicates whether the NEURON model uses adaptive time steps. `stimulus_start` and `stimulus_end` denotes the start and end time of any stimulus given to the neuron. `NeuronModel` loads the NEURON model from `mosinit.hoc`, sets the parameters of the model, evaluates the model and returns the somatic membrane potential of the neuron. `NeuronModel` therefore does not require a model function. An example of a NEURON model analysed with Uncertainpy is found in the *interneuron example*.

If changes are needed to the standard `NeuronModel`, such as measuring the voltage from other locations than the soma, or recalculate properties after the parameters have been set, the *Model* class with an appropriate model function should be used instead. Alternatively, `NeuronModel` can be subclassed and the existing methods customized as required. An example of the later is shown in */examples/bahl/*.

6.2.1 API Reference

class `uncertainpy.models.`**`NeuronModel`** (*file=u'mosinit.hoc', path=u", interpolate=True, stimulus_start=None, stimulus_end=None, name=None, ignore=False, run=None, record_from=u'soma', labels=[u'Time (ms)', u'Membrane potential (mV)'], suppress_graphics=True, logger_level=u'info', info={}, **model_kwargs*)

Class for Neuron simulator models.

Loads a Neuron simulation, runs it, and measures the voltage in the soma.

Parameters

- **file** (*str, optional*) – Filename of the Neuron model. Default is `"mosinit.hoc"`.
- **path** (*str, optional*) – Path to the Neuron model. If None, the file is considered to be in the current folder. Default is "".
- **stimulus_start** (*{int, float, None}, optional*) – The start time of any stimulus given to the neuron model. This is added to the info dictionary. If None, no stimulus_start is added to the info dictionary. Default is None.
- **stimulus_end** (*{int, float, None}, optional*) – The end time of any stimulus given to the neuron model. This is added to the info dictionary. If None, no stimulus_end is added to the info dictionary. Default is None.
- **interpolate** (*bool, optional*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **name** (*{None, str}, optional*) – Name of the model, if None the model gets the name of the current class. Default is None.
- **ignore** (*bool, optional*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. Default is False.
- **run** (*{None, callable}, optional*) – A function that implements the model. See the `run` method for requirements of the function. Default is None.
- **record_from** (*{str}, optional*) – Name of the section in the NEURON model where voltage should be recorded. Default is `"soma"`.
- **labels** (*list, optional*) – A list of label names for the axes when plotting the model. On the form `["x-axis", "y-axis", "z-axis"]`, with the number of axes that is correct for the model output. Default is `["Time (ms)", "Membrane potential (mv)"]`.
- **suppress_graphics** (*bool, optional*) – Suppress all graphics created by the Neuron model. Default is True.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed Default logger level is "info".
- **info** (*dict, optional*) – Dictionary added to info. Default is an empty dictionary.

- ****model_kwargs** – Any number of arguments passed to the model function when it is run.

Variables

- **`run`** (`uncertainpy.models.Model.run`) –
- **`labels`** (*`list`*) – A list of label names for the axes when plotting the model. On the form `["x-axis", "y-axis", "z-axis"]`, with the number of axes that is correct for the model output.
- **`interpolate`** (*`bool`*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **`suppress_graphics`** (*`bool`*) – Suppress all graphics created by the model.
- **`ignore`** (*`bool`*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. The model results are still postprocessed if a postprocessing is implemented. Default is False.

Raises `RuntimeError` – If no section with name `soma` is found in the Neuron model.

Notes

Measures the voltage in the section with name `soma`.

`evaluate`(***parameters*)

Run the model with parameters and default model_kwargs options, and validate the result.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the model, either setting them with Python, or assigning them to the simulator.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns None or numpy.nan.
- **values** (*array_like*) – Result of the model. Note that *values* myst either be regular (have the same number of points for different paramaters) or be able to be interpolated.
- *info, optional* – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`load_neuron`(*path, file*)

Import neuron and a neuron simulation file.

Parameters

- **file** (*str*) – Filename of the Neuron model. must be a `.hoc` file.
- **path** (*str*) – Path to the Neuron model.

Returns **h** – Neurons h object.

Return type Neuron object

Raises `ImportError` – If neuron is not installed.

`load_python`(*path*, *file*, *name*)

Import a Python neuron simulation located in function in *path/file* with name *name*.

Parameters

- **file** (*str*) – Filename of the Neuron model. must be a `.hoc` file.
- **path** (*str*) – Path to the Neuron model.
- **name** (*str*) – Name of the run function.

Returns **model** – A python function imported from *path/file* with name *name*.

Return type a run function

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`postprocess`(*time*, *values*, *info*)

Postprocessing of the time and results from the Neuron model is generally not needed. The direct model result except the info is returned.

Parameters

- **time** (*array_like*) – Time values of the Neuron model.
- **values** (*array_like*) – Voltage of the neuron.
- **info** (*dict*) – Dictionary with information needed by features.

Returns

- **time** (*array_like*) – Time values of the Neuron model.
- **values** (*array_like*) – Voltage of the neuron.

`run`

Run the model and return time and model result.

This method must either be implemented or set to a function and is responsible for running the model. See Notes for requirements.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the model, either setting them with Python, or assigning them to the simulator.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns None or numpy.nan.
- **values** (*array_like*) – Result of the model. Note that *values* myst either be regular (have the same number of points for different paramaters) or be able to be interpolated.
- *info, optional* – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

Raises `NotImplementedError` – If no run method have been implemented or set to a function.

Notes

The `run` method must either be implemented or set to a function. Both options have the following requirements:

1. **Input.** The model function takes a number of arguments which define the uncertain parameters of the model.
2. **Run the model.** The model must then be run using the parameters given as arguments.
3. **Output.** The model function must return at least two objects, the model time (or equivalent, if applicable) and model output. Additionally, any number of optional info objects can be returned. In Uncertainpy, we refer to the time object as `time`, the model output object as `values`, and the remaining objects as `info`. Note that while we refer to these objects as `time`, `values` and `info` in Uncertainpy, it does not matter what you call the objects returned by the run function.
   1. **Time** (`time`). The `time` can be interpreted as the x-axis of the model. It is used when interpolating (see below), and when certain features are calculated. We can return `None` if the model has no time associated with it.
   2. **Model output** (`values`). The model output must either be regular, or it must be possible to interpolate or postprocess the output to a regular form.
   3. **Additional info** (`info`). Some of the methods provided by Uncertainpy, such as the later defined model postprocessing, feature preprocessing, and feature calculations, require additional information from the model (e.g., the time a neuron receives an external stimulus). We recommend to use a single dictionary as info object, with key-value pairs for the information, to make debugging easier. Uncertainpy always uses a single dictionary as the `info` object. Certain features require that specific keys are present in this dictionary.

The model does not need to be implemented in Python, you can use any model/simulator as long as you are able to set the model parameters of the model from the run method Python and return the results from the model into the run method.

If you want to calculate features directly from the original model results, but still need to postprocess the model results to perform the uncertainty quantification, you can implement the postprocessing in the `postprocess` method.

See also:

`uncertainpy.features`

`uncertainpy.features.Features.preprocess` Preprocessing of model results before feature calculation

`uncertainpy.model.Model.postprocess` Postprocessing of model result.

`run_neuron`(***parameters*)

Load and run a Neuron simulation from a `.hoc` file and return the model voltage in soma.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model which are set in Neuron.

Returns

- **time** (*array*) – Time values of the model.
- **values** (*array*) – Voltage of the neuron. Note that *values* must either be regular (have the same number of points for different parameters) or be able to be interpolated.
- **info** (*dictionary*) – A dictionary with information needed by features. Efel features require `"stimulus_start"` and `"stimulus_end"` as keys, while spiking_features require `stimulus_start"`.

- **info** (*dictionary*) – A dictionary with information needed by features. `"stimulus_start"` and `"stimulus_end"` are returned in the info dictionary if they are given as parameters to `NeuronModel`.

Notes

Efel features require `"stimulus_start"` and `"stimulus_end"` as keys, while spiking_features require `stimulus_start"`.

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`run_python`(***parameters*)
Load and run a Python function that contains a Neuron simulation and return the model result. The Python neuron simulation is located in a function in *path/file* and name *name*.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model which are sent to the Python function.

Returns

- **time** (*array*) – Time values of the model.
- **values** (*array*) – Voltage of the neuron. Note that *values* must either be regular (have the same number of points for different parameters) or be able to be interpolated.
- **info** (*dictionary*) – A dictionary with information needed by features. If a info dictionary is returned by the model function it is updated with `"stimulus_start"` and `"stimulus_end"` if they are given as parameters to `NeuronModel`. If a info dictionary is not returned, a info dictionary is added as the third return argument.

Notes

Efel features require `"stimulus_start"` and `"stimulus_end"` as keys, while spiking_features require `stimulus_start"`.

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`set_parameters`(*parameters*)
Set parameters in the neuron model.

Parameters **parameters** (*dict*) – A dictionary with parameter names as keys and the parameter value as value.

`validate_postprocess`(*postprocess_result*)
Validate the results from `postprocess`.

This method ensures that `postprocess` returns *time* and *values*.

Parameters **model_results** – Any type of postprocessed model results returned by `postprocess`.

Raises

- `ValueError` – If the postprocessed model result does not fit the requirements.
- `TypeError` – If the postprocessed model result does not fit the requirements.

Notes

Tries to verify that *time* and *values* are returned from `postprocess`. `postprocess` must return two objects on the format: `return time, values`, where:

- **`time_postprocessed`** [`{None, numpy.nan, array_like}`.] The first object is the postprocessed time (or equivalent) of the model. We can return `None` if the model has no time. Note that the automatic interpolation of the postprocessed time can only be performed if a postprocessed time is returned (if an interpolation is required).
- **`values_postprocessed`** [`array_like`.] The second object is the postprocessed model output.

Both of these must be regular or on a form that can be interpolated.

See also:

`uncertainpy.models.Model.postprocess()`

`validate_run`(*model_result*)

Validate the results from `run`.

This method ensures `run` returns *time*, *values*, and optional info objects.

Parameters **model_results** – Any type of model results returned by `run`.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- `time` : `{None, numpy.nan, array_like}`. Time values of the model. If no time values it should return None or numpy.nan.
- `values` : `array_like` Result of the model.
- `info`, optional. Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

`uncertainpy.models.Model.run()`

6.3 NestModel

NEST is a simulator for large networks of spiking neurons. NEST models are supported through the `NestModel` class, another subclass of *Model*:

```
model = un.NestModel(run=nest_model_function)
```

`NestModel` requires the model function to be specified through the `run` argument, unlike `NeuronModel`. The NEST model function has the same requirements as a regular model function, except it is restricted to return only two objects: the final simulation time (denoted `simulation_end`), and a list of spike times for each neuron in the network, which we refer to as spiketrains (denoted `spiketrains`).

A spike train returned by a NEST model is a set of irregularly spaced time points where a neuron fired a spike. NEST models therefore require postprocessing to make the model output regular. Such a postprocessing is provided by the implemented `postprocess()` method, which converts a spiketrain to a list of zeros (no spike) and ones (a spike) for each time step in the simulation. For example, if a NEST simulation returns the spiketrain `[0, 2, 3.5]`, it means the neuron fired three spikes occurring at $t = 0, 2,$ and 3.5 ms. If the simulation have a time resolution of 0.5 ms and ends after 4 ms, `NestModel.postprocess` returns the postprocessed spiketrain `[1, 0, 0, 0, 1, 0, 0, 1, 0]`, and the postprocessed time array `[0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4]`. The final uncertainty quantification of a NEST network therefore predicts the probability for a spike to occur at any specific time point in the simulation. An example on how to use `NestModel` is found in the *Brunel exampel*.

6.3.1 API Reference

class `uncertainpy.models.`**`NestModel`**(*run=None, interpolate=False, ignore=False, labels=[u'Time (ms)', u'Neuron nr', u'Spiking probability'], logger_level=u'info', **model_kwargs*)

Class for NEST simulator models.

The `run` method must either be implemented or set to a function, and is responsible for running the NEST model.

Parameters

- **run** (*{None, function}, optional*) – A function that implements the model. See Note for requirements of the function. Default is None.
- **interpolate** (*bool, optional*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **ignore** (*bool, optional*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. Default is False.
- **labels** (*list, optional*) – A list of label names for the axes when plotting the model. On the form `["x-axis", "y-axis", "z-axis"]`, with the number of axes that is correct for the model output. Default is `["Time (ms)", "Neuron nr", "Spiking probability"]`.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed. Default logger level is "info".
- ****model_kwargs** – Any number of arguments passed to the model function when it is run.

Variables

- **`run`** (`uncertainpy.models.Model.run`) –
- **`labels`** (*`list, optional`*) – A list of label names for the axes when plotting the model.
- **`interpolate`** (*`bool`*) – True if the model is irregular, meaning it has a varying number of return values between different model evaluations, and an interpolation of the results is performed. Default is False.
- **`ignore`** (*`bool, optional`*) – Ignore the model results when calculating uncertainties, which means the uncertainty is not calculated for the model. The model results are still postprocessed. Default is False.

Raises `ImportError` – If nest is not installed.

See also:

`uncertainpy.models.NestModel.run`

`evaluate`(***parameters*)

Run the model with parameters and default model_kwargs options, and validate the result.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the model, either setting them with Python, or assigning them to the simulator.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model, if no time values returns None or numpy.nan.
- **values** (*array_like*) – Result of the model. Note that *values* myst either be regular (have the same number of points for different paramaters) or be able to be interpolated.
- *info, optional* – Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

`uncertainpy.models.Model.run()` Requirements for the model run function.

`postprocess`(*simulation_end*, *spiketrains*)

Postprocessing of the spiketrains from a Nest model.

For each neuron, convert a spiketrain to a list of the probability for a spike at each timestep, as well as creating a time array. For each timestep in the simulation the result is 0 if there is no spike and 1 if there is a spike.

Parameters

- **simulation_end** (*{int, float}*) – The final simulation time.
- **spiketrains** (*list*) – A list of spike trains for each neuron.

Returns

- **time** (*array*) – A time array of all time points in the Nest simulation.
- **spiketrains** (*list*) – A list of the probability for a spike at each timestep, for each neuron.

Example

In a simulation that gives the spiketrain `[0, 2, 3]`, with a time resolution of 0.5 ms and that ends after 4 ms, the resulting spike train become: `[1, 0, 0, 0, 1, 0, 1, 0, 0]`.

`run`

Run a Nest model and return the final simulation time and the spiketrains.

This method must either be implemented or set to a function and is responsible for running the model. See Notes for requirements.

Parameters ****parameters** (*A number of named arguments (name=value).*) – The parameters of the model. These parameters must be assigned to the NEST model.

Returns

- **simulation_end** (*{int, float}*) – The final simulation time.
- **spiketrains** (*list*) – A list of spike trains for each neuron.

Raises `NotImplementedError` – If no `run` method have been implemented or set to a function.

Notes

The `run` method must either be implemented or set to a function. Both options have the following requirements:

1. **Input.** The model function takes a number of arguments which define the uncertain parameters of the model.
2. **Run the model.** The NEST model must then be run using the parameters given as arguments.
3. **Output.** The model function must return:
    1. **Time** (`simulation_end`). The final simulation time of the NEST model.
    2. **Model output** (`spiketrains`). A list if spike trains from each recorded neuron.

The model results *simulation_end* and *spiketrains* are used to calculate the features, and is postprocessed to create a regular result before the calculating the uncertainty of the model.

See also:

`uncertainpy.model.Model.postprocess`

`set_parameters`(***parameters*)

Set all named arguments as attributes of the model class.

Parameters ****parameters** (*A number of named arguments (name=value).*) – All set as attributes of the class.

`validate_postprocess`(*postprocess_result*)

Validate the results from `postprocess`.

This method ensures that `postprocess` returns *time* and *values*.

Parameters **model_results** – Any type of postprocessed model results returned by `postprocess`.

Raises

- `ValueError` – If the postprocessed model result does not fit the requirements.
- `TypeError` – If the postprocessed model result does not fit the requirements.

Notes

Tries to verify that *time* and *values* are returned from `postprocess`. `postprocess` must return two objects on the format: `return time, values`, where:

- **`time_postprocessed`** [`{None, numpy.nan, array_like}`.] The first object is the postprocessed time (or equivalent) of the model. We can return `None` if the model has no time. Note that the automatic interpolation of the postprocessed time can only be performed if a postprocessed time is returned (if an interpolation is required).
- **`values_postprocessed`** [`array_like`.] The second object is the postprocessed model output.

Both of these must be regular or on a form that can be interpolated.

See also:

uncertainpy.models.Model.postprocess()

validate_run(*model_result*)

Validate the results from `run`.

This method ensures `run` returns *time*, *values*, and optional info objects.

Parameters **model_results** – Any type of model results returned by `run`.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- `time` : `{None, numpy.nan, array_like}`. Time values of the model. If no time values it should return None or numpy.nan.
- `values` : `array_like` Result of the model.
- `info`, optional. Any number of info objects that is passed on to feature calculations. It is recommended to use a single dictionary with the information stored as key-value pairs. This is what the implemented features requires, as well as require that specific keys to be present.

See also:

uncertainpy.models.Model.run()

6.4 Multiple model outputs

Uncertainpy is usable with multiple model outputs. However, it does unfortunately not have direct support for this, you have to use a small trick. Uncertainpy by default only performs an uncertainty quantification of the first model output returned. But you can return the additional model outputs in the info dictionary, and then define new features that extract each model output from the info dictionary, and then returns the additional model output.

Here is an example that shows how to do this:

```python
import uncertainpy as un
import chaospy as cp

# Example model with multiple outputs
def example_model(parameter_1, parameter_2):
    # Perform all model calculations here

    time = ...

    model_output_1 = ...
    model_output_2 = ...
    model_output_3 = ...
```

(continues on next page)

(continued from previous page)

```
    # We can store the additional model outputs in an info
    # dictionary
    info = {"model_output_2": model_output_2,
            "model_output_3": model_output_3}

    # Return time, model output and info dictionary
    # The first model output (model_output_1) is automatically used in the
    # uncertainty quantification
    return time, model_output_1, info
```

We can perform an uncertainty quantification of the other model outputs by creating a feature for each of the additional model outputs by extracting the output from the info dictionary and then return the output:

```
def model_output_2(time, model_output_1, info):
    return time, info["model_output_2"]

def model_output_3(time, model_output_1, info):
    return time, info["model_output_3"]

feature_list = [model_output_2, model_output_3]

# Define the parameter dictionary
parameters = {"parameter_1": cp.Uniform(),
            "parameter_2": cp.Uniform()}

# Set up the uncertainty quantification
UQ = un.UncertaintyQuantification(model=example_model,
                                parameters=parameters,
                                features=feature_list)

# Perform the uncertainty quantification using
# polynomial chaos with point collocation (by default)
data = UQ.quantify()
```

Alternatively, we can directly return all model outputs, but you are then unable to use the built-in features in Uncertainpy:

```
# Example model with multiple outputs
def example_model(parameter_1, parameter_2):
    # Perform all model calculations here

    time = ...

    model_output_1 = ...
    model_output_2 = ...
    model_output_3 = ...

    # Return time, model output and info dictionary
    # The first model output (model_output_1) is automatically used in the
    # uncertainty quantification
return time, model_output_1,  model_output_2, model_output_3


# We can perform an uncertainty quantification of the other model
# outputs by creating a feature for each of the additional
# model outputs by extracting the output from the info dictionary and
```

(continues on next page)

(continued from previous page)

```
# then return the output

def model_output_2(time, model_output_1, model_output_2, model_output_3):
    return time, model_output_2

def model_output_3(time, model_output_1, model_output_2, model_output_3):
    return time, model_output_3
```

CHAPTER 7

Parameters

The parameters of a model are defined by two properties they must have (i) a name and (ii) either a fixed value or a distribution. It is important that the name of the parameter is the same as the name given as the input argument in the model function. A parameter is considered uncertain if it has a probability distribution, and the distributions are given as Chaospy distributions. 64 different univariate distributions are defined in Chaospy. For a list of available distributions and detailed instructions on how to create probability distributions with Chaospy, see Section 3.3 in the Chaospy paper.

The parameters are defined by the *Parameters* class. `Parameters` takes the argument *parameters*. *parameters* can be on many different forms, but the most useful is a dictionary with the above information, the names of the parameters are the keys, and the fixed values or distributions of the parameters are the values. As an example, if we have two parameters, where the first is named `name_1` and has a uniform probability distributions in the interval $[8, 16]$, and the second is named `name_2` and has a fixed value 42, the list become:

```
import chaospy as cp
parameters = {"name_1": cp.Uniform(8, 16), "name_2": 42}
```

And `Parameters` is initialized:

```
parameters = un.Parameters(parameters=parameters)
```

The other possible forms that *parameters* can take are:

- `{name_1: parameter_object_1, name: parameter_object_2, ...}`
- `{name_1: value_1 or Chaospy distribution, name_2: value_2 or Chaospy distribution, ...}`
- `[parameter_object_1, parameter_object_2, ...]`,
- `[[name_1, value_1 or Chaospy distribution], ...]`.
- `[[name_1, value_1, Chaospy distribution or callable that returns a Chaospy distribution], ...]`

Where `name` is the name of the parameter and `parameter_object` is a `Parameter` object (see below). The *parameter* argument in `UncertaintyQuantification` is either `Parameters` object, or a `parameters` dictionary/list as shown above.

Each parameter in `Parameters` is a *Parameter* object. Each `Parameter` object is responsible for storing the name and fixed value and/or distribution of each parameter. It is initialized as:

```
parameter = Parameter(name="name_1", distribution=cp.Uniform(8, 16))
```

In general you should not need to use `Parameter`, it is mainly for internal use in Uncertainpy

7.1 API Reference

7.1.1 Parameters

class `uncertainpy.`**`Parameters`**(*parameters={}*, *distribution=None*)

A collection of parameters.

Has all standard dictionary methods implemented, such as items, value, contains and similar implemented. As such, behaves as an ordered dictionary.

Parameters

- **parameters** (*{dict {name: parameter_object}, dict of {name: value or Chaospy distribution}, . . .], list of Parameter instances, list [[name, value or Chaospy distribution], . . .], list [[name, value, Chaospy distribution or callable that returns a Chaospy distribution],. . .],}*) – List or dictionary of the parameters that should be created. On the form `parameters =`
  - `{name_1: parameter_object_1, name: parameter_object_2, ...}`
  - `{name_1: value_1 or Chaospy distribution, name_2: value_2 or Chaospy distribution, ...}`
  - `[parameter_object_1, parameter_object_2, ...],`
  - `[[name_1, value_1 or Chaospy distribution], ...].`
  - `[[name_1, value_1, Chaospy distribution or callable that returns a Chaospy distribution], ...]`
- **distribution** (*{None, multivariate Chaospy distribution}, optional*) – A multivariate distribution of all parameters, if it exists, it is used instead of individual distributions. Defaults to None.

Variables

- **`parameters`** (*`dict`*) – A dictionary of parameters with `name` as key and Parameter object as value.
- **`distribution`** (*`{None, multivariate Chaospy distribution}, optional`*) – A multivariate distribution of all parameters, if it exists, it is used instead of individual distributions. Defaults to None.

Notes

Both parameter values and parameter distributions must be set if uncertainpy.UncertaintyQuantification.quantify is run with single=True, meaning the uncertainty quantification should be performed with only one uncertain parameter at the time.

See also:

uncertainpy.Parameter

`__delitem__`(*name*)
Delete parameter with *name*.

Parameters **name** (*str*) – Name of parameter.

`__getitem__`(*name*)
Return Parameter object with *name*.

Parameters **name** (*str*) – Name of parameter.

Returns The parameter object with *name*.

Return type Parameter object

`__iter__`()
Iterate over the parameter objects.

Yields *Parameter object* – A parameter object.

`__len__`()
Get the number of parameters.

Returns The number of parameters.

Return type int

`__setitem__`(*name*, *parameter*)
Set parameter with *name*.

Parameters

- **name** (*str*) – Name of parameter.
- **parameter** (*Parameter object*) – The parameter object of *name*.

`__str__`()
Convert all parameters to a readable string.

Returns A readable string of all parameter objects.

Return type str

`clear`() → None. Remove all items from D.

`get`(*attribute=u'name'*, *parameter_names=None*)
Return attributes from all parameters.

Return a list of attributes (`name`, `value`, or `distribution`) from each parameters (parameters that have a distribution).

Parameters

- **attribute** (*{"name", "value", "distribution"}, optional*) – The name of the attribute to be returned from each uncertain parameter. Default is *name*.
- **parameter_names** (*{None, list, str}, optional*) – A list of all parameters of which attribute should be returned, or a string for a single parameter. If None, the attribute all parameters are returned. Default is None.

Returns List containing the *attribute* of each parameters.

Return type list

get_from_uncertain(*attribute=u'name'*)
Return attributes from uncertain parameters.

Return a list of attributes (`name`, `value`, or `distribution`) from each uncertain parameters (parameters that have a distribution).

Parameters attribute (*{"name", "value", "distribution"}, optional*) – The name of the attribute to be returned from each uncertain parameter. Default is *name*.

Returns List containing the *attribute* of each uncertain parameters.

Return type list

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

reset_parameter_file(*filename*)
Set all parameters to their value in a parameter file.

For all parameters, search *filename* for occurrences of `parameter_name = number` and replace `number` with value of that parameter.

Parameters filename (*str*) – Name of file.

set_all_distributions(*distribution*)
Set the distribution of all parameters.

Parameters distribution (*{None, Chaospy distribution, Function that returns a Chaospy distribution}*) – The distribution of the parameter.

set_distribution(*parameter*, *distribution*)
Set the distribution of a parameter.

Parameters

- **parameter** (*str*) – Name of parameter.
- **distribution** (*{None, Chaospy distribution, Function that returns a Chaospy distribution}*) – The distribution of the parameter.

set_parameters_file(*filename*, *parameters*)
Set listed parameters to their value in a parameter file.

For each parameter listed in *parameters*, search *filename* for occurrences of `parameter_name = number` and replace `number` with value of that parameter.

Parameters

- **filename** (*str*) – Name of file.
- **parameters** (*list*) – List of parameter names.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

7.1.2 Parameter

class `uncertainpy.`**Parameter**(*name*, *value=None*, *distribution=None*)
Parameter object, contains name of parameter, value of parameter and distribution of parameter.

Parameters

- **name** (*str*) – Name of the parameter.
- **value** (*float, int, None*) – The fixed value of the parameter. If you give a parameter a distribution, in most cases you do not need to give it a fixed value.
- **distribution** (*{None, Chaospy distribution, Function that returns a Chaospy distribution}, optional*) – The distribution of the parameter. A parameter is considered uncertain if it has a distribution. Defaults to None.

Variables

- **name** (`str`) – Name of the parameter.
- **value** (`float, int`) – The value of the parameter.
- **distribution** (`uncertainpy.Parameter.distribution`) – The distribution of the parameter. A parameter is considered uncertain if it has a distribution.

__str__()
Return a readable string describing the parameter.

Returns A string containing `name`, `value`, and if a parameter is uncertain.

Return type str

distribution
A Chaospy distribution or a function that returns a Chaospy distribution. If None the parameter has no distribution and is not considered uncertain.

Parameters distribution (*{None, Chaospy distribution, callable that returns a Chaospy distribution}, optional*) – The distribution of the parameter, used if the parameter is uncertain If it is a callable that returns a Chaospy distribution, the function sends *value* value to the function. Defaults to None.

Returns distribution – The distribution of the parameter, if None the parameter has no distribution and is not considered uncertain.

Return type {Chaospy distribution, None}

reset_parameter_file(*filename*)
Set all parameters to the original value in the parameter file, *filename*.

Parameters filename (*str*) – Name of file.

set_parameter_file(*filename*, *value*)
Set parameters to given value in a parameter file.

Search *filename* for occurrences of `name = number` and replace `number` with *value*.

Parameters

- **filename** (*str*) – Name of file.
- **value** (*float, int*) – New value to set in parameter file.

CHAPTER 8

Features

The activity of a biological system typically varies between recordings, even if the experimental conditions are maintained constant to the highest degree possible. Since the experimental data displays such variation, it is often meaningless (or even misguiding) to base the success of a computational model on a direct point-to-point comparison between the experimental data and model output (Druckmann et al., 2007; Van Geit et al., 2008). A common modeling practice is therefore to rather have the model reproduce essential features of the experimentally observed dynamics, such as the action potential shape, or action potential firing rate (Druckmann et al., 2007). Such features are typically more robust between different experimental measurements, or between different model simulations, than the raw data or raw model output, at least if sensible features have been chosen.

Uncertainpy takes this aspect of neural modeling into account, and is constructed so it can extract a set of features relevant for various common model types in neuroscience from the raw model output. Examples include the action potential shape in single neuron models, or the average interspike interval in network models. If we give the `features` argument to *UncertaintyQuantification*, Uncertainpy will perform uncertainty quantification and sensitivity analysis of the given features, in addition to the analysis of the "raw" output data. The value of feature based analysis is illustrated in the two examples on *a multi-compartment model of a thalamic interneuron* and *a sparsely connected recurrent network*.

The main class is *Features*. This class does not implement any specific features itself, but contain all common methods used by features. It is also used when creating custom features. Three sets of features comes pre-defined with Uncertainpy. Two sets of features for spiking models that returns voltage traces: *SpikingFeatures* and *EfelFeatures*. And one set of features for network models that return spiketrains *NetworkFeatures* Then there are two general classes for spiking (*GeneralSpikingFeatures*) and network features (*GeneralNetworkFeatures*) that implements common methods used by the two spiking features and network features respectively. These classes does not implement any specific models themselves.

8.1 Features

The `Features` class is used when creating custom features. Additionally it contains all common methods used by all features. The most common arguments to `Features` are:

```
list_of_feature_functions = [example_feature]

features = un.Features(new_features=list_of_feature_functions,
                       features_to_run=["example_feature"],
                       preprocess=example_preprocess,
                       interpolate=["example_feature"])
```

`new_features` is a list of Python functions that each calculates a specific feature, whereas `features_to_run` tells which of the features to perform uncertainty quantification of. If nothing is specified, the uncertainty quantification is by default performed on all features (`features_to_run="all"`). *`preprocess()`* requires a Python function that performs common calculations for all features. `interpolate` is a list of features that must be interpolated. As with models, Uncertainpy automatically interpolates the output of such features to a regular form. Below we first go into details on the requirements of a feature function, and then the requirements of a `preprocess` function.

8.1.1 Feature functions

A specific feature is given as a Python function. The outline of such a feature function is:

```
def example_feature(time, values, info):
    # Calculate the feature using time, values and info.

    # Return the feature times and values.
    return time_feature, values_feature
```

Feature functions have the following requirements:

1. **Input.** The feature function takes the objects returned by the model function as input, except in the case when a `preprocess` function is used (see below). In that case, the feature function instead takes the objects returned by the `preprocess` function as input `preprocess` is normally not used.
2. **Feature calculation.** The feature function calculates the value of a feature from the data given in `time`, `values` and optional `info` objects. As previously mentioned, in all built-in features in Uncertainpy, `info` is a dictionary containing required information as key-value pairs.
3. **Output.** The feature function must return two objects:
    1. **Feature time** (`time_feature`). The time (or equivalent) of the feature. We can return `None` instead for features where it is not relevant.
    2. **Feature values** (`values_feature`). The result of the feature calculation. As for the model output, the feature results must be regular, or able to be interpolated. If there are no feature results for a specific model evaluation (e.g., if the feature was spike width and there was no spike), the feature function can return `None`. The specific feature evaluation is then discarded in the uncertainty calculations.

As with models, we can as a shortcut give a list of feature functions as the `feature` argument in `UncertaintyQuantification`, instead of first having to create a `Features` instance.

8.1.2 Feature preprocessing

Some of the calculations needed to quantify features may overlap between different features. One example is finding the spike times from a voltage trace. The `preprocess` function is used to avoid having to perform the same calculations several times. An example outline of a `preprocess` function is:

```
def preprocess(time, values, info):
    # Perform all common feature calculations using time,
```

(continues on next page)

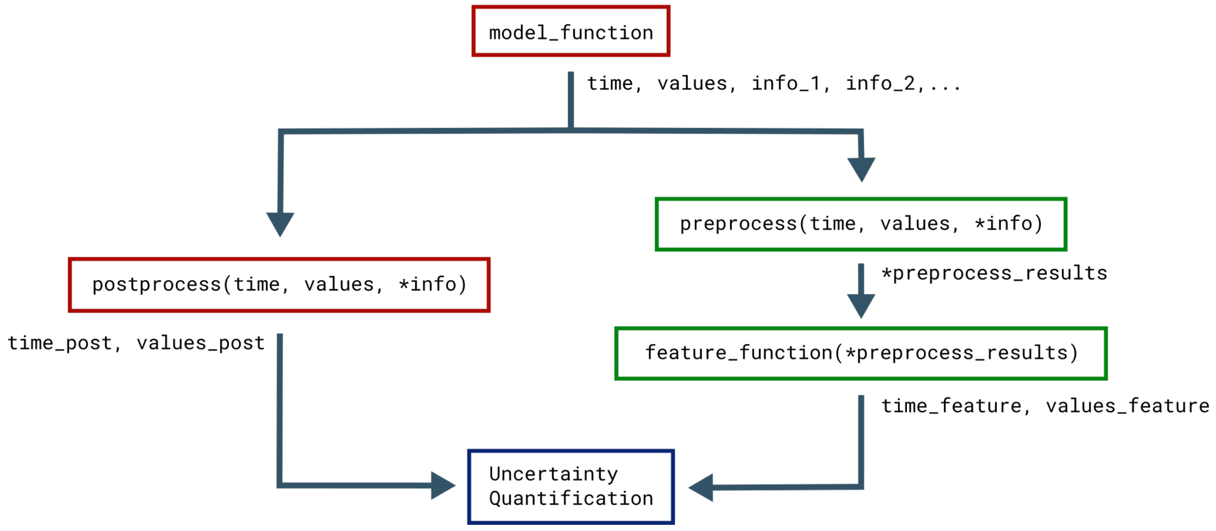
(continued from previous page)

```
    # values, and info returned by the model function.

    # Return the preprocessed model output and info.
    return time_preprocessed, values_preprocessed, info
```

The requirements for a `preprocess` function are:

1. **Input.** A `preprocess` function takes the objects returned by the model function as input.
2. **Preprocesssing.** The model output `time`, `values`, and additional `info` objects are used to perform all preprocess calculations.
3. **Output.** The `preprocess` function can return any number of objects as output. The returned preprocess objects are used as input arguments to the feature functions, so the two must be compatible.


model_function
time, values, info_1, info_2,...
preprocess(time, values, *info)
*preprocess_results
postprocess(time, values, *info)
time_post, values_post
feature_function(*preprocess_results)
time_feature, values_feature
Uncertainty Quantification

This figure illustrates how the objects returned by the model function are passed to `preprocess`, and the returned preprocess objects are used as input arguments in all feature functions. Functions associated with the model are in red while functions associated with features are in green. The preprocessing makes it so feature functions have different required input arguments depending on the feature class they are added to. As mentioned earlier, Uncertainpy comes with three built-in feature classes. These classes all take the `new_features` argument, so custom features can be added to each set of features. These feature classes perform a preprocessing, and therefore have different requirements for the input arguments of new feature functions. Additionally, certain features require specific keys to be present in the `info` dictionary. Each class has a `reference_feature` method that states the requirements for feature functions of that class in its docstring.

8.1.3 API Reference

class `uncertainpy.features.`**`Features`**(*new_features=None, features_to_run=u'all', new_utility_methods=None, interpolate=None, labels={}, preprocess=None, logger_level=u'info'*)

Class for calculating features of a model.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.

- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **new_utility_methods** (*{None, list}, optional*) – A list of new utility methods. All methods in this class that is not in the list of utility methods, is considered to be a feature. Default is None.
- **interpolate** (*{None, "all", str, list of feature names}, optional*) – Which features are irregular, meaning they have a varying number of time points between evaluations. An interpolation is performed on each irregular feature to create regular results. If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **features_to_run** (`list`) – Which features to calculate uncertainties for.
- **interpolate** (`list`) – A list of irregular features to be interpolated.
- **utility_methods** (`list`) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **labels** (`dictionary`) – Labels for the axes of each feature, used when plotting.

See also:

`uncertainpy.features.Features.reference_feature` reference_feature showing the requirements of a feature function.

add_features (*new_features*, *labels={}*)

Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`calculate_all_features`(**model_results*)
Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`calculate_feature`(*feature_name*, **preprocess_results*)
Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.
- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.

- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

calculate_features(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

features_to_run

Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

implemented_features()

Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

interpolate

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

labels

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

preprocess

Preprossesing of the time *time* and results *values* from the model, before the features are calculated.

No preprocessing is performed, and the direct model results are currently returned. If preprocessing is needed it should follow the below format.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns Returns any number of values that are sent to each feature. The values returned must compatible with the input arguments of all features.

Return type preprocess_results

Notes

Perform a preprossesing of the model results before the results are sent to the calculation of each feature. It is used to perform common calculations that each feature needs to perform, to reduce the number of necessary calculations. The values returned must therefore be compatible with the input arguments to each features.

See also:

`uncertainpy.models.Model.run` The model run method

reference_feature(**preprocess_results*)

An example feature. Feature function have the following requirements.

Parameters ***preprocess_results** – Variable length argument list. Is the values that `Features.preprocess` returns. By default `Features.preprocess` returns the same values as `Model.run` returns.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no

feature results return None or numpy.nan instead of *values* and that evaluation are disregarded.

See also:

uncertainpy.features.Features.preprocess() The features preprocess method.

uncertainpy.models.Model.run() The model run method

uncertainpy.models.Model.postprocess() The postprocessing method.

validate(*feature_name*, **feature_result*)

Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **time_feature** `[{None, numpy.nan, array_like}]` Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** `[{None, numpy.nan, array_like}]` The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

8.2 Spiking features

SpikingFeatures contains a set of features relevant for models of single neurons that receive an external stimulus and responds by eliciting a series of action potentials, also called spikes. Many of these features require the start time and end time of the stimulus, which must be returned as `info["stimulus_start"]` and `info["stimulus_start"]` in the model function. `info` is then used as an additional input argument in the calculation of each feature. `SpikingFeatures` implements a *preprocess()* method, which locates spikes in the model output.

The features included in the `SpikingFeatures` are briefly defined below. This set of features was taken from the previous work of Druckmann et al., 2007, with the addition of the number of action potentials during the stimulus period. We refer to the original publication for more detailed definitions.

1. `nr_spikes` – Number of action potentials (during stimulus period).
2. `spike_rate` – Action potential firing rate (number of action potentials divided by stimulus duration).
3. `time_before_first_spike` – Time from stimulus onset to first elicited action potential.

4. `accommodation_index` – Accommodation index (normalized average difference in length of two consecutive interspike intervals).
5. `average_AP_overshoot` – Average action potential peak voltage.
6. `average_AHP_depth` – Average afterhyperpolarization depth (average minimum voltage between action potentials).
7. `average_AP_width` – Average action potential width taken at midpoint between the onset and peak of the action potential.

A set of standard spiking features is already included in `SpikingFeatures`, but the user may want to add custom features. The *`preprocess()`* method changes the input given to the feature functions, and as such each spiking feature function has the following input arguments:

1. The `time` array returned by the model simulation.
2. An *Spikes* object (`spikes`) which contain the spikes found in the model output.
3. An `info` dictionary with `info["stimulus_start"]` and `info["stimulus_end"]` set.

The `Spikes` object is a preprocessed version of the model output, used as a container for `Spike` objects. In turn, each `Spike` object contain information of a single spike. This information includes a brief voltage trace represented by a `time` and a voltage (`V`) array that only includes the selected spike. The information in `Spikes` is used to calculate each feature. As an example, let us assume we want to create a feature that is the time at which the first spike in the voltage trace ends. Such a feature can be defined as follows:

```python
def first_spike_end_time(time, spikes, info):
    # Calculate the feature from the spikes object
    spike = spikes[0]               # Get the first spike
    values_feature = spike.t[-1]    # The last time point in the spike

    return None, values_feature
```

This feature may now be used as a feature function in the list given to the `new_features` argument.

From the set of both built-in and user defined features, we may select subsets of features that we want to use in the analysis of a model. Let us say we are interested in how the model performs in terms of the three features: `nr_spikes`, `average_AHP_depth` and `first_spike_end_time`. A spiking features object that calculates these features is created by:

```python
features_to_run = ["nr_spikes",
                   "average_AHP_depth",
                   "first_spike_end_time"]

features = un.SpikingFeatures(new_features=[first_spike_end_time],
                              features_to_run=features_to_run)
```

8.2.1 API Reference

class uncertainpy.features.**SpikingFeatures**(*new_features=None, features_to_run=u'all', interpolate=None, threshold=-30, end_threshold=-10, extended_spikes=False, trim=True, normalize=False, min_amplitude=0, min_duration=0, labels={}, strict=True, logger_level=u'info'*)

Spiking features of a model result, works with single neuron models and voltage traces.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.
- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **new_utility_methods** (*{None, list}, optional*) – A list of new utility methods. All methods in this class that is not in the list of utility methods, is considered to be a feature. Default is None.
- **interpolate** (*{None, "all", str, list of feature names}, optional*) – Which features are irregular, meaning they have a varying number of time points between evaluations. An interpolation is performed on each irregular feature to create regular results. If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **threshold** (*{float, int, "auto"}, optional*) – The threshold where the model result is considered to have a spike. If "auto" the threshold is set to the standard variation of the result. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Default is -10.
- **extended_spikes** (*bool, optional*) – If the found spikes should be extended further out than the threshold cuttoff. If True the spikes is considered to start and end where the derivative equals 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.
- **normalize** (*bool, optional*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value. Default is False.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **strict** (*bool, optional*) – If True, missing `"stimulus_start"` and `"stimulus_end"` from *info* raises a ValueError. If False the simulation start time is used as `"stimulus_start"` and the simulation end time is used for `"stimulus_end"`. Default is True.

- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **spikes** (`Spikes`) – A Spikes object that contain all spikes.
- **threshold** (`{float, int}`) – The threshold where the model result is considered to have a spike.
- **end_threshold** (`{int, float}`) – The end threshold for a spike relative to the threshold.
- **extended_spikes** (`bool`) – If the found spikes should be extended further out than the threshold cuttoff.
- **trim** (`bool`) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends.
- **normalize** (`bool`) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value.
- **min_amplitude** (`{int, float}`) – Minimum height for what should be considered a spike.
- **min_duration** (`{int, float}`) – Minimum duration for what should be considered a spike.
- **features_to_run** (`list`) – Which features to calculate uncertainties for.
- **interpolate** (`list`) – A list of irregular features to be interpolated.
- **utility_methods** (`list`) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **labels** (`dictionary`) – Labels for the axes of each feature, used when plotting.
- **strict** (`bool`) – If missing info values should raise an error.

Raises `ImportError` – If scipy is not installed.

Notes

The implemented features are:

nr_spikes	time_before_first_spike
spike_rate	average_AP_overshoot
average_AHP_depth	average_AP_width
accommodation_index	average_duration

Most of the feature are from: Druckmann, S., Banitt, Y., Gidon, A. A., Schurmann, F., Markram, H., and Segev, I. (2007). A novel multiple objective optimization framework for constraining conductance- based neuron models by experimental data. Frontiers in Neuroscience 1, 7-18. doi:10. 3389/neuro.01.1.1.001.2007

See also:

`uncertainpy.features.Features.reference_feature` reference_feature showing the requirements of a feature function.

`uncertainpy.features.Spikes` Class for finding spikes in the model result.

accommodation_index (*time, spikes, info*)

The accommodation index.

The accommodation index is the average of the difference in length of two consecutive interspike intervals normalized by the summed duration of the two interspike intervals.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – Not used in this feature.

Returns

- **time** (*None*)
- **accommodation_index** (*{float, None}*) – The accommodation index. Returns None if there are less than two spikes in the model result.

Notes

The accommodation index is defined as:

$$A = \frac{1}{N-k-1} \sum_{i=k}^{N} \frac{\text{ISI}_i - \text{ISI}_{i-1}}{\text{ISI}_i + \text{ISI}_{i-1}},$$

where ISI is the interspike interval, N the number of spikes, and k is defined as:

$$k = \min\left\{4, \frac{\text{Number of ISIs}}{5}\right\}.$$

add_features (*new_features, labels={}*)

Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`average_AHP_depth`(*time*, *spikes*, *info*)
The average action potential depth.

The minimum of the model result between two consecutive spikes (action potentials).

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – Not used in this feature.

Returns

- **time** (*None*)
- **average_AHP_depth** (*{float, None}*) – The average action potential depth. Returns None if there are no spikes in the model result.

`average_AP_overshoot`(*time*, *spikes*, *info*)
The average action potential overshoot,

The average of the absolute peak voltage values of all spikes (action potentials).

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – Not used in this feature.

Returns

- **time** (*None*)
- **average_AP_overshoot** (*{float, None}*) – The average action potential overshoot. Returns None if there are no spikes in the model result.

`average_AP_width`(*time*, *spikes*, *info*)
The average action potential width.

The average of the width of every spike (action potential) at the midpoint between the start and maximum of each spike.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – Not used in this feature.

Returns

- **time** (*None*)
- **average_AP_width** (*{float, None}*) – The average action potential width. Returns None if there are no spikes in the model result.

average_duration(*time*, *spikes*, *info*)
The average duration of an action potential, from the action potential onset to action potential termination.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – Not used in this feature.

Returns

- **time** (*None*)
- **average_AP_width** (*{float, None}*) – The average action potential width. Returns None if there are no spikes in the model result.

calculate_all_features(**model_results*)
Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

calculate_feature(*feature_name*, **preprocess_results*)
Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.
- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

`calculate_features`(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`calculate_spikes`(*time, values, threshold=-30, end_threshold=-10, extended_spikes=False, trim=True, normalize=False, min_amplitude=0, min_duration=0*)

Calculating spikes of a model result, works with single neuron models and voltage traces.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **values** (*array_like*) – Result of the model.
- **threshold** (*{float, int, "auto"}, optional*) – The threshold where the model result is considered to have a spike. If "auto" the threshold is set to the standard variation of the result. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Default is -10.
- **extended_spikes** (*bool, optional*) – If the found spikes should be extended further out than the threshold cuttoff. If True the spikes is considered to start and end where the derivative equals 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.

- **normalize** (*bool, optional*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value. Default is False.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it returns None or numpy.nan.
- **values** (*Spikes*) – The spikes found in the model results.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`uncertainpy.features.Spikes()` Class for finding spikes in the model result.

`features_to_run`

Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

`implemented_features``()`

Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

`interpolate`

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

`labels`

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

nr_spikes (*time*, *spikes*, *info*)

The number of spikes in the model result during the stimulus period.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – If `strict=True`, requires `info["stimulus_start"]` and `info['stimulus_end']` set.

Returns

- **time** (*None*)
- **nr_spikes** (*int*) – The number of spikes in the model result.

Raises

- `ValueError` – If strict is True and `"stimulus_start"` and `"stimulus_end"` are missing from *info*.
- `ValueError` – If stimulus_start >= stimulus_end.

preprocess (*time*, *values*, *info*)

Calculating spikes from the model result.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **values** (*array_like*) – Result of the model.
- **info** (*dictionary*) – A dictionary with info[“stimulus_start”] and info[“stimulus_end”].

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it returns None or numpy.nan.
- **values** (*Spikes*) – The spikes found in the model results.
- **info** (*dictionary*) – A dictionary with info[“stimulus_start”] and info[“stimulus_end”].

Notes

Also sets self.values = values, so features have access to self.values if necessary.

See also:

uncertainpy.models.Model.run() The model run method *`uncertainpy.features.Spikes()`* Class for finding spikes in the model result.

`reference_feature`(*time*, *spikes*, *info*)

An example of an GeneralSpikingFeature. The feature functions have the following requirements, and the input arguments must either be returned by `Model.run` or `SpikingFeatures.preprocess`.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – A dictionary with info["stimulus_start"] and info["stimulus_end"] set.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** (*array_like*) – The feature results, *values*. Returns None if there are no feature results and that evaluation are disregarded.

See also:

`uncertainpy.features.GeneralSpikingFeatures.preprocess()` The GeneralSpikingFeatures preprocess method.

`uncertainpy.models.Model.run()` The model run method

`spike_rate`(*time*, *spikes*, *info*)

The spike rate of the model result.

Number of spikes divided by the duration.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – If `strict=True`, requires `info["stimulus_start"]` and `info['stimulus_end']` set.

Returns

- **time** (*None*)
- **spike_rate** (*float*) – The spike rate of the model result.

Raises

- `ValueError` – If strict is True and `"stimulus_start"` and `"stimulus_end"` are missing from *info*.
- `ValueError` – If stimulus_start >= stimulus_end.

`time_before_first_spike`(*time*, *spikes*, *info*)

The time from the stimulus start to the first spike occurs.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.

- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – If `strict=True`, requires `info["stimulus_start"]` set.

Returns

- **time** (*None*)
- **time_before_first_spike** (*{float, None}*) – The time from the stimulus start to the first spike occurs. Returns None if there are no spikes on the model result.

Raises `ValueError` – If strict is True and `"stimulus_start"` and `"stimulus_end"` are missing from *info*.

validate (*feature_name*, **feature_result*)

Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **time_feature** `[{None, numpy.nan, array_like}]` Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** `[{None, numpy.nan, array_like}]` The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

8.3 Spikes

Spikes is responsible for locating spikes in a voltage trace, and is a container for all spikes found. Each spike is stored in a *Spike* object. `Spikes` is used in *SpikingFeatures*

8.3.1 API Reference

Spikes

class uncertainpy.features.**Spikes** (*time=None*, *V=None*, *threshold=-30*, *end_threshold=-10*, *extended_spikes=False*, *trim=True*, *normalize=False*, *min_amplitude=0*, *min_duration=0*, *xlabel=u"*, *ylabel=u"*)

Finds spikes in the given voltage trace and is a container for the resulting Spike objects.

Parameters

- **time** (*array_like*) – The time of the voltage trace.
- **V** (*array_like*) – The voltage trace.
- **threshold** (*{int, float, "auto"}*) – The threshold for what is considered a spike. If the voltage trace rise above and then fall below this *threshold* + *end_threshold* it is considered a spike. If "auto" the threshold is set to the standard deviation of the voltage trace. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Generally negative values give the best results. Default is -10.
- **extended_spikes** (*bool*) – If the spikes should be extended past the threshold, until the derivative of the voltage trace is below 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.
- **normalize** (*bool, optional*) – If the voltage trace should be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value. Default is False.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.
- **xlabel** (*str, optional*) – Label for the x-axis.
- **ylabel** (*str, optional*) – Label for the y-axis.

Variables

- **`spikes`** (*`list`*) – A list of Spike objects.
- **`nr_spikes`** (*`int`*) – The number of spikes.
- **`xlabel`** (*`str, optional`*) – Label for the x-axis.
- **`ylabel`** (*`str, optional`*) – Label for the y-axis.
- **`time`** (*`array_like`*) – The time of the voltage trace.
- **`V`** (*`array_like`*) – The voltage trace.

Notes

The spikes are found by finding where the voltage trace goes above the *threshold*, and then later falls below this *threshold* + *end_threshold*. The spike is considered to be everything within this interval.

The spike can be extended. If *extended_spikes* is True, the spike is extended around the above area until the derivative of the voltage trace falls below 0.5. This works badly with noisy voltage traces.

See also:

`Spike` The class for a single spike.

`find_spikes` Finding spikes in the voltage trace.

`consecutive`(*data*)
Returns the first consecutive array, from a discontinuous index array such as [2, 3, 4, 5, 12, 13, 14], which returns [2, 3, 4, 5]

Parameters **data** (*array_like*)

Returns The first consecutive array

Return type array_like

`find_spikes`(*time*, *V*, *threshold=-30*, *end_threshold=-10*, *extended_spikes=False*, *trim=True*, *normalize=False*, *min_amplitude=0*, *min_duration=0*)

Finds spikes in the given voltage trace.

Parameters

- **time** (*array_like*) – The time of the voltage trace.
- **V** (*array_like*) – The voltage trace.
- **threshold** (*{int, float, "auto"}*) – The threshold for what is considered a spike. If the voltage trace rise above and then fall below this *threshold* + *end_threshold* it is considered a spike. If "auto" the threshold is set to the standard deviation of the voltage trace. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Generally negative values give the best results. Default is -10.
- **extended_spikes** (*bool, optional*) – If the spikes should be extended past the threshold, until the derivative of the voltage trace is below 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.
- **normalize** (*bool, optional*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold must have a absolute value between [0, 1]. Default is False.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.

Raises

- `ValueError` – If the threshold is outside the interval [0, 1] when normalize=True.
- `ValueError` – If the absolute value of end_threshold is outside the interval [0, 1] when normalize=True.

Notes

The spikes are added to `self.spikes` and `self.nr_spikes` is updated.

The spikes are found by finding where the voltage trace goes above the *threshold*, and then later falls below this *threshold* + *end_threshold*. The spike is considered to be everything within this interval.

The spike can be extended. If *extended_spikes* is True, the spike is extended around the above area until the derivative of the voltage trace falls below 0.5. This works badly with noisy voltage traces.

`plot_spikes`(*save_name=None*)

Plot all spikes.

Parameters **save_name** (*{str, None}*) – Name of the plot file. If None, the plot is shown instead of saved to disk. Default is None.

`plot_voltage`(*save_name*)

Plot the voltage with the peak of each spike marked.

Parameters **save_name** (*{str, None}*) – Name of the plot file. If None, the plot is shown instead of saved to disk. Default is None.

Spike

class `uncertainpy.features.`**`Spike`**(*time*, *V*, *time_spike*, *V_spike*, *global_index*, *xlabel=u"*, *ylabel=u"*)

A single spike found in a voltage trace.

Parameters

- **time** (*array_like*) – The time array of the spike.
- **V** (*array_like*) – The voltage array of the spike.
- **time_spike** (*{float, int}*) – The timing of the peak of the spike.
- **V_spike** (*{float, int}*) – The voltage at the peak of the spike.
- **global_index** (*int*) – Index of the spike peak in the simulation.
- **xlabel** (*str, optional*) – Label for the x-axis.
- **ylabel** (*str, optional*) – Label for the y-axis.

Variables

- **`time`** (*`array_like`*) – The time array of the spike.
- **`V`** (*`array_like`*) – The voltage array of the spike.
- **`time_spike`** (*`{float, int}`*) – The timing of the peak of the spike.
- **`V_spike`** (*`{float, int}`*) – The voltage at the peak of the spike.
- **`global_index`** (*`int`*) – Index of the spike peak in the simulation.
- **`xlabel`** (*`str, optional`*) – Label for the x-axis.
- **`ylabel`** (*`str, optional`*) – Label for the y-axis.

`plot`(*save_name=None*)

Plot the spike.

Parameters **save_name** (*{str, None}*) – Name of the plot file. If None, the plot is shown instead of saved to disk. Default is None.

`trim`(*threshold*, *min_extent_from_peak=1*)

Remove the first and last values of the spike that is below *threshold*.

Parameters

- **threshold** (*{float, int}*) – Remove all values from each side of the spike that is bellow this value.
- **min_extent_from_peak** (*int, optional*) – Minimum extent of the spike in each direction from the peak.

8.4 EfelFeatures

An extensive set of features for single neuron voltage traces is found in the Electrophys Feature Extraction Library (eFEL). Uncertainpy has all features in the eFEL library contained in the `EfelFeatures` class. As with *SpikingFeatures*, many of the eFEL features require the start time and end time of the stimulus, which must be returned as `info["stimulus_start"]` and `info["stimulus_start"]` in the model function. eFEL currently contains 153 different features, we briefly list them here, but refer to the eFEL documentation for the definitions of each feature.

AHP1_depth_from_peak	AHP2_depth_from_peak	AHP_depth
AHP_depth_abs	AHP_depth_abs_slow	AHP_depth_diff
AHP_depth_from_peak	AHP_slow_time	AHP_time_from_peak
AP1_amp	AP1_begin_voltage	AP1_begin_width
AP1_peak	AP1_width	AP2_AP1_begin_width_diff
AP2_AP1_diff	AP2_AP1_peak_diff	AP2_amp
AP2_begin_voltage	AP2_begin_width	AP2_peak
AP2_width	AP_amplitude	AP_amplitude_change
AP_amplitude_diff	AP_amplitude_from_voltagebase	AP_begin_indices
AP_begin_time	AP_begin_voltage	AP_begin_width
AP_duration	AP_duration_change	AP_duration_half_width
AP_duration_half_width_change	AP_end_indices	AP_fall_indices
AP_fall_rate	AP_fall_rate_change	AP_fall_time
AP_height	AP_phaseslope	AP_phaseslope_AIS
AP_rise_indices	AP_rise_rate	AP_rise_rate_change
AP_rise_time	AP_width	APlast_amp
BAC_maximum_voltage	BAC_width	BPAPAmplitudeLoc1
BPAPAmplitudeLoc2	BPAPHeightLoc1	BPAPHeightLoc2
BPAPatt2	BPAPatt3	E10
E11	E12	E13
E14	E15	E16
E17	E18	E19
E2	E20	E21
E22	E23	E24
E25	E26	E27
E3	E39	E39_cod
E4	E40	E5
E6	E7	E8
E9	ISI_CV	ISI_log_slope
ISI_log_slope_skip	ISI_semilog_slope	ISI_values
Spikecount	Spikecount_stimint	adaptation_index
adaptation_index2	all_ISI_values	amp_drop_first_last
amp_drop_first_second	amp_drop_second_last	burst_ISI_indices
burst_mean_freq	burst_number	check_AISInitiation
decay_time_constant_after_stim	depolarized_base	doublet_ISI
fast_AHP	fast_AHP_change	interburst_voltage
inv_fifth_ISI	inv_first_ISI	inv_fourth_ISI
inv_last_ISI	inv_second_ISI	inv_third_ISI
inv_time_to_first_spike	irregularity_index	is_not_stuck
max_amp_difference	maximum_voltage	maximum_voltage_from_voltagebase
mean_AP_amplitude	mean_frequency	min_AHP_indices
min_AHP_values	min_voltage_between_spikes	minimum_voltage

Continued on next page

Table 1 – continued from previous page

number_initial_spikes	ohmic_input_resistance	ohmic_input_resistance_vb_ssse
peak_indices	peak_time	peak_voltage
sag_amplitude	sag_ratio1	sag_ratio2
single_burst_ratio	spike_half_width	spike_width2
steady_state_hyper	steady_state_voltage	steady_state_voltage_stimend
time_constant	time_to_first_spike	time_to_last_spike
time_to_second_spike	trace_check	voltage
voltage_after_stim	voltage_base	voltage_deflection

8.4.1 API Reference

class uncertainpy.features.**EfelFeatures**(*new_features=None*, *features_to_run=u'all'*, *interpolate=None*, *labels={}*, *strict=True*, *logger_level=u'info'*)

Calculating the mean value of each feature in the Electrophys Feature Extraction Library (eFEL), see: https://github.com/BlueBrain/eFEL.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.
- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **new_utility_methods** (*{None, list}, optional*) – A list of new utility methods. All methods in this class that is not in the list of utility methods, is considered to be a feature. Default is None.
- **interpolate** (*{None, "all", str, list of feature names}, optional*) – Which features are irregular, meaning they have a varying number of points between two evaluations. An interpolation is performed on each interpolate feature to create regular results. If `"all"`, all features interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **strict** (*bool, optional*) – If True, missing `"stimulus_start"` and `"stimulus_end"` from *info* raises a ValueError. If False the simulation start time is used as `"stimulus_start"` and the simulation end time is used for `"stimulus_end"`. The decay_time_constant_after_stim feature becomes disabled with False. Default is True
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **`features_to_run`** (*`list`*) – Which features to calculate uncertainties for.
- **`interpolate`** (*`list`*) – A list of irregular features to be interpolated.
- **`utility_methods`** (*`list`*) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **`labels`** (*`dictionary`*) – Labels for the axes of each feature, used when plotting.
- **`strict`** (*`bool`*) – If missing info values should raise an error.

Raises

- `ValueError` – If strict is True and `"stimulus_start"` and `"stimulus_end"` are missing from *info*.
- `ValueError` – If stimulus_start >= stimulus_end.
- `ImportError` – If Efel is not installed.

Notes

Efel features take the parameters `(time, values, info)` and require info["stimulus_start"] and info["stimulus_end"] to be set.

Implemented Efel features are:

AHP1_depth_from_peak	AHP2_depth_from_peak	AHP_depth
AHP_depth_abs	AHP_depth_abs_slow	AHP_depth_diff
AHP_depth_from_peak	AHP_slow_time	AHP_time_from_peak
AP1_amp	AP1_begin_voltage	AP1_begin_width
AP1_peak	AP1_width	AP2_AP1_begin_width_diff
AP2_AP1_diff	AP2_AP1_peak_diff	AP2_amp
AP2_begin_voltage	AP2_begin_width	AP2_peak
AP2_width	AP_amplitude	AP_amplitude_change
AP_amplitude_diff	AP_amplitude_from_voltagebase	AP_begin_indices
AP_begin_time	AP_begin_voltage	AP_begin_width
AP_duration	AP_duration_change	AP_duration_half_width
AP_duration_half_width_change	AP_end_indices	AP_fall_indices
AP_fall_rate	AP_fall_rate_change	AP_fall_time
AP_height	AP_phaseslope	AP_phaseslope_AIS
AP_rise_indices	AP_rise_rate	AP_rise_rate_change
AP_rise_time	AP_width	APlast_amp
APlast_width	BAC_maximum_voltage	BAC_width
BPAPAmplitudeLoc1	BPAPAmplitudeLoc2	BPAPHeightLoc1
BPAPHeightLoc2	BPAPatt2	BPAPatt3
E10	E11	E12
E13	E14	E15
E16	E17	E18
E19	E2	E20
E21	E22	E23
E24	E25	E26
E27	E3	E39
E39_cod	E4	E40

Continued on next page

Table 2 – continued from previous page

E5	E6	E7
E8	E9	ISI_CV
ISI_log_slope	ISI_log_slope_skip	ISI_semilog_slope
ISI_values	ISIs	Spikecount
Spikecount_stimint	adaptation_index	adaptation_index2
all_ISI_values	amp_drop_first_last	amp_drop_first_second
amp_drop_second_last	burst_ISI_indices	burst_mean_freq
burst_number	check_AISInitiation	decay_time_constant_after_stim
depolarized_base	doublet_ISI	fast_AHP
fast_AHP_change	initburst_sahp	initburst_sahp_ssse
initburst_sahp_vb	interburst_voltage	inv_fifth_ISI
inv_first_ISI	inv_fourth_ISI	inv_last_ISI
inv_second_ISI	inv_third_ISI	inv_time_to_first_spike
irregularity_index	is_not_stuck	max_amp_difference
maximum_voltage	maximum_voltage_from_voltagebase	mean_AP_amplitude
mean_frequency	min_AHP_indices	min_AHP_values
min_voltage_between_spikes	minimum_voltage	number_initial_spikes
ohmic_input_resistance	ohmic_input_resistance_vb_ssse	peak_indices
peak_time	peak_voltage	sag_amplitude
sag_ratio1	sag_ratio2	single_burst_ratio
spike_half_width	spike_width2	steady_state_hyper
steady_state_voltage	steady_state_voltage_stimend	time
time	time_constant	time_to_first_spike
time_to_last_spike	time_to_second_spike	trace_check
voltage	voltage	voltage_after_stim
voltage_base	voltage_deflection	voltage_deflection_begin
voltage_deflection_vb_ssse		

See also:

`uncertainpy.features.EfelFeatures.reference_feature` reference_feature showing the requirements of a Efel feature function.

`add_features`(*new_features*, *labels={}*)

Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`calculate_all_features`(**model_results*)

Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`calculate_feature`(*feature_name*, **preprocess_results*)

Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.
- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

calculate_features(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

uncertainpy.features.Features.calculate_feature() Method for calculating a single feature.

features_to_run

Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

implemented_features()

Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

interpolate

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

labels

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

preprocess

Preprossesing of the time *time* and results *values* from the model, before the features are calculated.

No preprocessing is performed, and the direct model results are currently returned. If preprocessing is needed it should follow the below format.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns Returns any number of values that are sent to each feature. The values returned must compatible with the input arguments of all features.

Return type preprocess_results

Notes

Perform a preprossesing of the model results before the results are sent to the calculation of each feature. It is used to perform common calculations that each feature needs to perform, to reduce the number of necessary calculations. The values returned must therefore be compatible with the input arguments to each features.

See also:

`uncertainpy.models.Model.run` The model run method

reference_feature(*time*, *values*, *info*)

An example of an Efel feature. Efel feature functions have the following requirements, and the given parameters must either be returned by `model.run` or `features.preprocess`.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **values** (*array_like*) – Result of the model.
- **info** (*dictionary*) – A dictionary with info["stimulus_start"] and info["stimulus_end"] set.

Returns

- **time** (*None*) – No mean Efel feature has time values, so None is returned instead.
- **values** (*array_like*) – The feature results, *values*. Returns None if there are no feature results and that evaluation are disregarded.

See also:

`uncertainpy.features.Features.preprocess()` The features preprocess method.

`uncertainpy.models.Model.run()` The model run method

validate (*feature_name*, **feature_result*)
Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **`time_feature`** [`{None, numpy.nan, array_like}`] Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **`values`** [`{None, numpy.nan, array_like}`] The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

8.5 NetworkFeatures

`NetworkFeatures` contains a set of features relevant for the output of network models and are calculated using the Elephant software. This set of features require that the model returns the simulation end time and a list of spiketrains, which are the times a given neuron spikes. The implemented features are:

1. `average_firing_rate` – Mean firing rate (for a single recorded neuron).
2. `instantaneous_rate` – Instantaneous firing rate (averaged over all recorded neurons within a small time window).
3. `mean_isi` – Average interspike interval (averaged over all recorded neurons).
4. `cv` – Coefficient of variation of the interspike interval (for a single recorded neuron).
5. `average_cv` – average coefficient of variation of the interspike interval (averaged over all recorded neurons).
6. `local_variation` – Local variation (variability of interspike intervals for a single recorded neuron).
7. `average_local_variation` – Mean local variation (variability of interspike intervals averaged over all recorded neurons).
8. `fanofactor` – Fanofactor (variability of spiketrains).
9. `victor_purpura_dist` – Victor purpura distance (spiketrain dissimilarity between two recorded neurons).
10. `van_rossum_dist` – Van rossum distance (spiketrain dissimilarity between two recorded neurons).
11. `binned_isi` – Histogram of the interspike intervals (for all recorded neurons).
12. `corrcoef` – Pairwise Pearson's correlation coefficients (between the spiketrains of two recorded neurons).
13. `covariance` – Covariance (between the spiketrains of two recorded neurons).

The use of the `NetworkFeatures` class in Uncertainpy follows the same logic as the use of the other feature classes, and custom features can easily be included. As with *SpikingFeatures*, `NetworkFeatures` implements a `preprocess()` method. This `preprocess` returns the following objects:

1. End time of the simulation (`end_time`).
2. A list of NEO spiketrains (`spiketrains`).

Each feature function therefore require the same objects as input arguments. Note that a `info` object is not used.

8.5.1 API Reference

class `uncertainpy.features.`**`NetworkFeatures`**(*new_features=None*, *features_to_run=u'all'*, *interpolate=None*, *labels={}*, *units=None*, *instantaneous_rate_nr_samples=50*, *isi_bin_size=1*, *corrcoef_bin_size=1*, *covariance_bin_size=1*, *logger_level=u'info'*)

Network features of a model result, works with all models that return the simulation end time, and a list of spiketrains.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.
- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **interpolate** (*{None, "all", str, list of feature names}, optional*) – Which features are irregular, meaning they have a varying number of time points between evaluations. An interpolation is performed on each irregular feature to create regular results. If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **units** (*{None, Quantities unit}, optional*) – The Quantities unit of the time in the model. If None, ms is used. The default is None.
- **instantaneous_rate_nr_samples** (*int*) – The number of samples used to calculate the instantaneous rate. Default is 50.
- **isi_bin_size** (*int*) – The size of each bin in the `binned_isi` method. Default is 1.
- **corrcoef_bin_size** (*int*) – The size of each bin in the `corrcoef` method. Default is 1.
- **covariance_bin_size** (*int*) – The size of each bin in the `covariance` method. Default is 1.

- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **`features_to_run`** (*`list`*) – Which features to calculate uncertainties for.
- **`interpolate`** (*`list`*) – A list of irregular features to be interpolated.
- **`utility_methods`** (*`list`*) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **`labels`** (*`dictionary`*) – Labels for the axes of each feature, used when plotting.
- **`logger`** (*`logging.Logger`*) – Logger object responsible for logging to screen or file.
- **`instantaneous_rate_nr_samples`** (*`int`*) – The number of samples used to calculate the instantaneous rate. Default is 50.
- **`isi_bin_size`** (*`int`*) – The size of each bin in the `binned_isi` method. Default is 1.
- **`corrcoef_bin_size`** (*`int`*) – The size of each bin in the `corrcoef` method. Default is 1.
- **`covariance_bin_size`** (*`int`*) – The size of each bin in the `covariance` method. Default is 1.

Notes

Implemented features are:

cv	average_cv	average_isi,
local_variation mean	local_variation	average_firing_rate
instantaneous_rate	fanofactor	van_rossum_dist
victor_purpura_dist	binned_isi	corrcoef
covariance		

All features in this set of features take the following input arguments:

simulation_end [float] The simulation end time

neo_spiketrains [list] A list of Neo spiketrains.

The model must return:

simulation_end [float] The simulation end time

spiketrains [list] A list of spiketrains, each spiketrain is a list of the times when a given neuron spikes.

Raises `ImportError` – If elephant or quantities is not installed.

See also:

`uncertainpy.features.Features.reference_feature` reference_feature showing the requirements of a feature function.

`add_features`(*new_features*, *labels={}*)

Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`average_cv` (*simulation_end*, *spiketrains*)
Calculate the average coefficient of variation.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **values** (*float*) – The average coefficient of variation of each spiketrain.

`average_firing_rate` (*simulation_end*, *spiketrains*)
Calculate the mean firing rate.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **average_firing_rate** (*float*) – The mean firing rate of all neurons.

`average_isi` (*simulation_end*, *spiketrains*)
Calculate the average interspike interval (isi) variation for each neuron.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **average_isi** (*float*) – The average interspike interval.

`average_local_variation`(*simulation_end*, *spiketrains*)

Calculate the average of the local variation.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **average_local_variation** (*float*) – The average of the local variation for each spiketrain.

`binned_isi`(*simulation_end*, *spiketrains*)

Calculate a histogram of the interspike interval.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*array*) – The center of each bin.
- **binned_isi** (*array*) – The binned interspike intervals.

`calculate_all_features`(**model_results*)

Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`calculate_feature`(*feature_name*, **preprocess_results*)

Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.

- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

calculate_features(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

corrcoef(*simulation_end, spiketrains*)

Calculate the pairwise Pearson's correlation coefficients.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **values** (*2D array*) – The pairwise Pearson's correlation coefficients.

covariance(*simulation_end, spiketrains*)

Calculate the pairwise covariances.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **values** (*2D array*) – The pairwise covariances.

`cv`(*simulation_end*, *spiketrains*)
Calculate the coefficient of variation for each neuron.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **values** (*array*) – The coefficient of variation for each spiketrain.

`fanofactor`(*simulation_end*, *spiketrains*)
Calculate the fanofactor.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **fanofactor** (*float*) – The fanofactor.

`features_to_run`
Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

`implemented_features()`
Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

`instantaneous_rate`(*simulation_end*, *spiketrains*)
Calculate the mean instantaneous firing rate.

Parameters

- **simulation_end** (*float*) – The simulation end time.

- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*array*) – Time of the instantaneous firing rate.
- **instantaneous_rate** (*float*) – The instantaneous firing rate.

`interpolate`

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

`labels`

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

`local_variation`(*simulation_end*, *spiketrains*)

Calculate the measure of local variation.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **local_variation** (*list*) – The local variation for each spiketrain.

`preprocess`(*simulation_end*, *spiketrains*)

Preprossesing of the simulation end time *simulation_end* and spiketrains *spiketrains* from the model, before the features are calculated.

Parameters

- **simulation_end** (*float*) – The simulation end time
- **spiketrains** (*list*) – A list of spiketrains, each spiketrain is a list of the times when a given neuron spikes.

Returns

- **simulation_end** (*float*) – The simulation end time
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Raises `ValueError` – If *simulation_end* is np.nan or None.

Notes

This preprocessing makes it so all features get the input *simulation_end* and *spiketrains*.

See also:

`uncertainpy.models.Model.run()` The model run method

`reference_feature`(*simulation_end*, *neo_spiketrains*)

An example of an GeneralNetworkFeature. The feature functions have the following requirements, and the given parameters must either be returned by `model.run` or `features.preprocess`.

Parameters

- **simulation_end** (*float*) – The simulation end time
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** (*array_like*) – The feature results, *values*. Returns None if there are no feature results and that evaluation are disregarded.

See also:

`uncertainpy.features.GeneralSpikingFeatures.preprocess()` The GeneralSpikingFeatures preprocess method.

`uncertainpy.models.Model.run()` The model run method

`validate`(*feature_name*, **feature_result*)

Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **`time_feature`** `[{None, numpy.nan, array_like}]` Time values, or equivalent, of the feature, if no time values return None or numpy.nan.

- **values** [{None, numpy.nan, array_like}] The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

van_rossum_dist (*simulation_end*, *spiketrains*)
Calculate van Rossum distance.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **van_rossum_dist** (*2D array*) – The van Rossum distance.

victor_purpura_dist (*simulation_end*, *spiketrains*)
Calculate the Victor-Purpura's distance.

Parameters

- **simulation_end** (*float*) – The simulation end time.
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*None*)
- **values** (*2D array*) – The Victor-Purpura's distance.

8.6 GeneralNetworkFeatures

`GeneralNetworkFeatures` implements the preprocessing of spiketrains, and create NEO spiketrains, but does not implement any features in itself. This set of features require that the model returns the simulation end time and a list of spiketrains, which are the times a given neuron spikes. The `preprocess()` method changes the input given to the feature functions, and as such each network feature function has the following input arguments:

1. End time of the simulation (`end_time`).
2. A list of NEO spiketrains (`spiketrains`).

8.6.1 API Reference

class uncertainpy.features.**GeneralNetworkFeatures**(*new_features=None*, *features_to_run=u'all'*, *interpolate=None*, *labels={}*, *units=None*, *logger_level=u'info'*)
Class for creating NEO spiketrains from a list of spiketrains, for network models. The model must return the simulation end time and a list of spiketrains.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.

- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **new_utility_methods (*{None, list}, optional*) – A list of new utility methods. All methods in this class that is n**
  the list of utility methods, is considered to be a feature. Default is None.

  interpolate [{None, "all", str, list of feature names}, optional] Which features are irregular, meaning they have a varying number of time points between evaluations. An interpolation is performed on each irregular feature to create regular results. If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **units** (*{None, Quantities unit}, optional*) – The Quantities unit of the time in the model. If None, ms is used. The default is None.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **`features_to_run`** (*`list`*) – Which features to calculate uncertainties for.
- **`interpolate`** (*`list`*) – A list of irregular features.
- **`utility_methods`** (*`list`*) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **`labels`** (*`dictionary`*) – Labels for the axes of each feature, used when plotting.

Notes

All features in this set of features take the following input arguments:

simulation_end [float] The simulation end time

neo_spiketrains [list] A list of Neo spiketrains.

The model must return:

simulation_end [float] The simulation end time

spiketrains [list] A list of spiketrains, each spiketrain is a list of the times when a given neuron spikes.

Raises `ImportError` – If neo or quantities is not installed.

See also:

GeneralNetworkFeatures.preprocess

GeneralNetworkFeatures.reference_feature reference_feature showing the requirements of a feature function.

add_features (*new_features*, *labels={}*)
Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

uncertainpy.features.Features.reference_feature() reference_feature showing the requirements of a feature function.

calculate_all_features (**model_results*)
Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

uncertainpy.features.Features.calculate_feature() Method for calculating a single feature.

calculate_feature (*feature_name*, **preprocess_results*)
Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.
- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

`calculate_features`(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`features_to_run`

Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

implemented_features()

Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

interpolate

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

labels

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

preprocess(*simulation_end*, *spiketrains*)

Preprossesing of the simulation end time *simulation_end* and spiketrains *spiketrains* from the model, before the features are calculated.

Parameters

- **simulation_end** (*float*) – The simulation end time
- **spiketrains** (*list*) – A list of spiketrains, each spiketrain is a list of the times when a given neuron spikes.

Returns

- **simulation_end** (*float*) – The simulation end time
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Raises `ValueError` – If *simulation_end* is np.nan or None.

Notes

This preprocessing makes it so all features get the input *simulation_end* and *spiketrains*.

See also:

uncertainpy.models.Model.run() The model run method

reference_feature(*simulation_end*, *neo_spiketrains*)

An example of an GeneralNetworkFeature. The feature functions have the following requirements, and the given parameters must either be returned by `model.run` or `features.preprocess`.

Parameters

- **simulation_end** (*float*) – The simulation end time
- **neo_spiketrains** (*list*) – A list of Neo spiketrains.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** (*array_like*) – The feature results, *values*. Returns None if there are no feature results and that evaluation are disregarded.

See also:

uncertainpy.features.GeneralSpikingFeatures.preprocess() The GeneralSpikingFeatures preprocess method.

uncertainpy.models.Model.run() The model run method

validate(*feature_name*, **feature_result*)

Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **`time_feature`** `[{None, numpy.nan, array_like}]` Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **`values`** `[{None, numpy.nan, array_like}]` The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

8.7 GeneralSpikingFeatures

`GeneralSpikingFeatures` implements the preprocessing of voltage traces, and locate spikes in the voltage traces, but does not implement any features in itself. The *`preprocess()`* method changes the input given to the feature functions, and as such each spiking feature function has the following input arguments:

1. The `time` array returned by the model simulation.
2. An *Spikes* object (`spikes`) which contain the spikes found in the model output.
3. An `info` dictionary with `info["stimulus_start"]` and `info["stimulus_end"]` set.

8.7.1 API Reference

class `uncertainpy.features.`**`GeneralSpikingFeatures`**(*new_features=None, features_to_run=u'all', interpolate=None, threshold=-30, end_threshold=-10, extended_spikes=False, trim=True, normalize=False, min_amplitude=0, min_duration=0, labels={}, logger_level=u'info'*)

Class for calculating spikes of a model, works with single neuron models and voltage traces.

Parameters

- **new_features** (*{None, callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`. If None, no features are added. Default is None.
- **features_to_run** (*{"all", None, str, list of feature names}, optional*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list `[]`, no features are calculated. If str, only that feature is calculated. If list of feature names, all the listed features are calculated. Default is `"all"`.
- **new_utility_methods** (*{None, list}, optional*) – A list of new utility methods. All methods in this class that is not in the list of utility methods, is considered to be a feature. Default is None.
- **interpolate** (*{None, "all", str, list of feature names}, optional*) – Which features are irregular, meaning they have a varying number of time points between evaluations. An interpolation is performed on each irregular feature to create regular results. If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.
- **threshold** (*{float, int, "auto"}, optional*) – The threshold where the model result is considered to have a spike. If "auto" the threshold is set to the standard variation of the result. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Generally negative values give the best results. Default is -10.
- **extended_spikes** (*bool, optional*) – If the found spikes should be extended further out than the threshold cuttoff. If True the spikes is considered to start and end where the derivative equals 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.
- **normalize** (*bool, optional*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value. Default is False.

- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.
- **labels** (*dictionary, optional*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **spikes** (*Spikes object*) – A Spikes object that contain all spikes.
- **threshold** (*{float, int, "auto"}, optional*) – The threshold where the model result is considered to have a spike. If "auto" the threshold is set to the standard variation of the result. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Default is -10.
- **extended_spikes** (*bool*) – If the found spikes should be extended further out than the threshold cuttoff.
- **trim** (*bool*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends.
- **normalize** (*bool*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.
- **features_to_run** (*list*) – Which features to calculate uncertainties for.
- **interpolate** (*list*) – A list of irregular features to be interpolated.
- **utility_methods** (*list*) – A list of all utility methods implemented. All methods in this class that is not in the list of utility methods is considered to be a feature.
- **labels** (*dictionary*) – Labels for the axes of each feature, used when plotting.

See also:

uncertainpy.features.Features.reference_feature reference_feature showing the requirements of a feature function.

uncertainpy.features.Spikes Class for finding spikes in the model result.

add_features (*new_features*, *labels={}*)
Add new features.

Parameters

- **new_features** (*{callable, list of callables}*) – The new features to add. The feature functions have the requirements stated in `reference_feature`.
- **labels** (*dictionary, optional*) – A dictionary with the labels for the new features. The keys are the feature function names and the values are a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

Raises `TypeError` – Raises a TypeError if *new_features* is not callable or list of callables.

Notes

The features added are not added to `features_to_run`. `features_to_run` must be set manually afterwards.

See also:

uncertainpy.features.Features.reference_feature() reference_feature showing the requirements of a feature function.

calculate_all_features (**model_results*)
Calculate all implemented features.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": t, "values": U}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

uncertainpy.features.Features.calculate_feature() Method for calculating a single feature.

calculate_feature (*feature_name*, **preprocess_results*)
Calculate feature with *feature_name*.

Parameters

- **feature_name** (*str*) – Name of feature to calculate.

- ***preprocess_results** – The values returned by `preprocess`. These values are sent as input arguments to each feature. By default preprocess returns the values that `model.run()` returns, which contains *time* and *values*, and then any number of optional *info* values. The implemented features require that *info* is a single dictionary with the information stored as key-value pairs. Certain features require specific keys to be present.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values returns None or numpy.nan.
- **values** (*array_like*) – The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated.

Raises `TypeError` – If *feature_name* is a utility method.

See also:

`uncertainpy.models.Model.run()` The model run method

`calculate_features`(**model_results*)

Calculate all features in `features_to_run`.

Parameters ***model_results** – Variable length argument list. Is the values that `model.run()` returns. By default it contains *time* and *values*, and then any number of optional *info* values.

Returns **results** – A dictionary where the keys are the feature names and the values are a dictionary with the time values *time* and feature results on *values*, on the form `{"time": time, "values": values}`.

Return type dictionary

Raises `TypeError` – If *feature_name* is a utility method.

Notes

Checks that the feature returns two values.

See also:

`uncertainpy.features.Features.calculate_feature()` Method for calculating a single feature.

`calculate_spikes`(*time, values, threshold=-30, end_threshold=-10, extended_spikes=False, trim=True, normalize=False, min_amplitude=0, min_duration=0*)

Calculating spikes of a model result, works with single neuron models and voltage traces.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **values** (*array_like*) – Result of the model.
- **threshold** (*{float, int, "auto"}, optional*) – The threshold where the model result is considered to have a spike. If "auto" the threshold is set to the standard variation of the result. Default is -30.
- **end_threshold** (*{int, float}, optional*) – The end threshold for a spike relative to the threshold. Default is -10.

- **extended_spikes** (*bool, optional*) – If the found spikes should be extended further out than the threshold cuttoff. If True the spikes is considered to start and end where the derivative equals 0.5. Default is False.
- **trim** (*bool, optional*) – If the spikes should be trimmed back from the termination threshold, so each spike is equal the threshold at both ends. Default is True.
- **normalize** (*bool, optional*) – If the voltage traceshould be normalized before the spikes are found. If normalize is used threshold must be between [0, 1], and the end_threshold a similar relative value. Default is False.
- **min_amplitude** (*{int, float}, optional*) – Minimum height for what should be considered a spike. Default is 0.
- **min_duration** (*{int, float}, optional*) – Minimum duration for what should be considered a spike. Default is 0.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it returns None or numpy.nan.
- **values** (*Spikes*) – The spikes found in the model results.

See also:

`uncertainpy.features.Features.reference_feature()` reference_feature showing the requirements of a feature function.

`uncertainpy.features.Spikes()` Class for finding spikes in the model result.

`features_to_run`

Which features to calculate uncertainties for.

Parameters **new_features_to_run** (*{"all", None, str, list of feature names}*) – Which features to calculate uncertainties for. If `"all"`, the uncertainties are calculated for all implemented and assigned features. If None, or an empty list , no features are calculated. If str, only that feature is calculated. If list of feature names, all listed features are calculated. Default is `"all"`.

Returns A list of features to calculate uncertainties for.

Return type list

`implemented_features``()`

Return a list of all callable methods in feature, that are not utility methods, does not starts with "_" and not a method of a general python object.

Returns A list of all callable methods in feature, that are not utility methods.

Return type list

`interpolate`

Features that require an interpolation.

Which features are interpolated, meaning they have a varying number of time points between evaluations. An interpolation is performed on each interpolated feature to create regular results.

Parameters **new_interpolate** (*{None, "all", str, list of feature names}*) – If `"all"`, all features are interpolated. If None, or an empty list, no features are interpolated. If str, only that feature is interpolated. If list of feature names, all listed features are interpolated. Default is None.

Returns A list of irregular features to be interpolated.

Return type list

labels

Labels for the axes of each feature, used when plotting.

Parameters **new_labels** (*dictionary*) – A dictionary with key as the feature name and the value as a list of labels for each axis. The number of elements in the list corresponds to the dimension of the feature. Example:

```
new_labels = {"0d_feature": ["x-axis"],
              "1d_feature": ["x-axis", "y-axis"],
              "2d_feature": ["x-axis", "y-axis", "z-axis"]
             }
```

preprocess (*time*, *values*, *info*)

Calculating spikes from the model result.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **values** (*array_like*) – Result of the model.
- **info** (*dictionary*) – A dictionary with info["stimulus_start"] and info["stimulus_end"].

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it returns None or numpy.nan.
- **values** (*Spikes*) – The spikes found in the model results.
- **info** (*dictionary*) – A dictionary with info["stimulus_start"] and info["stimulus_end"].

Notes

Also sets self.values = values, so features have access to self.values if necessary.

See also:

uncertainpy.models.Model.run() The model run method

uncertainpy.features.Spikes() Class for finding spikes in the model result.

reference_feature (*time*, *spikes*, *info*)

An example of an GeneralSpikingFeature. The feature functions have the following requirements, and the input arguments must either be returned by `Model.run` or `SpikingFeatures.preprocess`.

Parameters

- **time** (*{None, numpy.nan, array_like}*) – Time values of the model. If no time values it is None or numpy.nan.
- **spikes** (*Spikes*) – Spikes found in the model result.
- **info** (*dictionary*) – A dictionary with info["stimulus_start"] and info["stimulus_end"] set.

Returns

- **time** (*{None, numpy.nan, array_like}*) – Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** (*array_like*) – The feature results, *values*. Returns None if there are no feature results and that evaluation are disregarded.

See also:

uncertainpy.features.GeneralSpikingFeatures.preprocess() The GeneralSpikingFeatures preprocess method.

uncertainpy.models.Model.run() The model run method

validate (*feature_name*, **feature_result*)

Validate the results from `calculate_feature`.

This method ensures each returns *time*, *values*.

Parameters

- **model_results** – Any type of model results returned by `run`.
- **feature_name** (*str*) – Name of the feature, to create better error messages.

Raises

- `ValueError` – If the model result does not fit the requirements.
- `TypeError` – If the model result does not fit the requirements.

Notes

Tries to verify that at least, *time* and *values* are returned from `run`. `model_result` should follow the format: `return time, values, info_1, info_2, ...`. Where:

- **time_feature** [`{None, numpy.nan, array_like}`] Time values, or equivalent, of the feature, if no time values return None or numpy.nan.
- **values** [`{None, numpy.nan, array_like}`] The feature results, *values* must either be regular (have the same number of points for different paramaters) or be able to be interpolated. If there are no feature results return None or `numpy.nan` instead of *values* and that evaluation are disregarded.

CHAPTER 9

Data

Uncertainpy stores all results from the uncertainty quantification and sensitivity analysis in `UncertaintyQuantification.data`, as a `Data` object. The `Data` class works similarly to a Python dictionary. The name of the model or feature is the key, while the values are `DataFeature` objects that stores each statistical metric in in the table below as attributes. Results can be saved and loaded through `Data.save` and `Data.load`.

Calculated statistical metric	Symbol	Variable
Model and feature evaluations	U	`evaluations`
Model and feature times	t	`time`
Mean	$\mathbb{E}$	`mean`
Variance	$\mathbb{V}$	`variance`
5th percentile	P_5	`percentile_5`
95th percentile	P_{95}	`percentile_95`
First order Sobol indices	S	`sobol_first`
Total order Sobol indices	S_T	`sobol_total`
Average of the first order Sobol indices	$\widehat{S}$	`sobol_first_average`
Average of the total order Sobol indices	$\widehat{S}_T$	`sobol_total_average`

An example: if we have performed uncertainty quantification of a spiking neuron model with the number of spikes as one of the features, we get load the data file and get the variance of the number of spikes by typing:

```
data = un.Data()
data.load("filename")
variance = data["nr_spikes"].variance
```

9.1 API reference

9.1.1 Data

class uncertainpy.**Data** (*filename=None*, *backend=u'auto'*, *logger_level=u'info'*)

Store the results of each statistical metric calculated from the uncertainty quantification and sensitivity analysis for each model/features.

Has all standard dictionary methods, such as items, value, contains and so implemented. Can be indexed as a regular dictionary with model/feature names as keys and returns a DataFeature object that contains the data for all statistical metrics for that model/feature. Additionally it contains information on how the calculations was performed

Parameters

- **filename** (*str, optional*) – Name of the file to load data from. If None, no data is loaded. Default is None.
- **backend** (*{"auto", "hdf5", "exdir"}, optional*) – The fileformat used to save and load data to/from file. "auto" assumes the filenamess ends with either ".h5" for HDF5 files or ".exdir" for Exdir files. If unknown fileextension defaults to saving as HDF5 files. "hdf5" saves and loads files from HDF5 files. "exdir" saves and loads files from Exdir files. Default is "auto".
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed Default logger level is "info".

Variables

- **uncertain_parameters** (*list*) – A list of the uncertain parameters in the uncertainty quantification.
- **model_name** (*str*) – Name of the model.
- **incomplete** (*list*) – List of all model/features that have missing model/feature evaluations.
- **error** (*list*) – List of all model/features that were irregular, but not set to be interpolated.
- **method** (*str*) – A string that describes the method used to perform the uncertainty quantification.
- **data** (*dictionary*) – A dictionary with a DataFeature for each model/feature.
- **data_information** (*list*) – List of attributes containing additional information.

Notes

The statistical metrics calculated for each feature and model in Uncertainpy are:

- `evaluations` - the results from the model/feature evaluations.
- `time` - the time of the model/feature.
- `mean` - the mean of the model/feature.
- `variance`. - the variance of the model/feature.
- `percentile_5` - the 5th percentile of the model/feature.
- `percentile_95` - the 95th percentile of the model/feature.

- `sobol_first` - the first order Sobol indices (sensitivity) of the model/feature.
- `sobol_first_average` - the average of the first order Sobol indices (sensitivity) of the model/feature.
- `sobol_total` - the total order Sobol indices (sensitivity) of the model/feature.
- `sobol_total_average` - the average of the total order Sobol indices (sensitivity) of the model/feature.

Raises `ValueError` – If unsupported backend is chosen.

See also:

uncertainpy.DataFeature

`__delitem__`(*feature*)
Delete data for *feature*.

Parameters **feature** (*str*) – Name of feature.

`__getitem__`(*feature*)
Get the DataFeature containing the data for *feature*.

Parameters **feature** (*str*) – Name of feature/model.

Returns The DataFeature containing the data for *feature*.

Return type DataFeature

`__iter__`()
Iterate over each feature/model that has not errored.

Yields *str* – Name of feature/model.

`__len__`()
Get the number of model/features that have not errored.

Returns The number of model/features that have not errored.

Return type int

`__setitem__`(*feature*, *data*)
Set *data* for *feature*. *Data* must be a DataFeature object.

Parameters

- **feature** (*str*) – Name of feature/model.
- **data** (*DataFeature*) – DataFeature with the data for *feature*.

Raises `ValueError` – If *data* is not a DataFeature.

`__str__`()
Convert all data to a readable string.

Returns A human readable string of all stored data.

Return type str

`add_features`(*features*)
Add features (which contain no data).

Parameters **features** (*{str, list}*) – Name of feature to add, or list of features to add.

`clear`()
Clear all data.

`get`(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_labels (*feature*)
Get labels for a *feature*. If no labels are defined, returns a list with the correct number of empty strings.

Parameters **feature** (*str*) – Name of the model or a feature.

Returns A list of labels for plotting, `[x-axis, y-axis, z-axis]`. If no labels are defined (labels = []), returns a list with the correct number of empty strings.

Return type list

items () → list of D's (key, value) pairs, as 2-tuples

iteritems () → an iterator over the (key, value) items of D

iterkeys () → an iterator over the keys of D

itervalues () → an iterator over the values of D

keys () → list of D's keys

load (*filename*)
Load data from a HDF5 or Exdir file with name *filename*.

Parameters **filename** (*str*) – Name of the file to load data from.

Raises

- `ImportError` – If h5py is not installed.
- `ImportError` – If Exdir is not installed.

ndim (*feature*)
Get the number of dimensions of a *feature*.

Parameters **feature** (*str*) – Name of the model or a feature.

Returns The number of dimensions of the model/feature result. Returns None if the feature has no evaluations or only contains nan.

Return type int, None

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem () → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

remove_only_invalid_features ()
Remove all features that only have invalid results (NaN).

save (*filename*)
Save data to a HDF5 or Exdir file with name *filename*.

Parameters **filename** (*str*) – Name of the file to load data from.

Raises

- `ImportError` – If h5py is not installed.
- `ImportError` – If Exdir is not installed.

seed
Seed used in the calculations.

Parameters **new_seed** (*{None, int}*) – Seed used in the calculations. If None, converted to "".

Returns **seed** – Seed used in the calculations.

Return type {int, str}

setdefault (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update ([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values () → list of D's values

9.1.2 DataFeature

class uncertainpy.**DataFeature** (*name*, *evaluations=None*, *time=None*, *mean=None*, *variance=None*, *percentile_5=None*, *percentile_95=None*, *sobol_first=None*, *sobol_first_average=None*, *sobol_total=None*, *sobol_total_average=None*, *labels=[]*)

Store the results of each statistical metric calculated from the uncertainty quantification and sensitivity analysis for a single model/feature.

The statistical metrics can be retrieved as attributes. Additionally, DataFeature implements all standard dictionary methods, such as items, value, contains and so implemented. This means it can be indexed as a regular dictionary with the statistical metric names as keys and returns the values for that statistical metric.

Parameters

- **name** (*str*) – Name of the model/feature.
- **evaluations** (*{None, array_like}, optional.*) – Feature or model result. Default is None.
- **time** (*{None, array_like}, optional.*) – Time evaluations for feature or model. Default is None.
- **mean** (*{None, array_like}, optional.*) – Mean of the feature or model results. Default is None.
- **variance** (*{None, array_like}, optional.*) – Variance of the feature or model results. Default is None.
- **percentile_5** (*{None, array_like}, optional.*) – 5 percentile of the feature or model results. Default is None.
- **percentile_95** (*{None, array_like}, optional.*) – 95 percentile of the feature or model results. Default is None.
- **sobol_first** (*{None, array_like}, optional.*) – First order sensitivity of the feature or model results. Default is None.
- **sobol_first_average** (*{None, array_like}, optional.*) – First order sensitivity of the feature or model results. Default is None.
- **sobol_total** (*{None, array_like}, optional.*) – Total effect sensitivity of the feature or model results. Default is None.
- **sobol_total_average** (*{None, array_like}, optional.*) – Average of the total effect sensitivity of the feature or model results. Default is None.
- **labels** (*list, optional.*) – A list of labels for plotting, `[x-axis, y-axis, z-axis]` Default is `[]`.

Variables

- **name** (`str`) – Name of the model/feature.

- **evaluations** (*{None, array_like}*) – Feature or model output.
- **time** (*{None, array_like}*) – Time values for feature or model.
- **mean** (*{None, array_like}*) – Mean of the feature or model results.
- **variance** (*{None, array_like}*) – Variance of the feature or model results.
- **percentile_5** (*{None, array_like}*) – 5 percentile of the feature or model results.
- **percentile_95** (*{None, array_like}*) – 95 percentile of the feature or model results.
- **sobol_first** (*{None, array_like}*) – First order Sobol indices (sensitivity) of the feature or model results.
- **sobol_first_average** (*{None, array_like}*) – Average of the first order Sobol indices of the feature or model results.
- **sobol_total** (*{None, array_like}*) – Total order Sobol indices (sensitivity) of the feature or model results.
- **sobol_total_average** (*{None, array_like}*) – Average of the total order Sobol indices of the feature or model results.
- **labels** (*list*) – A list of labels for plotting, `[x-axis, y-axis, z-axis]`.

Notes

The statistical metrics calculated in Uncertainpy are:

- `evaluations` - the results from the model/feature evaluations.
- `time` - the time of the model/feature.
- `mean` - the mean of the model/feature.
- `variance`. - the variance of the model/feature.
- `percentile_5` - the 5th percentile of the model/feature.
- `percentile_95` - the 95th percentile of the model/feature.
- `sobol_first` - the first order Sobol indices (sensitivity) of the model/feature.
- `sobol_first_average` - the average of the first order Sobol indices (sensitivity) of the model/feature.
- `sobol_total` - the total order Sobol indices (sensitivity) of the model/feature.
- `sobol_total_average` - the average of the total order Sobol indices (sensitivity) of the model/feature.

__delitem__(*statistical_metric*)
Delete data for *statistical_metric* (set to None).

Parameters **statistical_metric** (*str*) – Name of the statistical metric.

__getitem__(*statistical_metric*)
Get the data for *statistical_metric*.

Parameters **statistical_metric** (*str*) – Name of the statistical metric.

Returns The data for *statistical_metric*.

Return type {array_like, None}

`__iter__`()
Iterate over each statistical metric with data.

Yields *str* – Name of the statistical metric.

`__len__`()
Get the number of data types with data.

Returns The number of data types with data.

Return type int

`__setitem__`(*statistical_metric*, *data*)
Set the data for the statistical metric.

Parameters

- **statistical_metric** (*str*) – Name of the statistical metric.
- **data** (*{array_like, None}*) – The data for the statistical metric.

`clear`() → None. Remove all items from D.

`get`(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`get_metrics`()
Get the names of all statistical metrics that contain data (not None).

Returns List of the names of all statistical metric that contain data.

Return type list

`items`() → list of D's (key, value) pairs, as 2-tuples

`iteritems`() → an iterator over the (key, value) items of D

`iterkeys`() → an iterator over the keys of D

`itervalues`() → an iterator over the values of D

`keys`() → list of D's keys

`ndim`()
Get the number of dimensions the data of a data type. Returns None if no evaluations or all evaluations contain numpy.nan.

Parameters **feature** (*str*) – Name of the model or a feature.

Returns The number of dimensions of the data of the data type.

Return type int

`pop`(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

`popitem`() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

`setdefault`(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

`update`([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

`values`() → list of D's values

CHAPTER 10

Distribution

Functions (that work as closures) used to set the distribution of a parameter to an *interval* around their original value through for example `set_all_distributions()`. An example:

```
# Define a parameter list
parameter_list = [["parameter_1", -67],
                  ["parameter_2", 22]]

# Create the parameters
parameters = un.Parameters(parameter_list)

# Set all parameters to have a uniform distribution
# within a 5% interval around their fixed value
parameters.set_all_distributions(un.uniform(0.05))
```

10.1 API Reference

`uncertainpy.`**`uniform`**`(`*interval*`)`

A closure that creates a function that takes a *parameter* as input and returns a uniform distribution with *interval* around *parameter*.

Parameters **interval** (*int, float*) – The interval of the uniform distribution around *parameter*.

Returns **distribution** – A function that takes *parameter* as input and returns a uniform distribution with *interval* around this *parameter*.

Return type function

Notes

This function ultimately calculates:

```
cp.Uniform(parameter - abs(interval/2.*parameter),
           parameter + abs(interval/2.*parameter)).
```

`uncertainpy.`**`normal`**`(`*interval*`)`

A closure that creates a function that takes a *parameter* as input and returns a Gaussian distribution with standard deviation *interval*parameter* around *parameter*.

Parameters **interval** (*int, float*) – The interval of the standard deviation `interval*parameter` for the Gaussian distribution.

Returns **distribution** – A function that takes a *parameter* as input and returns a Gaussian distribution standard deviation `interval*parameter`.

Return type function

Notes

This function ultimately calculates:

```
cp.Normal(parameter, abs(interval*parameter))
```

CHAPTER 11

Plotting

PlotUncertainty creates plot of the data from the uncertainty quantification and sensitivity analysis. `PlotUncertainpy` plots the results for all zero and one dimensional statistical metrics, and some of the two dimensional statistical metrics It is intended as a quick way to get an overview of the data, and does not create publication ready plots. Custom plots of the data can easily be created by retrieving the results from the *Data* class.

11.1 API Reference

class uncertainpy.plotting.**PlotUncertainty**(*filename=None*, *folder=u'figures/'*, *figureformat=u'.png'*, *logger_level=u'info'*)

Plotting the results from the uncertainty quantification and sensitivity analysis.

Parameters

- **filename** (*{None, str}, optional*) – The name of the data file. If given the file is loaded. If None, no file is loaded. Default is None.
- **folder** (*str, optional*) – The folder where to save the plots. Creates a new folder if it does not exist. Default is "figures/".
- **figureformat** (*str, optional*) – The format to save the plots in. Given as ".xxx". All formats supported by Matplotlib are available. Default is ".png",
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed Default logger level is "info".

Variables

- **folder** (*str*) – The folder where to save the plots.
- **figureformat** (*str, optional*) – The format to save the plots in. Given as ".xxx". All formats supported by Matplotlib are available.
- **data** (*Data*) – A data object that contains the results from the uncertainty quantification. Contains all model and feature values, as well as all calculated statistical metrics.

`all_evaluations` (*foldername=u'evaluations'*)
Plot all evaluations for all model and features.

Parameters **foldername** (*str, optional*) – Name of folder where to save all plots. The folder is created if it does not exist. Default folder is named "evaluations".

`attribute_feature_1d` (*feature=None*, *attribute=u'mean'*, *attribute_name=u'mean'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot a 1 dimensional attribute for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **attribute** (*{"mean", "variance"}, optional*) – Attribute to plot, either the mean or variance. Default is "mean".
- **attribute_name** (*str*) – Name of the attribute, used as title and name of the plot. Default is "mean".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.
- `ValueError` – If the attribute is not a supported attribute, either "mean" or "variance".

`attribute_feature_2d` (*feature=None*, *attribute=u'mean'*, *attribute_name=u'mean'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot a 2 dimensional attribute for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **attribute** (*{"mean", "variance"}, optional*) – Attribute to plot, either the mean or variance. Default is "mean".
- **attribute_name** (*str*) – Name of the attribute, used as title and name of the plot. Default is "mean".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 2 dimensional.
- `ValueError` – If the attribute is not a supported attribute, either "mean" or "variance".

`average_sensitivity` (*feature*, *sensitivity=u'first'*, *hardcopy=True*, *show=False*)
Plot the average of the sensitivity for a specific model/feature.

Parameters

- **feature** (*{None, str}*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".
- `ValueError` – If feature does not exist.

`average_sensitivity_all`(*sensitivity=u'first'*, *hardcopy=True*, *show=False*)
Plot the average of the sensitivity for all model/features.

Parameters

- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

`average_sensitivity_grid`(*sensitivity=u'first'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot the average of the sensitivity for all model/features in their own plots in the same figure.

Parameters

- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

`convert_sensitivity`(*sensitivity*)
Convert a sensitivity str to the correct sensitivity attribute, and a full name.

Parameters sensitivity (*{"sobol_first", "first", "sobol_total", "total", None}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices.

Returns

- **sensitivity** (*str*) – Name of the sensitivity attribute. Either sobol_first", "sobol_total", or the unchanged input.
- **full_text** (*str*) – Complete name of the sensitivity. Either "", or "first order Sobol indices" or "total order Sobol indices".

evaluations(*feature=None*, *foldername=u"*, ***plot_kwargs*)
Plot all evaluations for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **foldername** (*str, optional*) – Name of folder where to save all plots. The folder is created if it does not exist. Default folder is named "featurename_evaluations".
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `NotImplementedError` – If the model/feature have more than 2 dimensions.
- `AttributeError` – If the dimensions of the evaluations is not valid.

evaluations_0d(*feature=None*, *foldername=u"*, ***plot_kwargs*)
Plot all 0D evaluations for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **foldername** (*str, optional*) – Name of folder where to save all plots. The folder is created if it does not exist.Default folder is named "featurename_evaluations".
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the evaluations are not 0 dimensional.

evaluations_1d(*feature=None*, *foldername=u"*, ***plot_kwargs*)
Plot all 1D evaluations for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **foldername** (*str, optional*) – Name of folder where to save all plots. The folder is created if it does not exist. Default folder is named "featurename_evaluations".
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the evaluations are not 1 dimensional.

evaluations_2d(*feature=None*, *foldername=u"*, ***plot_kwargs*)
Plot all 2D evaluations for a specific model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **foldername** (*str, optional*) – Name of folder where to save all plots. The folder is created if it does not exist. Default folder is named "featurename_evaluations".
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the evaluations are not 2 dimensional.

feature_0d(*feature*, *sensitivity=u'first'*, *hardcopy=True*, *show=False*, *max_legend_size=5*)

Plot all attributes (mean, variance, p_05, p_95 and sensitivity of it exists) for a 0 dimensional model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total", None}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. If None, no sensitivity is plotted. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- **max_legend_size** (*int, optional*) – The max number of legends in a row. Default is 5.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 0 dimensional.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", "total" or None.

features_0d(*sensitivity=u'first'*, *hardcopy=True*, *show=False*)

Plot the results for all 0 dimensional model/features.

Parameters

- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

features_1d(*sensitivity=u'first'*)

Plot all data for all 1 dimensional model/features.

For each model/feature plots `mean_1d`, `variance_1d`, `mean_variance_1d`, and `prediction_interval_1d`. If sensitivity also plot `sensitivity_1d`, `sensitivity_1d_combined`, and `sensitivity_1d_grid`.

Parameters **sensitivity** (*{"sobol_first", "first", "sobol_total", "total", None}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. If None, no sensitivity is plotted. Default is "first".

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", "total" or None.

See also:

uncertainpy.plotting.PlotUncertainty.mean_1d(), *uncertainpy.plotting.PlotUncertainty.variance_1d()*, *uncertainpy.plotting.PlotUncertainty.mean_variance_1d()*, *uncertainpy.plotting.PlotUncertainty.prediction_interval_1d()*, *uncertainpy.plotting.PlotUncertainty.sensitivity_1d()*, *uncertainpy.plotting.PlotUncertainty.sensitivity_1d_combined()*, *uncertainpy.plotting.PlotUncertainty.sensitivity_1d_grid()*

features_2d()

Plot all implemented plots for all 2 dimensional model/features. For each model/feature plots `mean_2d`, and `variance_2d`.

Raises `ValueError` – If a Datafile is not loaded.

folder

The folder where to save all plots.

Parameters **new_folder** (*str*) – Name of new folder where to save all plots. The folder is created if it does not exist.

load(*filename*)

Load data from a HDF5 or Exdir file with name *filename*.

Parameters **filename** (*str*) – Name of the file to load data from.

mean_1d(*feature*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the mean for a specific 1 dimensional model/feature.

Parameters

- **feature** (*str*) – The name of the model/feature.
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.

`mean_2d` (*feature*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot the mean for a specific 2 dimensional model/feature.

Parameters

- **feature** (*str*) – The name of the model/feature.
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 2 dimensional.

`mean_variance_1d` (*feature=None*, *new_figure=True*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot the mean and variance for a specific 1 dimensional model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.

`plot` (*condensed=True*, *sensitivity=u'first'*)
Plot the subset of data that shows all information in the most concise way, with the chosen sensitivity.

Parameters

- **condensed** (*bool, optional*) – If the results should be plotted in the most concise way. If not, all plots are created. Default is True.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. If None, no sensitivity is plotted. Default is "first".

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", "total", or None.

`plot_all` (*sensitivity=u'first'*)
Plot the results for all model/features, with the chosen sensitivity.

Parameters sensitivity (*{"sobol_first", "first", "sobol_total", "total", None}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. If None, no sensitivity is plotted. Default is "first".

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", "total", or None.

plot_all_sensitivities()

Plot the results for all model/features, with all sensitivities.

Raises `ValueError` – If a Datafile is not loaded.

plot_condensed(*sensitivity=u'first'*)

Plot the subset of data that shows all information in the most concise way, with the chosen sensitivity.

Parameters **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. If None, no sensitivity is plotted. Default is "first".

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", "total", or None.

prediction_interval_1d(*feature=None*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the prediction interval for a specific 1 dimensional model/feature.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.

sensitivity_1d(*feature=None*, *sensitivity=u'first'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the sensitivity for a specific 1 dimensional model/feature. The Sensitivity for each parameter is plotted in sepearate figures.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

sensitivity_1d_combined(*feature=None*, *sensitivity=u'first'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the sensitivity for a specific 1 dimensional model/feature. The Sensitivity for each parameter is plotted in the same plot.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

sensitivity_1d_grid(*feature=None*, *sensitivity=u'first'*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the sensitivity for a specific 1 dimensional model/feature. The Sensitivity for each parameter is plotted in the same figure, but separate plots.

Parameters

- **feature** (*{None, str}, optional*) – The name of the model/feature. If None, the name of the model is used. Default is None.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – Which Sobol indices to plot. "sobol_first" and "first" is the first order Sobol indices, while "sobol_total" and "total" are the total order Sobol indices. Default is "first".
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.
- `ValueError` – If sensitivity is not one of "sobol_first", "first", "sobol_total", or "total".

variance_1d(*feature*, *hardcopy=True*, *show=False*, ***plot_kwargs*)

Plot the variance for a specific 1 dimensional model/feature.

Parameters

- **feature** (*str*) – The name of the model/feature.

- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 1 dimensional.

`variance_2d`(*feature*, *hardcopy=True*, *show=False*, ***plot_kwargs*)
Plot the variance for a specific 2 dimensional model/feature.

Parameters

- **feature** (*str*) – The name of the model/feature.
- **hardcopy** (*bool, optional*) – If the plot should be saved to file. Default is True.
- **show** (*bool, optional*) – If the plot should be shown on screen. Default is False.
- ****plot_kwargs, optional** – Matplotlib plotting arguments.

Raises

- `ValueError` – If a Datafile is not loaded.
- `ValueError` – If the model/feature is not 2 dimensional.

CHAPTER 12

Logging

Uncertainpy uses the logging module to log to both file and to screen. All loggers are named `class_instance.__module__ + "." + class_instance.__class__.__name__`. An example, the logger in a `Data```object is named ``uncertainpy.data.Data`. If the the module name does not start with "uncertainpy.", "uncertainpy." as added as a prefix.

A file handler is only added to the logging by `UncertaintyQuantification`. If level is set to None, no logging in Uncertainpy is set up and the logging can be customized as necessary by using the logging module. This should only be done if you know what you are doing. Be warned that logging is performed in parallel. If the `MultiprocessLoggingHandler()` is not used when trying to write to a single log file, Uncertainpy will hang. This happens because several processes try to log to the same file.

Logging can easily be added to custom models and features by:

```
# Import the functions and libraries needed
from uncertainpy.utils import create_logger
import logging

# Set up a logger. This adds a screen handlers to the "uncertainpy" logger
# if it does not already exist
# All log messages with level "info" or higher will be logged.
setup_logger("uncertainpy.logger_name", level="info")

# Get the logger recently created
logger = logging.getLogger("uncertainpy.logger_name")

# Log a message with the level "info".
logger.info("info logging message here")
```

Note that if you want to use the logger setup in Uncertainpy, the name of your loggers should start with `uncertainpy.`.

12.1 API Reference

class `uncertainpy.utils.logger.`**`MultiprocessLoggingHandler`**(*filename*, *mode*)
Adapted from: https://stackoverflow.com/questions/641420/how-should-i-log-while-using-multiprocessing-in-python

`close`()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

`emit`(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

`setFormatter`(*fmt*)
Set the formatter for this handler.

class `uncertainpy.utils.logger.`**`MyFormatter`**(*fmt=u'%(levelno)s: %(msg)s'*)
The logging formater.

`format`(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class `uncertainpy.utils.logger.`**`TqdmLoggingHandler`**(*stream=None*)
Set logging so logging to stream works with Tqdm, logging now uses tqdm.write.

`emit`(*record*)
Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

`uncertainpy.utils.logger.`**`add_file_handler`**(*name=u'uncertainpy'*, *filename=u'uncertainpy.log'*)
Add file handler to logger with *name*, if no file handler already exists for the given logger.

Parameters

- **name** (*str, optional*) – Name of the logger. Default name is "uncertainpy".
- **filename** (*str*) – Name of the logfile. If None, no logging to file is performed. Default is "uncertainpy.log".

`uncertainpy.utils.logger.`**`add_screen_handler`**(*name=u'uncertainpy'*)
Adds a logging to console (a console handler) to logger with *name*, if no screen handler already exists for the given logger.

Parameters **name** (*str, optional*) – Name of the logger. Default name is "uncertainpy".

`uncertainpy.utils.logger.`**`get_logger`**`(`*class_instance*`)`

Get a logger with name given from *class_instance*: `class_instance.__module__ + "." + class_instance.__class__.__name__`.

Parameters **class_instance** (*instance*) – Class instance used to get the logger name.

Returns **logger** – The logger object.

Return type Logger object

`uncertainpy.utils.logger.`**`has_handlers`**`(`*logger*`)`

See if this logger has any handlers configured.

Loop through all handlers for this logger and its parents in the logger hierarchy. Return True if a handler was found, else False. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger which is checked for the existence of handlers.

Returns True if the logger or any parent logger has handlers attached.

Return type bool

`uncertainpy.utils.logger.`**`setup_logger`**`(`*name*, *level=u'info'*`)`

Create a logger with *name*.

Parameters

- **name** (*str*) – Name of the logger
- **level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logger is set up. Default logger level is info.

`uncertainpy.utils.logger.`**`setup_module_logger`**`(`*class_instance*, *level=u'info'*`)`

Create a logger with a name from the current class. "uncertainpy." is added to the beginning of the name if the module name does not start with "uncertainpy.". If no handlers, adds handlers to the logger named uncertainpy.

Parameters

- **class_instance** (*instance*) – Class instance used to set the logger name. `class_instance.__module__ + "." + class_instance.__class__.__name__`.
- **level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logger level is set. Setting logger level overwrites the logger level set from configuration file. Default logger level is "info".

CHAPTER 13

Utilities

Various utility functions.

13.1 API Reference

`uncertainpy.utils.utility.`**`contains_nan`**`(`*values*`)`

Checks if `None` or `numpy.nan` exists in *values*. Returns `True` if any there are at least one occurrence of `None` or `numpy.nan`.

Parameters **values** (*array_like, list, number*) – *values* where to check for occurrences of `None` or `np.nan`. Can be irregular and have any number of nested elements.

Returns `True` if *values* has at least one occurrence of `None` or `numpy.nan`.

Return type bool

`uncertainpy.utils.utility.`**`is_regular`**`(`*values*`)`

Test if *values* is regular or not, meaning it has a varying length of nested elements.

Parameters **values** (*array_like, list, number*) – *values* to check if it is regular or not, meaning it has a varying length of nested elements.

Returns True if the feature is regular or False if the feature is irregular.

Return type bool

Notes

Does not ignore `numpy.nan`, so `[numpy.nan, [1, 2]]` returns False.

`uncertainpy.utils.utility.`**`lengths`**`(`*values*`)`

Get the lengths of a list and all its sublists.

Parameters **values** (*list*) – List where we want to find the lengths of the list and all sublists.

Returns A list with the lengths of the list and all sublists.

Return type list

`uncertainpy.utils.utility.`**`none_to_nan`**(*values*)

Converts `None` values in *values* to `np.nan`.

Parameters **values** (*array_like, list, number*) – Values where to convert occurrences of `None` converted to `np.nan`. Can be irregular and have any number of nested elements.

Returns **values** – *values* with all occurrences of `None` converted to `np.nan`.

Return type array_like, list, number

`uncertainpy.utils.utility.`**`set_nan`**(*values*, *index*)

Set the index of a arbitrarly nested list to nan

Parameters

- **values** (*array_like, list, number*) – Values where to set index to `numpy.nan`. Can be irregular and have any number of nested elements.
- **index** (*array_like, list, number*) – Index where to set *values* to `numpy.nan`.

CHAPTER 14

Core

This module contains the classes that are responsible for running the model and calculate features of the model, both in parallel (*RunModel* and *Parallel*), as well as the class for performing the uncertainty calculations (*UncertaintyCalculations*). It also contains the base classes that are responsible for setting and updating parameters, models and features across classes (*Base and ParameterBase*).

14.1 UncertaintyCalculations

`UncertaintyCalculations` is the class responsible for performing the uncertainty calculations. Here we explain how they are performed as well as well as which options the user have to customize the calculations An insight into how the calculations are performed is not required to use Uncertainpy. In most cases, the default settings works fine. In addition to the customization options shown below, Uncertainpy has support for implementing entirely custom uncertainty quantification and sensitivity analysis methods. This is only recommended for expert users, as knowledge of both Uncertainpy and uncertainty quantification is needed.

14.1.1 Quasi-Monte Carlo method

To use the quasi-Monte Carlo method, we call `quantify()` with `method="mc"`, and the optional argument `nr_mc_samples`:

```
data = UQ.quantify(
    method="mc",
    nr_mc_samples=10**4,
)
```

By default, the quasi-Monte Carlo method quasi-randomly draws *10000* parameter samples from the joint multivariate probability distribution of the parameters ρ_Q using Hammersley sampling (Hammersley, 1960). As the name indicates, the number of samples is specified by the `nr_mc_samples` argument. The model is evaluated for each of these parameter samples, and features are calculated for each model evaluation (when applicable). To speed up the calculations, Uncertainpy uses the multiprocess Python package (McKerns et al., 2012) to perform this step in parallel.

When model and feature calculations are done, Uncertainpy calculates the mean, variance, and 5th and 95th percentile (which gives the *90%* prediction interval) for the model output as well as for each feature.

14.1.2 Polynomial chaos expansions

To use polynomial chaos expansions we use `quantify()` with the argument `method="pc"`, which takes a set of optional arguments (default are values specified):

```
data = UQ.quantify(
    method="pc",
    pc_method="collocation",
    rosenblatt=False,
    polynomial_order=4,
    nr_collocation_nodes=None,
    quadrature_order=None,
    nr_pc_mc_samples=10**4,
)
```

As previously mentioned, Uncertainpy allows the user to select between point collocation (`pc_method="collocation"`) and pseudo-spectral projections (`pc_method="spectral"`). The goal is to create separate polynomial chaos expansions *hat{U}* for the model and each feature. In both methods, Uncertainpy creates the orthogonal polynomial $\boldsymbol{\phi}_n$ using $\rho_{\boldsymbol{Q}}$ and the three-term recurrence relation if available, otherwise the discretized Stieltjes method (Stieltjes, 1884) is used. Uncertainpy uses a third order polynomial expansion, changed with `polynomial_order`. The polynomial $\boldsymbol{\phi}_n$ is shared between the model and all features, since they have the same uncertain input parameters, and therefore the same $\rho_{\boldsymbol{Q}}$. Only the polynomial coefficients c_n differ between the model and each feature.

The two polynomial chaos methods differ in terms of how they calculate c_n. For point collocation Uncertainpy uses $2(N_p + 1)$ collocation nodes, as recommended by (Hosder et al., 2007), where *N_p* is the number of polynomial chaos expansion factors. The number of collocation nodes can be customized with `nr_collocation_nodes`, but the new number of nodes must be chosen carefully. The collocation nodes are sampled from $\rho_{\boldsymbol{Q}}$ using Hammersley sampling (Hammersley, 1960). The model and features are calculated for each of the collocation nodes. As with the quasi-Monte Carlo method, this step is performed in parallel. The polynomial coefficients c_n are calculated using Tikhonov regularization (Rifkin and Lipert, 2007) from the model and feature results.

For the pseudo-spectral projection, Uncertainpy chooses nodes and weights using a quadrature scheme, instead of choosing nodes from $\rho_{\boldsymbol{Q}}$. The quadrature scheme used is Leja quadrature with a Smolyak sparse grid (Narayan and Jakeman, 2014; Smolyak, 1963). The Leja quadrature is of order two greater than the polynomial order, but can be changed with `quadrature_order`. The model and features are calculated for each of the quadrature nodes. As before, this step is performed in parallel. The polynomial coefficients c_n are then calculated from the quadrature nodes, weights, and model and feature results.

When Uncertainpy has derived $\hat{U}$ for the model and features, it uses $\hat{U}$ to compute the mean, variance, and the first and total order Sobol indices. The first and total order Sobol indices are also summed and normalized. Finally, Uncertainpy uses $\hat{U}$ as a surrogate model, and performs a quasi-Monte Carlo method with Hammersley sampling and `nr_pc_mc_samples=10**4` samples to find the 5th and 95th percentiles.

If the model parameters have a dependent joint multivariate distribution, the Rosenblatt transformation must be used by setting `rosenblatt=True`. To perform the transformation Uncertainpy chooses $\rho_{\boldsymbol{R}}$ to be a multivariate independent normal distribution, which is used instead of $\rho_{\boldsymbol{Q}}$ to perform the polynomial chaos expansions. Both the point collocation method and the pseudo-spectral method are performed as described above. The only difference is that we use $\rho_{\boldsymbol{R}}$ instead of $\rho_{\boldsymbol{Q}}$, and use the Rosenblatt transformation to transform the selected nodes from $\boldsymbol{R}$ to $\boldsymbol{Q}$, before they are used in the model evaluation.

14.1.3 API Reference

class uncertainpy.core.**UncertaintyCalculations**(*model=None, parameters=None, features=None, create_PCE_custom=None, custom_uncertainty_quantification=None, CPUs=u'max', logger_level=u'info'*)

Perform the calculations for the uncertainty quantification and sensitivity analysis.

This class performs the calculations for the uncertainty quantification and sensitivity analysis of the model and features. It implements both quasi-Monte Carlo methods and polynomial chaos expansions using either point collocation or pseudo-spectral method. Both of the polynomial chaos expansion methods have support for the rosenblatt transformation to handle dependent variables.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}, optional*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **parameters** (*{dict {name: parameter_object}, dict of {name: value or Chaospy distribution}, …], list of Parameter instances, list [[name, value or Chaospy distribution], …], list [[name, value, Chaospy distribution or callable that returns a Chaospy distribution],…],}*) – List or dictionary of the parameters that should be created. On the form `parameters =`
  - `{name_1: parameter_object_1, name: parameter_object_2, ...}`
  - `{name_1: value_1 or Chaospy distribution, name_2: value_2 or Chaospy distribution, ...}`
  - `[parameter_object_1, parameter_object_2, ...],`
  - `[[name_1, value_1 or Chaospy distribution], ...].`
  - `[[name_1, value_1, Chaospy distribution or callable that returns a Chaospy distribution], ...]`
- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated. Default is None.
- **create_PCE_custom** (*callable, optional*) – A custom method for calculating the polynomial chaos approximation. For the requirements of the function see `UncertaintyCalculations.create_PCE_custom`. Overwrites existing `create_PCE_custom` method. Default is None.
- **custom_uncertainty_quantification** (*callable, optional*) – A custom method for calculating uncertainties. For the requirements of the function see `UncertaintyCalculations.custom_uncertainty_quantification`. Overwrites existing `custom_uncertainty_quantification` method. Default is None.
- **CPUs** (*{int, None, "max"}, optional*) – The number of CPUs to use when calculating the model and features. If None, no multiprocessing is used. If "max", the maximum number of CPUs on the computer (multiprocess.cpu_count()) is used. Default is "max".
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed. Default logger level is "info".

Variables

- **model** (*Model or Model subclass*) – The model to perform uncertainty quantification on.
- **parameters** (*Parameters*) – The uncertain parameters.
- **features** (*Features or Features subclass*) – The features of the model to perform uncertainty quantification on.
- **runmodel** (*RunModel*) – Runmodel object responsible for evaluating the model and calculating features.

See also:

uncertainpy.features.Features, *uncertainpy.Parameter*, *uncertainpy.Parameters*, *uncertainpy.models.Model*, *uncertainpy.core.RunModel*

uncertainpy.models.Model.run Requirements for the model run function.

analyse_PCE (*U_hat*, *distribution*, *data*, *nr_samples=10000*)

Calculate the statistical metrics from the polynomial chaos approximation.

Parameters

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.
- **nr_samples** (*int, optional*) – Number of samples for the Monte Carlo sampling of the polynomial chaos approximation. Default is 10**4.

Returns **data** – The *data* parameter given as input with the statistical metrics added.

Return type Data

Notes

The *data* parameter should contain (but not necessarily) the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`
7. `data.errored`

When returned *data* additionally contains:

8. `data["model/features"].mean`
9. `data["model/features"].variance`
10. `data["model/features"].percentile_5`
11. `data["model/features"].percentile_95`
12. `data["model/features"].sobol_first`, if more than 1 parameter

13. `data["model/features"].sobol_total`, if more than 1 parameter
14. `data["model/features"].sobol_first_average`, if more than 1 parameter
15. `data["model/features"].sobol_total_average`, if more than 1 parameter

See also:

`uncertainpy.Data()`

`average_sensitivity`(*data*, *sensitivity=u'sobol_first'*)

Calculate the average of the sensitivities for the model and all features and add them to *data*. Ignores any occurrences of numpy.NaN.

Parameters

- **data** (*Data*) – A data object with all model and feature evaluations, as well as all calculated statistical metrics.
- **sensitivity** (*{"sobol_first", "first", "sobol_total", "total"}, optional*) – The sensitivity to normalize and sum. "sobol_first" and "1" are for the first order Sobol indice while "sobol_total" and "t" is for the total order Sobol indices. Default is "sobol_first".

Returns **data** – The *data* object with the average of the sensitivities for the model and all features added.

Return type Data

See also:

`uncertainpy.Data()`

`convert_uncertain_parameters`(*uncertain_parameters=None*)

Converts uncertain_parameter(s) to a list of uncertain parameter(s), and checks if it is a legal set of uncertain parameter(s).

Parameters **uncertain_parameters** (*{None, str, list}, optional*) – The name(s) of the uncertain parameters to use. If None, a list of all uncertain parameters are returned. Default is None.

Returns **uncertain_parameters** – A list with the name of all uncertain parameters.

Return type list

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

See also:

`uncertainpy.Parameters()`

`create_PCE_collocation`(*uncertain_parameters=None*, *polynomial_order=4*, *nr_collocation_nodes=None*, *allow_incomplete=True*)

Create the polynomial approximation *U_hat* using pseudo-spectral projection.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.

- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.

Returns

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

The returned *data* should contain (but not necessarily) the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`
7. `data.errored`

The model and feature do not necessarily give results for each node. The collocation method is robust towards missing values as long as the number of results that remain is high enough.

The polynomial chaos expansion method for uncertainty quantification approximates the model with a polynomial that follows specific requirements. This polynomial can be used to quickly calculate the uncertainty and sensitivity of the model.

To create the polynomial chaos expansion we first find the polynomials using the three-therm recurrence relation if available, otherwise the discretized Stieltjes method is used. Then we use point collocation to find the expansion coefficients for the model and each feature of the model.

In point collocation we require the polynomial approximation to be equal the model at a set of collocation nodes. This results in a set of linear equations for the polynomial coefficients we can solve. We choose *nr_collocation_nodes* collocation nodes with Hammersley sampling from the *distribution*. We evaluate the model and each feature in parallel, and solve the resulting set of linear equations with Tikhonov regularization.

See also:

`uncertainpy.Data()`, *`uncertainpy.Parameters()`*

`create_PCE_collocation_rosenblatt`(*uncertain_parameters=None, polynomial_order=4, nr_collocation_nodes=None, allow_incomplete=True*)

Create the polynomial approximation *U_hat* using pseudo-spectral projection and the Rosenblatt transformation. Works for dependend uncertain parameters.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.

Returns

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

The returned *data* should contain (but not necessarily) the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`

The model and feature do not necessarily give results for each node. The collocation method is robust towards missing values as long as the number of results that remain is high enough.

The polynomial chaos expansion method for uncertainty quantification approximates the model with a polynomial that follows specific requirements. This polynomial can be used to quickly calculate the uncertainty and sensitivity of the model.

We use the Rosenblatt transformation to transform from dependent to independent variables before we create the polynomial chaos expansion. We first find the polynomials from the independent distributions using the three-therm recurrence relation if available, otherwise the discretized Stieltjes method is used. Then we use the point collocation with the Rosenblatt transformation to find the expansion coefficients for the model and each feature of the model.

In point collocation we require the polynomial approximation to be equal the model at a set of collocation nodes. This results in a set of linear equations for the polynomial coefficients we can solve. We choose *nr_collocation_nodes* collocation nodes with Hammersley sampling from the independent distribution. We then transform the nodes using the Rosenblatte transformation and evaluate the model and each feature in parallel. We solve the resulting set of linear equations with Tikhonov regularization.

See also:

uncertainpy.Data(), *uncertainpy.Parameters()*

create_PCE_custom

A custom method for calculating the polynomial chaos approximation. Must follow the below requirements.

Parameters

- **self** (*UncertaintyCalculation*) – An explicit self is required as the first argument. self can be used inside the custom function.
- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- ****kwargs** – Any number of optional arguments.

Returns

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

This method can be implemented to create a custom method to calculate the polynomial chaos expansion. The method must calculate and return the return arguments described above.

The returned *data* should contain (but not necessarily) the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`

The method *analyse_PCE* is called after the polynomial approximation has been created.

Usefull methods in Uncertainpy are:

1. uncertainpy.core.Uncertaintycalculations.convert_uncertain_parameters
2. uncertainpy.core.Uncertaintycalculations.create_distribution
3. uncertainpy.core.RunModel.run

See also:

uncertainpy.Data, *uncertainpy.Parameters*

uncertainpy.core.Uncertaintycalculations.convert_uncertain_parameters

Converts uncertain parameters to allowed list

uncertainpy.core.Uncertaintycalculations.create_distribution Creates the uncertain parameter distribution

uncertainpy.core.RunModel.run Runs the model

create_PCE_spectral(*uncertain_parameters=None*, *polynomial_order=4*, *quadrature_order=None*, *allow_incomplete=True*)

Create the polynomial approximation *U_hat* using pseudo-spectral projection.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.

Returns

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

The returned *data* should contain (but not necessarily) the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`
7. `data.errored`

The model and feature do not necessarily give results for each node. The pseudo-spectral methods is sensitive to missing values, so *allow_incomplete* should be used with care.

The polynomial chaos expansion method for uncertainty quantification approximates the model with a polynomial that follows specific requirements. This polynomial can be used to quickly calculate the uncertainty and sensitivity of the model.

To create the polynomial chaos expansion we first find the polynomials using the three-therm recurrence relation if available, otherwise the discretized Stieltjes method is used. Then we use the pseudo-spectral projection to find the expansion coefficients for the model and each feature of the model.

Pseudo-spectral projection is based on least squares minimization and finds the expansion coefficients through numerical integration. The integration uses a quadrature scheme with weights and nodes. We use Leja quadrature with Smolyak sparse grids to reduce the number of nodes required. For each of the nodes we evaluate the model and calculate the features, and the polynomial approximation is created from these results.

See also:

uncertainpy.Data(), *uncertainpy.Parameters()*

create_PCE_spectral_rosenblatt(*uncertain_parameters=None*, *polynomial_order=4*, *quadrature_order=None*, *allow_incomplete=True*)

Create the polynomial approximation *U_hat* using pseudo-spectral projection and the Rosenblatt transformation. Works for dependend uncertain parameters.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.

Returns

- **U_hat** (*dict*) – A dictionary containing the polynomial approximations for the model and each feature as chaospy.Poly objects.
- **distribution** (*chaospy.Dist*) – The multivariate distribution for the uncertain parameters.
- **data** (*Data*) – A data object containing the values from the model evaluation and feature calculations.

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

data **should contain (but not necessarily) the following, if** applicable:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`
7. `data.errored`

The model and feature do not necessarily give results for each node. The pseudo-spectral methods is sensitive to missing values, so *allow_incomplete* should be used with care.

The polynomial chaos expansion method for uncertainty quantification approximates the model with a polynomial that follows specific requirements. This polynomial can be used to quickly calculate the uncertainty and sensitivity of the model.

We use the Rosenblatt transformation to transform from dependent to independent variables before we create the polynomial chaos expansion. We first find the polynomials from the independent distributions using the three-therm recurrence relation if available, otherwise the discretized Stieltjes method is used. Then we use the pseudo-spectral projection with the Rosenblatt transformation to find the expansion coefficients for the model and each feature of the model.

Pseudo-spectral projection is based on least squares minimization and finds the expansion coefficients through numerical integration. The integration uses a quadrature scheme with weights and nodes. We use Leja quadrature with Smolyak sparse grids to reduce the number of nodes required. We use the Rosenblatt transformation to transform the quadrature nodes before they are sent to the model evaluation. For each of the nodes we evaluate the model and calculate the features, and the polynomial approximation is created from these results.

See also:

`uncertainpy.Data()`, `uncertainpy.Parameters()`

`create_distribution`(*uncertain_parameters=None*)

Create a joint multivariate distribution for the selected parameters from univariate distributions.

Parameters **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the joint multivariate distribution. If None, the joint multivariate distribution for all uncertain parameters is created. Default is None.

Returns **distribution** – The joint multivariate distribution for the given parameters.

Return type chaospy.Dist

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

If a multivariate distribution is defined in the Parameters.distribution, that multivariate distribution is returned. Otherwise the joint multivariate distribution for the selected parameters is created from the univariate distributions.

See also:

`uncertainpy.Parameters()`

`create_mask`(*evaluations*)

Mask evaluations that do not give results (anything but np.nan or None).

Parameters **evaluations** (*array_like*) – Evaluations for the model.

Returns

- **masked_evaluations** (*list*) – The evaluations that have results (not numpy.nan or None).
- **mask** (*boolean array*) – The mask itself, used to create the masked arrays.

`create_masked_evaluations`(*data, feature*)

Mask all model and feature evaluations that do not give results (anything but np.nan) and the corresponding nodes.

Parameters

- **data** (*Data*) – A Data object with evaluations for the model and each feature. Must contain *data[feature].evaluations*.
- **feature** (*str*) – Name of the feature or model to mask.

Returns

- **masked_evaluations** (*list*) – The evaluations that have results (not numpy.nan or None).
- **mask** (*boolean array*) – The mask itself, used to create the masked arrays.

`create_masked_nodes` (*data*, *feature*, *nodes*)

Mask all model and feature evaluations that do not give results (anything but np.nan) and the corresponding nodes.

Parameters

- **data** (*Data*) – A Data object with evaluations for the model and each feature. Must contain *data[feature].evaluations*.
- **feature** (*str*) – Name of the feature or model to mask.
- **nodes** (*array_like*) – The nodes used to evaluate the model.

Returns

- **masked_evaluations** (*array_like*) – The evaluations which have results.
- **mask** (*boolean array*) – The mask itself, used to create the masked arrays.
- **masked_nodes** (*array_like*) – The nodes that correspond to the evaluations with results.

`create_masked_nodes_weights` (*data*, *feature*, *nodes*, *weights*)

Mask all model and feature evaluations that do not give results (anything but numpy.nan) and the corresponding nodes.

Parameters

- **data** (*Data*) – A Data object with evaluations for the model and each feature. Must contain *data[feature].evaluations*.
- **nodes** (*array_like*) – The nodes used to evaluate the model.
- **feature** (*str*) – Name of the feature or model to mask.
- **weights** (*array_like*) – Weights corresponding to each node.

Returns

- **masked_evaluations** (*array_like*) – The evaluations which have results.
- **mask** (*boolean array*) – The mask itself, used to create the masked arrays.
- **masked_nodes** (*array_like*) – The nodes that correspond to the evaluations with results.
- **masked_weights** (*array_like*) – Masked weights that correspond to evaluations with results.

`custom_uncertainty_quantification`

A custom uncertainty quantification method. Must follow the below requirements.

Parameters

- **self** (*UncertaintyCalculation*) – An explicit self is required as the first argument. self can be used inside the custom function.

- ****kwargs** – Any number of optional arguments.

Returns **data** – A Data object with calculated uncertainties.

Return type Data

Notes

Usefull methods in Uncertainpy are:

1. uncertainpy.core.Uncertaintycalculations.convert_uncertain_parameters - Converts uncertain parameters to an allowed list.
2. uncertainpy.core.Uncertaintycalculations.create_distribution - Creates the uncertain parameter distribution
3. uncertainpy.core.RunModel.run - Runs the model and all features.

See also:

`uncertainpy.Data`

`uncertainpy.core.Uncertaintycalculations.convert_uncertain_parameters` Converts uncertain parameters to list

`uncertainpy.core.Uncertaintycalculations.create_distribution` Create uncertain parameter distribution

`uncertainpy.core.RunModel.run` Runs the model

`dependent`(*distribution*)
Check if a distribution is dependent or not.

Parameters **distribution** (*chaospy.Dist*) – A Chaospy probability distribution.

Returns **dependent** – True if the distribution is dependent, False if is independent.

Return type bool

`features`
Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

`uncertainpy.features.Features`, `uncertainpy.features.GeneralSpikingFeatures`, `uncertainpy.features.SpikingFeatures`, `uncertainpy.features.GeneralNetworkFeatures`, `uncertainpy.features.NetworkFeatures`

`mc_calculate_sobol`(*evaluations*, *nr_uncertain_parameters*, *nr_samples*)
Calculate the Sobol indices.

Parameters

- **evaluations** (*array_like*) – The model evaluations, evaluated for the samples created by SALIB.sample.saltelli.

- **nr_uncertain_parameters** (*int*) – Number of uncertain parameters.
- **nr_samples** (*int*) – Number of samples used in the Monte Carlo sampling.

Returns

- **sobol_first** (*list*) – The first order Sobol indices for each uncertain parameter.
- **sobol_total** (*list*) – The total order Sobol indices for each uncertain parameter.

`model`

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

`uncertainpy.models.Model`, `uncertainpy.models.Model.run`, `uncertainpy.models.NestModel`, `uncertainpy.models.NeuronModel`

`monte_carlo`(*uncertain_parameters=None*, *nr_samples=10000*, *seed=None*, *allow_incomplete=True*)

Perform an uncertainty quantification using the quasi-Monte Carlo method.

Parameters

- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **nr_samples** (*int, optional*) – Number of samples for the quasi-Monte Carlo sampling. Default is 10**4.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.
- **allow_incomplete** (*bool, optional*) – If the uncertainty quantification should be performed for features or models with incomplete evaluations. Default is True.

Returns **data** – A data object with all model and feature evaluations, as well as all calculated statistical metrics.

Return type Data

Raises `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.

Notes

The returned *data* should contain the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`

7. `data.errored`
8. `data["model/features"].mean`
9. `data["model/features"].variance`
10. `data["model/features"].percentile_5`
11. `data["model/features"].percentile_95`
12. `data["model/features"].sobol_first`, if more than 1 parameter
13. `data["model/features"].sobol_total`, if more than 1 parameter
14. `data["model/features"].sobol_first_average`, if more than 1 parameter
15. `data["model/features"].sobol_total_average`, if more than 1 parameter

In the quasi-Monte Carlo method we quasi-randomly draw `(nr_samples/2)*(nr_uncertain_parameters + 2)` (nr_samples=10**4 by default) parameter samples using Saltelli's sampling scheme ([1]). We require this number of samples to be able to calculate the Sobol indices. We evaluate the model for each of these parameter samples and calculate the features from each of the model results. This step is performed in parallel to speed up the calculations. Then we use nr_samples‘ of the model and feature results to calculate the mean, variance, and 5th and 95th percentile for the model and each feature. Lastly, we use all calculated model and each feature results to calculate the Sobol indices using Saltellie's approach.

References

See also:

`uncertainpy.Data()`, *`uncertainpy.Parameters()`*

`parameters`

Model parameters.

Parameters new_parameters (*{None, Parameters instance, list of Parameter instances, list [[name, value, distribution], . . .]}*) – Either None, a Parameters instance or a list of the parameters that should be created. The two lists are similar to the arguments sent to Parameters. Default is None.

Returns parameters – Parameters of the model. If None, no parameters have been set.

Return type {None, Parameters}

See also:

`uncertainpy.Parameter`, *`uncertainpy.Parameters`*

`polynomial_chaos`(*method=u'collocation'*, *rosenblatt=u'auto'*, *uncertain_parameters=None*, *polynomial_order=4*, *nr_collocation_nodes=None*, *quadrature_order=None*, *nr_pc_mc_samples=10000*, *allow_incomplete=True*, *seed=None*, ***custom_kwargs*)

Perform an uncertainty quantification and sensitivity analysis using polynomial chaos expansions.

Parameters

- **method** (*{"collocation", "spectral", "custom"}, optional*) – The method to use when creating the polynomial chaos approximation. "collocation" is the point collocation method "spectral" is pseudo-spectral projection, and "custom" is the custom polynomial method. Default is "collocation".

[1] Saltelli, A., P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola (2010). "Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index." Computer Physics Communications, 181(2):259-270, doi:10.1016/j.cpc.2009.09.018.

- **rosenblatt** (*{"auto", bool}, optional*) – If the Rosenblatt transformation should be used. The Rosenblatt transformation must be used if the uncertain parameters have dependent variables. If "auto" the Rosenblatt transformation is used if there are dependent parameters, and it is not used of the parameters have independent distributions. Default is "auto".
- **uncertain_parameters** (*{None, str, list}, optional*) – The uncertain parameter(s) to use when creating the polynomial approximation. If None, all uncertain parameters are used. Default is None.
- **polynomial_order** (*int, optional*) – The polynomial order of the polynomial approximation. Default is 4.
- **nr_collocation_nodes** (*{int, None}, optional*) – The number of collocation nodes to choose, if point collocation is used. If None, *nr_collocation_nodes* = 2* number of expansion factors + 2. Default is None.
- **quadrature_order** (*{int, None}, optional*) – The order of the Leja quadrature method, if pseudo-spectral projection is used. If None, `quadrature_order = polynomial_order + 2`. Default is None.
- **nr_pc_mc_samples** (*int, optional*) – Number of samples for the Monte Carlo sampling of the polynomial chaos approximation.
- **allow_incomplete** (*bool, optional*) – If the polynomial approximation should be performed for features or models with incomplete evaluations. Default is True.
- **seed** (*int, optional*) – Set a random seed. If None, no seed is set. Default is None.

Returns **data** – A data object with all model and feature values, as well as all calculated statistical metrics.

Return type Data

Raises

- `ValueError` – If a common multivariate distribution is given in Parameters.distribution and not all uncertain parameters are used.
- `ValueError` – If *method* not one of "collocation", "spectral" or "custom".
- `NotImplementedError` – If "custom" is chosen and have not been implemented.

Notes

The returned *data* should contain the following:

1. `data["model/features"].evaluations`
2. `data["model/features"].time`
3. `data["model/features"].labels`
4. `data.model_name`
5. `data.incomplete`
6. `data.method`
7. `data.errored`
8. `data["model/features"].mean`
9. `data["model/features"].variance`
10. `data["model/features"].percentile_5`

11. `data["model/features"].percentile_95`
12. `data["model/features"].sobol_first`, if more than 1 parameter
13. `data["model/features"].sobol_total`, if more than 1 parameter
14. `data["model/features"].sobol_first_average`, if more than 1 parameter
15. `data["model/features"].sobol_total_average`, if more than 1 parameter

The model and feature do not necessarily give results for each node. The collocation method is robust towards missing values as long as the number of results that remain is high enough. The pseudo-spectral method on the other hand, is sensitive to missing values, so *allow_incomplete* should be used with care in that case.

The polynomial chaos expansion method for uncertainty quantification approximates the model with a polynomial that follows specific requirements. This polynomial can be used to quickly calculate the uncertainty and sensitivity of the model.

To create the polynomial chaos expansion we first find the polynomials using the three-therm recurrence relation if available, otherwise the discretized Stieltjes method is used. Then we use point collocation or pseudo-spectral projection to find the expansion coefficients for the model and each feature of the model.

In point collocation we require the polynomial approximation to be equal the model at a set of collocation nodes. This results in a set of linear equations for the polynomial coefficients we can solve. We choose *nr_collocation_nodes* collocation nodes with Hammersley sampling from the *distribution*. We evaluate the model and each feature in parallel, and solve the resulting set of linear equations with Tikhonov regularization.

Pseudo-spectral projection is based on least squares minimization and finds the expansion coefficients through numerical integration. The integration uses a quadrature scheme with weights and nodes. We use Leja quadrature with Smolyak sparse grids to reduce the number of nodes required. For each of the nodes we evaluate the model and calculate the features, and the polynomial approximation is created from these results.

If we have dependent uncertain parameters we must use the Rosenblatt transformation. We use the Rosenblatt transformation to transform from dependent to independent variables before we create the polynomial chaos expansion. We first find the polynomials from the independent distributions using the three-term recurrence relation if available, otherwise the discretized Stieltjes method is used

Both pseudo-spectral projection and point collocation is performed using the independent distribution, the only difference is that we use the Rosenblatt transformation to transform the nodes from the independent distribution to the dependent distribution.

See also:

`uncertainpy.Data()`, *`uncertainpy.Parameters()`*

`separate_output_values`(*evaluations*, *nr_uncertain_parameters*, *nr_samples*)

Notes

Separate the output from the model evaluations, evaluated for the samples created by SALIB.sample.saltelli.

Parameters

- **evaluations** (*array_like*) – The model evaluations, evaluated for the samples created by SALIB.sample.saltelli.
- **nr_uncertain_parameters** (*int*) – Number of uncertain parameters.

- **nr_samples** (*int*) – Number of samples used in the Monte Carlo sampling.

Returns

- **A** (*array_like*) – The A sample matrix from saltellie et. al. 2010.
- **B** (*array_like*) – The B sample matrix from saltellie et. al. 2010.
- **AB** (*array_like*) – The AB sample matrix from saltellie et. al. 2010.

Notes

Adapted from SALib/analyze/sobol.py:

https://github.com/SALib/SALib/blob/master/SALib/analyze/sobol.py

14.2 Base and ParameterBase

These classes enable setting and updating the model, features and parameters (not in all classes) across classes from the top of the hierarchy (*UncertaintyQuantification*) and down (*Parallel*). To add updating of the current class, as well as the classes further down the setters can be overridden. One example of this from *RunModel*):

```
@ParameterBase.model.setter
def model(self, new_model):
    ParameterBase.model.fset(self, new_model)

    self._parallel.model = self.model
```

14.2.1 API Reference

Base

class uncertainpy.core.**Base**(*model=None*, *features=None*, *logger_level=u'info'*)

Set and update features and model.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}, optional*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all listed features will be calculated. Default is None.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed. Default logger level is "info".

Variables

- **model** (*uncertainpy.Model or subclass of uncertainpy.Model*) – The model to perform uncertainty quantification on.
- **features** (*uncertainpy.Features or subclass of uncertainpy.Features*) – The features of the model to perform uncertainty quantification on.

See also:

uncertainpy.features.Features, *uncertainpy.models.Model*

uncertainpy.models.Model.run Requirements for the model run function.

features

Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

uncertainpy.features.Features, *uncertainpy.features.GeneralSpikingFeatures*, *uncertainpy.features.SpikingFeatures*, *uncertainpy.features.GeneralNetworkFeatures*, *uncertainpy.features.NetworkFeatures*

model

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

uncertainpy.models.Model, *uncertainpy.models.Model.run*, *uncertainpy.models.NestModel*, *uncertainpy.models.NeuronModel*

ParameterBase

class uncertainpy.core.**ParameterBase**(*model=None*, *parameters=None*, *features=None*, *logger_level=u'info'*)

Set and update features, model and parameters.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}, optional*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **parameters** (*{dict {name: parameter_object}, dict of {name: value or Chaospy distribution}, . . .], list of Parameter instances, list [[name, value or Chaospy distribution], . . .], list [[name, value, Chaospy distribution or callable that returns a Chaospy distribution],. . .],}*) – List or dictionary of the parameters that should be created. On the form `parameters =`
  - `{name_1: parameter_object_1, name: parameter_object_2, ...}`
  - `{name_1: value_1 or Chaospy distribution, name_2: value_2 or Chaospy distribution, ...}`
  - `[parameter_object_1, parameter_object_2, ...],`

  - `[[name_1, value_1 or Chaospy distribution], ...]`.
  - `[[name_1, value_1, Chaospy distribution or callable that returns a Chaospy distribution], ...]`

- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated. Default is None.
- **logger_level** (*{"info", "debug", "warning", "error", "critical"}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. Default is *"info"*.

Variables

- **`model`** (*`Model or Model subclass`*) – The model to perform uncertainty quantification on.
- **`parameters`** (*`Parameters`*) – The uncertain parameters.
- **`features`** (*`Features or subclass of Features`*) – The features of the model to perform uncertainty quantification on.
- **`logger_level`** (*`{"info", "debug", "warning", "error", "critical", None}`*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging is performed.

See also:

`uncertainpy.features.Features`, *`uncertainpy.models.Model`*

`uncertainpy.models.Model.run` Requirements for the model run function.

`features`

Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

`uncertainpy.features.Features`, *`uncertainpy.features.GeneralSpikingFeatures`*, *`uncertainpy.features.SpikingFeatures`*, *`uncertainpy.features.GeneralNetworkFeatures`*, *`uncertainpy.features.NetworkFeatures`*

`model`

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

uncertainpy.models.Model, *uncertainpy.models.Model.run*, *uncertainpy.models.NestModel*, *uncertainpy.models.NeuronModel*

parameters
Model parameters.

Parameters **new_parameters** (*{None, Parameters instance, list of Parameter instances, list [[name, value, distribution], . . .]}*) – Either None, a Parameters instance or a list of the parameters that should be created. The two lists are similar to the arguments sent to Parameters. Default is None.

Returns **parameters** – Parameters of the model. If None, no parameters have been set.

Return type {None, Parameters}

See also:

uncertainpy.Parameter, *uncertainpy.Parameters*

14.3 Parallel

Parallel calculates the model and features of the model for one specific set of model parameters. `Parallel` is the class that is run in parallel.

14.3.1 API Reference

class `uncertainpy.core.`**`Parallel`**(*model=None*, *features=None*, *logger_level=u'info'*)
Calculates the model and features of the model for one set of model parameters. Is the class that is run in parallel.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}, optional*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated. Default is None.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed Default logger level is "info".

Variables

- **`model`** (*uncertainpy.Parallel.model*) –
- **`features`** (*uncertainpy.Parallel.features*) –

See also:

uncertainpy.features.Features, *uncertainpy.models.Model*

uncertainpy.models.Model.run Requirements for the model run function.

`create_interpolations`(*result*)
Create an interpolation.

Model or feature *result* s that have a varying number of time steps, are interpolated. Interpolation is only performed for one dimensional *result*. Zero dimensional *result* does not need to be interpolated, and support for interpolating two dimensional and above *result* have currently not been implemented. Adds a *"interpolation"* key-value pair to *result*.

Parameters **result** (*dict*) – The model and feature results. The model and each feature each has a dictionary with the time values, `"time"`, and model/feature results, `"values"`. An example:

```
result = {model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8, 9,␣
↪10]),
                       "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9])},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}
```

Returns

result – If an interpolation has been created, those features/model have "interpolation" and the corresponding interpolation object added to each features/model dictionary. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                    [0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                              "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}
```

Return type dict

Notes

If either model or feature results are irregular, the results must be interpolated for Chaospy to be able to create the polynomial approximation. For 1D results this is done with scipy: `InterpolatedUnivariateSpline(time, U, k=3)`.

`features`

Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

`uncertainpy.features.Features`, *`uncertainpy.features.GeneralSpikingFeatures`*, *`uncertainpy.features.SpikingFeatures`*, *`uncertainpy.features.GeneralNetworkFeatures`*, *`uncertainpy.features.NetworkFeatures`*

`interpolation_1d`(*result*, *feature*)

Create an interpolation for an 1D result.

Parameters **result** (*dict*) – The model and feature results. The model and each feature each has a dictionary with the time values, `"time"`, and model/feature results, `"values"`. An example:

```
result = {model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8, 9,␣
↪10]),
                       "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9])},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}
```

Returns **interpolation** – The result of the interpolation. If either the time or values contain None or numpy.nan, None is returned.

Return type {scipy.interpolate.fitpack2.InterpolatedUnivariateSpline, None}

Raises

- `ValueError` – If the values of the feature are not 1D.
- `ValueError` – If the time of the feature is not 1D.

Notes

The interpolation is performed using scipy: `InterpolatedUnivariateSpline(time, values, k=3)`.

`model`

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

`uncertainpy.models.Model`, *`uncertainpy.models.Model.run`*, *`uncertainpy.models.NestModel`*, *`uncertainpy.models.NeuronModel`*

`run`(*model_parameters*)

Run a model and calculate features from the model output, return the results.

The model is run and each feature of the model is calculated from the model output, *time* (time values) and *values* (model result). The results are interpolated if they are irregular, meaning they return a varying number of steps. An interpolation is created and added to results for the model/features that are irregular. Each instance of None is converted to `numpy.nan`.

Parameters **model_parameters** (*dictionary*) – All model parameters as a dictionary. These parameters are sent to model.run().

Returns

result – The model and feature results. The model and each feature each has a dictionary with the time values, `"time"`, and model/feature results, `"values"`. If an interpolation has been created, those features/model also has `"interpolation"` added. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
```

(continues on next page)

(continued from previous page)

```
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                                "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                                "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                               "time": np.nan}}
```

Return type dictionary

Notes

Time *time* and result *values* are calculated from the model. Then sent to model.postprocess, and the post-processed result from model.postprocess is added to result. *time* and *values* are sent to features.preprocess and the preprocessed results is used to calculate each feature.

See also:

uncertainpy.utils.utility.none_to_nan() Method for converting from None to NaN

uncertainpy.features.Features.preprocess() preprocessing model results before features are calculated

uncertainpy.models.Model.postprocess() posteprocessing of model results

14.4 RunModel

RunModel is responsible for running the model in parallel for all selected sets of parameters. It runs *Parallel* in Parallel. `RunModel` organizes the results in a *Data* object.

14.4.1 API Reference

class uncertainpy.core.**RunModel**(*model*, *parameters*, *features=None*, *logger_level=u'info'*, *CPUs=u'max'*)

Calculate model and feature results for a series of different model parameters, and store them in a Data object.

Parameters

- **model** (*{None, Model or Model subclass instance, model function}, optional*) – Model to perform uncertainty quantification on. For requirements see Model.run. Default is None.
- **parameters** (*{dict {name: parameter_object}, dict of {name: value or Chaospy distribution}, . . .], list of Parameter instances, list [[name, value or Chaospy distribution], . . .], list [[name, value, Chaospy distribution or callable that returns a Chaospy distribution],. . .],}*) – List or dictionary of the parameters that should be created. On the form `parameters =`
  - `{name_1: parameter_object_1, name: parameter_object_2, ...}`
  - `{name_1: value_1 or Chaospy distribution, name_2: value_2 or Chaospy distribution, ...}`
  - `[parameter_object_1, parameter_object_2, ...],`

  - `[[name_1, value_1 or Chaospy distribution], ...]`.
  - `[[name_1, value_1, Chaospy distribution or callable that returns a Chaospy distribution], ...]`

- **features** (*{None, Features or Features subclass instance, list of feature functions}, optional*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated. Default is None.
- **logger_level** (*{"info", "debug", "warning", "error", "critical", None}, optional*) – Set the threshold for the logging level. Logging messages less severe than this level is ignored. If None, no logging to file is performed. Default logger level is "info".
- **CPUs** (*{int, None, "max"}, optional*) – The number of CPUs to use when calculating the model and features. If None, no multiprocessing is used. If "max", the maximum number of CPUs on the computer (multiprocess.cpu_count()) is used. Default is "max".

Variables

- **model** (*uncertainpy.Model or subclass of uncertainpy.Model*) – The model to perform uncertainty quantification on.
- **parameters** (*uncertainpy.Parameters*) – The uncertain parameters.
- **features** (*uncertainpy.Features or subclass of uncertainpy.Features*) – The features of the model to perform uncertainty quantification on.
- **CPUs** (*int*) – The number of CPUs used when calculating the model and features.

See also:

uncertainpy.features.Features, *uncertainpy.Parameter*, *uncertainpy.Parameters*, *uncertainpy.models.Model*

uncertainpy.models.Model.run Requirements for the model run function.

apply_interpolation(*results, feature*)

Perform interpolation of one model/feature using the interpolation objects created by Parallel.

Parameters

- **results** (*list*) – A list where each element is a result dictionary for each set of model evaluations. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7,␣
↪8, 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,
↪ 8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6,␣
↪7, 8, 9, 10]),
```

(continues on next page)

(continued from previous page)

```
                              "time": array([0, 1, 2, 3, 4, 5, 6,
↪ 7, 8, 9]),
                              "interpolation": scipy␣
↪interpolation object},
         "feature_invalid": {"values": np.nan,
                             "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

- **feature** (*str*) – Name of a feature or the model.

Returns

- **time** (*array_like*) – The time array with the greatest number of time steps.
- **interpolated_results** (*list*) – A list containing all interpolated model/features results. Interpolated at the points of the time results with the greatest number of time steps.

Notes

Chooses the time array with the highest number of time points and use this time array to interpolate the model/feature results in each of those points. If an interpolation is None, gives numpy.nan instead.

create_model_parameters(*nodes*, *uncertain_parameters*)

Combine nodes (values) with the uncertain parameter names to create a list of dictionaries corresponding to the model values for each model evaluation.

Parameters

- **nodes** (*array*) – A series of different set of parameters. The model and each feature is evaluated for each set of parameters in the series.
- **uncertain_parameters** (*list*) – A list of names of the uncertain parameters.

Returns

model_parameters – A list where each element is a dictionary with the model parameters for a single evaluation. An example:

```
model_parameter = {"parameter 1": value 1, "parameter 2": value 2, .
↪..}
model_parameters = [model_parameter 1, model_parameter 2, ...]
```

Return type list

evaluate_nodes(*nodes*, *uncertain_parameters*)

Evaluate the the model and calculate the features for the nodes (values) for the uncertain parameters.

Parameters

- **nodes** (*array*) – The values for the uncertain parameters to evaluate the model and features for.
- **uncertain_parameters** (*list*) – A list of the names of all uncertain parameters.

Returns

results – A list where each element is a result dictionary for each set of model evaluations. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                               "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

Return type list

Raises `ImportError` – If xvfbwrapper is not installed.

features

Features to calculate from the model result.

Parameters **new_features** (*{None, Features or Features subclass instance, list of feature functions}*) – Features to calculate from the model result. If None, no features are calculated. If list of feature functions, all will be calculated.

Returns **features** – Features to calculate from the model result. If None, no features are calculated.

Return type {None, Features object}

See also:

uncertainpy.features.Features, *uncertainpy.features.GeneralSpikingFeatures*, *uncertainpy.features.SpikingFeatures*, *uncertainpy.features.GeneralNetworkFeatures*, *uncertainpy.features.NetworkFeatures*

is_regular (*results*, *feature*)

Test if *feature* in *results* is regular or not, meaning it has a varying number of values for each evaluation. Ignores results that contains numpy.nan.

Parameters

- **results** (*list*) – A list where each element is a result dictionary for each set of model evaluations. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7,␣
↪8, 9, 10]),
                           "time": array([0, 1, 2, 3, 4, 5, 6, 7,
↪ 8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6,␣
↪7, 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,
↪ 7, 8, 9]),
                               "interpolation": scipy␣
↪interpolation object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

- **feature** (*str*) – Name of a feature or the model.

Returns True if the feature is regular or False if the feature is irregular.

Return type bool

model

Model to perform uncertainty quantification on. For requirements see Model.run.

Parameters **new_model** (*{None, Model or Model subclass instance, model function}*) – Model to perform uncertainty quantification on.

Returns **model** – Model to perform uncertainty quantification on.

Return type Model or Model subclass instance

See also:

uncertainpy.models.Model, *uncertainpy.models.Model.run*, *uncertainpy.models.NestModel*, *uncertainpy.models.NeuronModel*

parameters

Model parameters.

Parameters **new_parameters** (*{None, Parameters instance, list of Parameter instances, list [[name, value, distribution], ...]}*) – Either None, a Parameters instance or a list of the parameters that should be created. The two lists are similar to the arguments sent to Parameters. Default is None.

Returns **parameters** – Parameters of the model. If None, no parameters have been set.

Return type {None, Parameters}

See also:

uncertainpy.Parameter, *uncertainpy.Parameters*

regularize_nan_results(*results*)

Regularize arrays with that only contain numpy.nan values.

Make each result for each feature have the same the same shape, if they only contain numpy.nan values.

Parameters results (*list*) – A list where each element is a result dictionary for each set of model evaluations. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                               "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                               "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

Returns

results – A list with where the only nan results have been regularized. On the form:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                            "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
```

(continues on next page)

(continued from previous page)

```
                                "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                                "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

Return type list

results_to_data (*results*)

Store *results* in a Data object.

Stores the time and (interpolated) results for the model and each feature in a Data object. Performs the interpolation calculated in Parallel, if the result is irregular.

Parameters **results** (*list*) – A list where each element is a result dictionary for each set of model evaluations. An example:

```
result = {self.model.name: {"values": array([1, 2, 3, 4, 5, 6, 7, 8,
↪ 9, 10]),
                             "time": array([0, 1, 2, 3, 4, 5, 6, 7,␣
↪8, 9])},
          "feature1d": {"values": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature0d": {"values": 1,
                        "time": np.nan},
          "feature2d": {"values": array([[0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9],
                                         [0, 1, 2, 3, 4, 5, 6, 7, 8,
↪ 9]]),
                        "time": array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9])},
          "feature_adaptive": {"values": array([1, 2, 3, 4, 5, 6, 7,
↪ 8, 9, 10]),
                                "time": array([0, 1, 2, 3, 4, 5, 6,␣
↪7, 8, 9]),
                                "interpolation": scipy interpolation␣
↪object},
          "feature_invalid": {"values": np.nan,
                              "time": np.nan}}

results = [result 1, result 2, ..., result N]
```

Returns **data** – A Data object with time and (interpolated) results for the model and each feature.

Return type Data object

Notes

Sets the following in data, if applicable: 1. `data["model/features"].evaluations`, which contains all `values` 2. `data["model/features"].time` 3. `data["model/features"].labels` 4. `data.model_name`

See also:

uncertainpy.Data()

run(*nodes*, *uncertain_parameters*)

Evaluate the the model and calculate the features for the nodes (values) for the uncertain parameters. The results are interpolated as necessary.

Parameters

- **nodes** (*array*) – A series of different set of parameters. The model and each feature is evaluated for each set of parameters in the series.
- **uncertain_parameters** (*list*) – A list of names of the uncertain parameters.

Returns **data** – A Data object with time and (interpolated) results for the model and each feature.

Return type Data object

See also:

uncertainpy.Data()

CHAPTER 15

Theory

15.1 The problem definition

Consider a model U that depends on space $\boldsymbol{x}$ and time t, has D uncertain input parameters $\boldsymbol{Q} = [Q_1, Q_2, \ldots, Q_D]$, and gives the output Y:

$$Y = U(\boldsymbol{x}, t, \boldsymbol{Q}).$$

The output Y can be any value within the output space Ω_Y and has an unknown probability density function ρ_Y. The goal of an uncertainty quantification is to describe the unknown ρ_Y through statistical metrics. We are only interested in the input and output of the model, and we ignore all details on how the model works. The model U is thus considered a black box, and may represent any model, for example a spiking neuron model that returns a voltage trace, or a network model that return a spike train.

We assume the model includes uncertain parameters that can be described by a multivariate probability density function $\rho_{\boldsymbol{Q}}$. Examples of parameters that can be uncertain in neuroscience are the conductance of a single ion channel, or the synaptic weight between two species of neurons in a network. If the uncertain parameters are independent, the multivariate probability density function $\rho_{\boldsymbol{Q}}$ can be given as separate univariate probability density functions ρ_{Q_i}, one for each uncertain parameter Q_i. The joint multivariate probability density function for the independent uncertain parameters is then:

$$\rho_{\boldsymbol{Q}} = \prod_{i=1}^{D} \rho_{Q_i}.$$

In cases where the uncertain input parameters are dependent, the multivariate probability density function $\rho_{\boldsymbol{Q}}$ must be defined directly. We assume the probability density functions are known, and are not here concerned with how they are determined. They may be the product of a series of measurements, a parameter estimation, or educated guesses made by experts.

15.2 Uncertainty quantification

The goal of an uncertainty quantification is to describe the unknown distribution of the model output ρ_Y through statistical metrics. The two most common statistical metrics used in this context are the mean $\mathbb{E}$ (also called the expectation value) and the variance $\mathbb{V}$. The mean is defined as:

$$\mathbb{E}[Y] = \int_{\Omega_Y} y\rho_Y(y)dy,$$

and tells us the expected value of the model output Y. The variance is defined as:

$$\mathbb{V}[Y] = \int_{\Omega_Y} (y - \mathbb{E}[Y])^2 \rho_Y(y)dy,$$

and tells us how much the output varies around the mean.

Another useful metric is the $(100 \cdot x)$-th percentile P_x of Y, which defines a value below which $100 \cdot x$ percent of the simulation outputs are located. For example, 5% of the simulations of a model will give an output lower than the 5-th percentile. The $(100 \cdot x)$-th percentile is defined as:

$$x = \int_{-\infty}^{P_x} \rho_Y(y)dy.$$

We can combine two percentiles to create a prediction interval I_x, which is a range of values such that a $100 \cdot x$ percentage of the outputs Y occur within this range:

$$I_x = \left[P_{(x/2)}, P_{(1-x/2)}\right].$$

The 90% prediction interval gives us the interval within 90% of the Y outcomes occur, which also means that 5% of the outcomes are above and 5% below this interval.

15.3 Sensitivity analysis

Sensitivity analysis quantifies how much of the uncertainty in the model output each uncertain parameter is responsible for. It is the computational equivalent of analysis of variance (ANOVA) performed by experimentalists (Archer et al., 1997). For a review of different sensitivity analysis methods, see Hamby (1994); Borgonovo and Plischke (2016). Several different sensitivity measures exist, but Uncertainpy uses the commonly used Sobol sensitivity indices (Sobol, 1990). The Sobol sensitivity indices quantify how much of the variance in the model output each uncertain parameter is responsible for. If a parameter has a low sensitivity index, variations of this parameter results in comparatively small variations in the final model output. On the other hand, if a parameter has a high sensitivity index, a change in this parameter leads to a dramatic change in the model output.

A sensitivity analysis provides a better understanding of the relationship between the parameters and output of a model. This can be useful in a model reduction context. For example, a parameter with a low sensitivity index can essentially be set to any fixed value (within the explored distribution), without affecting the variance of the model much. In some cases, such an analysis can justify leaving out entire mechanisms from a model. For example, if a single neuron model is insensitive to the conductance of a given ion channel g_x, this ion channel could possibly be removed from the model without changing the model behavior much. Additionally, a model-based sensitivity analysis can guide the experimental focus, so that special care is taken to obtain accurate measures of parameters with high sensitivity indices, while more crude measures are acceptable for parameters with low sensitivity indices.

There exist several types of Sobol indices. The first order Sobol sensitivity index S measures the direct effect each parameter has on the variance of the model:

$$S_i = \frac{\mathbb{V}[\mathbb{E}[Y|Q_i]]}{\mathbb{V}[Y]}.$$

Here, $\mathbb{E}[Y|Q_i]$ denotes the expected value of the output Y when parameter Q_i is fixed. The first order Sobol sensitivity index tells us the expected reduction in the variance of the model when we fix parameter Q_i. The sum of the first order Sobol sensitivity indices can not exceed one (Glen and Isaacs, 2012).

Higher order sobol indices exist, and give the sensitivity due interactions between a parameter Q_i and various other parameters. It is customary to only calculate the first and total order indices (Saltelli et al., 2010). The total Sobol sensitivity index S_{Ti} includes the sensitivity of both first order effects as well as the sensitivity due to interactions (covariance) between a given parameter Q_i and all other parameters (Homma and Saltelli, 1996). It is defined as:

$$S_{Ti} = 1 - \frac{\mathbb{V}[\mathbb{E}[Y|Q_{-i}]]}{\mathbb{V}[Y]},$$

where Q_{-i} denotes all uncertain parameters except Q_i. The sum of the total Sobol sensitivity indices is equal to or greater than one (Glen and Isaacs, 2012). If no higher order interactions are present, the sum of both the first and total order sobol indices are equal to one.

We might want to compare Sobol indices across different features (see in *Features*). This can be problematic when we have features with different number of output dimensions. In the case of a zero dimensional output the Sobol indices is a single number, while for a one dimensional output we get Sobol indices for each point in time. To better be able to compare the Sobol indices across such features, we therefore calculate the average of both the first order Sobol indices $\widehat{S}$, and the total order Sobol indices $\widehat{S}_T$.

15.4 (Quasi-)Monte Carlo methods

A typical way to obtain the statistical metrics mentioned above is to use (quasi-)Monte Carlo methods. We give a brief overview of these methods here, for more comprehensive reviews see Lemieux, (2009); Rubinstein and Kroese (2016).

The general idea behind the standard Monte Carlo method is quite simple. A set of parameters is pseudo-randomly drawn from the joint multivariate probability density function $\rho_{\boldsymbol{Q}}$ of the parameters. The model is then evaluated for the sampled parameter set. This process is repeated thousand of times, and statistical metrics such as the mean and variance are computed for the resulting series of model outputs. The problem with the standard Monte Carlo method is that a very high number of model evaluations is required to get reliable statistics. If the model is computationally expensive, the Monte Carlo method may require insurmountable computer power.

Quasi-Monte Carlo methods improve upon the standard Monte Carlo method by using variance-reduction techniques to reduce the number of model evaluations needed. These methods are based on increasing the coverage of the sampled parameter space by distributing the samples more evenly. Fewer samples are then required to get a given accuracy. Instead of pseudo-randomly selecting parameters from $\rho_{\boldsymbol{Q}}$, the samples are selected using a low-discrepancy sequence such as the Hammersley sequence (Hammersley, 1960). Quasi-Monte Carlo methods are faster than the Monte Carlo method, as long as the number of uncertain parameters is sufficiently small (Lemieux, 2009).

Uncertainpy allows quasi-Monte Carlo methods to be used to compute the statistical metrics. When this option is chosen, the metrics are computed as follows. With N model evaluations, which gives the results $\boldsymbol{Y} = [Y_1, Y_2, \ldots, Y_N]$, the mean is given by

$$\mathbb{E}[\boldsymbol{Y}] \approx \frac{1}{N}\sum_{i=1}^{N} Y_i,$$

and the variance by

$$\mathbb{V}[\boldsymbol{Y}] \approx \frac{1}{N-1}\sum_{i=1}^{N} (Y_i - \mathbb{E}[Y])^2.$$

Prediction intervals are found by sorting the model evaluations $\boldsymbol{Y}$ in an increasing order, and then find the $(100 \cdot x/2)$-th and $(100 \cdot (1 - x/2))$-th percentiles. The Sobol indices can be calculated using the method in (Saltelli et al., 2010). The total number of samples N_t required by this method is:

$$N_t = N(D + 2)$$

15.5 Polynomial chaos expansions

A recent mathematical framework for estimating uncertainty is that of polynomial chaos expansions (Xiu and Hesthaven, 2005). Polynomial chaos expansions can be seen as a subset of polynomial approximation methods. For a review of polynomial chaos expansions see (Xiu, (2010)). Polynomial chaos expansions are much faster than (quasi-)Monte Carlo methods as long as the number of uncertain parameters is relatively low, typically smaller than about twenty (Crestaux et al.,2009). This is the case for many neuroscience models, and even for models with a higher number of uncertain parameters, the analysis could be performed for selected subsets of the parameters.

The general idea behind polynomial chaos expansions is to approximate the model U with a polynomial expansion $\hat{U}$:

$$U \approx \hat{U}(\boldsymbol{x}, t, \boldsymbol{Q}) = \sum_{n=0}^{N_p-1} c_n(\boldsymbol{x}, t)\boldsymbol{\phi}_n(\boldsymbol{Q}),$$

where $\boldsymbol{\phi}_n$ denote polynomials and c_n denote expansion coefficients. The number of expansion factors N_p is given by

$$N_p = \binom{D+p}{p},$$

where p is the polynomial order. The number of expansion coefficients in the multivariate case ($D > 1$) is greater than the polynomial order. This is because the multivariate polynomial is created by multiplying univariate polynomials together. The polynomials $\phi_n(\boldsymbol{Q})$ are chosen so they are orthogonal with respect to the probability density function $\rho_{\boldsymbol{Q}}$, which ensures useful statistical properties.

When creating the polynomial chaos expansion, the first step is to find the orthogonal polynomials $\boldsymbol{\phi}_n$, which in Uncertainpy is done using the so called three-term recurrence relation (Xiu, 2010). The next step is to estimate the expansion coefficients c_n. The non-intrusive methods for doing this can be divided into two classes, point-collocation methods and pseudo-spectral projection methods, both of which are implemented in Uncertainpy.

Point collocation is the default method used in Uncertainpy. This method is based on demanding that the polynomial approximation is equal to the model output evaluated at a set of collocation nodes drawn from the joint probability density function $\rho_{\boldsymbol{Q}}$. This demand results in a set of linear equations for the polynomial coefficients c_n, which can be solved by the use of regression methods. The regression method used in Uncertainpy is Tikhonov regularization (Rifkin and Lippert, 2007).

Pseudo-spectral projection methods are based on least squares minimization in the orthogonal polynomial space, and finds the expansion coefficients c_n through numerical integration. The integration uses a quadrature scheme with weights and nodes, and the model is evaluated at these nodes. The quadrature method used in Uncertainpy is Leja quadrature, with Smolyak sparse grids to reduce the number of nodes required (Narayan and Jakeman, 2014; Smolyak, 1963). Pseudo-spectral projection methods are only used in Uncertainpy when requested by the user.

Several of the statistical metrics of interest can be obtained directly from the polynomial chaos expansion $\hat{U}$. The mean is simply

$$\mathbb{E}[U] \approx \mathbb{E}[\hat{U}] = c_0,$$

and the variance is

$$\mathbb{V}[U] \approx \mathbb{V}[\hat{U}] = \sum_{n=1}^{N_p-1} \gamma_n c_n^2,$$

where γ_n is a normalization factor defined as

$$\gamma_n = \mathbb{E}\left[\boldsymbol{\phi}_n^2(\boldsymbol{Q})\right].$$

The first and total order Sobol indices can also be calculated directly from the polynomial chaos expansion (Sudret, 2008; Crestaux et al.,2009). On the other hand, the percentiles and prediction interval must be estimated using $\hat{U}$ as a surrogate model, and then perform the same procedure as for the (quasi-)Monte Carlo methods.

15.6 Dependency between uncertain parameters

One of the underlying assumptions when creating the polynomial chaos expansion is that the model parameters are independent. However, dependent parameters in neuroscience models are quite common (Achard and De Schutter, 2006). Fortunately, models containing dependent parameters can be analyzed with Uncertainpy by the aid of the Rosenblatt transformation from Chaospy (Rosenblatt, 1952; Feinberg and Langtangen, 2015). The idea is to use the Rosenblatt transformation to create a reformulated model $\widetilde{U}(\boldsymbol{x}, t, \boldsymbol{R})$, that takes an arbitrary independent parameter set $\boldsymbol{R}$ as input, instead of the original dependent parameter set $\boldsymbol{Q}$. We use the Rosenblatt transformation to transform from $\boldsymbol{R}$ to $\boldsymbol{Q}$, which makes it so $\widetilde{U}$ give the same output (and statistics) as the original model:

$$\widetilde{U}(\boldsymbol{x}, t, \boldsymbol{R}) = U(\boldsymbol{x}, t, \boldsymbol{Q}).$$

We can then perform polynomial chaos expansion as normal on the reformulated model, since it has independent parameters.

The Rosenblatt transformation $T_{\boldsymbol{Q}}$ transforms the random variable $\boldsymbol{Q}$ to the random variable $\boldsymbol{H}$, which in a statistical context behaves as if it were drawn uniformly from the unit hypercube $[0, 1]^D$.

$$T_{\boldsymbol{Q}}(\boldsymbol{Q}) = \boldsymbol{H}.$$

Here, $T_{\boldsymbol{Q}}$ denotes a Rosenblatt transformation which is uniquely defined by ρ_Q (the probability distribution of $\boldsymbol{Q}$). We can use the Rosenblatt transformation to transform from $\boldsymbol{R}$ to $\boldsymbol{Q}$ (through $\boldsymbol{H}$) to regain our original parameters:

$$\begin{aligned} T_{\boldsymbol{Q}}(\boldsymbol{Q}) &= \boldsymbol{H} = T_{\boldsymbol{R}}(\boldsymbol{R}) \\ \boldsymbol{Q} &= T_{\boldsymbol{Q}}^{-1}(T_{\boldsymbol{R}}(\boldsymbol{R})). \end{aligned}$$

Using this relation between $\boldsymbol{R}$ and $\boldsymbol{Q}$ in we can reformulate our model to take $\boldsymbol{R}$ as input, but still give the same results:

$$U(\boldsymbol{x}, t, \boldsymbol{Q}) = U(\boldsymbol{x}, t, T_{\boldsymbol{Q}}^{-1}(T_{\boldsymbol{R}}(\boldsymbol{R}))) = \widetilde{U}(\boldsymbol{x}, t, \boldsymbol{R}).$$

The statistical analysis can now be performed on this reformulated model $\widetilde{U}$ as before.

Here we give an overview of the theory behind uncertainty quantification and sensitivity analysis with a focus on (quasi-)Monte Carlo methods and polynomial chaos expansions, the methods implemented in Uncertainpy.

Uncertainty quantification and sensitivity analysis provide rigorous procedures to analyse and characterize the effects of parameter uncertainty on the output of a model. The methods for uncertainty quantification and sensitivity analysis can be divided into global and local methods. Local methods keep all but one model parameter fixed and explore how much the model output changes due to variations in that single parameter. Global methods, on the other hand, allows the entire parameter space to vary simultaneously. Global methods can therefore identify complex dependencies between the model parameters in terms of how they affect the model output.

The global methods can be further divided into intrusive and non-intrusive methods. Intrusive methods require changes to the underlying model equations, and are often challenging to implement. Models in neuroscience are often created with the use of advanced simulators such as NEST and NEURON. Modifying the underlying equations of models using these simulators is a complicated task best avoided. Non-intrusive methods, on the other hand, consider the model as a black box, and can be applied to any model without needing to modify the model equations or implementation. Global, non-intrusive methods are therefore the methods of choice in Uncertainpy. The uncertainty calculations in Uncertainpy is based on the Python package Chaospy, which provides global non-intrusive methods for uncertainty quantification and sensitivity analysis.

We start by introducing the *problem definition*. Next, we introduce the statistical measurements for *uncertainty quantification* and *sensitivity analysis*. Further, we give an introduction to *(quasi-)Monte Carlo methods* and *Polynomial chaos expansions*, the two methods used to calculate the uncertainty and sensitivity in Uncertainpy. We next explain how Uncertainpy handle cases with *dependent model parameters*. We note that insight into this theory is not a prerequisite for using Uncertainpy.

CHAPTER 16

Uncertainpy paper

The Uncertainpy paper can be found here: Tennøe S, Halnes G, and Einevoll GT (2018) Uncertainpy: A Python Toolbox for Uncertainty Quantification and Sensitivity Analysis in Computational Neuroscience. Front. Neuroinform. 12:49. doi: 10.3389/fninf.2018.00049.

CHAPTER 17

Getting started

CHAPTER 18

Examples

This is a collection of examples that shows the use of Uncertainpy for a few different case studies.

- *A simple cooling coffee cup*
- *A cooling coffee cup with dependent parameters*
- *The Hodgkin-Huxley model*
- *A multi-compartment model of a thalamic interneuron*
- *A sparsely connected recurrent network*

CHAPTER 19

Frequently asked questions

- *Multiple model outputs*.

CHAPTER 20

Content of Uncertainpy

This is the content of Uncertainpy and contains instructions for how to use all classes and functions, along with their API.

- *UncertaintyQuantification*
- ***Models***
  - *General models*
  - *Nest models*
  - *Neuron models*
  - *Multiple model outputs*
- *Parameters*
- ***Features***
  - *General features*
  - *Spiking features*
  - *Spikes* (used by the spiking features)
  - *Electrophys Feature Extraction Library (eFEL) features*
  - *Network features*
  - *General spiking features*
  - *General network features*
- *Data*
- *Utility distributions*
- *Plotting*
- *Logging*
- *Utilities*

CHAPTER 21

Theory

Here we give an overview of the theory behind uncertainty quantification and sensitivity analysis with a focus on (quasi-)Monte Carlo methods and polynomial chaos expansions, the methods implemented in Uncertainpy.

Python Module Index

Index