SALib Documentation

Release 1.4.5.post1.dev26+gb20ee38.d20210925

Jon Herman, Will Usher and others

Sep 25, 2021

CONTENTS

Python implementations of commonly used sensitivity analysis methods, including Sobol, Morris, and FAST methods. Useful in systems modeling to calculate the effects of model inputs or exogenous factors on outputs of interest.

CHAPTER

ONE

SUPPORTED METHODS

- Sobol Sensitivity Analysis ([Sobol 2001], [Saltelli 2002], [Saltelli et al. 2010])
- Method of Morris, including groups and optimal trajectories ([Morris 1991], [Campolongo et al. 2007])
- Fourier Amplitude Sensitivity Test (FAST) ([Cukier et al. 1973], [Saltelli et al. 1999])
- Random Balance Designs - Fourier Amplitude Sensitivity Test (RBD-FAST) ([Tarantola et al. 2006], [Elmar Plischke 2010], [Tissot et al. 2012])
- Delta Moment-Independent Measure ([Borgonovo 2007], [Plischke et al. 2013])
- Derivative-based Global Sensitivity Measure (DGSM) ([Sobol and Kucherenko 2009])
- Fractional Factorial Sensitivity Analysis ([Saltelli et al. 2008])
- High Dimensional Model Representation ([Li et al. 2010])

1.1 Getting Started

1.1.1 Installing SALib

To install the latest stable version of SALib using pip, together with all the dependencies, run the following command:

```
pip install SALib
```

To install the latest development version of SALib, run the following commands. Note that the development version may be unstable and include bugs. We encourage users use the latest stable version.

```
git clone https://github.com/SALib/SALib.git
cd SALib
python setup.py develop
```

1.1.2 Installing Prerequisite Software

SALib requires NumPy, SciPy, and matplotlib installed on your computer. Using pip, these libraries can be installed with the following command:

```
pip install numpy
pip install scipy
pip install matplotlib
```

The packages are normally included with most Python bundles, such as Anaconda and Canopy. In any case, they are installed automatically when using pip or setuptools to install SALib.

1.1.3 Testing Installation

To test your installation of SALib, run the following command

```
pytest
```

Alternatively, if you'd like also like a taste of what SALib provides, start a new interactive Python session and copy/paste the code below.

```
from SALib.sample import saltelli
from SALib.analyze import sobol
from SALib.test_functions import Ishigami
import numpy as np

# Define the model inputs
problem = {
    'num_vars': 3,
    'names': ['x1', 'x2', 'x3'],
    'bounds': [[-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359]]
}

# Generate samples
param_values = saltelli.sample(problem, 1024)

# Run model (example)
Y = Ishigami.evaluate(param_values)

# Perform analysis
Si = sobol.analyze(problem, Y, print_to_console=True)

# Print the first-order sensitivity indices
print(Si['S1'])
```

If installed correctly, the last line above will print three values, similar to `[ 0.30644324 0.44776661 -0.00104936]`.

1.2 Basics

1.2.1 What is Sensitivity Analysis?

According to Wikipedia, sensitivity analysis is "the study of how the uncertainty in the output of a mathematical model or system (numerical or otherwise) can be apportioned to different sources of uncertainty in its inputs." The sensitivity of each input is often represented by a numeric value, called the *sensitivity index*. Sensitivity indices come in several forms:

1. First-order indices: measures the contribution to the output variance by a single model input alone.

2. Second-order indices: measures the contribution to the output variance caused by the interaction of two model inputs.
3. Total-order index: measures the contribution to the output variance caused by a model input, including both its first-order effects (the input varying alone) and all higher-order interactions.

1.2.2 What is SALib?

SALib is an open source library written in Python for performing sensitivity analyses. SALib provides a decoupled workflow, meaning it does not directly interface with the mathematical or computational model. Instead, SALib is responsible for generating the model inputs, using one of the `sample` functions, and computing the sensitivity indices from the model outputs, using one of the `analyze` functions. A typical sensitivity analysis using SALib follows four steps:

1. Determine the model inputs (parameters) and their sample range.
2. Run the `sample` function to generate the model inputs.
3. Evaluate the model using the generated inputs, saving the model outputs.
4. Run the `analyze` function on the outputs to compute the sensitivity indices.

SALib provides several sensitivity analysis methods, such as Sobol, Morris, and FAST. There are many factors that determine which method is appropriate for a specific application, which we will discuss later. However, for now, just remember that regardless of which method you choose, you need to use only two functions: `sample` and `analyze`. To demonstrate the use of SALib, we will walk you through a simple example.

1.2.3 An Example

In this example, we will perform a Sobol' sensitivity analysis of the Ishigami function, shown below. The Ishigami function is commonly used to test uncertainty and sensitivity analysis methods because it exhibits strong nonlinearity and nonmonotonicity.

$$f(x) = sin(x_1) + asin^2(x_2) + bx_3^4 sin(x_1)$$

Importing SALib

The first step is the import the necessary libraries. In SALib, the `sample` and `analyze` functions are stored in separate Python modules. For example, below we import the `saltelli` sample function and the `sobol` analyze function. We also import the Ishigami function, which is provided as a test function within SALib. Lastly, we import `numpy`, as it is used by SALib to store the model inputs and outputs in a matrix.

```
from SALib.sample import saltelli
from SALib.analyze import sobol
from SALib.test_functions import Ishigami
import numpy as np
```

Defining the Model Inputs

Next, we must define the model inputs. The Ishigami function has three inputs, x_1, x_2, x_3 where $x_i \in [-\pi, \pi]$. In SALib, we define a `dict` defining the number of inputs, the names of the inputs, and the bounds on each input, as shown below.

```
problem = {
    'num_vars': 3,
    'names': ['x1', 'x2', 'x3'],
    'bounds': [[-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359]]
}
```

Generate Samples

Next, we generate the samples. Since we are performing a Sobol' sensitivity analysis, we need to generate samples using the Saltelli sampler, as shown below.

```
param_values = saltelli.sample(problem, 1024)
```

Here, `param_values` is a NumPy matrix. If we run `param_values.shape`, we see that the matrix is 8000 by 3. The Saltelli sampler generated 8000 samples. The Saltelli sampler generates $N * (2D + 2)$ samples, where in this example N is 1024 (the argument we supplied) and D is 3 (the number of model inputs). The keyword argument `calc_second_order=False` will exclude second-order indices, resulting in a smaller sample matrix with $N*(D+2)$ rows instead.

Run Model

As mentioned above, SALib is not involved in the evaluation of the mathematical or computational model. If the model is written in Python, then generally you will loop over each sample input and evaluate the model:

```
Y = np.zeros([param_values.shape[0]])

for i, X in enumerate(param_values):
    Y[i] = evaluate_model(X)
```

If the model is not written in Python, then the samples can be saved to a text file:

```
np.savetxt("param_values.txt", param_values)
```

Each line in `param_values.txt` is one input to the model. The output from the model should be saved to another file with a similar format: one output on each line. The outputs can then be loaded with:

```
Y = np.loadtxt("outputs.txt", float)
```

In this example, we are using the Ishigami function provided by SALib. We can evaluate these test functions as shown below:

```
Y = Ishigami.evaluate(param_values)
```

Perform Analysis

With the model outputs loaded into Python, we can finally compute the sensitivity indices. In this example, we use `sobol.analyze`, which will compute first, second, and total-order indices.

```
Si = sobol.analyze(problem, Y)
```

`Si` is a Python `dict` with the keys `"S1"`, `"S2"`, `"ST"`, `"S1_conf"`, `"S2_conf"`, and `"ST_conf"`. The `_conf` keys store the corresponding confidence intervals, typically with a confidence level of 95%. Use the keyword argument `print_to_console=True` to print all indices. Or, we can print the individual values from `Si` as shown below.

```
print(Si['S1'])

[ 0.316832  0.443763 0.012203 ]
```

Here, we see that x1 and x2 exhibit first-order sensitivities but x3 appears to have no first-order effects.

```
print(Si['ST'])

[ 0.555860  0.441898   0.244675]
```

If the total-order indices are substantially larger than the first-order indices, then there is likely higher-order interactions occurring. We can look at the second-order indices to see these higher-order interactions:

```
print("x1-x2:", Si['S2'][0,1])
print("x1-x3:", Si['S2'][0,2])
print("x2-x3:", Si['S2'][1,2])

x1-x2: 0.0092542
x1-x3: 0.2381721
x2-x3: -0.0048877
```

We can see there are strong interactions between x1 and x3. Some computing error will appear in the sensitivity indices. For example, we observe a negative value for the x2-x3 index. Typically, these computing errors shrink as the number of samples increases.

The output can then be converted to a Pandas DataFrame for further analysis.

```
total_Si, first_Si, second_Si = Si.to_df()

# Note that if the sample was created with `calc_second_order=False`
# Then the second order sensitivities will not be returned
# total_Si, first_Si = Si.to_df()
```

Basic Plotting

Basic plotting facilities are provided for convenience.

```
Si.plot()
```

The `plot()` method returns matplotlib axes objects to allow later adjustment.

1.2.4 Another Example

When the model you want to analyse depends on parameters that are not part of the sensitivity analysis, like position or time, the analysis can be performed for each time/position "bin" separately.

Consider the example of a parabola:

$$f(x) = a + bx^2$$

The parameters a and b will be subject to the sensitivity analysis, but x will be not.

We start with a set of imports:

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

from SALib.sample import saltelli
from SALib.analyze import sobol
```

and define the parabola:

```
def parabola(x, a, b):
    """Return y = a + b*x**2."""
    return a + b*x**2
```

The `dict` describing the problem contains therefore only a and b:

```
problem = {
    'num_vars': 2,
    'names': ['a', 'b'],
    'bounds': [[0, 1]]*2
}
```

The triad of sampling, evaluating and analysing becomes:

```
# sample
param_values = saltelli.sample(problem, 2**6)

# evaluate
x = np.linspace(-1, 1, 100)
y = np.array([parabola(x, *params) for params in param_values])

# analyse
sobol_indices = [sobol.analyze(problem, Y) for Y in y.T]
```

Note how we analysed for each x separately.

Now we can extract the first-order Sobol indices for each bin of x and plot:

```
S1s = np.array([s['S1'] for s in sobol_indices])

fig = plt.figure(figsize=(10, 6), constrained_layout=True)
gs = fig.add_gridspec(2, 2)

ax0 = fig.add_subplot(gs[:, 0])
ax1 = fig.add_subplot(gs[0, 1])
```

(continues on next page)

(continued from previous page)

```
ax2 = fig.add_subplot(gs[1, 1])

for i, ax in enumerate([ax1, ax2]):
    ax.plot(x, S1s[:, i],
            label=r'S1$_\mathregular{{{}}}$'.format(problem["names"][i]),
            color='black')
    ax.set_xlabel("x")
    ax.set_ylabel("First-order Sobol index")

    ax.set_ylim(0, 1.04)

    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()

    ax.legend(loc='upper right')

ax0.plot(x, np.mean(y, axis=0), label="Mean", color='black')

# in percent
prediction_interval = 95

ax0.fill_between(x,
                 np.percentile(y, 50 - prediction_interval/2., axis=0),
                 np.percentile(y, 50 + prediction_interval/2., axis=0),
                 alpha=0.5, color='black',
                 label=f"{prediction_interval} % prediction interval")

ax0.set_xlabel("x")
ax0.set_ylabel("y")
ax0.legend(title=r"$y=a+b\cdot x^2$",
           loc='upper center')._legend_box.align = "left"

plt.show()
```

With the help of the plots, we interprete the Sobol indices. At $x = 0$, the variation in y can be explained to 100 % by parameter a as the contribution to y from bx^2 vanishes. With larger $|x|$, the contribution to the variation from parameter b increases and the contribution from parameter a decreases.

1.3 Advanced Examples

1.3.1 Group sampling (Sobol and Morris methods only)

It is sometimes useful to perform sensitivity analysis on groups of input variables to reduce the number of model runs required, when variables belong to the same component of a model, or there is some reason to believe that they should behave similarly.

Groups can be specified in two ways for the Sobol and Morris methods.

First, as a fourth column in the parameter file:

```
# name lower_bound upper_bound group_name
P1 0.0 1.0 Group_1
P2 0.0 5.0 Group_2
P3 0.0 5.0 Group_2
...etc.
```

Or in the *problem* dictionary:

```
problem = {
    'groups': ['Group_1', 'Group_2', 'Group_2'],
    'names': ['x1', 'x2', 'x3'],
    'num_vars': 3,
    'bounds': [[-3.14, 3.14], [-3.14, 3.14], [-3.14, 3.14]]
}
```

`groups` is a list of strings specifying the group name to which each variable belongs. The rest of the code stays the same:

```
param_values = saltelli.sample(problem, 1024)
Y = Ishigami.evaluate(param_values)
Si = sobol.analyze(problem, Y, print_to_console=True)
```

But the output is printed by group:

```
            ST   ST_conf
Group_1  0.555309  0.084058
Group_2  0.684332  0.057449
            S1   S1_conf
Group_1  0.316696  0.056797
Group_2  0.456497  0.079049
                        S2   S2_conf
(Group_1, Group_2)  0.238909  0.127195
```

The output can then be converted to a Pandas DataFrame for further analysis.

```
total_Si, first_Si, second_Si = Si.to_df()
```

1.3.2 Generating alternate distributions

In Essential basic functionality, we generate a uniform sample of parameter space.

```
from SALib.sample import saltelli
from SALib.analyze import sobol
from SALib.test_functions import Ishigami
import numpy as np

problem = {
    'num_vars': 3,
    'names': ['x1', 'x2', 'x3'],
    'bounds': [[-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359],
               [-3.14159265359, 3.14159265359]]
```

(continues on next page)

(continued from previous page)

```
}

param_values = saltelli.sample(problem, 1024)
```

SALib is also capable of generating alternate sampling distributions by specifying a `dists` entry in the `problem` specification.

As implied in the basic example, a uniform distribution is the default.

When an entry for `dists` is not 'unif', the `bounds` entry does not indicate parameter bounds but sample-specific metadata.

`bounds` definitions for available distributions:

- **unif: uniform distribution** e.g. `[-np.pi, np.pi]` defines the lower and upper bounds
- **triang: triangular with width (scale) and location of peak.** Location of peak is in percentage of width. Lower bound assumed to be zero.

  e.g. `[3, 0.5]` assumes 0 to 3, with a peak at 1.5
- norm: normal distribution with mean and standard deviation
- lognorm: lognormal with ln-space mean and standard deviation

An example specification is shown below:

```
problem = {
    'names': ['x1', 'x2', 'x3'],
    'num_vars': 3,
    'bounds': [[-np.pi, np.pi], [1.0, 0.2], [3, 0.5]],
    'groups': ['G1', 'G2', 'G1'],
    'dists': ['unif', 'lognorm', 'triang']
}
```

1.4 Concise API Reference

This page documents the sensitivity analysis methods supported by SALib.

1.4.1 FAST - Fourier Amplitude Sensitivity Test

SALib.sample.fast_sampler.**sample**(*problem*, *N*, *M=4*, *seed=None*)

Generate model inputs for the extended Fourier Amplitude Sensitivity Test (eFAST).

Returns a NumPy matrix containing the model inputs required by the extended Fourier Amplitude sensitivity test. The resulting matrix contains N * D rows and D columns, where D is the number of parameters. The samples generated are intended to be used by *SALib.analyze.fast.analyze()*.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 4)

References

SALib.analyze.fast.**analyze**(*problem*, *Y*, *M=4*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)

Performs the extended Fourier Amplitude Sensitivity Test (eFAST) on model outputs.

Returns a dictionary with keys 'S1' and 'ST', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: *fast_sampler* : *SALib.sample.fast_sampler.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 4)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = fast_sampler.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = fast.analyze(problem, Y, print_to_console=False)
```

1.4.2 RBD-FAST - Random Balance Designs Fourier Amplitude Sensitivity Test

SALib.sample.latin.**sample**(*problem*, *N*, *seed=None*)

Generate model inputs using Latin hypercube sampling (LHS).

Returns a NumPy matrix containing the model inputs generated by Latin hypercube sampling. The resulting matrix contains N rows and D columns, where D is the number of parameters.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate

References

SALib.analyze.rbd_fast.**analyze**(*problem*, *X*, *Y*, *M=10*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)

Performs the Random Balanced Design - Fourier Amplitude Sensitivity Test (RBD-FAST) on model outputs.

Returns a dictionary with keys 'S1', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: all samplers

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – A NumPy array containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = rbd_fast.analyze(problem, X, Y, print_to_console=False)
```

1.4.3 Method of Morris

SALib.sample.morris.**sample**(*problem: Dict*, *N: int*, *num_levels: int = 4*, *optimal_trajectories: Optional[int] = None*, *local_optimization: bool = True*, *seed: Optional[int] = None*) → numpy.ndarray

Generate model inputs using the Method of Morris

Returns a NumPy matrix containing the model inputs required for Method of Morris. The resulting matrix has $(G+1)*T$ rows and D columns, where D is the number of parameters, G is the number of groups (if no groups are selected, the number of parameters). T is the number of trajectories N, or *optimal_trajectories* if selected. These model inputs are intended to be used with *SALib.analyze.morris.analyze()*.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of trajectories to generate
- **num_levels** (*int, default=4*) – The number of grid levels (should be even)
- **optimal_trajectories** (*int*) – The number of optimal trajectories to sample (between 2 and N)

- **local_optimization** (*bool, default=True*) – Flag whether to use local optimization according to Ruano et al. (2012) Speeds up the process tremendously for bigger N and num_levels. If set to `False` brute force method is used, unless `gurobipy` is available
- **seed** (*int*) – Seed to generate a random number

Returns sample_morris – Returns a numpy.ndarray containing the model inputs required for Method of Morris. The resulting matrix has $(G/D + 1) * N/T$ rows and D columns, where D is the number of parameters.

Return type numpy.ndarray

References

SALib.analyze.morris.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *num_levels: int = 4*, *seed=None*) → numpy.ndarray

Perform Morris Analysis on model outputs.

Returns a dictionary with keys 'mu', 'mu_star', 'sigma', and 'mu_star_conf', where each entry is a list of parameters containing the indices in the same order as the parameter file.

Notes

Compatible with: *morris* : `SALib.sample.morris.sample()`

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – The NumPy matrix containing the model inputs of dtype=float
- **Y** (*numpy.array*) – The NumPy array containing the model outputs of dtype=float
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 1000)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **num_levels** (*int*) – The number of grid levels, must be identical to the value passed to SALib.sample.morris (default 4)
- **seed** (*int*) – Seed to generate a random number

Returns

Si – A dictionary of sensitivity indices containing the following entries.

- *mu* - the mean elementary effect
- *mu_star* - the absolute of the mean elementary effect
- *sigma* - the standard deviation of the elementary effect
- *mu_star_conf* - the bootstrapped confidence interval
- *names* - the names of the parameters

Return type dict

References

Examples

```
>>> X = morris.sample(problem, 1000, num_levels=4)
>>> Y = Ishigami.evaluate(X)
>>> Si = morris.analyze(problem, X, Y, conf_level=0.95,
>>>                     print_to_console=True, num_levels=4)
```

1.4.4 Sobol' Sensitivity Analysis

SALib.sample.saltelli.**sample**(*problem: Dict*, *N: int*, *calc_second_order: bool = True*, *skip_values: Optional[int] = None*)

Generates model inputs using Saltelli's extension of the Sobol' sequence.

The Sobol' sequence is a popular quasi-random low-discrepancy sequence used to generate uniform samples of parameter space.

Returns a NumPy matrix containing the model inputs using Saltelli's sampling scheme. Saltelli's scheme extends the Sobol' sequence in a way to reduce the error rates in the resulting sensitivity index calculations. If *calc_second_order* is False, the resulting matrix has `N * (D + 2)` rows, where `D` is the number of parameters. If *calc_second_order* is *True*, the resulting matrix has `N * (2D + 2)` rows. These model inputs are intended to be used with *SALib.analyze.sobol.analyze()*.

Notes

The initial points of the Sobol' sequence has some repetition (see Table 2 in Campolongo [1]), which can be avoided by setting the *skip_values* parameter. Skipping values reportedly improves the uniformity of samples. It has been shown, however, that naively skipping values may reduce accuracy, increasing the number of samples needed to achieve convergence (see Owen [2]).

A recommendation adopted here is that both *skip_values* and *N* be a power of 2, where *N* is the desired number of samples (see [2] and discussion in [5] for further context). It is also suggested therein that `skip_values >= N`.

The method now defaults to setting *skip_values* to a power of two that is `>= N`. If *skip_values* is provided, the method now raises a UserWarning in cases where sample sizes may be sub-optimal according to the recommendation above.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate. Ideally a power of 2 and <= *skip_values*.
- **calc_second_order** (*bool*) – Calculate second-order sensitivities (default True)
- **skip_values** (*int or None*) – Number of points in Sobol' sequence to skip, ideally a value of base 2 (default: a power of 2 >= N, or 16; whichever is greater)

References

SALib.analyze.sobol.**analyze**(*problem*, *Y*, *calc_second_order=True*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *parallel=False*, *n_processors=None*, *keep_resamples=False*, *seed=None*)

Perform Sobol Analysis on model outputs.

Returns a dictionary with keys 'S1', 'S1_conf', 'ST', and 'ST_conf', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file. If calc_second_order is True, the dictionary also contains keys 'S2' and 'S2_conf'.

Notes

Compatible with: *saltelli* : *SALib.sample.saltelli.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **calc_second_order** (*bool*) – Calculate second-order sensitivities (default True)
- **num_resamples** (*int*) – The number of resamples (default 100)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **keep_resamples** (*bool*) – Whether or not to store intermediate resampling results (default False)

References

Examples

```
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
```

1.4.5 Delta Moment-Independent Measure

SALib.sample.latin.**sample**(*problem*, *N*, *seed=None*)

Generate model inputs using Latin hypercube sampling (LHS).

Returns a NumPy matrix containing the model inputs generated by Latin hypercube sampling. The resulting matrix contains N rows and D columns, where D is the number of parameters.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate

References

SALib.analyze.delta.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *seed: Optional[int] = None*) → Dict

Perform Delta Moment-Independent Analysis on model outputs.

Returns a dictionary with keys 'delta', 'delta_conf', 'S1', and 'S1_conf', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: all samplers

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – A NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **num_resamples** (*int*) – The number of resamples when computing confidence intervals (default 10)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = delta.analyze(problem, X, Y, print_to_console=True)
```

1.4.6 Derivative-based Global Sensitivity Measure (DGSM)

SALib.analyze.dgsm.**analyze**(*problem*, *X*, *Y*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)

Calculates Derivative-based Global Sensitivity Measure on model outputs.

Returns a dictionary with keys 'vi', 'vi_std', 'dgsm', and 'dgsm_conf', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: *finite_diff* : *SALib.sample.finite_diff.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – The NumPy array containing the model outputs
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 1000)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = finite_diff.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = dgsm.analyze(problem, Y, print_to_console=False)
```

1.4.7 Fractional Factorial

SALib.sample.ff.**sample**(*problem*, *seed=None*)

Generates model inputs using a fractional factorial sample

Returns a NumPy matrix containing the model inputs required for a fractional factorial analysis. The resulting matrix has D columns, where D is smallest power of 2 that is greater than the number of parameters. These model inputs are intended to be used with *SALib.analyze.ff.analyze()*.

The problem file is padded with a number of dummy variables called `dummy_0` required for this procedure. These dummy variables can be used as a check for errors in the analyze procedure.

This algorithm is an implementation of that contained in Saltelli et al [Saltelli et al. 2008]

Parameters **problem** (*dict*) – The problem definition

Returns **sample**

Return type numpy.array

References

SALib.analyze.ff.**analyze**(*problem*, *X*, *Y*, *second_order=False*, *print_to_console=False*, *seed=None*)
Perform a fractional factorial analysis

Returns a dictionary with keys 'ME' (main effect) and 'IE' (interaction effect). The techniques bulks out the number of parameters with dummy parameters to the nearest 2**n. Any results involving dummy parameters could indicate a problem with the model runs.

Notes

Compatible with: *ff* : *SALib.sample.ff.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – The NumPy array containing the model outputs
- **second_order** (*bool, default=False*) – Include interaction effects
- **print_to_console** (*bool, default=False*) – Print results directly to console

Returns **Si** – A dictionary of sensitivity indices, including main effects `ME`, and interaction effects `IE` (if `second_order` is True)

Return type dict

References

Examples

```
>>> X = sample(problem)
>>> Y = X[:, 0] + (0.1 * X[:, 1]) + ((1.2 * X[:, 2]) * (0.2 + X[:, 0]))
>>> analyze(problem, X, Y, second_order=True, print_to_console=True)
```

1.4.8 PAWN Sensitivity Analysis

SALib.analyze.pawn.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *S: int = 10*, *print_to_console: bool = False*, *seed: Optional[int] = None*)
Performs PAWN sensitivity analysis.

Calculates the min, mean, median, max, and coefficient of variation (CV).

CV is (standard deviation / mean), and so lower values indicate little change over the slides, and larger values indicate large variations across the slides.

Notes

Compatible with: all samplers

This implementation ignores all NaNs.

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – A NumPy array containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **S** (*int*) – Number of slides (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = pawn.analyze(problem, X, Y, S=10, print_to_console=False)
```

1.4.9 High-Dimensional Model Representation

SALib.analyze.hdmr.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *maxorder: int = 2*, *maxiter: int = 100*, *m: int = 2*, *K: int = 20*, *R: Optional[int] = None*, *alpha: float = 0.95*, *lambdax: float = 0.01*, *print_to_console: bool = True*, *seed: Optional[int] = None*) → Dict

High-Dimensional Model Representation (HDMR) using B-spline functions.

HDMR is used for variance-based global sensitivity analysis (GSA) with correlated and uncorrelated inputs. This function uses as input

- a N x d matrix of N different d-vectors of model inputs (factors/parameters)
- a N x 1 vector of corresponding model outputs

Returns: - each factor's first, second, and third order sensitivity coefficient

(separated in total, structural and correlative contributions),

- an estimate of their 95% confidence intervals (from bootstrap method)
- the coefficients of the significant B-spline basis functions that govern output,
- Y (determined by an F-test of the error residuals of the HDMR model (emulator) with/without a given first, second and/or third order B-spline). These coefficients define an emulator that can be used to predict the output, Y, of the original (CPU-intensive) model for any d-vector of model inputs. For uncorrelated model inputs (columns of X are independent), the HDMR sensitivity indices reduce to a single index (= structural contribution), consistent with their values derived from commonly used variance-based GSA methods.

Notes

Compatible with: all samplers

Contributed by @sahin-abdullah (sahina@uci.edu)

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs, N rows by d columns
- **Y** (*numpy.array*) – The NumPy array containing the model outputs for each row of X
- **maxorder** (*int (1-3, default: 2)*) – Maximum HDMR expansion order
- **maxiter** (*int (1-1000, default: 100)*) – Max iterations backfitting
- **m** (*int (2-10, default: 2)*) – Number of B-spline intervals
- **K** (*int (1-100, default: 20)*) – Number of bootstrap iterations
- **R** (*int (100-N/2, default: N/2)*) – Number of bootstrap samples. Will be set to length of *Y* if *K* is set to 1.
- **alpha** (*float (0.5-1)*) – Confidence interval F-test
- **lambdax** (*float (0-10, default: 0.01)*) – Regularization term
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*bool*) – Set a seed value

Returns

Si – Sa: Uncorrelated contribution Sa_conf: Confidence interval of Sa Sb: Correlated contribution Sb_conf: Confidence interval of Sb S: Total contribution of a particular term S_conf: Confidence interval of S ST: Total contribution of a particular dimension/parameter ST_conf: Confidence interval of ST Sa: Uncorrelated contribution select: Number of selection (F-Test) Em: Result set

C1: First order coefficient C2: Second order coefficient C3: Third Order coefficient

Return type *ResultDict*,

References

Examples

```
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = hdmr.analyze(problem, X, Y, **options)
```

1.5 License

1.6 Developers

- Jon Herman <jdherman8@gmail.com>
- Will Usher <william.usher@ouce.ox.ac.uk>
- Chris Mutel
- Bernardo Trindade
- Dave Hadka
- Matt Woodruff
- Fernando Rios
- Dan Hyams
- xantares
- Abdullah Sahin <sahina@uci.edu>
- Takuya Iwanaga

1.7 How to cite SAlib

If you would like to use our software, please cite it using the following:

> Herman, J. and Usher, W. (2017) SALib: An open-source Python library for sensitivity analysis. Journal of Open Source Software, 2(9). doi:10.21105/joss.00097

|paper status|

If you use BibTeX, cite using the following entry:

```
@article{Herman2017,
    doi = {10.21105/joss.00097},
    url = {https://doi.org/10.21105/joss.00097},
    year  = {2017},
    month = {jan},
    publisher = {The Open Journal},
    volume = {2},
    number = {9},
    author = {Jon Herman and Will Usher},
    title = {{SALib}: An open-source Python library for Sensitivity Analysis},
    journal = {The Journal of Open Source Software}
}
```

1.8 Projects that use SALib

Many projects now use the Global Sensitivity Analysis features provided by SALib. Here is a selection:

1.8.1 Software

- The City Energy Analyst
- pynoddy
- savvy
- rhodium
- pySur
- EMA workbench
- Brain/Circulation Model Developer
- DAE Tools
- agentpy
- uncertainpy
- CLIMADA

1.8.2 Blogs

- Sensitivity Analyis in Python
- Sensitivity Analysis with SALib
- Running Sobol using SALib
- Extensions of SALib for more complex sensitivity analyses

1.8.3 Videos

- PyData Presentation on SALib

If you would like to be added to this list, please submit a pull request, or create an issue.

Many thanks for using SALib.

1.9 Changelog

This changelog follows the format defined at: https://keepachangelog.com/en/1.0.0/

1.9.1 [1.4.5]

- Adjusted Saltelli sampling to follow recommendation of Owen (2020) (http://arxiv.org/abs/2008.08051; https://github.com/scipy/scipy/pull/10844#issuecomment-672186615)
- Initial support for parallel analysis
- Updated Sobol' G-function analytic results (PR #464, Issues #335 #461)
- Sobol' analysis: Optional storage of intermediate resample results to allow analysis of variation (PR #462)

Documentation

- Updated Salteli sampling examples to use powers of 2 following recommendations
- Added `citations.cff` file

Development

- Upgrade PyScaffold to v4
- Replaced recommonmark with MyST (PR #466)

1.9.2 [1.4.0]

Shortlist of changes since v1.3.x

Added

- High Dimensional Model Representation (HDMR) method (PR #275)
- PAWN method (PR #415)
- Support for sampling/analysis method chaining (PR #339)
- Support for truncated normal distribution (PR #383)
- Confidence Interval estimation for FAST-based methods (PR #375)
- Initial support for parallel model evaluation

Documentation

- Defining non-uniform sampling now explicitly documented
- Many general documentation improvements
- Added FAQ

Development

- Generalized support for non-uniform sampling methods (PR #346)

1.9.3 [1.3.13]

Added

- Many documentation improvements
- Explicitly mention extended FAST in documentation
- Saltelli sampling: Warnings displayed when selected samples do not meet requirements (PR #416).
- Group sampling and analysis enabled for Sobol' and morris
- Enhanced DataFrame support for groups

Bug Fixes:

- Conversion to DataFrame when groups are defined with Sobol' results (PR #413 and Issue #387)

1.9.4 [1.3.0]

Added

- Various minor performance enhancements (PR #253 #264)
- Added some visualisation methods (PR #259)
- Tidying up of the Command Line Interfance, and num samples (PR #260 #291)
- Improved efficiency of summing distances in local optimization (PR #246)
- Revamped fast method for consistency (PR #239)
- Updated Sobol-G function to modified G-function (#269)

Bug Fixes:

- Method of morris didn't adjust with levels above 4 (PR #252)
- Add missing seed option for morris sampling
- Handle singular value matrix cases (PR #251)
- Fixed typo (#205)
- Updated import of scipy comb function (PR #243)

Documentation:

- Update documentation for Morris and DSGM methods (#261 #266)

Development Features:

- Updated PyScaffold to version 3.2.2 (#267)
- Updated Travis and package config (#285)

1.9.5 [1.1.0]

New Features:

- Refactored Method of Morris so the Ruano et al. local approach is default

Bug Fixes:

- Inputs to morris.analyze are provided as floats
- Removed calls to standard random library as inconsistent between Python 2 & 3
- First row in Sobol sequences should be zero, not empty

Documentation:

- Added a Code of Conduct
- Added DAETools, BCMD and others to citations - thanks for using SALib!
- Removed misleading keyword arguments in docs and readme examples
- Updated documentation for Method of Morris following refactor
- Improved existing documentation where lacking e.g. for fractional factorial method

Development Features:

- Implemented automatic deployment to PyPi
- Fixed a bug preventing automatic deployment to PyPi upon tagging a branch
- Removed postgres from travis config

1.9.6 [1.0.0]

Release of our stable version of SALIB to coincide with an submission to JOSS:

- Added a paper for submission to the Journal of Open-source Software
- Updated back-end for documentation on read-the-docs
- Updated the back-end for version introspection using PyScaffold, rather than versioneer
- Updated the Travis-CI scripts
- Moved the tests out of the SALib package and migrated to using pytest

1.9.7 [0.7.1]

Improvements to Morris sampling and Sobol groups/distributions

- Adds improved sampling for the Morris method (thanks to @JoerivanEngelen) and group sampling/analysis for the Sobol method (thanks to @calvinwhealton).
- @calvinwhealton has also added non-uniform distributions to the Sobol sampling. This will be a baseline for adding this to the other methods in the future.
- Also includes several minor bug fixes.

1.9.8 [0.7.0]

New documentation, doc strings and installation requirements

- @dhadka has kindly contributed a wealth of documentation to the project, including doc strings in every module
- no longer test for numpy <1.8.0 and matplotlib < 1.4.3, and these requirements are implemented in a new setup script.

1.9.9 [0.6.3]

Parallel option for Sobol method

- New option to run analyze.sobol function in parallel using multiprocessing

1.9.10 [0.6.2]

This release does not contain any new functionality, but SALib now is citable using a Digital Object Identifier (DOI), which can be found in the readme.

Some minor updates are included:

- morris: sigma has been removed from the grouped-morris results and plots, replaced by mu_star_conf - a bootstrapped confidence interval. Mu_star_conf is not equivalent to sigma when used in the non-grouped method of morris, but its all we have when using groups.
- some minor updates to the tests in the plotting module

1.9.11 [0.6.0]

- Set up to include and test plotting functions
- Specific plotting functions for Morris
- Fractional Factorial SA from Saltelli et al.
- Repo transferred to SALib organization, update setup and URLs
- Small bugfixes

1.9.12 [0.5.0]

- Vectorized bootstrap calculations for Morris and Sobol
- Optional trajectory optimization with Gurobi, and tests for it
- Several minor bugfixes
- Starting with v0.5, SALib is released under the MIT license.

1.9.13 [0.4.0]

- Better Python API without requiring file read/write to the OS. Consistent functional API to sampling methods so that they return numpy matrices. Analysis methods now accept numpy matrices instead of data file names. This does not change the CLI at all, but makes it much easier to use from native Python.
- Also expanded tests for regression and the Sobol method.

1.9.14 [0.3.0]

Improvements to Morris sampling and analysis methods, some bugfixes to make consistent with previous versions of the methods.

1.9.15 [0.2.0]

Improvements to Morris sampling methods (support for group sampling, and optimized trajectories). Much better test coverage, and fixed Python 3 compatibility.

1.9.16 [0.1.0]

First numbered release. Contains reasonably well-tested versions of the Sobol, Morris, and FAST methods. Also contains newer additions of DGSM and delta methods which are not as well-tested yet. Contains setup.py for installation.

1.10 SALib

1.10.1 SALib package

Subpackages

SALib.analyze package

Submodules

SALib.analyze.common_args module

SALib.analyze.common_args.**create**(*cli_parser=None*)

SALib.analyze.common_args.**run_cli**(*cli_parser*, *run_analysis*, *known_args=None*)

SALib.analyze.common_args.**setup**(*parser*)

SALib.analyze.delta module

SALib.analyze.delta.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *seed: Optional[int] = None*) → Dict

Perform Delta Moment-Independent Analysis on model outputs.

Returns a dictionary with keys 'delta', 'delta_conf', 'S1', and 'S1_conf', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: all samplers

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – A NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **num_resamples** (*int*) – The number of resamples when computing confidence intervals (default 10)

- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = delta.analyze(problem, X, Y, print_to_console=True)
```

SALib.analyze.delta.**bias_reduced_delta**(*Y*, *Ygrid*, *X*, *m*, *num_resamples*, *conf_level*)
Plischke et al. 2013 bias reduction technique (eqn 30)

SALib.analyze.delta.**calc_delta**(*Y*, *Ygrid*, *X*, *m*)
Plischke et al. (2013) delta index estimator (eqn 26) for d_hat.

SALib.analyze.delta.**cli_action**(*args*)

SALib.analyze.delta.**cli_parse**(*parser*)

SALib.analyze.delta.**sobol_first**(*Y*, *X*, *m*)

SALib.analyze.delta.**sobol_first_conf**(*Y*, *X*, *m*, *num_resamples*, *conf_level*)

SALib.analyze.dgsm module

SALib.analyze.dgsm.**analyze**(*problem*, *X*, *Y*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)
Calculates Derivative-based Global Sensitivity Measure on model outputs.

Returns a dictionary with keys 'vi', 'vi_std', 'dgsm', and 'dgsm_conf', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: *finite_diff* : *SALib.sample.finite_diff.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – The NumPy array containing the model outputs
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 1000)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = finite_diff.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = dgsm.analyze(problem, Y, print_to_console=False)
```

SALib.analyze.dgsm.**calc_dgsm**(*base*, *perturbed*, *x_delta*, *bounds*, *num_resamples*, *conf_level*)
v_i sensitivity measure following Sobol and Kucherenko (2009). For comparison, total order S_tot <= dgsm

SALib.analyze.dgsm.**calc_vi_mean**(*base*, *perturbed*, *x_delta*)
Calculate v_i mean.

Same as calc_vi_stats but only returns the mean.

SALib.analyze.dgsm.**calc_vi_stats**(*base*, *perturbed*, *x_delta*)
Calculate v_i mean and std.

v_i sensitivity measure following Sobol and Kucherenko (2009) For comparison, Morris mu* < sqrt(v_i)

Same as calc_vi_mean but returns standard deviation as well.

SALib.analyze.dgsm.**cli_action**(*args*)

SALib.analyze.dgsm.**cli_parse**(*parser*)

SALib.analyze.fast module

SALib.analyze.fast.**analyze**(*problem*, *Y*, *M=4*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)
Performs the extended Fourier Amplitude Sensitivity Test (eFAST) on model outputs.

Returns a dictionary with keys 'S1' and 'ST', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: *fast_sampler* : *SALib.sample.fast_sampler.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 4)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = fast_sampler.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = fast.analyze(problem, Y, print_to_console=False)
```

SALib.analyze.fast.**bootstrap**(*Y*, *N*, *M*, *omega_0*, *resamples*, *conf_level*)

SALib.analyze.fast.**cli_action**(*args*)

SALib.analyze.fast.**cli_parse**(*parser*)
Add method specific options to CLI parser.

Parameters **parser** (*argparse object*) –

Returns

Return type Updated argparse object

SALib.analyze.fast.**compute_orders**(*outputs*, *N*, *M*, *omega*)

SALib.analyze.ff module

Created on 30 Jun 2015

@author: will2

SALib.analyze.ff.**analyze**(*problem*, *X*, *Y*, *second_order=False*, *print_to_console=False*, *seed=None*)
Perform a fractional factorial analysis

Returns a dictionary with keys 'ME' (main effect) and 'IE' (interaction effect). The techniques bulks out the number of parameters with dummy parameters to the nearest 2**n. Any results involving dummy parameters could indicate a problem with the model runs.

Notes

Compatible with: *ff* : *SALib.sample.ff.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs
- **Y** (*numpy.array*) – The NumPy array containing the model outputs
- **second_order** (*bool, default=False*) – Include interaction effects
- **print_to_console** (*bool, default=False*) – Print results directly to console

Returns **Si** – A dictionary of sensitivity indices, including main effects `ME`, and interaction effects `IE` (if `second_order` is True)

Return type dict

References

Examples

```
>>> X = sample(problem)
>>> Y = X[:, 0] + (0.1 * X[:, 1]) + ((1.2 * X[:, 2]) * (0.2 + X[:, 0]))
>>> analyze(problem, X, Y, second_order=True, print_to_console=True)
```

SALib.analyze.ff.**cli_action**(*args*)

SALib.analyze.ff.**cli_parse**(*parser*)

SALib.analyze.ff.**interactions**(*problem*, *Y*)

Computes the second order effects

Computes the second order effects (interactions) between all combinations of pairs of input factors

Parameters

- **problem** (*dict*) – The problem definition
- **Y** (*numpy.array*) – The NumPy array containing the model outputs

Returns

- **ie_names** (*list*) – The names of the interaction pairs
- **IE** (*list*) – The sensitivity indices for the pairwise interactions

SALib.analyze.ff.**to_df**(*self*)

Conversion method to Pandas DataFrame. To be attached to ResultDict.

Returns **main_effect, inter_effect** – A tuple of DataFrames for main effects and interaction effects. The second element (for interactions) will be *None* if not available.

Return type tuple

SALib.analyze.hdmr module

SALib.analyze.hdmr.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *maxorder: int = 2*, *maxiter: int = 100*, *m: int = 2*, *K: int = 20*, *R: Optional[int] = None*, *alpha: float = 0.95*, *lambdax: float = 0.01*, *print_to_console: bool = True*, *seed: Optional[int] = None*) → Dict

High-Dimensional Model Representation (HDMR) using B-spline functions.

HDMR is used for variance-based global sensitivity analysis (GSA) with correlated and uncorrelated inputs. This function uses as input

- a N x d matrix of N different d-vectors of model inputs (factors/parameters)
- a N x 1 vector of corresponding model outputs

Returns: - each factor's first, second, and third order sensitivity coefficient

(separated in total, structural and correlative contributions),

- an estimate of their 95% confidence intervals (from bootstrap method)
- the coefficients of the significant B-spline basis functions that govern output,

- Y (determined by an F-test of the error residuals of the HDMR model (emulator) with/without a given first, second and/or third order B-spline). These coefficients define an emulator that can be used to predict the output, Y, of the original (CPU-intensive) model for any d-vector of model inputs. For uncorrelated model inputs (columns of X are independent), the HDMR sensitivity indices reduce to a single index (= structural contribution), consistent with their values derived from commonly used variance-based GSA methods.

Notes

Compatible with: all samplers

Contributed by @sahin-abdullah (sahina@uci.edu)

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.matrix*) – The NumPy matrix containing the model inputs, N rows by d columns
- **Y** (*numpy.array*) – The NumPy array containing the model outputs for each row of X
- **maxorder** (*int (1-3, default: 2)*) – Maximum HDMR expansion order
- **maxiter** (*int (1-1000, default: 100)*) – Max iterations backfitting
- **m** (*int (2-10, default: 2)*) – Number of B-spline intervals
- **K** (*int (1-100, default: 20)*) – Number of bootstrap iterations
- **R** (*int (100-N/2, default: N/2)*) – Number of bootstrap samples. Will be set to length of *Y* if *K* is set to 1.
- **alpha** (*float (0.5-1)*) – Confidence interval F-test
- **lambdax** (*float (0-10, default: 0.01)*) – Regularization term
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*bool*) – Set a seed value

Returns

Si – Sa: Uncorrelated contribution Sa_conf: Confidence interval of Sa Sb: Correlated contribution Sb_conf: Confidence interval of Sb S: Total contribution of a particular term S_conf: Confidence interval of S ST: Total contribution of a particular dimension/parameter ST_conf: Confidence interval of ST Sa: Uncorrelated contribution select: Number of selection (F-Test) Em: Result set

C1: First order coefficient C2: Second order coefficient C3: Third Order coefficient

Return type *ResultDict*,

References

Examples

```
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = hdmr.analyze(problem, X, Y, **options)
```

SALib.analyze.hdmr.**cli_action**(*args*)

SALib.analyze.hdmr.**cli_parse**(*parser*)

SALib.analyze.morris module

SALib.analyze.morris.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *num_levels: int = 4*, *seed=None*) → numpy.ndarray

Perform Morris Analysis on model outputs.

Returns a dictionary with keys 'mu', 'mu_star', 'sigma', and 'mu_star_conf', where each entry is a list of parameters containing the indices in the same order as the parameter file.

Notes

Compatible with: *morris* : `SALib.sample.morris.sample()`

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – The NumPy matrix containing the model inputs of dtype=float
- **Y** (*numpy.array*) – The NumPy array containing the model outputs of dtype=float
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 1000)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **num_levels** (*int*) – The number of grid levels, must be identical to the value passed to SALib.sample.morris (default 4)
- **seed** (*int*) – Seed to generate a random number

Returns

Si – A dictionary of sensitivity indices containing the following entries.

- *mu* - the mean elementary effect
- *mu_star* - the absolute of the mean elementary effect
- *sigma* - the standard deviation of the elementary effect
- *mu_star_conf* - the bootstrapped confidence interval
- *names* - the names of the parameters

Return type dict

References

Examples

```
>>> X = morris.sample(problem, 1000, num_levels=4)
>>> Y = Ishigami.evaluate(X)
>>> Si = morris.analyze(problem, X, Y, conf_level=0.95,
>>>                     print_to_console=True, num_levels=4)
```

SALib.analyze.morris.**cli_action**(*args*)

SALib.analyze.morris.**cli_parse**(*parser*)

SALib.analyze.pawn module

SALib.analyze.pawn.**analyze**(*problem: Dict*, *X: numpy.ndarray*, *Y: numpy.ndarray*, *S: int = 10*, *print_to_console: bool = False*, *seed: Optional[int] = None*)

Performs PAWN sensitivity analysis.

Calculates the min, mean, median, max, and coefficient of variation (CV).

CV is (standard deviation / mean), and so lower values indicate little change over the slides, and larger values indicate large variations across the slides.

Notes

Compatible with: all samplers

This implementation ignores all NaNs.

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – A NumPy array containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **S** (*int*) – Number of slides (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = pawn.analyze(problem, X, Y, S=10, print_to_console=False)
```

SALib.analyze.pawn.**cli_action**(*args*)

SALib.analyze.pawn.**cli_parse**(*parser*)

SALib.analyze.rbd_fast module

SALib.analyze.rbd_fast.**analyze**(*problem*, *X*, *Y*, *M=10*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *seed=None*)

Performs the Random Balanced Design - Fourier Amplitude Sensitivity Test (RBD-FAST) on model outputs.

Returns a dictionary with keys 'S1', where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file.

Notes

Compatible with: all samplers

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.array*) – A NumPy array containing the model inputs
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)

References

Examples

```
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = rbd_fast.analyze(problem, X, Y, print_to_console=False)
```

SALib.analyze.rbd_fast.**bootstrap**(*X_d*, *Y*, *M*, *resamples*, *conf_level*)

SALib.analyze.rbd_fast.**cli_action**(*args*)

SALib.analyze.rbd_fast.**cli_parse**(*parser*)

SALib.analyze.rbd_fast.**compute_first_order**(*permuted_outputs*, *M*)

SALib.analyze.rbd_fast.**permute_outputs**(*X*, *Y*)

Permute the output according to one of the inputs as in [_2]

References

SALib.analyze.rbd_fast.**unskew_S1**(*S1*, *M*, *N*)

Unskew the sensivity indice (Jean-Yves Tissot, Clémentine Prieur (2012) "Bias correction for the estimation of sensitivity indices based on random balance designs.", Reliability Engineering and System Safety, Elsevier, 107, 205-213. doi:10.1016/j.ress.2012.06.010)

SALib.analyze.sobol module

SALib.analyze.sobol.**Si_list_to_dict**(*S_list*, *D: int*, *num_resamples: int*, *keep_resamples: bool*, *calc_second_order: bool*)

Convert the parallel output into the regular dict format for printing/returning

SALib.analyze.sobol.**Si_to_pandas_dict**(*S_dict*)

Convert Si information into Pandas DataFrame compatible dict.

Parameters **S_dict** (ResultDict) – Sobol sensitivity indices

See also:

Si_list_to_dict

Returns **tuple** – Total and first order are dicts. Second order sensitivities contain a tuple of parameter name combinations for use as the DataFrame index and second order sensitivities. If no second order indices found, then returns tuple of (None, None)

Return type of total, first, and second order sensitivities.

Examples

```
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
>>> T_Si, first_Si, (idx, second_Si) = sobol.Si_to_pandas_dict(Si, problem)
```

SALib.analyze.sobol.**analyze**(*problem*, *Y*, *calc_second_order=True*, *num_resamples=100*, *conf_level=0.95*, *print_to_console=False*, *parallel=False*, *n_processors=None*, *keep_resamples=False*, *seed=None*)

Perform Sobol Analysis on model outputs.

Returns a dictionary with keys ‘S1’, ‘S1_conf’, ‘ST’, and ‘ST_conf’, where each entry is a list of size D (the number of parameters) containing the indices in the same order as the parameter file. If calc_second_order is True, the dictionary also contains keys ‘S2’ and ‘S2_conf’.

Notes

Compatible with: *saltelli* : *SALib.sample.saltelli.sample()*

Parameters

- **problem** (*dict*) – The problem definition
- **Y** (*numpy.array*) – A NumPy array containing the model outputs
- **calc_second_order** (*bool*) – Calculate second-order sensitivities (default True)
- **num_resamples** (*int*) – The number of resamples (default 100)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **keep_resamples** (*bool*) – Whether or not to store intermediate resampling results (default False)

References

Examples

```
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
```

SALib.analyze.sobol.**cli_action**(*args*)

SALib.analyze.sobol.**cli_parse**(*parser*)

SALib.analyze.sobol.**create_Si_dict**(*D: int*, *num_resamples: int*, *keep_resamples: bool*, *calc_second_order: bool*)
initialize empty dict to store sensitivity indices

SALib.analyze.sobol.**create_task_list**(*D*, *calc_second_order*, *n_processors*)
Create list with one entry (key, parameter 1, parameter 2) per sobol index (+conf.). This is used to supply parallel tasks to multiprocessing.Pool

SALib.analyze.sobol.**first_order**(*A*, *AB*, *B*)
First order estimator following Saltelli et al. 2010 CPC, normalized by sample variance

SALib.analyze.sobol.**second_order**(*A*, *ABj*, *ABk*, *BAj*, *B*)
Second order estimator following Saltelli 2002

SALib.analyze.sobol.**separate_output_values**(*Y*, *D*, *N*, *calc_second_order*)

SALib.analyze.sobol.**sobol_parallel**(*Z*, *A*, *AB*, *BA*, *B*, *r*, *tasks*)

SALib.analyze.sobol.**to_df**(*self*)
Conversion method to Pandas DataFrame. To be attached to ResultDict.

Returns List

Return type of Pandas DataFrames in order of Total, First, Second

Example

```
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
>>> total_Si, first_Si, second_Si = Si.to_df()
```

SALib.analyze.sobol.**total_order**(*A*, *AB*, *B*)
Total order estimator following Saltelli et al. 2010 CPC, normalized by sample variance

Module contents

SALib.plotting package

Submodules

SALib.plotting.bar module

SALib.plotting.bar.**plot**(*Si_df*, *ax=None*)
Create bar chart of results.

Parameters **Si_df** (*) –

Returns * ax

Return type matplotlib axes object

Examples

```
>>> from SALib.plotting.bar import plot as barplot
>>> from SALib.test_functions import Ishigami
>>>
>>> # See README for example problem specification
>>>
>>> X = saltelli.sample(problem, 512)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=False)
>>> total, first, second = Si.to_df()
>>> barplot(total)
```

SALib.plotting.hdmr module

Created on Dec 20, 2019

@author: @sahin-abdullah

This submodule produces two diffent figures: (1) emulator vs simulator, (2) regression lines of first order component functions

SALib.plotting.hdmr.**plot**(*Si*)

SALib.plotting.morris module

Created on 29 Jun 2015

@author: @willu47

This module provides the basic infrastructure for plotting charts for the Method of Morris results

The procedures should build upon and return an axes instance:

```
import matplotlib.pyplot as plt
Si = morris.analyze(problem, param_values, Y, conf_level=0.95,
                    print_to_console=False, num_levels=10)

# set plot style etc.
fig, ax = plt.subplots(1)
p = SALib.plotting.morris.horizontal_bar_plot(ax, Si, {'marker':'x'})
p.show()
```

SALib.plotting.morris.**covariance_plot**(*ax*, *Si*, *opts=None*, *unit=''*)
Plots mu* against sigma or the 95% confidence interval

SALib.plotting.morris.**horizontal_bar_plot**(*ax*, *Si*, *opts=None*, *sortby='mu_star'*, *unit=''*)
Updates a matplotlib axes instance with a horizontal bar plot of mu_star, with error bars representing mu_star_conf.

SALib.plotting.morris.**sample_histograms**(*fig*, *input_sample*, *problem*, *opts=None*)
Plots a set of subplots of histograms of the input sample

Module contents

SALib.sample package

Subpackages

SALib.sample.morris package

Submodules

SALib.sample.morris.brute module

class SALib.sample.morris.brute.**BruteForce**
Bases: *SALib.sample.morris.strategy.Strategy*

Implements the brute force optimisation strategy

brute_force_most_distant(*input_sample: numpy.ndarray*, *num_samples: int*, *num_params: int*, *k_choices: int*, *num_groups: Optional[int] = None*) → List
Use brute force method to find most distant trajectories

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_samples** (*int*) – The number of samples to generate
- **num_params** (*int*) – The number of parameters
- **k_choices** (*int*) – The number of optimal trajectories
- **num_groups** (*int, default=None*) – The number of groups

Returns

Return type list

find_maximum(*scores*, *N*, *k_choices*)
Finds the *k_choices* maximum scores from *scores*

Parameters

- **scores** (*numpy.ndarray*) –
- **N** (*int*) –
- **k_choices** (*int*) –

Returns

Return type list

find_most_distant(*input_sample: numpy.ndarray*, *num_samples: int*, *num_params: int*, *k_choices: int*, *num_groups: Optional[int] = None*) → numpy.ndarray
Finds the 'k_choices' most distant choices from the 'num_samples' trajectories contained in 'input_sample'

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_samples** (*int*) – The number of samples to generate
- **num_params** (*int*) – The number of parameters
- **k_choices** (*int*) – The number of optimal trajectories
- **num_groups** (*int, default=None*) – The number of groups

Returns

Return type numpy.ndarray

static grouper(*n*, *iterable*)

static mappable(*combos*, *pairwise*, *distance_matrix*)

Obtains scores from the distance_matrix for each pairwise combination held in the combos array

Parameters

- **combos** (*numpy.ndarray*) –
- **pairwise** (*numpy.ndarray*) –
- **distance_matrix** (*numpy.ndarray*) –

static nth(*iterable*, *n*, *default=None*)

Returns the nth item or a default value

Parameters

- **iterable** (*iterable*) –
- **n** (*int*) –
- **default** (*default=None*) – The default value to return

SALib.sample.morris.local module

class SALib.sample.morris.local.**LocalOptimisation**

Bases: *SALib.sample.morris.strategy.Strategy*

Implements the local optimisation algorithm using the Strategy interface

add_indices(*indices: Tuple*, *distance_matrix: numpy.ndarray*) → List

Adds extra indices for the combinatorial problem.

Parameters

- **indices** (*tuple*) –
- **distance_matrix** (*numpy.ndarray (M,M)*) –

Example

```
>>> add_indices((1,2), numpy.array((5,5)))
[(1, 2, 3), (1, 2, 4), (1, 2, 5)]
```

find_local_maximum(*input_sample: numpy.ndarray*, *N: int*, *num_params: int*, *k_choices: int*, *num_groups: Optional[int] = None*) → List

Find the most different trajectories in the input sample using a local approach

An alternative by Ruano et al. (2012) for the brute force approach as originally proposed by Campolongo et al. (2007). The method should improve the speed with which an optimal set of trajectories is found tremendously for larger sample sizes.

Parameters

- **input_sample** (*np.ndarray*) –
- **N** (*int*) – The number of trajectories
- **num_params** (*int*) – The number of factors
- **k_choices** (*int*) – The number of optimal trajectories to return
- **num_groups** (*int, default=None*) – The number of groups

Returns

Return type list

get_max_sum_ind(*indices_list: List[Tuple]*, *distances: numpy.ndarray*, *i: Union[str, int]*, *m: Union[str, int]*) → Tuple

Get the indices that belong to the maximum distance in *distances*

Parameters

- **indices_list** (*list*) – list of tuples
- **distances** (*numpy.ndarray*) – size M
- **i** (*int*) –
- **m** (*int*) –

Returns

Return type list

sum_distances(*indices: Tuple*, *distance_matrix: numpy.ndarray*) → numpy.ndarray

Calculate combinatorial distance between a select group of trajectories, indicated by indices

Parameters

- **indices** (*tuple*) –
- **distance_matrix** (*numpy.ndarray (M,M)*) –

Returns

Return type numpy.ndarray

Notes

This function can perhaps be quickened by calculating the sum of the distances. The calculated distances, as they are right now, are only used in a relative way. Purely summing distances would lead to the same result, at a perhaps quicker rate.

SALib.sample.morris.morris module

Generate a sample using the Method of Morris

Three variants of Morris' sampling for elementary effects is supported:

- Vanilla Morris
- **Optimised trajectories when `optimal_trajectories=True` (using** Campolongo's enhancements from 2007 and optionally Ruano's enhancement from 2012; `local_optimization=True`)
- **Groups with optimised trajectories when `optimal_trajectories=True` and** the problem definition specifies groups (note that `local_optimization` must be `False`)

At present, optimised trajectories is implemented using either a brute-force approach, which can be very slow, especially if you require more than four trajectories, or a local method based which is much faster. Both methods now implement working with groups of factors.

Note that the number of factors makes little difference, but the ratio between number of optimal trajectories and the sample size results in an exponentially increasing number of scores that must be computed to find the optimal combination of trajectories. We suggest going no higher than 4 from a pool of 100 samples with the brute force approach. With local_optimization = True (which is default), it is possible to go higher than the previously suggested 4 from 100.

SALib.sample.morris.morris.**sample**(*problem: Dict*, *N: int*, *num_levels: int = 4*, *optimal_trajectories: Optional[int] = None*, *local_optimization: bool = True*, *seed: Optional[int] = None*) → numpy.ndarray

Generate model inputs using the Method of Morris

Returns a NumPy matrix containing the model inputs required for Method of Morris. The resulting matrix has $(G+1)*T$ rows and D columns, where D is the number of parameters, G is the number of groups (if no groups are selected, the number of parameters). T is the number of trajectories N, or *optimal_trajectories* if selected. These model inputs are intended to be used with *SALib.analyze.morris.analyze()*.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of trajectories to generate
- **num_levels** (*int, default=4*) – The number of grid levels (should be even)
- **optimal_trajectories** (*int*) – The number of optimal trajectories to sample (between 2 and N)
- **local_optimization** (*bool, default=True*) – Flag whether to use local optimization according to Ruano et al. (2012) Speeds up the process tremendously for bigger N and num_levels. If set to `False` brute force method is used, unless `gurobipy` is available
- **seed** (*int*) – Seed to generate a random number

Returns sample_morris – Returns a numpy.ndarray containing the model inputs required for Method of Morris. The resulting matrix has $(G/D + 1) * N/T$ rows and D columns, where D is the number of parameters.

Return type numpy.ndarray

References

SALib.sample.morris.strategy module

Defines a family of algorithms for generating samples

The sample a for use with *SALib.analyze.morris.analyze*, encapsulate each one, and makes them interchangeable.

Example

```
>>> localoptimisation = LocalOptimisation()
>>> context = SampleMorris(localoptimisation)
>>> context.sample(input_sample, num_samples, num_params, k_choices, groups)
```

class SALib.sample.morris.strategy.**SampleMorris**(*strategy*)

Bases: object

Computes the optimum *k_choices* of trajectories from the input_sample.

Parameters **strategy** (*Strategy*) –

sample(*input_sample*, *num_samples*, *num_params*, *k_choices*, *num_groups*)

Computes the optimum k_choices of trajectories from the input_sample.

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_samples** (*int*) – The number of samples to generate
- **num_params** (*int*) – The number of parameters
- **k_choices** (*int*) – The number of optimal trajectories
- **num_groups** (*int*) – The number of groups

Returns An array of optimal trajectories

Return type numpy.ndarray

class SALib.sample.morris.strategy.**Strategy**

Bases: object

Declare an interface common to all supported algorithms. *SampleMorris* uses this interface to call the algorithm defined by a ConcreteStrategy.

static check_input_sample(*input_sample*, *num_params*, *num_samples*)

Check the *input_sample* is valid

Checks input sample is:

- the correct size
- values between 0 and 1

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_params** (*int*) –

- **num_samples** (*int*) –

compile_output(*input_sample*, *num_samples*, *num_params*, *maximum_combo*, *num_groups=None*)
Picks the trajectories from the input

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_samples** (*int*) –
- **num_params** (*int*) –
- **maximum_combo** (*list*) –
- **num_groups** (*int*) –

static compute_distance(*m*, *l*)
Compute distance between two trajectories

Parameters

- **m** (*np.ndarray*) –
- **l** (*np.ndarray*) –

Returns

Return type numpy.ndarray

compute_distance_matrix(*input_sample*, *num_samples*, *num_params*, *num_groups=None*, *local_optimization=False*)
Computes the distance between each and every trajectory

Each entry in the matrix represents the sum of the geometric distances between all the pairs of points of the two trajectories

If the *groups* argument is filled, then the distances are still calculated for each trajectory,

Parameters

- **input_sample** (*numpy.ndarray*) – The input sample of trajectories for which to compute the distance matrix
- **num_samples** (*int*) – The number of trajectories
- **num_params** (*int*) – The number of factors
- **num_groups** (*int, default=None*) – The number of groups
- **local_optimization** (*bool, default=False*) – If True, fills the lower triangle of the distance matrix

Returns **distance_matrix**

Return type numpy.ndarray

static run_checks(*number_samples*, *k_choices*)
Runs checks on *k_choices*

sample(*input_sample*, *num_samples*, *num_params*, *k_choices*, *num_groups=None*)
Computes the optimum k_choices of trajectories from the input_sample.

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_samples** (*int*) – The number of samples to generate

- **num_params** (*int*) – The number of parameters
- **k_choices** (*int*) – The number of optimal trajectories
- **num_groups** (*int, default=None*) – The number of groups

Returns

Return type numpy.ndarray

Module contents

Submodules

SALib.sample.common_args module

SALib.sample.common_args.**create**(*cli_parser=None*)
Create CLI parser object.

Parameters **cli_parser** (*function [optional]*) – Function to add method specific arguments to parser

Returns

Return type argparse object

SALib.sample.common_args.**run_cli**(*cli_parser*, *run_sample*, *known_args=None*)
Run sampling with CLI arguments.

Parameters

- **cli_parser** (*function*) – Function to add method specific arguments to parser
- **run_sample** (*function*) – Method specific function that runs the sampling
- **known_args** (*list [optional]*) – Additional arguments to parse

Returns

Return type argparse object

SALib.sample.common_args.**setup**(*parser*)
Add common sampling options to CLI parser.

Parameters **parser** (*argparse object*) –

Returns

Return type Updated argparse object

SALib.sample.directions module

SALib.sample.fast_sampler module

SALib.sample.fast_sampler.**cli_action**(*args*)
Run sampling method

Parameters **args** (*argparse namespace*) –

SALib.sample.fast_sampler.**cli_parse**(*parser*)
Add method specific options to CLI parser.

Parameters **parser** (*argparse object*) –

Returns

Return type Updated argparse object

SALib.sample.fast_sampler.**sample**(*problem*, *N*, *M=4*, *seed=None*)
Generate model inputs for the extended Fourier Amplitude Sensitivity Test (eFAST).

Returns a NumPy matrix containing the model inputs required by the extended Fourier Amplitude sensitivity test. The resulting matrix contains N * D rows and D columns, where D is the number of parameters. The samples generated are intended to be used by *SALib.analyze.fast.analyze()*.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate
- **M** (*int*) – The interference parameter, i.e., the number of harmonics to sum in the Fourier series decomposition (default 4)

References

SALib.sample.ff module

The sampling implementation of fractional factorial method

This implementation is based on the formulation put forward in [Saltelli et al. 2008]

SALib.sample.ff.**cli_action**(*args*)
Run sampling method

Parameters **args** (*argparse namespace*) –

SALib.sample.ff.**extend_bounds**(*problem*)
Extends the problem bounds to the nearest power of two

Parameters **problem** (*dict*) – The problem definition

SALib.sample.ff.**find_smallest**(*num_vars*)
Find the smallest exponent of two that is greater than the number of variables

Parameters **num_vars** (*int*) – Number of variables

Returns **x** – Smallest exponent of two greater than *num_vars*

Return type int

SALib.sample.ff.**generate_contrast**(*problem*)
Generates the raw sample from the problem file

Parameters **problem** (*dict*) – The problem definition

SALib.sample.ff.**sample**(*problem*, *seed=None*)
Generates model inputs using a fractional factorial sample

Returns a NumPy matrix containing the model inputs required for a fractional factorial analysis. The resulting matrix has D columns, where D is smallest power of 2 that is greater than the number of parameters. These model inputs are intended to be used with *SALib.analyze.ff.analyze()*.

The problem file is padded with a number of dummy variables called `dummy_0` required for this procedure. These dummy variables can be used as a check for errors in the analyze procedure.

This algorithm is an implementation of that contained in Saltelli et al [Saltelli et al. 2008]

Parameters **problem** (*dict*) – The problem definition

Returns **sample**

Return type numpy.array

References

SALib.sample.finite_diff module

SALib.sample.finite_diff.**cli_action**(*args*)
Run sampling method

Parameters **args** (*argparse namespace*) –

SALib.sample.finite_diff.**cli_parse**(*parser*)
Add method specific options to CLI parser.

Parameters **parser** (*argparse object*) –

Returns

Return type Updated argparse object

SALib.sample.finite_diff.**sample**(*problem: Dict*, *N: int*, *delta: float = 0.01*, *seed: Optional[int] = None*, *skip_values: int = 1024*) → numpy.ndarray
Generate matrix of samples for derivative-based global sensitivity measure (dgsm). Start from a QMC (Sobol') sequence and finite difference with delta % steps

Parameters

- **problem** (*dict*) – SALib problem specification
- **N** (*int*) – Number of samples
- **delta** (*float*) – Finite difference step size (percent)
- **seed** (*int or None*) – Random seed value
- **skip_values** (*int*) – How many values of the Sobol sequence to skip

Returns **np.array**

Return type DGSM sequence

References

SALib.sample.latin module

SALib.sample.latin.**cli_action**(*args*)
Run sampling method

Parameters **args** (*argparse namespace*) –

SALib.sample.latin.**sample**(*problem*, *N*, *seed=None*)
Generate model inputs using Latin hypercube sampling (LHS).

Returns a NumPy matrix containing the model inputs generated by Latin hypercube sampling. The resulting matrix contains N rows and D columns, where D is the number of parameters.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate

References

SALib.sample.saltelli module

SALib.sample.saltelli.**cli_action**(*args*)
Run sampling method

Parameters **args** (*argparse namespace*) –

SALib.sample.saltelli.**cli_parse**(*parser*)
Add method specific options to CLI parser.

Parameters **parser** (*argparse object*) –

Returns

Return type Updated argparse object

SALib.sample.saltelli.**sample**(*problem: Dict*, *N: int*, *calc_second_order: bool = True*, *skip_values: Optional[int] = None*)
Generates model inputs using Saltelli's extension of the Sobol' sequence.

The Sobol' sequence is a popular quasi-random low-discrepancy sequence used to generate uniform samples of parameter space.

Returns a NumPy matrix containing the model inputs using Saltelli's sampling scheme. Saltelli's scheme extends the Sobol' sequence in a way to reduce the error rates in the resulting sensitivity index calculations. If *calc_second_order* is False, the resulting matrix has `N * (D + 2)` rows, where `D` is the number of parameters. If *calc_second_order* is *True*, the resulting matrix has `N * (2D + 2)` rows. These model inputs are intended to be used with *SALib.analyze.sobol.analyze()*.

Notes

The initial points of the Sobol' sequence has some repetition (see Table 2 in Campolongo [1]), which can be avoided by setting the *skip_values* parameter. Skipping values reportedly improves the uniformity of samples. It has been shown, however, that naively skipping values may reduce accuracy, increasing the number of samples needed to achieve convergence (see Owen [2]).

A recommendation adopted here is that both *skip_values* and *N* be a power of 2, where *N* is the desired number of samples (see [2] and discussion in [5] for further context). It is also suggested therein that `skip_values >= N`.

The method now defaults to setting *skip_values* to a power of two that is `>= N`. If *skip_values* is provided, the method now raises a UserWarning in cases where sample sizes may be sub-optimal according to the recommendation above.

Parameters

- **problem** (*dict*) – The problem definition
- **N** (*int*) – The number of samples to generate. Ideally a power of 2 and <= *skip_values*.
- **calc_second_order** (*bool*) – Calculate second-order sensitivities (default True)

- **skip_values** (*int or None*) – Number of points in Sobol' sequence to skip, ideally a value of base 2 (default: a power of 2 >= N, or 16; whichever is greater)

References

SALib.sample.sobol_sequence module

SALib.sample.sobol_sequence.**index_of_least_significant_zero_bit**(*value*)

SALib.sample.sobol_sequence.**sample**(*N*, *D*)
Generate (N x D) numpy array of Sobol sequence samples

Module contents

SALib.scripts package

Submodules

SALib.scripts.salib module

Command-line utility for SALib

SALib.scripts.salib.**main**()

SALib.scripts.salib.**parse_subargs**(*module*, *parser*, *method*, *opts*)
Attach argument parser for action specific options.

Parameters

- **module** (*module*) – name of module to extract action from
- **parser** (*argparser*) – argparser object to attach additional arguments to
- **method** (*str*) – name of method (morris, sobol, etc). Must match one of the available submodules
- **opts** (*list*) – A list of argument options to parse

Returns subargs

Return type argparser namespace object

Module contents

SALib.test_functions package

Submodules

SALib.test_functions.Ishigami module

SALib.test_functions.Ishigami.**evaluate**(*X: numpy.ndarray*, *A: float = 7.0*, *B: float = 0.1*) → numpy.ndarray
Non-monotonic Ishigami-Homma three parameter test function:

f(x) = sin(x_{1}) + A sin(x_{2})^2 + Bx^{4}_{3}sin(x_{1})

This test function is commonly used to benchmark global sensitivity methods as variance-based sensitivities of this function can be analytically determined.

See listed references below.

In [2], the expected first-order indices are:

x1: 0.3139 x2: 0.4424 x3: 0.0

when A = 7, B = 0.1 when conducting Sobol' analysis with the Saltelli sampling method with a sample size of 1000.

Parameters

- **X** (*np.ndarray*) – An *N*D* array holding values for each parameter, where *N* is the number of samples and *D* is the number of parameters (in this case, three).
- **A** (*float*) – Constant *A* parameter
- **B** (*float*) – Constant *B* parameter

Returns **Y**

Return type np.ndarray

References

SALib.test_functions.Sobol_G module

SALib.test_functions.Sobol_G.**evaluate**(*values*, *a=None*, *delta=None*, *alpha=None*)
Modified Sobol G-function.

Reverts to original Sobol G-function if delta and alpha are not given.

Parameters

- **values** (*numpy.ndarray*) – input variables
- **a** (*numpy.ndarray*) – parameter values
- **delta** (*numpy.ndarray*) – shift parameters
- **alpha** (*numpy.ndarray*) – curvature parameters

Returns **Y**

Return type Result of G-function

SALib.test_functions.Sobol_G.**sensitivity_index**(*a*, *alpha=None*)

SALib.test_functions.Sobol_G.**total_sensitivity_index**(*a*, *alpha=None*)

SALib.test_functions.lake_problem module

SALib.test_functions.lake_problem.**evaluate**(*values: numpy.ndarray*, *nvars: int = 100*, *seed=101*)
Evaluate the Lake Problem with an array of parameter values.

Parameters

- **values** (*np.ndarray,*) – model inputs in the (column) order of a, q, b, mean, stdev, delta, alpha
- **nvars** (*int,*) – number of decision variables to simulate (default: 100)

Returns **np.ndarray**

Return type max_P, utility, inertia, reliability

SALib.test_functions.lake_problem.**evaluate_lake**(*values: numpy.ndarray*, *seed=101*) → numpy.ndarray
Evaluate the Lake Problem with an array of parameter values.

References

Parameters **values** (*np.ndarray,*) – model inputs in the (column) order of a, q, b, mean, stdev

where * *a* is rate of anthropogenic pollution * *q* is exponent controlling recycling rate * *b* is decay rate for phosphorus * *mean* and * *stdev* set the log normal distribution of *eps*, see [2]

Returns

Return type np.ndarray, of Phosphorus pollution over time *t*

SALib.test_functions.lake_problem.**lake_problem**(*X: Union[float, numpy.array]*, *a: Union[float, numpy.array] = 0.1*, *q: Union[float, numpy.array] = 2.0*, *b: Union[float, numpy.array] = 0.42*, *eps: Union[float, numpy.array] = 0.02*) → float
Lake Problem as given in Hadka et al., (2015) and Kwakkel (2017) modified for use as a test function.

The *mean* and *stdev* parameters control the log normal distribution of natural inflows (*epsilon* in [1] and [2]).

References

Parameters

- **X** (*float or np.ndarray*) – normalized concentration of Phosphorus at point in time
- **a** (*float or np.ndarray*) – rate of anthropogenic pollution (0.0 to 0.1)
- **q** (*float or np.ndarray*) – exponent controlling recycling rate (2.0 to 4.5).
- **b** (*float or np.ndarray*) – decay rate for phosphorus (0.1 to 0.45, where default 0.42 is irreversible, as described in [1])
- **eps** (*float or np.ndarray*) – natural inflows of phosphorus (pollution), see [3]

Returns

Return type * float, phosphorus pollution for a point in time

SALib.test_functions.linear_model_1 module

SALib.test_functions.linear_model_1.**evaluate**(*values*)
Linear model (#1) used in Li et al., (2010).

y = x1 + x2 + x3 + x4 + x5

Parameters **values** (*np.array*) –

References

SALib.test_functions.linear_model_2 module

SALib.test_functions.linear_model_2.**evaluate**(*values*)
Linear model (#2) used in Li et al., (2010).

y = 5x1 + 4x2 + 3x3 + 2x4 + x5

Parameters **values** (*np.array*) –

References

SALib.test_functions.oakley2004 module

SALib.test_functions.oakley2004.**evaluate**(*X: numpy.ndarray*, *A: numpy.ndarray*, *M: numpy.ndarray*) → numpy.ndarray
Test function taken from Oakley and O'Hagan (2004) (see Eqn. 21 in [1])

References

Module contents

SALib.util package

Submodules

SALib.util.problem module

class SALib.util.problem.**ProblemSpec**(**args*, ***kwargs*)
Bases: dict

Dictionary-like object representing an SALib Problem specification.

property analysis

analyze(*func*, **args*, ***kwargs*)
Analyze sampled results using given function.

Parameters

- **func** (*function,*) – Analysis method to use. The provided function must accept the problem specification as the first parameter, X values if needed, Y values, and return a numpy array.

- ***args** (*list,*) – Additional arguments to be passed to *func*
- **nprocs** (*int,*) – If specified, attempts to parallelize model evaluations
- ****kwargs** (*dict,*) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

analyze_parallel(*func*, **args*, *nprocs=None*, ***kwargs*)
Analyze sampled results using the given function in parallel.

Parameters

- **func** (*function,*) – Analysis method to use. The provided function must accept the problem specification as the first parameter, X values if needed, Y values, and return a numpy array.
- ***args** (*list,*) – Additional arguments to be passed to *func*
- **nprocs** (*int,*) – Number of processors to use. Capped to the number of outputs or available processors.
- ****kwargs** (*dict,*) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

evaluate(*func*, **args*, ***kwargs*)
Evaluate a given model.

Parameters

- **func** (*function,*) – Model, or function that wraps a model, to be run/evaluated. The provided function is required to accept a numpy array of inputs as its first parameter and must return a numpy array of results.
- ***args** (*list,*) – Additional arguments to be passed to *func*
- **nprocs** (*int,*) – If specified, attempts to parallelize model evaluations
- ****kwargs** (*dict,*) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

evaluate_distributed(*func*, **args*, *nprocs=1*, *servers=None*, *verbose=False*, ***kwargs*)
Distribute model evaluation across a cluster.

Usage Conditions: * The provided function needs to accept a numpy array of inputs as

its first parameter

- The provided function must return a numpy array of results

Parameters

- **func** (*function,*) – Model, or function that wraps a model, to be run in parallel
- **nprocs** (*int,*) – Number of processors to use for each node. Defaults to 1.
- **servers** (*list[str] or None,*) – IP addresses or alias for each server/node to use.
- **verbose** (*bool,*) – Display job execution statistics. Defaults to False.

- ***args** (*list*,) – Additional arguments to be passed to *func*
- ****kwargs** (*dict*,) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

evaluate_parallel(*func*, **args*, *nprocs=None*, ***kwargs*)
Evaluate model locally in parallel.

All detected processors will be used if *nprocs* is not specified.

Parameters

- **func** (*function*,) – Model, or function that wraps a model, to be run in parallel. The provided function needs to accept a numpy array of inputs as its first parameter and must return a numpy array of results.
- **nprocs** (*int*,) – Number of processors to use. Capped to the number of available processors.
- ***args** (*list*,) – Additional arguments to be passed to *func*
- ****kwargs** (*dict*,) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

plot()
Plot results.

Returns axes

Return type matplotlib axes object

property results

sample(*func*, **args*, ***kwargs*)
Create sample using given function.

Parameters

- **func** (*function*,) – Sampling method to use. The given function must accept the SALib problem specification as the first parameter and return a numpy array.
- ***args** (*list*,) – Additional arguments to be passed to *func*
- ****kwargs** (*dict*,) – Additional keyword arguments passed to *func*

Returns self

Return type ProblemSpec object

property samples

set_results(*results: numpy.ndarray*)
Set previously available model results.

set_samples(*samples: numpy.ndarray*)
Set previous samples used.

to_df()
Convert results to Pandas DataFrame.

SALib.util.results module

class SALib.util.results.**ResultDict**(**args*, ***kwargs*)
Bases: dict

Dictionary holding analysis results.

Conversion methods (e.g. to Pandas DataFrames) to be attached as necessary by each implementing method

plot(*ax=None*)
Create bar chart of results

to_df()
Convert dict structure into Pandas DataFrame.

SALib.util.util_funcs module

SALib.util.util_funcs.**avail_approaches**(*pkg*)
Create list of available modules.

Parameters **pkg** (*module*) – module to inspect

Returns **method** – A list of available submodules

Return type list

SALib.util.util_funcs.**read_param_file**(*filename*, *delimiter=None*)
Unpacks a parameter file into a dictionary

Reads a parameter file of format:

```
Param1,0,1,Group1,dist1
Param2,0,1,Group2,dist2
Param3,0,1,Group3,dist3
```

(Group and Dist columns are optional)

Returns a dictionary containing:

- names - the names of the parameters
- bounds - a list of lists of lower and upper bounds
- **num_vars - a scalar indicating the number of variables** (the length of names)
- groups - a list of group names (strings) for each variable
- **dists - a list of distributions for the problem,** None if not specified or all uniform

Parameters

- **filename** (*str*) – The path to the parameter file
- **delimiter** (*str, default=None*) – The delimiter used in the file to distinguish between columns

Module contents

A set of utility functions

class SALib.util.**ResultDict**(**args*, ***kwargs*)
Bases: dict

Dictionary holding analysis results.

Conversion methods (e.g. to Pandas DataFrames) to be attached as necessary by each implementing method

plot(*ax=None*)
Create bar chart of results

to_df()
Convert dict structure into Pandas DataFrame.

SALib.util.**avail_approaches**(*pkg*)
Create list of available modules.

Parameters **pkg** (*module*) – module to inspect

Returns **method** – A list of available submodules

Return type list

SALib.util.**read_param_file**(*filename*, *delimiter=None*)
Unpacks a parameter file into a dictionary

Reads a parameter file of format:

```
Param1,0,1,Group1,dist1
Param2,0,1,Group2,dist2
Param3,0,1,Group3,dist3
```

(Group and Dist columns are optional)

Returns a dictionary containing:

- names - the names of the parameters
- bounds - a list of lists of lower and upper bounds
- **num_vars - a scalar indicating the number of variables** (the length of names)
- groups - a list of group names (strings) for each variable
- **dists - a list of distributions for the problem,** None if not specified or all uniform

Parameters

- **filename** (*str*) – The path to the parameter file
- **delimiter** (*str, default=None*) – The delimiter used in the file to distinguish between columns

SALib.util.**scale_samples**(*params: numpy.ndarray*, *problem: Dict*)
Scale samples based on specified distribution (defaulting to uniform).

Adds an entry to the problem specification to indicate samples have been scaled to maintain backwards compatibility (*sample_scaled*).

Parameters

- **params** (*np.ndarray,*) – numpy array of dimensions *num_params*-by-N, where N is the number of samples
- **problem** (*dictionary,*) – SALib problem specification

Returns

Return type np.ndarray, scaled samples

Module contents

1.11 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

N

P

R

S

T

U