econsa Documentation

Release 0.01

Linda Maokomatanda

Jun 23, 2021

CONTENTS

I prefer true but imperfect knowledge, even if it leaves much undetermined and unpredictable, to a pretence of exact knowledge that is likely to be false.

– Friedrich von Hayek, Nobel Prize Lecture

econsa is an open-source package for economists that facilitates the sound analysis of computational economic models. It offers suitable methods for uncertainty propagation and global sensitivity analysis.

With conda available on your path, installing and testing econsa is as simple as typing

```
$ conda install -c opensourceeconomics econsa
$ python -c "import econsa; econsa.test()"
```

CHAPTER

ONE

SUPPORTED BY

1.1 Motivation

Computational economic models clearly specify an individual's objective and the institutional and informational constraints of their economic environment under which they operate. Fully-parameterized computational implementations of the economic model are estimated on observed data as to reproduce the observed individual decisions and experiences. Based on the results, researchers can quantify the importance of competing economic mechanisms in determining economic outcomes and forecast the effects of alternative policies before their implementation ([16]).

The uncertainties involved in such an analysis are ubiquitous. Any such model is subject to misspecification, its numerical implementation introduces approximation error, the data is subject to measurement error, and the estimated parameters remain partly uncertain.

A proper accounting of the uncertainty is a prerequisite for the use of computational models in most disciplines ([1, 11]) and has long been recognized in economics as well ([3, 5, 9]). However, in practice economists analyze the implications of the estimated model, economists display incredible certitude ([10]) as all uncertainty is disregarded. As a result, flawed findings are accepted as truth and contradictory results are competing. Both have the potential to undermine the public trust in research in the long run.

Any computational economic model $\boldsymbol{M}$ provides a mapping between its input parameters $\boldsymbol{x}$ and the quantities of interest y.

$$\boldsymbol{M} : \boldsymbol{x} \in \mathcal{D}_{\boldsymbol{X} \mapsto y = \boldsymbol{M}(\boldsymbol{x})}$$

We follow [2] and use the **Economic Order Quantity (EOQ)** model ([6]) as a running example throughout our documentation. We thus start by explaining its basic setup first and then discuss uncertainty propagation and sensitivity analysis.

1.1.1 EOQ model

The **EOQ** inventory management model provides a stylized representation of the decision problem faced by a firm that needs to determine the order quantity of a product that minimizes the unit cost per piece. The unit cost T depends on the price of the product C, the size of the order X as each comes with a fixed cost S, and an annual capital cost R expressed as a percentage of the value of the inventory. Core simplifications of the model include a constant monthly demand M over the year and the delivery of each order in full when inventory reaches zero.

We can then derive the unit cost as follows:

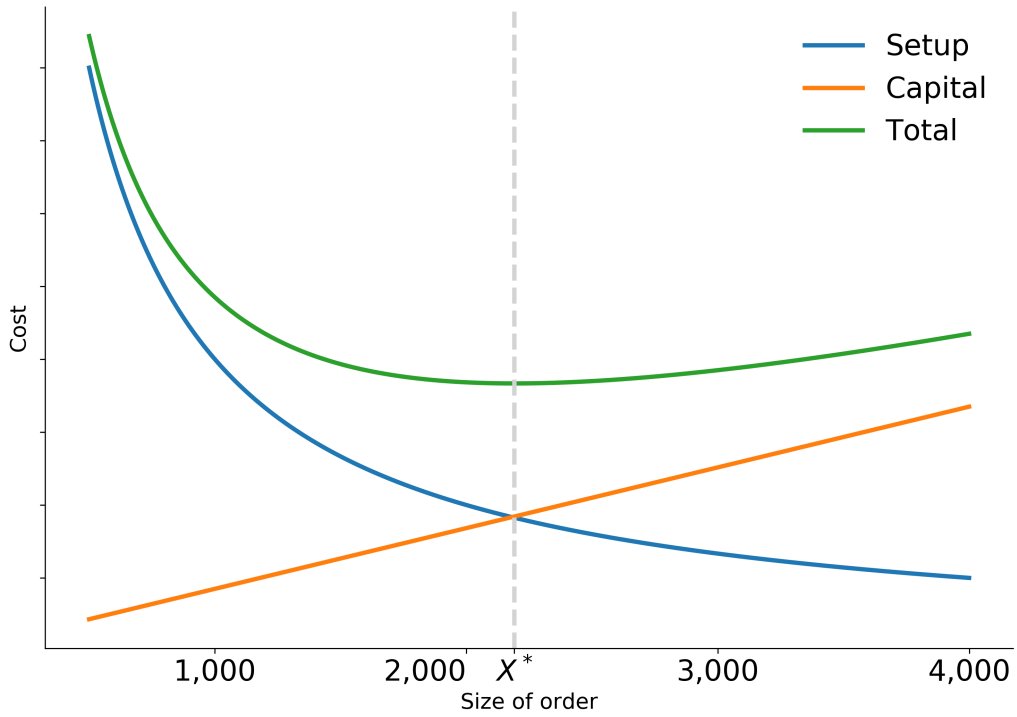
$$T = \underbrace{\frac{1}{12 \times M} \times R \times \frac{C \times X + S}{2}}_{\text{Part I}} + \underbrace{\frac{S}{X} + C}_{\text{Part II}}.$$

The first part of the equation denotes the capital cost of one unit in storage. It is computed based on the ratio of the value of the average stock and the total number of ordered units during the year. The second part captures each unit's cost as part of an order of size X.

The economic order quantity X^* is determined as:

$$X^* = \sqrt{\frac{24 \times M \times S}{R \times C}}.$$

The figure below reproduces the fundamental economic trade-offs of the model for a fixed parameterization of M, C, S, and R. An increase in the size of order X results in a decrease in the setup cost per unit, but at the same time, capital cost increases as the stock of inventory increase.



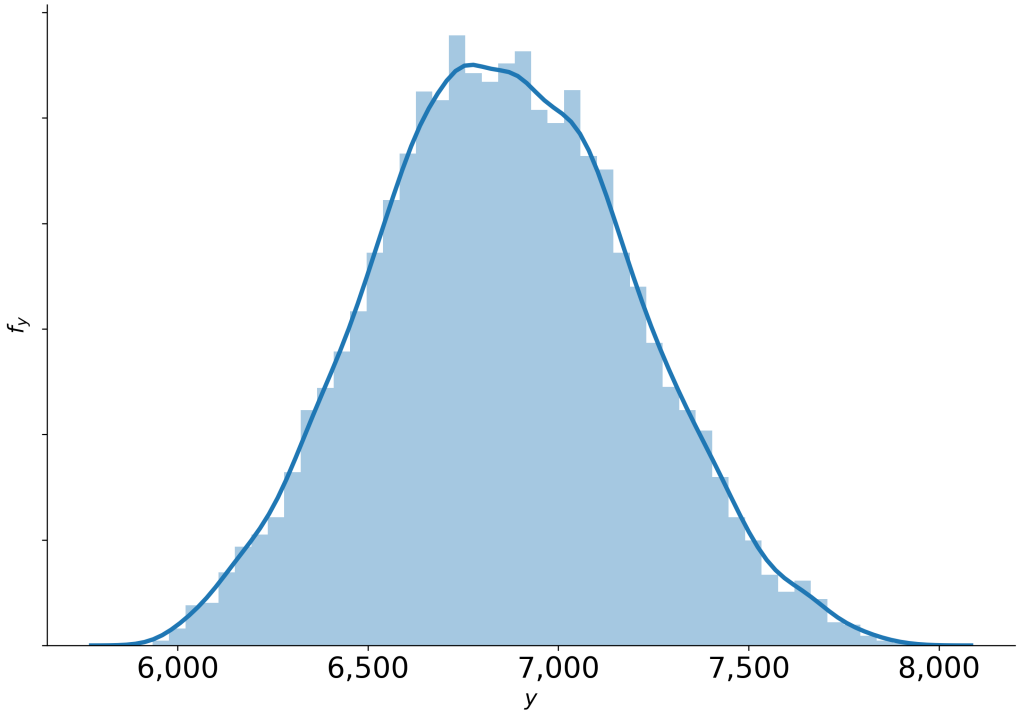
Going forward, we treat the annual interest and depreciation rate R as an exogenous parameter and set it to 10%. We can map the example to our more general notation by denoting the optimal order quantity as y and collecting the three remaining input parameters in $\mathbf{x}$ as follows:

$$\mathbf{x} = (x_1, x_2, x_3)^T = (M, C, S)^T.$$

1.1.2 Uncertainty propagation

We start from a probabilistic model for the input parameters that is informed by, for example, expert knowledge or the outcome of a calibration. We treat the model parameters $\mathbf{X}$ as a simple random vector with a joint probability density function $f_{\mathbf{X}}$. We are not particularly interested in the uncertainty of each individual parameter of the model. Instead we seek to learn about the induced distribution of the model output Y as the uncertainty about the model parameters $\mathbf{X}$ propagates through the computational model $\mathbf{M}$. We want to study the statistical properties of Y.

We now return to the example of the **EOQ** model. We specify a uniform distribution centered around $\mathbf{x^0} = (M, C, S) = (1230, 0.0135, 2.15)$ and spread the support 10% above and below the center. We solve for the optimal economic order quantity Y for $1,000$ random input parameters and end up with the distribution f_Y below.



1.1.3 Qualitative sensitivity analysis

Elementary effects

1.1.4 Quantitative sensitivity analysis

When analyzing (complex) computational models it is often unclear from the model specification alone how the inputs of the model contribute to the outputs. As we've seen in the previous tutorial on *Qualitative sensitivity analysis*, a first step is to sort the inputs by their respective order of importance on the outputs. In many cases however, we would like to learn by how much the individual inputs contribute to the output in relation to the other inputs. This can be done using Sobol indices ([13]). Classical Sobol indices are designed to work on models with independent input variables. However, since in economics this independence assumption would be very questionable, we focus on so called generalized Sobol indices, as those proposed by [8], that also work in the case of dependent inputs.

Generalized Sobol indices

Say we have a model $\mathcal{M} : \mathbb{R}^n \to \mathbb{R}, x \mapsto \mathcal{M}(x)$ and we are interested in analyzing the variance of its output on a given subset $U \subset \mathbb{R}^n$, i.e. we want to analyze

$$D := \text{Var}(\mathcal{M}|_U) := \int_U (\mathcal{M}(x) - \mu_U)^2 f_X(x) \mathrm{d}x$$

where $\mu_U := \int_U \mathcal{M}(x) f_X(x) \mathrm{d}x$ denotes the restricted mean of the model and f_X denotes the probability density function imposed on the input parameters. For the sake of brevity let us assume that $\mathcal{M}$ is already restricted so that we can drop the dependence on S. To analyze the effect of a single variable, or more general a subset of variable, consider partitioning the model inputs as $(y, z) = x$. The construction of Sobol and generalized Sobol indices starts with noticing that we can decompose the overall variance as

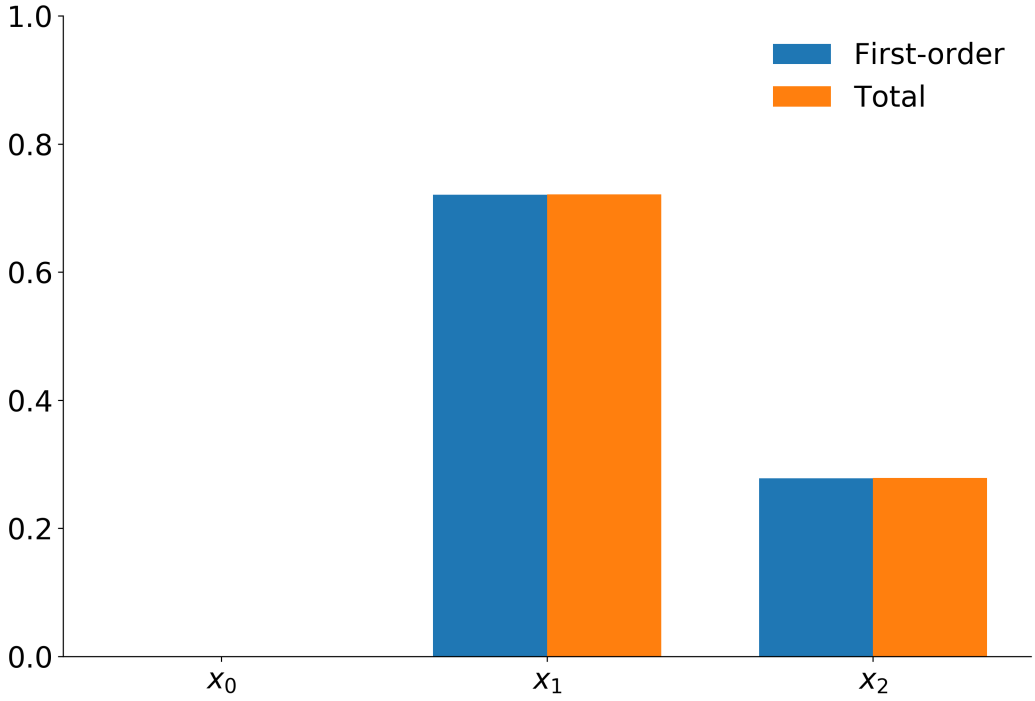
$$D = \text{Var}_y(\mathbb{E}_z [\mathcal{M}(y, z) \mid y]) + \mathbb{E}_y [\text{Var}_z(\mathcal{M}(y, z) \mid y)]$$

which implies that

$$1 = \frac{\text{Var}_y(\mathbb{E}_z [\mathcal{M}(y, z) \mid y])}{D} + \frac{\mathbb{E}_y [\text{Var}_z(\mathcal{M}(y, z) \mid y)]}{D} =: S_y + S_z^T$$

We call S_y the *first order effect index* of the subset y and we call S_z^T the *total effect* of the subset z. Notice that for each partition of the input space y and z, the above provides a way of computing the fraction of explained variance. For the sake of clarity, assume y represent only a single input variable. Then S_y can be interpreted as the effect of y on the variability of $\mathcal{M}$ **without** considering any interaction effects with other variables. While S_y^T can be thought of as representing the effect of y on the variance via itself **and** all other input variables.

Again, we now apply this to the **EOQ** model. Given the current limits to our implementation and the fact that the parameters of the model need to remain positive, we specify that the parameters follow a normal distribution with a very small variance.



Shapely values

In this overview, we give brief notational insights on variance-based sensitivity analysis as well as the Shapley value's theoratical framework ([14]). We follow the framework on variance-based sensitivity analysis and Shapley values developed by [14].

Variance-based Sensitivity Analysis (SA) can be illustrated in the following manner. Consider a model with k inputs denoted by $X_K = \{X_1, X_2, X_3, \ldots, X_k\}$ where $K = \{1, 2, \ldots, k\}$. Consider also X_J, which indicates the vector of inputs included in the index set $J \subseteq X$. The uncertainty in X_K is represented by the joint cumulative distribution G_K. Furthermore, we denote the joint distribution of inputs included in the index set J as G_J and the marginal distribution of each X_i as G_i. The model is treated as a blackbox, and only the model response is analysed. The model response Y is a function of the inputs, i.e., $Y = f(X_K)$ and therefore $f(X_K)$ is stochastic due to the uncertainty in X_K although $f(\cdot)$ is deterministic. Often, $f(\cdot)$ has a complex structure, and does not have a closed form expression. The overall uncertainty in the model output Y caused by X_K is $Var[Y]$, where the variance is calculated with respect to the joint distribution G_K. The Shapley value then, helps us to quantify how much of $Var[Y]$ can be attributed to each each X_i.

An analogous framework to the one developed for variance-based sensitivity analysis above is apparent in the specification of the Shapley value. Formally, a *k-player game* with the set of players $K = \{1, 2, \ldots, k\}$ is defined as a real valued function that maps a subset of K to its corresponding cost (or value), i.e., $c : 2^K \rightarrow \mathbb{R}$ with $c(\emptyset) = 0$. With this in mind, $c(J)$ then, represents the cost that arises when the players in the subset J of K participate in the game. The Shapley value for player i with respect to $c(\cdot)$ is defined as

$$v_i = \sum_{J \subseteq K \setminus \{i\}} \frac{(k - |J| - 1)!|J|!}{k!} \cdot (c(J \cup \{i\}) - c(J)),$$

where $|J|$ indicates the size of J. In other words, v_i is the incremental cost of including player i in set J averaged over all sets $J \subseteq K \setminus \{i\}$. The Shapley value gives equal weight to each k subset sizes and equal weights amongst the subsets of the same size, which is important in determining the fairness of the variance allocation in the calculation of Shapley effects in variance-based sensitivity analysis ([14]). Reconciling the two frameworks by direct comparison, we can think of the set of K players as the set of inputs of $f(\cdot)$ and define $c(\cdot)$ so that for $J \subseteq K$, $c(J)$ measures the variance of $c(J)$ caused by the uncertainty of the inputs in J.

The ideal $c(\cdot)$ should satisfy the conditions: $c(\emptyset) = 0$ and $c(K) = Var[Y]$. Two such candidates for such $c(\cdot)$ can be considered, and have been shown to be equivalent are equivalent ([14]). The first cost function is

$$\tilde{c}(J) = Var[E[Y|X_J]].$$

This cost function satisfies the two conditions from above and was originally put forth by [12] and later adopted by [14] in their paper. The cost function can be rewritten as $\tilde{c}(J) = Var[Y] - E[Var[Y|X_J]]$, and interpreted as the expected reduction in the output variance when the values of X_J are known. The second cost function that satisfies the required conditions is

$$c(J) = E[Var[Y|X_{-J}]]$$

where $X_{-J} = X_{K \setminus J}$. $c(J)$ is interpreted as the expected remaining variance in Y when the values of X_{-J} are known. In this case, the incremental cost $c(J \cup \{i\}) - c(J)$ can be interpreted as the expected decrease in the variance of Y conditional on the known input values of X_i out of all the unknown inputs in $J \cup \{i\}$.

Although both cost functions result in the same Shapley values, their resultant estimators from Monte Carlo simulation are different. [15] reveal that the Monte Carlo estimator that results from the simulation of $\tilde{c}(J)$ can be severely biased if the inner level sample size used to estimate the conditional expectation is not large enough. Given the already computationally demanding structure of microeconomic models, this added computational complexity is costly. In contrast however, the estimator of $c(J)$ is unbiased for all sample sizes. Because of this added feature, we follow [14] in using the cost function $c(J)$ rather that $\tilde{c}(J)$. We therefore define the *Shapley effect* of the i_{th} input, Sh_i, as the Shapley value obtained by applying the cost function $c(J)$ to the Shapley value equation. Indeed, any Shapley value

defined by the satisfaction of the two conditions: $c(\emptyset) = 0$ and $c(K) = Var[Y]$ imply that

$$\sum_{k}^{i=1} Sh_i = Var[Y],$$

even if there is dependence or structural interactions amongst the elements in X_K. Throughout the package, we use Sh_i to denote the Shapley effect and v_i to denote the generic Shapley value.

Quantile based sensitivity measures

This part will be written by Yulei Li as part of her thesis.

1.2 Methods

Here we explain and document in detail, the methods we implement in the `econsa` package to perform sensitivity analysis and uncertainty quantification. An insight into how the calculations are performed is not a prerequisite for using `econsa`, in most cases, the default settings works fine. Global Sensitivity Analysis can be divided into two categories: quali- and quantitative methods. `econsa` implements both methods as a comprehensive to ensure flexibility depending on your model requirements, features and specifications.

1.2.1 Qualitative sensitivity analysis

`econsa` applies the methods in [4] to calculate morris indices for models with dependent parameters. The Elementary Effects (EE), also known as the Morris method, is a qualitative way to screen inputs and helps to determine the set of influential and non-influential inputs. Shapely values on the other hand, ...

Constributor: Janos Gabler (janosg)

Calculate morris indices for models with dependent parameters.

We convert frequently between iid uniform, iid standard normal and multivariate normal variables. To not get confused, we use the following naming conventions:

-u refers to to uniform variables -z refers to standard normal variables -x refers to multivariate normal variables.

`econsa.morris.`**`elementary_effects`**(*func*, *params*, *cov*, *n_draws*, *sampling_scheme='sobol'*, *n_cores=1*)

Calculate Morris Indices of a model described by func.

The distribution of the parameters is assumed to be multivariate normal, with mean `params["value"]` and covariance matrix `cov`.

The algorithm is based on Ge and Menendez, 2017, (GM17): Extending Morris method for qualitative global sensitivity analysis of models with dependent inputs.

Parameters

- **`func`** (*`function`*) – Function that maps parameters into a quantity of interest.
- **`params`** (*`pd.DataFrame`*) – DataFrame with arbitrary index. There must be a column called value that contains the mean of the parameter distribution.
- **`cov`** (*`pd.DataFrame`*) – Both the index and the columns are the same as the index of params. The covariance matrix of the parameter distribution.
- **`n_draws`** (*`int`*) – Number of draws
- **`sampling_scheme`** (*`str`*) – One of ["sobol", "random"]. Default: "sobol"

Returns

- **mu_ind** (*float*) – Absolute mean of independent part of elementary effects
- **sigma_ind** (*float*) – Standard deviation of independent part of elementary effects

1.2.2 Quantitative sensitivity analysis

`econsa` provides several algorithms for quantitative sensitivity analysis.

Sobol indices

We implement the methods outlined in [8].

Constributor: Tim Mensinger (timmens)

Shapley values

We implement the methods outlined in [12].

Constributor: Linda Maokomatanda (lindamaok899)

Capabilities for computation of Shapley effects.

This module contains functions to estimate shapley effects for models with dependent inputs.

`econsa.shapley.get_shapley`(*method*, *model*, *x_all*, *x_cond*, *n_perms*, *n_inputs*, *n_output*, *n_outer*, *n_inner*)

Shapley value function.

This function calculates Shapley effects and their standard errors for models with both dependent and independent inputs. We allow for two ways to calculate Shapley effects: by examining all permutations of the given inputs or alternatively, by randomly sampling permutations of inputs.

The function is a translation of the exact and random permutation funtions in the `sensitivity` package in R, and takes the method (exact or random) as an argument and therefore estimates shapley effects in both ways.

The functions where obtained from R's *sensitiity* package for the shapleyPermEx and shapleyPermRand functions.

Contributor: Linda Maokomatanda

Parameters

- **`method`** (*`string`*) – Specifies which method you want to use to estimate shapley effects, the `exact` or `random` permutations method. When the number of inputs is small, it is better to use the `exact` method, and `random` otherwise.
- **`model`** (*`string`*) – The model/function you will calculate the shapley effects on.
- **`x_all`** (*`string (n)`*) – A function that takes *n* as an argument and generates a n-sample of a d-dimensional input vector.
- **`x_cond`** (*`string (n, Sj, Sjc, xjc)`*) – A function that takes *n, Sj, Sjc, xjc* as arguments and generates a n- sample an input vector corresponding to the indices in *Sj* conditional on the input values *xjc* with the index set *Sjc*.
- **`n_perms`** (*`scalar`*) – This is an input for the number of permutations you want the model to make. For the `exact` method, this argument is none as the number of permutations is determined by how many inputs you have, and for the `random` method, this is determined exogeniously.

- **n_inputs** (*scalar*) – The number of input vectors for which shapley estimates are being estimated.
- **n_output** (*scalar*) – Monte Carlo (MC) sample size to estimate the output variance of the model output *Y*.
- **n_outer** (*scalar*) – The outer Monte Carlo sample size to estimate the cost function for *c(J) = E[Var[Y|X]]*.
- **n_inner** (*scalar*) – The inner Monte Carlo sample size to estimate the cost function for *c(J) = Var[Y|X]*.

Returns **effects** – n dimensional DataFrame with the estimated shapley effects, the standard errors and the confidence intervals for the input vectors.

Return type DataFrame

Quantile based sensitivity measures

We implement the methods outlined in [7].

Constributor: Yulei Li (Yuleii)

Capabilities for quantile-based sensitivity analysis.

This module contains functions to calculate the global sensitivity measures based on quantiles of the output introduced by Kucherenko et al.(2019). Both the brute force and double loop reordering MC estimators are included.

econsa.quantile_measures.**mc_quantile_measures**(*estimator*, *func*, *n_params*, *loc*, *scale*, *dist_type*, *n_draws*, *sampling_scheme='sobol'*, *seed=0*, *skip=0*)

Compute the MC/QMC estimators of quantile-based global sensitivity measures.

The algorithm is described in Section 4 of Kucherenko et al.(2019).

Parameters

- **estimator** (*string*) – Specify the Monte Carlo estimator. One of ["brute force", "DLR"], where "DLR" denotes to the double loop reordering approach.
- **func** (*callable*) – Objective function to estimate the quantile-based measures. Must be broadcastable.
- **n_params** (*int*) – Number of parameters of objective function.
- **loc** (*float or np.ndarray*) – The location(*loc*) keyword passed to scipy.stats.norm function to shift the location of "standardized" distribution. Specifically, for normal distribution it specifies the mean array with the length of *n_params*.
- **scale** (*float or np.ndarray*) – The *scale* keyword passed to scipy.stats.norm function to adjust the scale of "standardized" distribution. Specifically, for normal distribution it specifies the covariance matrix of shape (n_params, n_params).
- **dist_type** (*str*) – The distribution type of inputs. Options are "Normal", "Exponential" and "Uniform".
- **n_draws** (*int*) – Number of Monte Carlo draws. For double loop reordering estimator, S. Kucherenko and S. Song(2017). suggests that *n_draws* should always be equal to 2^p to preserve the uniformity properties , where p is an integer. In this function p should be integers between 6 and 15 if *estimator* is "DLR".
- **sampling_scheme** (*str, optional*) – One of ["random", "sobol"], default "sobol".
- **seed** (*int, optional*) – Random number generator seed.

- **skip** (*int, optional*) – Number of values to skip of Sobol sequence. Default is *0*.

Returns **df_measures** – DataFrame containing quantile-based sensitivity measures.

Return type pd.DataFrame

1.2.3 Sampling methods

Capabilities for sampling of random input parameters.

This module contains functions to sample random input parameters.

`econsa.sampling.`**`cond_mvn`**(*mean*, *cov*, *dependent_ind*, *given_ind=None*, *given_value=None*, *check_cov=True*)

Conditional mean and variance function.

This function provides the conditional mean and variance-covariance matrix of [Y given X], where $Z = (X, Y)$ is the fully-joint multivariate normal distribution with mean equal to `mean` and covariance matrix `cov`.

This is a translation of the main function of R package condMVNorm.

Parameters

- **mean** (*array_like*) – The mean vector of the multivariate normal distribution.
- **cov** (*array_like*) – Symmetric and positive-definite covariance matrix of the multivariate normal distribution.
- **dependent_ind** (*int or array_like*) – The indices of dependent variables.
- **given_ind** (*array_like, optional*) – The indices of independent variables (default value is *None*). If not specified return unconditional values.
- **given_value** (*array_like, optional*) – The conditioning values (default value is *None*). Should be the same length as `given_ind`, otherwise throw an error.
- **check_cov** (*bool, optional*) – Check that `cov` is symmetric, and all eigenvalue is positive (default value is *True*).

Returns

- **cond_mean** (*numpy.ndarray*) – The conditional mean of dependent variables.
- **cond_cov** (*numpy.ndarray*) – The conditional covariance matrix of dependent variables.

Examples

```
>>> mean = np.array([1, 1, 1])
>>> cov = np.array([[4.0677098, -0.9620331, 0.9897267],
...                 [-0.9620331, 2.2775449, 0.7475968],
...                 [0.9897267, 0.7475968, 0.7336631]])
>>> dependent_ind = [0, ]
>>> given_ind = [1, 2]
>>> given_value = [1, -1]
>>> cond_mean, cond_cov = cond_mvn(mean, cov, dependent_ind, given_ind, given_value)
>>> np.testing.assert_almost_equal(cond_mean, -4.347531, decimal=6)
>>> np.testing.assert_almost_equal(cond_cov, 0.170718, decimal=6)
```

Conditional sampling from Gaussian copula.

This module contains functions to sample random input parameters from a Gaussian copula.

econsa.copula.**cond_gaussian_copula**(*cov*, *dependent_ind*, *given_ind*, *given_value_u*, *size=1*)
Conditional sampling from Gaussian copula.

This function provides the probability distribution of conditional sample drawn from a Gaussian copula, given covariance matrix and a uniform random vector.

Parameters

- **cov** (*array_like*) – Covariance matrix of the desired sample.
- **dependent_ind** (*int or array_like*) – The indices of dependent variables.
- **given_ind** (*array_like*) – The indices of independent variables.
- **given_value_u** (*array_like*) – The given random vector (u) that is uniformly distributed between 0 and 1.
- **size** (*int*) – Number of draws from the conditional distribution. (default value is *1*)

Returns **cond_quan** – The conditional sample ($G(u)$) that is between 0 and 1, and has the same length as dependent_ind.

Return type numpy.ndarray

Examples

```
>>> np.random.seed(123)
>>> cov = np.array([[ 3.290887,  0.465004, -3.411841],
...                 [ 0.465004,  3.962172, -0.574745],
...                 [-3.411841, -0.574745,  4.063252]])
>>> dependent_ind = 2
>>> given_ind = [0, 1]
>>> given_value_u = [0.0596779, 0.39804426]
>>> condi_value_u = cond_gaussian_copula(cov, dependent_ind, given_ind, given_value_
↪u)
>>> np.testing.assert_almost_equal(condi_value_u[0], 0.856504, decimal=6)
```

1.2.4 Utility functions

This page includes useful functions that are not categorised.

Correlation

Map arbitrary correlation matrix to Gaussian.

This module implements methods from two papers to map arbitrary correlation matrix into correlation matrix for Gaussian copulas.

econsa.correlation.**gc_correlation**(*marginals*, *corr*, *order=15*, *force_calc=False*)
Correlation for Gaussian copula.

This function implements the algorithm outlined in Section 4.2 of [K2012] to map arbitrary correlation matrix to an correlation matrix for Gaussian copula. For special combination of distributions, use the values from Table 4. of [L1986].

Since chaospy's copula functions only accept positive definite correlation matrix, this function also checks the output, and transforms to nearest positive definite matrix if it is not already.

Numerical integration is calculated with Gauss-Hermite quadrature ([D1984]).

Parameters

- **marginals** (*chaospy.distributions*) – Marginal distributions of the correlated variables. All marginals must be chaospy distributions, otherwise returns error.
- **corr** (*array_like*) – The correlation matrix to be transformed.
- **order** (*int, optional*) – The order of grids used to generate for integration. The total number of used points is calculated as $(\text{order}+1)^2$. Values larger than 20 are not recommended. (default value is *15*)
- **force_calc** (*bool, optional*) – When *True*, calculate the covariances ignoring all special combinations of marginals (default value is *False*).

Returns **gc_corr** – The transformed correlation matrix that is ready to be fed into a Gaussian copula.

Return type numpy.ndarray

References

Examples

```
>>> corr = [[1.0, 0.6, 0.3], [0.6, 1.0, 0.0], [0.3, 0.0, 1.0]]
>>> marginals = [cp.Normal(1.00), cp.Uniform(lower=-4.00), cp.Normal(4.20)]
>>> corr_transformed = gc_correlation(marginals, corr)
>>> copula = cp.Nataf(cp.J(*marginals), corr_transformed)
>>> corr_copula = np.corrcoef(copula.sample(1000000))
>>> np.testing.assert_almost_equal(corr, corr_copula, decimal=6)
```

1.3 Tutorials

We provide several tutorials that showcase the use case for `econsa`.

1.3.1 Sampling

We show how to construct correlated sample with Gaussian copula.

[1]:
```
import chaospy as cp
import numpy as np

from econsa.correlation import gc_correlation
```

First we specify the marginal distributions and correlation matrix.

[2]:
```
corr = [[1.0, 0.6, 0.2], [0.6, 1.0, 0.0], [0.2, 0.0, 1.0]]
marginals = (
    cp.Normal(mu=1230),
    cp.Normal(mu=0.0135),
    cp.Uniform(lower=1.15, upper=3.15),
)
```

We then transform the correlation matrix using equation (4.5) in Kucherenko et al. (2012).

[3]:
```
corr_transformed = gc_correlation(marginals, corr)
```

Now we are ready to use transformed correlation matrix to sample from a Gaussian copula.

[4]:
```
copula = cp.Nataf(cp.J(*marginals), corr_transformed)
corr_copula = np.corrcoef(copula.sample(100000))

np.round(corr_copula, decimals=4)
```

[4]:
```
array([[ 1.    ,  0.6018,  0.2013],
       [ 0.6018,  1.    , -0.0018],
       [ 0.2013, -0.0018,  1.    ]])
```

1.3.2 Uncertainty propagation

We show how to conduct uncertainty propagation for the **EOQ** model. We can simply import the core function from `temfpy`.

[1]:
```
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import chaospy as cp

from temfpy.uncertainty_quantification import eoq_model
from econsa.correlation import gc_correlation
```

Setup

We specify a uniform distribution centered around $\mathbf{x^0} = (M, C, S) = (1230, 0.0135, 2.15)$ and spread the support 10% above and below the center.

[2]:
```
marginals = list()
for center in [1230, 0.0135, 2.15]:
    lower, upper = 0.9 * center, 1.1 * center
    marginals.append(cp.Uniform(lower, upper))
```

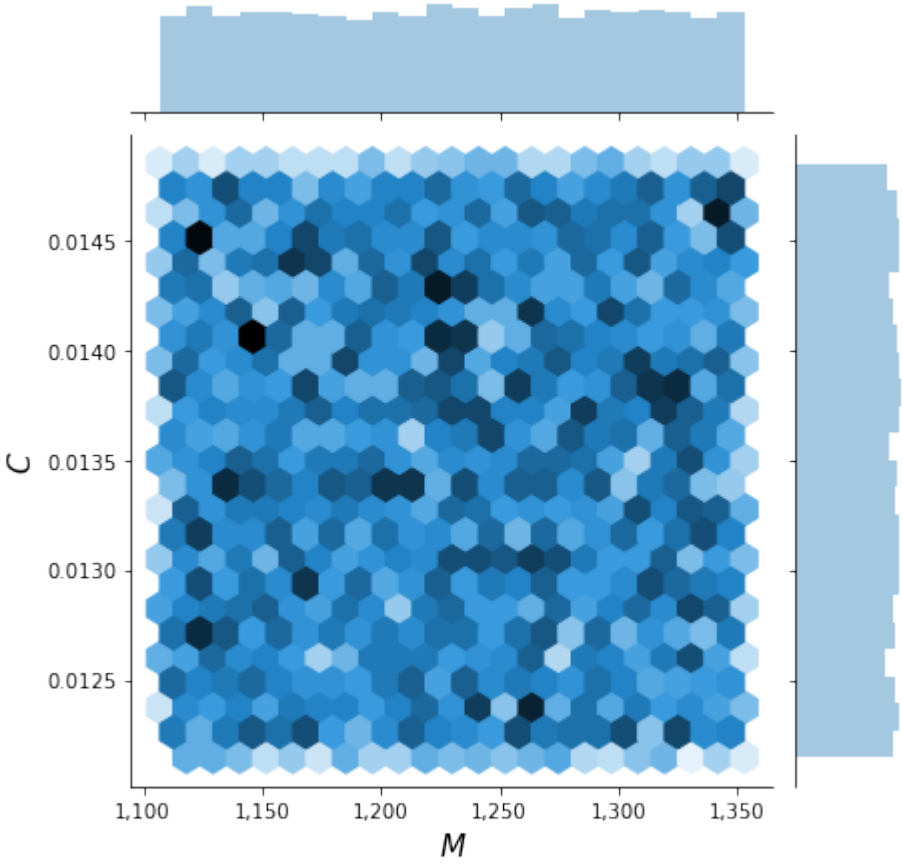
Independent parameters

We now construct a joint distribution for the the independent input parameters and draw a sample of $1,000$ random samples.

[3]:
```
distribution = cp.J(*marginals)
sample = distribution.sample(10000, rule="random")
```

The briefly inspect the joint distribution of M and C.

[5]:
```
plot_joint(sample)
```

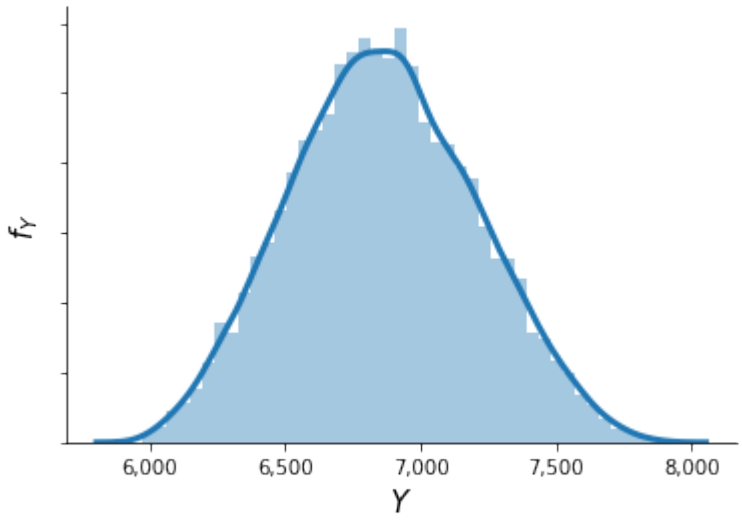


We are now ready to compute the optimal economic order quantity for each draw.

[6]:
```
y = eoq_model(sample)
```

This results in the following distribution f_Y.

[8]:
```
plot_quantity(y)
```



Depdendent paramters

We now consider dependent parameters with the following correlation matrix.

[9]:
```
corr = [[1.0, 0.6, 0.2], [0.6, 1.0, 0.0], [0.2, 0.0, 1.0]]
```

We approximate their joint distribution using a Gaussian copula. This requires us to map the correlation matrix of the parameters to the correlation matrix of the copula.

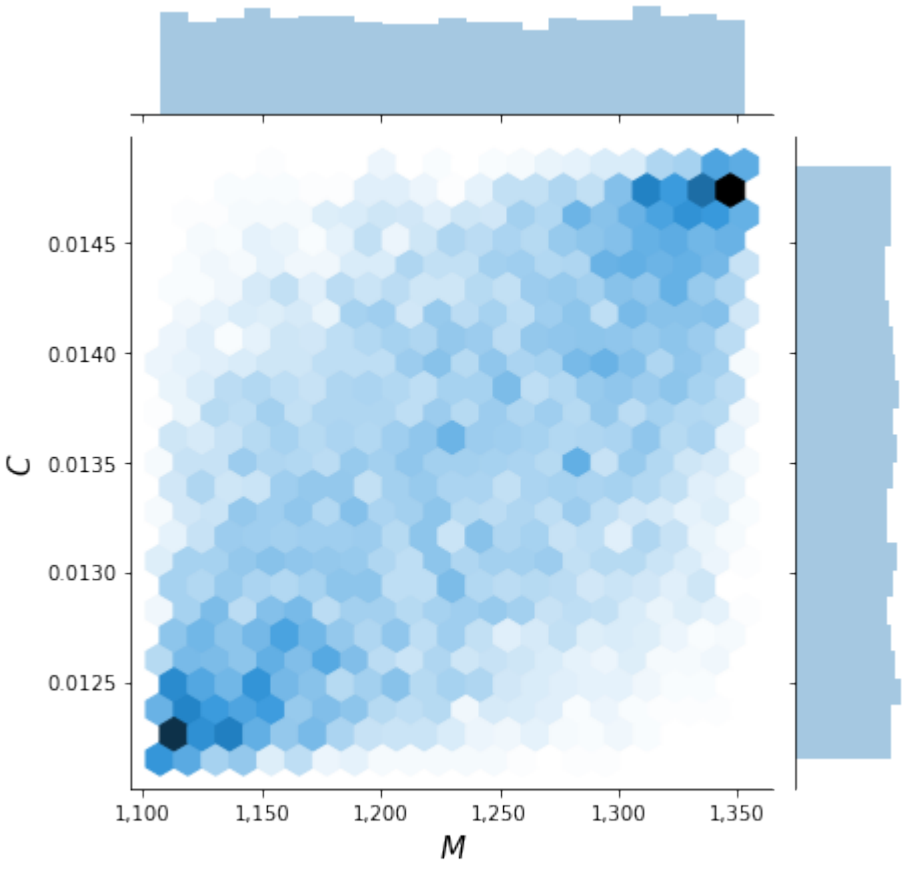
[10]:
```
corr_copula = gc_correlation(marginals, corr)
copula = cp.Nataf(distribution, corr)
```

We are ready to sample from the distribution.

[11]:
```
sample = copula.sample(10000, rule="random")
```

Again, we briefly inspect the joint distribution which now clearly shows a dependence pattern.

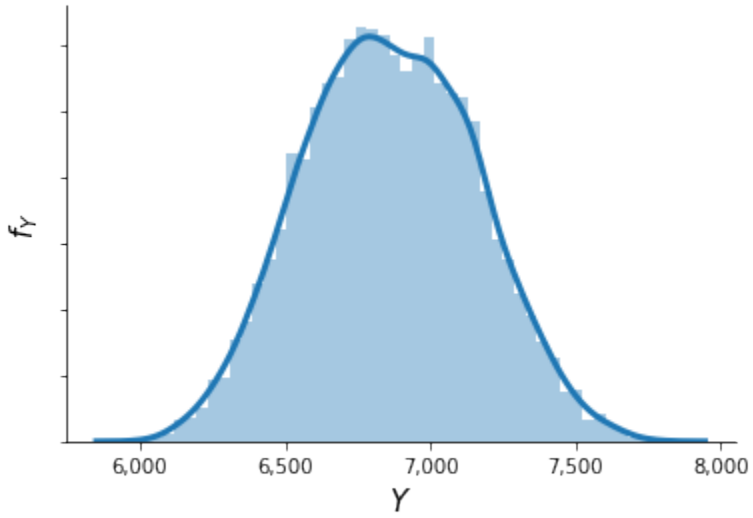
[12]:
```
plot_joint(sample)
```



[13]:
```
y = eoq_model(sample)
```

This now results in a distribution of f_Y where the peak is flattened out.

[14]:
```
plot_quantity(y)
```



1.3.3 Qualitative sensitivity analysis

Morris Method

We showcase the use of `econsa` for qualitative sensitivity analysis.

[1]:
```
from econsa.morris import elementary_effects
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

The module `morris` implements the extended Morris method as proposed by Ge & Menendez (2017). They extend the Morris method in the sense, that their algorithm takes dependency among inputs into account.

For illustration purposes consider the Morris method for independent inputs only.

Let $x = \{x_1, \ldots, x_k\}$ denote a sample of values assigned to the X_i's. $f(x)$ is then the model output obtained for the values in x. Now consider a second sample $x_{\Delta_i} = \{x_1, \ldots, x_{i-1}, x_i + \Delta, x_{i+1}, \ldots, x_k\}$ that is identical to x up to input x_i which is varied by Δ. Then, one elementary effect for input i is derived by

$$EE_i = \frac{f(x_{\Delta_i}) - f(x)}{\Delta}.$$

The above elementary effect is computed N times, each for a varying Δ. The actual sensitivity measures resulting from the Morris method are the mean, denoted by μ_i^*, and the standard deviation, denoted by σ_i, taken from the N elementary effects per input i.

$$\mu_i^* = \frac{1}{N} \sum_{r=1}^{N} |EE_{i,r}|$$

$$\sigma_i = \sqrt{\frac{1}{N-1} \sum_{r=1}^{N} (EE_{i,r} - \mu_i)^2}$$

The derivation of the extended Morris indices is more complicated and we get *four* sensitivity indices indstead of only two: *independent* and *full* Morris indices, $(\mu_i^{*,\ ind}, \mu_i^{*,\ full}, \sigma_i^{ind}, \sigma_i^{full})$, which are computed analogously to the Morris indices under input independence, but are based on different elementary effects:

- EE_i^{ind} denotes *independent* elementary effects for input i, effects that exclude the contributions attributable to the dependence between input X_i and X_j for $i \neq j$, and
- EE_i^{full} denotes *full* elementary effects for input i, that include the effects due to correlation with other inputs.

The implementation of the algorithm used in `econsa` uses the *radial design* and the *inverse Nataf transformation* as described in Ge & Menendez (2017).

For applying the Morris method, we need to specify the following arguments:

- `func`: The model for which we want to calculate the Morris indices. Note how the data needs to be accessed within the function. See below example.
- `params`: The mean values of the inputs.
- `cov`: The variance-covariance matrix of the inputs.
- `n_draws`: Number of draws, which corresponds to N above.

Additional arguments are optional.

Note that the current implementation of the Morris method in `econsa` does allow for Gaussian (i.e. normally distributed) inputs only.

The `func` argument

`func` is the implementation of the model we want to conduct sensitivity analysis for. The Morris method can be applied to all models that return a unique value for a given set of realisations of the model inputs.

The model implemented by `func` needs to access the inputs in the following way, if the input names are specified in `params` and `cov`:

```
m = x["value"]["m"]
c = x["value"]["c"]
s = x["value"]["s"].
```

Alternatively we can access them via the index as well:

```
m = x["value"][0]
c = x["value"][1]
s = x["value"][2].
```

[2]:
```
def eoq_model_morris(x, r=0.1):
    """EOQ Model that accesses data as expected by elementary_effects."""
    m = x["value"]["m"]
    c = x["value"]["c"]
    s = x["value"]["s"]

    # Need to ensure that there exists a solution (i.e. no NaNs).
    if m < 0:
        m = 0
    elif c < 0:
        raise ValueError
```

(continues on next page)

(continued from previous page)

```
    elif s < 0:
        s = 0
    else:
        pass

    return np.sqrt((24 * m * s) / (r * c))
```

The params and cov arguments

Specify the input names in the data frames `params` and `cov` to display the input names in the output of `elementary_effects`. `params` is a vector of means of the normally distributed model inputs. `params` needs to be a `pandas.DataFrame` with a colum called `"value"`, which contains the means of the inputs.

`cov` is the corresponding variance-covariance matrix. The variance-covariance matrix describes the dependence structure of the inputs. As `params`, `cov` needs to be a `pandas.DataFrame`. Indices need to be the same as in `params`.

[3]:
```
names = ['m', 'c', 's']
params = pd.DataFrame(data=np.array([5.345, 0.0135, 2.15]), columns=['value'],␣
↪index=names)
cov = pd.DataFrame(data=np.diag([1, 0.000001, 0.01]), columns=names, index=names)
```

The n_draws argument

`n_draws` is the number of elementary effects we want to use for the computation of the Morris indices. The total computational cost of the extended Morris method amounts to $3kN$, where k denotes the number of inputs and N the number of draws (`n_draws`).

[4]:
```
n_draws = 100
```

The sampling_scheme and seed arguments

By specifying `sampling_scheme` we can choose how uniformly distributed samples are drawn. The uniformly distributed samples are then transfomed to dependently and normally distributed samples. "sobol" is used to sample from a low-discrepancy seuqence which generates more evenly distributed samples. When using "random", we get pseudo-random samples. `seed` denotes the corresponding seed when generating random numbers. The default is that `sampling_scheme` = "sobol" and `seed` = 1.

The n_cores argument

Parallelising code is done by the Python built-in `multiprocessing` module, where `n_cores` is the number of cores employed. The default is that that `n_cores` is set to 1.

[5]:
```
results = elementary_effects(eoq_model_morris, params, cov, n_draws)
```

The output

The output of `elementary_effects` is a dictionary containing the four sensitivity indices derived from the `n_draws` elementary effects: $(\mu_i^{*,\,ind}, \mu_i^{*,\,full}, \sigma_i^{ind}, \sigma_i^{full})$. The Morris indices are accessed as shown below.

Independent Morris indices $(\mu_i^{*,\,ind}, \sigma_i^{ind})$

[6]:
```
morris_ind = pd.DataFrame(pd.concat((results['mu_ind'], results['sigma_ind']), axis=1))
morris_ind.columns = ['mu', 'sigma']
morris_ind
```

[6]:
```
           mu      sigma
m  152.932694  54.844400
c   57.953365  17.314969
s   33.104776   7.963128
```

Full Morris indices $(\mu_i^{*,\,full}, \sigma_i^{full})$

[7]:
```
morris_full = pd.DataFrame(pd.concat((results['mu_corr'], results['sigma_corr']),␣
↪axis=1))
morris_full.columns = ['mu', 'sigma']
morris_full
```

[7]:
```
            mu        sigma
m  2369.122666  2523.492389
c  3270.066313  8210.375153
s  2888.326540  3814.720109
```

Plotting the results

The input ranking is conducted based on $(\mu_i^{*,\,ind}, \mu_i^{*,\,full})$.

[8]:
```
def plot_morris_indices(morris_full, morris_ind):
    fig, ax = plt.subplots(2, 1)
    sns.set_style("whitegrid")

    sns.scatterplot(x=morris_full['mu'], y=morris_full['sigma'], data=morris_full,␣
↪ax=ax[0])
    sns.scatterplot(x=morris_ind['mu'], y=morris_ind['sigma'], data=morris_full,␣
↪ax=ax[1])

    ax[0].set_title('Full Morris indices')

    ax[0].text(x=morris_full['mu'].iloc[0] + 20, y=morris_full['sigma'].iloc[0], s='m')
    ax[0].text(x=morris_full['mu'].iloc[1] + 20, y=morris_full['sigma'].iloc[1], s='c')
    ax[0].text(x=morris_full['mu'].iloc[2] + 20, y=morris_full['sigma'].iloc[2], s='s')

    ax[1].set_title('Independent Morris indices')
```

(continues on next page)

(continued from previous page)

```
    ax[1].text(x=morris_ind['mu'].iloc[0] + 2, y=morris_ind['sigma'].iloc[0], s='m')
    ax[1].text(x=morris_ind['mu'].iloc[1] + 2, y=morris_ind['sigma'].iloc[1], s='c')
    ax[1].text(x=morris_ind['mu'].iloc[2] + 2, y=morris_ind['sigma'].iloc[2], s='s')

    plt.tight_layout()
    plt.show()
```

[9]:
```
plot_morris_indices(morris_full, morris_ind)
```

Interpretation

The input ranking based on $(\mu_i^{*,\,ind}, \mu_i^{*,\,full})$ differs when independent or full indices are considered.

The ranking in ascending order according to *full* indices is $c - s - m$, whereas the ranking based on *independent* inidces is $m - c - s$. For input m this means that the variance contribution due to the isolated effect of m is much larger than the contribution due to dependence with other inputs. Inputs c and s, though, exhibit large effects due to dependence.

Since none of the inputs is close to zero, we can conclude that all three inputs are important in terms of their output variance contribution.

[ ]:
```
```

1.3.4 Quantitative sensitivity analysis

Generalized Sobol Indices

Here we show how to compute generalized Sobol indices on the **EOQ** model using the algorithm presented in Kucherenko et al. 2012. We import our model function from `temfpy` and use the Kucherenko indices function from `econsa`.

[1]:
```
import matplotlib.pyplot as plt  # noqa: F401
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from temfpy.uncertainty_quantification import eoq_model

# TODO: Reactivate once Tim's PR is ready.
# from econsa.kucherenko import kucherenko_indices  # noqa: E265
```

The function `kucherenko_indices` expects the input function to be broadcastable over rows, that is, a row represents the input arguments for one evaluation. For sampling around the mean parameters we specify a diagonal covariance matrix, where the variances depend on the scaling of the mean. Since the variances of the parameters are unknown prior to our analysis we choose values such that the probability of sampling negative values is negligible. We do this since the **EOQ** model is not defined for negative parameters and the normal sampling does not naturally account for bounds.

[2]:
```
def eoq_model_transposed(x):
    """EOQ Model but with variables stored in columns."""
    return eoq_model(x.T)

mean = np.array([1230, 0.0135, 2.15])
cov = np.diag([1, 0.000001, 0.01])

# indices = kucherenko_indices( # noqa: E265
#     func=eoq_model_transposed, # noqa: E265
#     sampling_mean=mean,  # noqa: E265
#     sampling_cov=cov,  # noqa: E265
#     n_draws=1_000_000,  # noqa: E265
#     sampling_scheme="sobol",  # noqa: E265
# )  # noqa: E265
```

Now we are ready to inspect the results.

[4]:
```
# fig  # noqa: E265
```

Shapley Effects

`econsa` offers an implementation of the algorithm for the computation of Shapley effects as propsoed by Song et al. (2016). Here we show how to compute Shapley effects using the **EOQ** model as referenced above. We adjust the model in `temfpy` to accomodate an n-dimensional array for use in the context of the Shapley effects as implemented in `econsa`.

[6]:
```
# import necessary packages and functions
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import chaospy as cp

from econsa.shapley import get_shapley
from econsa.shapley import _r_condmvn
```

Sampling via `x_all` and `x_cond`, and the model of interest `model`

First, we load all neccesary objects for the estimation of Shapley effects. The following objects are needed in the case of Gaussian model inputs.

- The functions `x_all` and `x_cond` for (un-)conditional sampling. These functions depend on the distribution from which we are sampling from. For the purposes of this illustration, we will sample from a multivariate normal distribution, but the functions can be tailored to the user's specific needs.
- A mean vector and covariance matrix of the model inputs. They are necessary for sampling conducted by the above two functions in the case of a Gaussian distribution.
- The `model` the user wishes to perform sensitivity analysis (SA) on that maps model inputs to a model output. Here we consider the **EOQ** model.

[7]:
```
# Mean vector and covariance matrix of the model inputs.
n_inputs = 3
mean = np.array([5.345, 0.0135, 2.15])
cov = np.diag([1, 0.000001, 0.01])
```

Choosing `n_perms`

Since we are conducting SA on a model with three inputs, the number of permutations on which the computation algorithm is based is $3! = 6$. For larger number of inputs it might be worthwhile to consider only a subset of all permutations. E.g. for a model with 10 inputs, there are 3,628,800 permutations. Considering all permutations could be computationally infeasible. Thus, `get_shapley` allows the user to set a specific number of permutations by the argument `n_perms`.

Choosing the number of Monte Carlo (MC) runs `n_output`, `n_outer`, and `n_inner`

N_V, N_O, and N_I denote the function arguments `n_output`, `n_outer`, and `n_inner`, respectively. For the algorithm by Song et al. (2016) these three MC simulations are needed. The number of model evaluations required for the estimation of Shapley effects by `get_shapley` are given by

$$N_V + m \cdot N_I \cdot N_O \cdot (k - 1),$$

where m denotes the number of permutations, `n_perms`, and k the number of inputs, `n_inputs`.

Song et al. (2016) show that choosing $N_I = 3$ is optimal. N_V needs to be large enough to reliably estimate the total output variance, $V[Y]$. Given these choices, N_O is chosen to consume the rest of the computational budget.

[8]:
```
# Model for which senstivity analysis is being performed.
def eoq_model_ndarray(x, r=0.1):
    """EOQ Model that accepts ndarray."""
    m = x[:, 0]
    c = x[:, 1]
    s = x[:, 2]
    return np.sqrt((24 * m * s) / (r * c))
```

[9]:
```
# Function for unconditional sampling.
def x_all(n):
    distribution = cp.MvNormal(mean, cov)
    return distribution.sample(n)
```

(continues on next page)

(continued from previous page)

```
# Function for conditional sampling in the case of Gaussian inputs.
def x_cond(n, subset_j, subsetj_conditional, xjc):
    if subsetj_conditional is None:
        cov_int = np.array(cov)
        cov_int = cov_int.take(subset_j, axis = 1)
        cov_int = cov_int[subset_j]
        distribution = cp.MvNormal(mean[subset_j], cov_int)
        return distribution.sample(n)
    else:
        return _r_condmvn(n, mean=mean, cov=cov, dependent_ind=subset_j, given_
↪ind=subsetj_conditional, x_given=xjc)
```

[11]:
```
# Estimate Shapley effects using the exact method.
np.random.seed(1234)
method = "exact"
n_perms = None
n_output = 10 ** 4
n_outer = 10 ** 3
n_inner = 3

exact_shapley = get_shapley(method, eoq_model_ndarray, x_all, x_cond, n_perms, n_inputs,␣
↪n_output, n_outer, n_inner)
```

[12]:
```
exact_shapley
```

[12]:

	Shapley effects	std. errors	CI_min	CI_max
X1	0.828423	0.008709	0.811353	0.845493
X2	0.130322	0.004717	0.121077	0.139568
X3	0.041255	0.012115	0.017509	0.065000

[13]:
```
# Estimate Shapley effects using the random method.
np.random.seed(1234)
method = "random"
n_perms = 5
n_output = 10 ** 4
n_outer = 10 ** 3
n_inner = 3

random_shapley = get_shapley(method, eoq_model_ndarray, x_all, x_cond, n_perms, n_inputs,
↪ n_output, n_outer, n_inner)
```

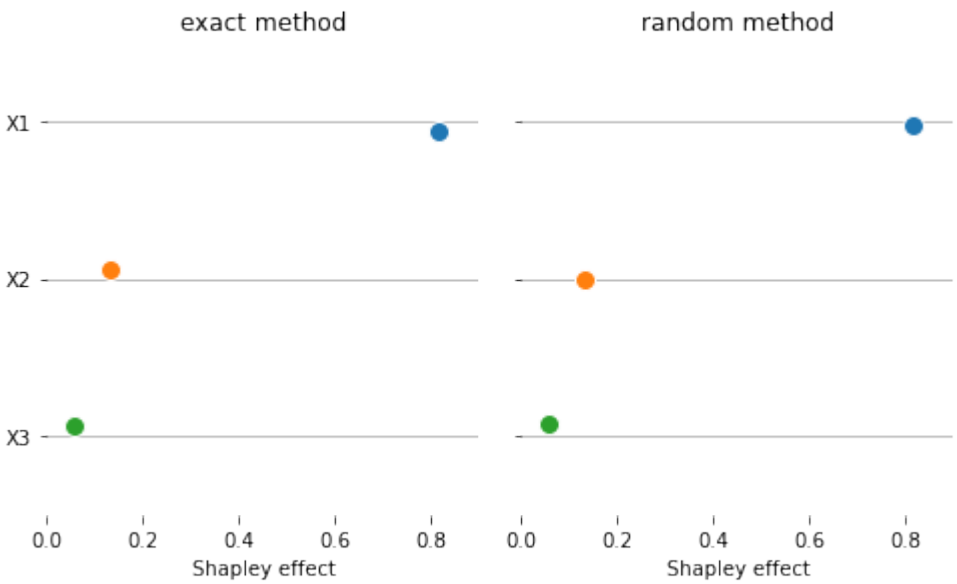
[15]:
```
random_shapley
```

[15]:

	Shapley effects	std. errors	CI_min	CI_max
X1	0.817302	0.015392	0.787134	0.847470
X2	0.129110	0.003608	0.122037	0.136182
X3	0.053588	0.013530	0.027069	0.080108

Now we plot the ranking of the Shapley values below.

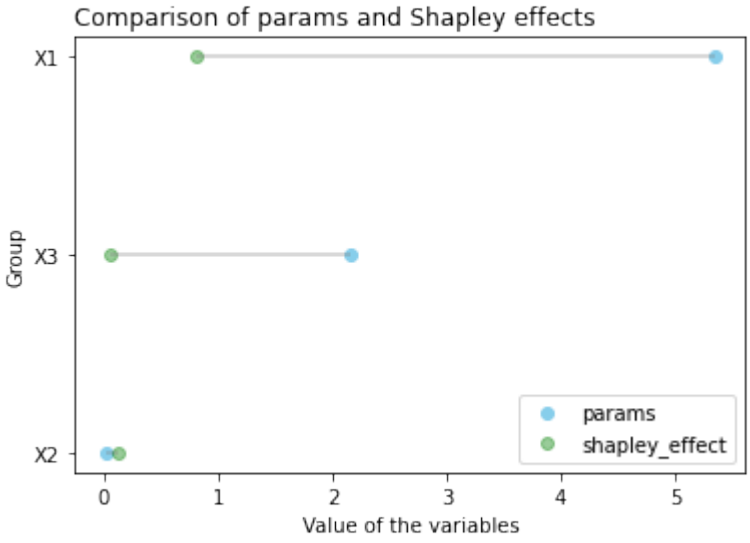
[17]:
```
# plot for exact and random permutations methods
ranking(data = df)
```



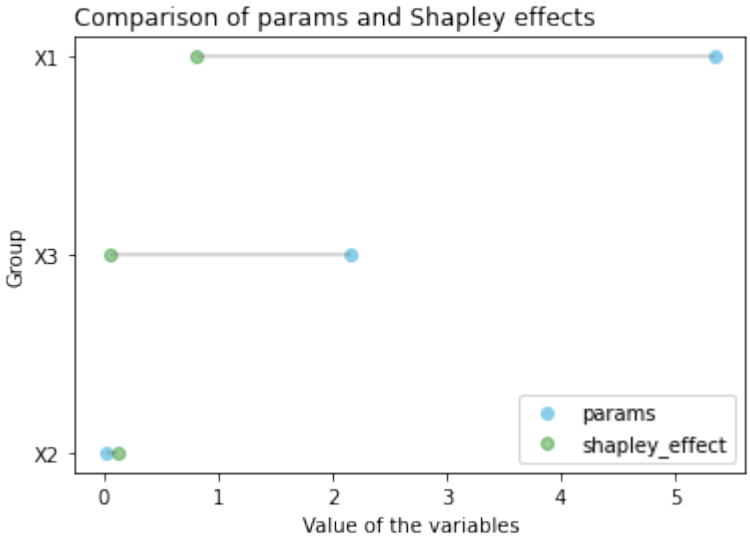
As noticed above, both methods produce the same ranking. Sometimes, it is neccesary to compare the parameter estimates with their parameter values. A typical application is hypothesis testing, that is, whether the parameter estimates are significant, and what the contribution of significant / insignificant estimates is to the output variance as reflected by their Shapley ranking.

We can plot the parameter estimates together with their Shapley ranking as shown below:

[19]:
```
# plot exact method comparison
ranking_params_shapley(ordered_df)
```



[21]:
```
# plot random method comparison
ranking_params_shapley(ordered_df)
```



When do we use randomly sampled permutations? Choosing `method`

The `exact` method is good for use when the number of parameters is low, depending on the computational time it takes to estimate the model in question. If it is computationally inexpensive to estimate the model for which sensitivity analysis is required, then the `exact` method is always preferable, otherwise the `random` is recommended. A good way to proceed if one suspects that the computational time required to estimate the model is high, having a lot of parameters to conduct SA on is always to commence the exercise with a small number of parameters, e.g. 3, then get a benchmark of the Shapley effects using the `exact` method. Having done that, repeat the exercise using the `random` method on the same vector of parameters, calibrating the `n_perms` argument to make sure that the results produced by the `random` method are the same as the `exact` one. Once this is complete, scale up the exercise using the `random` method, increasing the number of parameters to the desired parameter vector.

Quantile Based Sensitivity Measures

We show how to compute global sensitivity measures based on quantiles of model's output.

[19]:
```
# import necessary packages and functions
import numpy as np
import pandas as pd
import chaospy as cp
import matplotlib.pyplot as plt
import seaborn as sns

from temfpy.uncertainty_quantification import eoq_model
from econsa.quantile_measures import mc_quantile_measures
```

Firstly, we specify the parameters of the function. This function `mc_quantile_measures` is capable of computing numerical results of quantile-based sensitivity measures on various user-provided models. Here we take **EOQ model** from `temfpy` as an example, where the model function is adjusted to accommodate an n-dimensional array. For multivariate distributed samples, the `loc` and `scale` keywords are denoted by a mean vector and a covariance matrix respectively. Considering both efficient computation and good convergence, we set `n_draws` equal to 3000 and 2^{13} for brute force and double loop reordering estimators correspondently. Note that the double loop reordering estimator is more efficient than the brute force estimator.

[20]:
```
# model to perform quantile based sensitivity ananlysis
def eoq_model_transposed(x):
    """EOQ Model but with variables stored in columns."""
    return eoq_model(x.T)
```

[21]:
```
# mean and covaraince matrix inputs
mean = np.array([5.345, 0.0135, 2.15])
cov = np.diag([1, 0.000001, 0.01])
n_params = len(mean)
dist_type = "Normal"
```

Then we are ready to calculate the numerical results using the algorithm presented in Kucherenko et al. 2019.

[22]:
```
# compute quantile measures using brute force estimator
bf_measures = mc_quantile_measures("brute force", eoq_model_transposed, n_params, mean,␣
↪cov, dist_type, 3000,)
```

[23]:
```
bf_measures
```

[23]:

		x_1	x_2	x_3
Measures	alpha			
q_1	0.020	64.010462	10.595192	6.522037
	0.052	53.565287	11.764802	7.126140
	0.084	47.203635	11.849184	7.315335
	0.116	43.584702	11.945758	7.390247
	0.148	40.746792	12.074625	7.493912
...		...	...	...
Q_2	0.852	0.851180	0.108055	0.040765
	0.884	0.857788	0.102897	0.039315
	0.916	0.863797	0.099314	0.036889
	0.948	0.871352	0.093822	0.034825
	0.980	0.880632	0.089250	0.030119

[124 rows x 3 columns]

[24]:
```
# compute quantile measures using double loop reordering estimator
dlr_measures = mc_quantile_measures("DLR", eoq_model_transposed, n_params, mean, cov,␣
↪dist_type, 2 ** 13,)
```

[25]:
```
dlr_measures
```

[25]:

		x_1	x_2	x_3
Measures	alpha			
q_1	0.020	64.179809	10.414823	6.285291
	0.052	52.267399	11.037341	6.778817
	0.084	46.311146	11.211302	6.917412
	0.116	42.563103	11.407816	7.013578
	0.148	40.321098	11.638289	7.183969
...		...	...	...
Q_2	0.852	0.865417	0.097312	0.037271
	0.884	0.871006	0.093422	0.035572
	0.916	0.876810	0.089801	0.033388
	0.948	0.883911	0.084970	0.031119

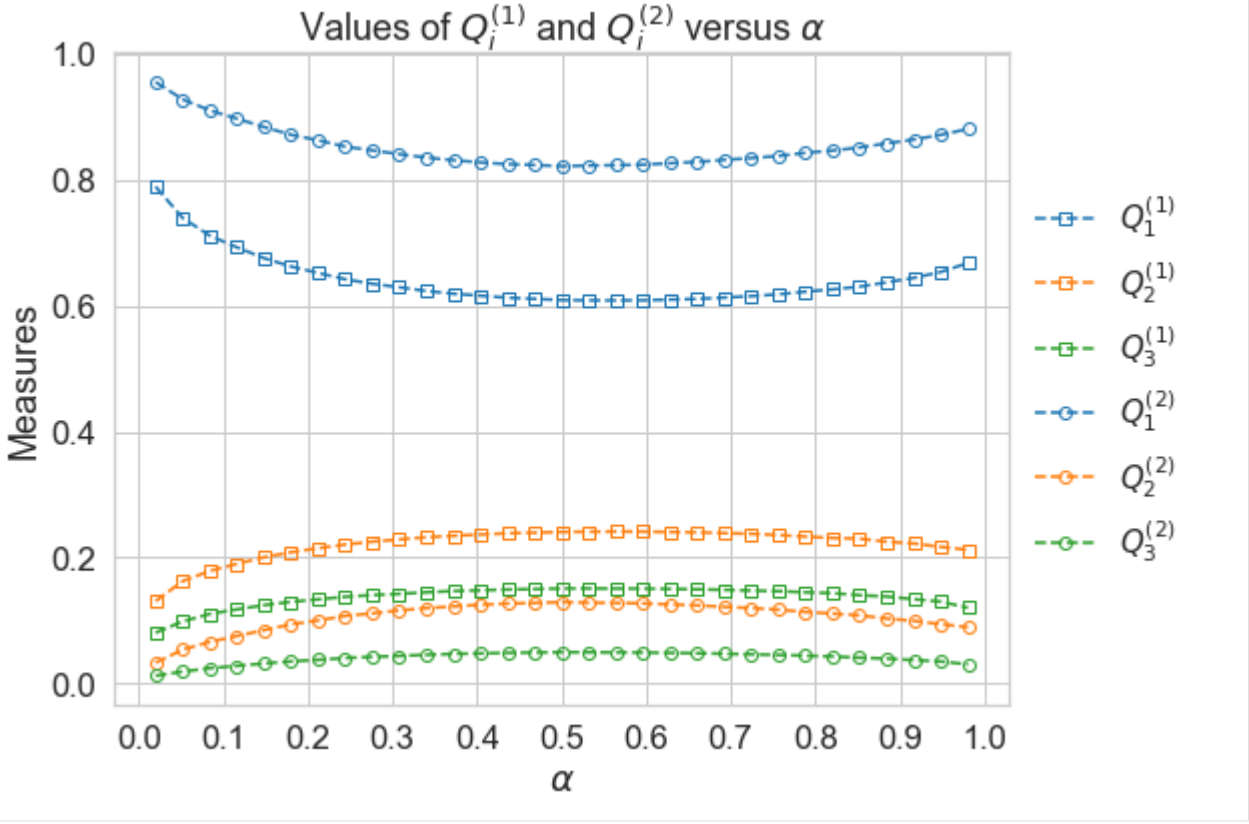
(continues on next page)

(continued from previous page)

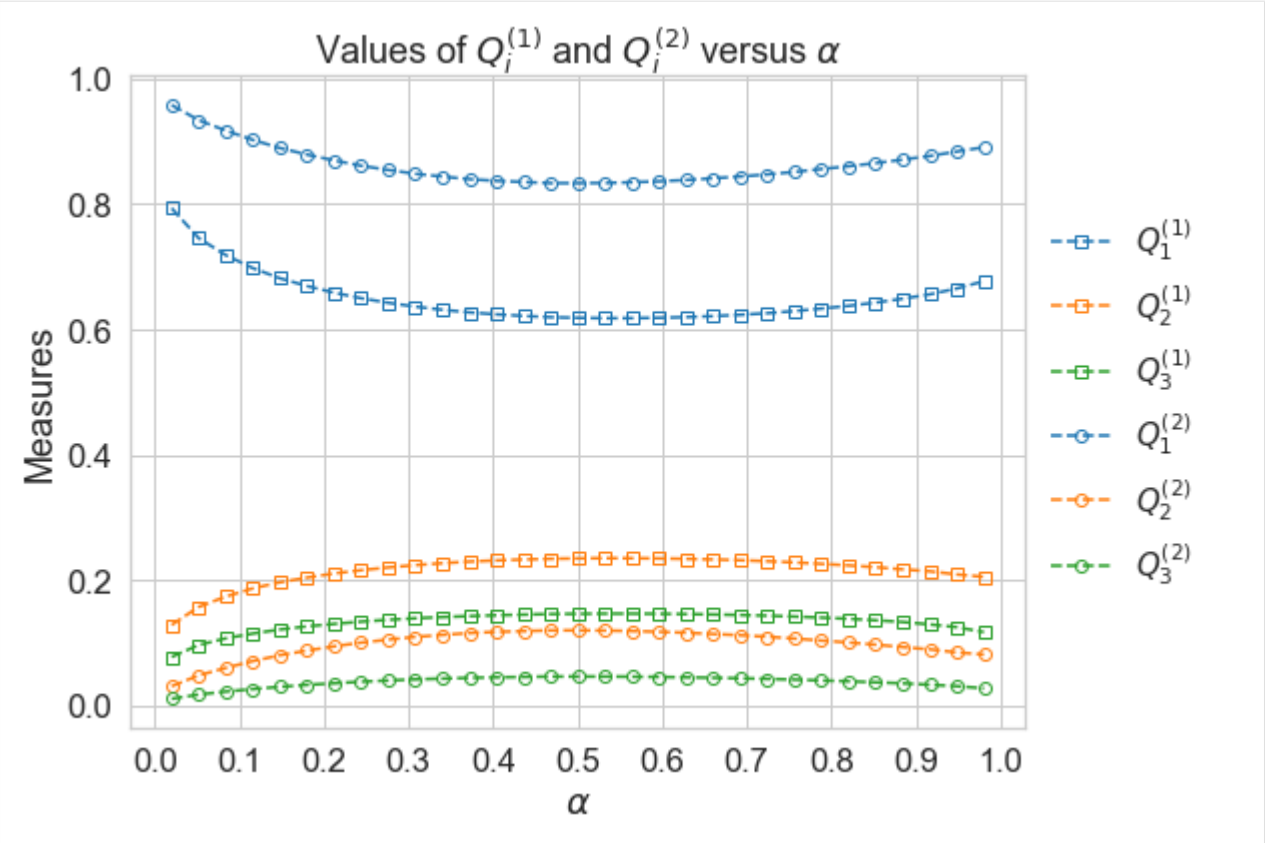
```
         0.980   0.890728   0.081605  0.027667

[124 rows x 3 columns]
```

Now we are able to visualize the results.

[27]:
```
# plot brute force estimates
plot_quantile_measures(bf_measures)
```



[28]:
```
# plot double loop reordering estimates
plot_quantile_measures(dlr_measures)
```



The brute force estimator and DLR estimator generate the same ranking of variables for all quantiles: x_1, x_2, x_3(in descending order). At $\alpha = 0.5$, measures Q_i reachs their minimun:$i = 1$ and maximum:$i = 2, 3$.

1.4 Published versions

1.5 Acknowledgements

econsa is developed and maintained as part of the OpenSourceEconomics initiative.

Project Manager

- Philipp Eisenhauer (peisenha)

Developers

- Linda Maokomatanda (lindamaok899)
- Janos Gabler (janosg)
- Tim Mensinger (timmens)
- Leiqiong Wan (loikein)
- Yulei Li (Yuleii)
- Benedikt Müller(bhmueller)

1.6 Related work

We are drawing on related work throughout.

1.6.1 Software

- Feinberg, J., and Langtangen, H. P. (2015). *Chaospy: An open source tool for designing methods of uncertainty quantification*. *Journal of Computational Science, 11*, 46-57.
- Herman, J., and Usher, W. (2017). *SALib: An open-source Python library for sensitivity analysis*. *Journal of Open Source Software, 2* (9).
- Tennoe S., Halnes G., and Einevoll G.T. (2018). *Uncertainpy: A Python toolbox for uncertainty quantification and sensitivity analysis in computational neuroscience*. *Frontiers in Neuroinformatics, 12*, 49.

1.6.2 Books

- Ghanem, R., Higdon, D., and Owhadi, H. (2017). *Handbook of uncertainty quantification*. Cham, Switzerland: Springer International Publishing.
- Saltelli et al. (2008). *Global sensitivity analysis: The primer*. Chichester, UK: John Wiley & Sons Ltd.
- Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M. (2004). *Sensitivity analysis in practice: A guide to assessing scientific models*. Chichester, UK: John Wiley & Sons Ltd.
- Smith, R.C. (2014). *Uncertainty quantification: Theory, implementation, and applications*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
- Sullivan, T.J. (2015). *Introduction to uncertainty quantification*. Cham, Switzerland: Springer International Publishing.

1.6.3 Popular science

- King, M., and Kay, J. (2020). *Radical uncertainty: Decision-making for an unknowable future*. London, UK: The Bridge Street Press.

1.7 Bibliography

BIBLIOGRAPHY

[K2012] Kucherenko, S., Tarantola, S., & Annoni, P. (2012). Estimation of global sensitivity indices for models with dependent variables. Computer Physics Communications, 183(4), 937–946.

[L1986] Liu, P., & Der Kiureghian, A. (1986). Multivariate distribution models with prescribed marginals and covariances. Probabilistic Engineering Mechanics, 1(2), 105–112.

[D1984] Davis, P. J., & Rabinowitz, P. (1984). Methods of numerical integration (2nd ed.). Academic Press.

[1] Marvin L. Adams, editor. *Assessing the reliability of complex models: Mathematical and statistical foundations of verification, validation, and uncertainty quantification*. National Academies Press, Washington, D.C., 2012.

[2] Emanuele Borgonovo and Elmar Plischke. Sensitivity analysis: A review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016.

[3] Fabio Canova. Statistical inference in calibrated models. *Journal of Applied Econometrics*, 9(1):123–144, 1994.

[4] Qiao Ge and Monica Menendez. An efficient sensitivity analysis approach for computationally expensive microscopic traffic simulation models. *International Journal of Transportation*, 2(2):49–64, 2014.

[5] Lars Peter Hansen and James J. Heckman. The empirical foundations of calibration. *Journal of economic perspectives*, 10(1):87–104, 1996.

[6] Ford W. Harris. How many parts to make at once. *Operations Research*, 38(6):947–950, 1990.

[7] Sergei Kucherenko, Shufang Song, and Lu Wang. Quantile based global sensitivity measures. *Reliability Engineering & System Safety*, 185:35–48, 2019.

[8] Sergei Kucherenko, Stefano Tarantola, and Paola Annoni. Estimation of global sensitivity indices for models with dependent variables. *Computer Physics Communications*, 183(4):937–946, 2012.

[9] Finn E. Kydland. On the econometrics of world business cycles. *European Economic Review*, 36(2-3):476–482, 1992.

[10] Charles F. Manski. Communicating uncertainty in policy analysis. *Proceedings of the National Academy of Sciences*, 116(16):7634–7641, 2019.

[11] William L. Oberkampf and Christopher J. Roy. *Verification and validation in scientific computing*. Cambridge University Press, Cambridge, UK, 2010.

[12] Art B. Owen. Sobol' indices and shapley value. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):245–251, 2014.

[13] I. Sobol. On sensitivity estimation for nonlinear mathematical models. *Math. Modelling & Comp. Exp*, 1993.

[14] Eunhye Song, Barry L Nelson, and Jeremy Staum. Shapley effects for global sensitivity analysis: theory and computation. *SIAM/ASA Journal on Uncertainty Quantification*, 4(1):1060–1083, 2016.

[15] Yunpeng Sun, Daniel W Apley, and Jeremy Staum. Efficient nested simulation for estimating the variance of a conditional expectation. *Operations research*, 59(4):998–1007, 2011.

[16] Kenneth I. Wolpin. *The limits to inference without theory*. MIT University Press, Cambridge, MA, 2013.

PYTHON MODULE INDEX

e

INDEX