SALib Documentation

Release 1.5.3.dev0+g55e7079a9.d20251012

Jon Herman, Will Usher and others

Oct 12, 2025

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)
- PAWN (Pianosi and Wagener 2018, Pianosi and Wagener 2015)
- Regional Sensitivity Analysis (based on Hornberger and Spear, 1981, Saltelli et al. 2008, Pianosi et al., 2016)

CHAPTER

TWO

GETTING STARTED

2.1 Getting Started

2.1.1 Installing SALib

To install the latest stable version of SALib via pip from PyPI. 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
pip install .
```

2.1.2 Installing Prerequisite Software

Core dependencies include: - NumPy - SciPy - pandas - matplotlib

These should be installed automatically alongside SALib but otherwise they can be installed with the following command:

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

The packages are normally included with most Python bundles, such as Anaconda and Canopy.

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

(continues on next page)

(continued from previous page)

```
# 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 = sample(problem, 1024)

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

# Perform analysis
Si = 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 `[ 0.31683154 0.44376306 0.01220312]`.

2.2 Basics

2.2.1 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.

2.2.2 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.

2.2.3 An Example

In this example, we will perform a Sobol' sensitivity analysis of the Ishigami function (shown below) using the core SALib functions. The example is repeated in the next tutorial using an object-oriented interface which some may find easier to use.

The Ishigami function is commonly used to test uncertainty and sensitivity analysis methods because it exhibits strong nonlinearity and nonmonotonicity.

$$f(x) = \sin(x_1) + a \sin^2(x_2) + b x_3^4 \sin(x_1)$$

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.

2.2.4 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])
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 interpret 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.

2.3 SALib Interface Basics

The earlier example is repeated on this page with the SALib *ProblemSpec* interface. The interface provides an object-oriented approach to using SALib which some may find easier to use. Note that the interface does add computational overhead (thereby increasing runtimes) as it provides additional checks for users.

Developers who wish to integrate SALib into their own work may wish to use the core API as detailed in the first example instead.

2.3.1 The Ishigami function

As a reminder, we will perform a Sobol' sensitivity analysis of the Ishigami function, shown below.

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

The ProblemSpec Interface

SALib provides a single interface from which all methods can be accessed. The interface is called the `ProblemSpec` (the Problem Specification). One advantage to the interface is the method chaining approach which offers a concise workflow. The sampling, evaluation and analysis trifecta can be run with:

```
import numpy as np

from SALib.test_functions import Ishigami
from SALib import ProblemSpec


sp = ProblemSpec({
        "names": ["x1", "x2", "x3"],
        "groups": None,
        "bounds": [[-np.pi, np.pi]] * 3,
        "outputs": ["Y"],
    })

(
    sp.sample_sobol(1024)
    .evaluate(Ishigami.evaluate)
    .analyze_sobol()
)

total_Si, first_Si, second_Si = sp.to_df()

sp.plot()
sp.heatmap()
```

Each step in the above is outlined in the sections below.

Importing SALib

First, the necessary packages are imported.

Here, we import numpy as well as the Ishigami test function (to use for this example) and the `ProblemSpec` interface.

```
import numpy as np

from SALib.test_functions import Ishigami
from SALib import ProblemSpec
```

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` which holds the names of the inputs, and the bounds on each input. Optionally, the expected output(s) can also be named.

```
sp = ProblemSpec({
    'names': ['x1', 'x2', 'x3'],
    'bounds': [
        [-np.pi, np.pi],  # bounds for x1
        [-np.pi, np.pi],  # ... x2
        [-np.pi, np.pi]   # ... x3
    ],
    'outputs': ['Y']
})
```

As seen above, the `ProblemSpec` simply wraps around a `dict`.

Here, the default is to assume all inputs are uniformly distributed.

See *Advanced Examples* on how to provide further details, including alternate distributions.

> **Note**
>
> If `outputs` is not provided, then SALib will automatically create generic names.
>
> `Y` for a single output `Y1, Y2, ... Yn` for multiple outputs

Method Chaining

Since we are performing a Sobol' sensitivity analysis, we need to generate samples using the Sobol' sampler, run the model, then analyze the outputs.

In the example above, all the steps are expressed as a method chain

```
(
    sp.sample_sobol(1024)
    .evaluate(Ishigami.evaluate)
    .analyze_sobol()
)
```

That said, each step can be run individually. Note that the `sample_` and `analyze_` methods are shown with default arguments.

```
sp.sample_sobol(1024, calc_second_order=True)
sp.evaluate(Ishigami.evaluate)
sp.analyze_sobol(print_to_console=False, calc_second_order=True)
```

The `samples`, `results`, and results of `analysis` are all held inside the `sp` object. If needed, these may be extracted via their respective attributes.

```
X = sp.samples
y = sp.results
S = sp.analysis
```

Internally, all data is handled as a numpy array/matrix.

Tip

All sampling, evaluation and analysis methods may be accessed through the ProblemSpec interface and follow a standard pattern.

- Sampling methods can be accessed with `sp.sample_[name of method]`
- Likewise, for analysis methods use `sp.analyze_[name of method]`

See the documentation of each method for further information.

Generating Samples

In this example we are using the Sobol' sampling method (shown below with the default value for `calc_second_order`).

```
sp.sample_sobol(1024, calc_second_order=True)
```

If we run `sp.samples.shape`, we will see the matrix is 8192 by 3. In other words, the Sobol' sampler generated $N * (2D + 2)$ 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.

Note

Specific sampling methods have their own requirements and behaviours. The documentation for each method lists a brief overview and includes references to provide further details.

A generic `sp.sample` method is also available, allowing use of your own sampling function.

```
sp.sample(my_sampler, *args, **kwargs)
```

The provided function must follow two requirements.

1. A `ProblemSpec` must be accepted as its first argument. This can simply be defined as a dictionary following the same format as outlined above.
2. The function must return a numpy array.

If a sample has already been defined, this can be provided to the interface like so:

```
sp.set_samples(X)
```

where `X` is a numpy array.

> **Note**
>
> `sp.set_results(Y)` can be used to set existing results.

> **Warning**
>
> Care must be taken to avoid inappropriately mix-and-matching sampling and analysis methods. For example, Sobol' analysis must be conducted with a Sobol' sample.

Running a Model

If the model is written in Python, and is written such that it can accept a numpy array as an input in its first position, then it may be called directly with the interface. Here, we use the Ishigami function as an example.

```
sp.evaluate(Ishigami.evaluate)
```

> **Note**
>
> SALib also supports parallel model evaluation with *sp.evaluate_parallel()*. It is assumed that all results can be held in memory.

The Ishigami module provides an `evaluate` function that looks something like:

```
def evaluate(X: np.ndarray, A: float = 7.0, B: float = 0.1):
```

The actual implementation can be seen here.

Note that the inputs (`X`) is the first argument.

> **Tip**
>
> For user-provided models, a wrapper can be written. A wrapper is a function that accepts parameters in the expected order, then runs the model itself.
>
> See also: *Another Example* , functools.partial

Note that SALib does not require direct interaction with the model.

If the model is written in Python, then it may be run manually without SALib. 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)
```

(continues on next page)

(continued from previous page)

```
# Provide the results to the interface
sp.set_results(Y)
```

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

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

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)

# Provide the results to the interface
sp.set_results(Y)
```

Perform Analysis

With the model outputs loaded, we can finally compute the sensitivity indices. In this example, we use Sobol' analysis, which will compute first, second, and total-order indices.

```
sp.analyze_sobol()
```

We see an overview of the results once we print out the interface:

```
print(sp)


Samples:
    3 parameters: ['x1', 'x2', 'x3']
    8192 evaluations

Outputs:
    1 outputs: ['Y']
    8192 evaluations

Analysis:
        ST   ST_conf
x1  0.557271  0.078640
x2  0.442311  0.040564
x3  0.247103  0.025728

        S1   S1_conf
x1  0.317728  0.060368
x2  0.442253  0.056459
x3  0.002556  0.054109

                S2   S2_conf
(x1, x2) -0.000604  0.071442
(x1, x3)  0.247521  0.096797
(x2, x3) -0.002954  0.072420
```

Here `ST`, `S1`, and `S2` relate to the total, first-order, and second-order sensitivity indices respectively. Those ending with *_conf* indicate the corresponding confidence intervals, typically with a confidence level of 95%.

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

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:

```
y_S2 = sp.analysis['S2']
print("x1-x2:", y_S2[0,1])
print("x1-x3:", y_S2[0,2])
print("x2-x3:", y_S2[1,2])

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

Results can also be extracted as Pandas DataFrames for further analysis.

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

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()
```

For multi-output models, sensitivity results for individual outputs can be extracted:

```
sp.analysis['Y1']['S1']  # First order for Y1
sp.analysis['Y2']['S2']  # Second order for Y2
```

Basic Plotting

Basic plotting facilities are provided for convenience.

```
Si.plot()
```

All plotting methods will return matplotlib axes objects to allow later adjustment.

In the example below, the figure is created, the y-axis for the first subplot is set to use log scale, and the figure size is adjusted with `matplotlib`.

```
import matplotlib.pyplot as plt


axes = sp.plot()
axes[0].set_yscale('log')
fig = plt.gcf()  # get current figure
fig.set_size_inches(10, 4)
plt.tight_layout()
```

In addition to the basic `plot()` command, SALib can also produce a basic heatmap.

```
sp.heatmap()
```

2.3.2 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. This can be useful for the purpose of factor mapping, to identify where in parameter space the model is sensitive to.

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 import ProblemSpec
```

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:

```
sp = ProblemSpec({
    'names': ['a', 'b'],
    'bounds': [[0, 1]]*2,
})
```

The triad of sampling, evaluating and analysing becomes:

```
# Create "bins" of x
x = np.linspace(-1, 1, 100)

# Create wrapper (runs each a, b combination separately)
def wrapped_parabola(ab, x=x):
    y = np.zeros((ab.shape[0], x.shape[0]))
    for i, (a, b) in enumerate(ab):
        y[i,:] = parabola(x, a, b)

    return y

(
    sp.sample_sobol(2**6)
    .evaluate(wrapped_parabola)
    .analyze_sobol()
)
```

Note how we analyzed for each x separately.

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

```
# Get first order sensitivities for all outputs
S1s = np.array([sp.analysis[_y]['S1'] for _y in sp['outputs']])

# Get model outputs
y = sp.results

# Set up figure
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])
ax2 = fig.add_subplot(gs[1, 1])

# Populate figure subplots
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 interpret 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.

2.4 Advanced Examples

2.4.1 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

|            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()
```

2.4.2 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]]
}

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
- logunif: logarithmic uniform with lower and upper bounds
- **triang: triangular with lower and upper bounds, as well as**
  location of peak The location of peak is in percentage of width e.g. `[1.0, 3.0, 0.5]` indicates 1.0 to 3.0 with a peak at 2.0

  A soon-to-be deprecated two-value format assumes the lower bound to be 0 e.g. `[3, 0.5]` assumes 0 to 3, with a peak at 1.5
- norm: normal distribution with mean and standard deviation
- truncnorm: truncated normal distribution with lower and upper bounds, 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']
}
```

2.5 Wrapping an existing model

SALib performs sensitivity analysis for any model that can be expressed in the form of $f(X) = Y$, where X is a matrix of inputs (often referred to as the model's factors).

The analysis methods are independent of the model and can be applied non-intrusively such that it does not matter what f is.

Typical model implementations take the form of $f(a, b, c, ...) = Y$. In other words, each model factor is supplied as a separate argument to the function. In such cases it is necessary to write a wrapper to allow use with SALib. This is illustrated here with a simple linear function:

```
def linear(a, b, x):
    """Return y = a + b + x"""
    return a + b + x
```

As SALib expects a (numpy) matrix of factors, we simply "wrap" the function above like so:

```
def wrapped_linear(X, func=linear):
    """g(X) = Y, where X := [a b x] and g(X) := f(X)"""
    # We transpose to obtain each column (the model factors) as separate variables
    a, b, x = X.T

    # Then call the original model
    return func(a, b, x)
```

Note

Wrapped function is an argument

Note here that the model being "wrapped" is also passed in as an argument. This will be revisited further down below.

Tip

Interfacing with external models/programs

Here we showcase interacting with models written in Python. If the model is an external program, this is where interfacing code would be written.

A pragmatic approach could be to use subprocess to start the external program, then read in the results.

Constants, which SALib should not consider, can be expressed by defining default keyword arguments (for flexibility) or otherwise defined within the wrapper function itself.

```
def wrapped_linear_w_constant(X, a=10, func=linear):
    """f(X, a) = Y, where X := [b x] and a = 10"""
    # We transpose to obtain each column as separate variables
    b, x = X.T

    # Then call the original model
    return func(a, b, x)
```

Note that the first argument to any function provided to SALib is assumed to be a numpy array of shape $N * D$, where

D is the number of model factors (dimensions) and N is the number of their combinations. The argument name is, by convention, denoted as `X`. This is to maximize compatibility with all methods provided in SALib as they expect the first argument to hold the model factor values. Using `functools.partial()` from the *functools* package to create wrappers can be useful.

In this example, the model (`linear()`) can be used with both scalar inputs or *numpy* arrays. In cases where a, b or x are a vector of inputs, *numpy* will automatically vectorize the calculation. There are many cases where the model is not (or cannot be easily) expressed in a vectorizable form. In such cases, simply apply a `for` loop as in the example below.

```
import numpy as np
from SALib import ProblemSpec


def linear(a: float, b: float, x: float) -> float:
    return a + b * x


def wrapped_linear(X: np.ndarray, func=linear) -> np.ndarray:
    N, D = X.shape
    results = np.empty(N)
    for i in range(N):
        a, b, x = X[i, :]
        results[i] = func(a, b, x)

    return results


sp = ProblemSpec({
    'names': ['a', 'b', 'x'],
    'bounds': [
        [-1, 0],
        [-1, 0],
        [-1, 1],
    ],
})

(
    sp.sample_sobol(2**6)
    .evaluate(wrapped_linear)
    .analyze_sobol()
)

sp.to_df()

# [         ST   ST_conf
#  a  0.173636  0.072142
#  b  0.167933  0.059599
#  x  0.654566  0.208328,
#            S1   S1_conf
#  a  0.182788  0.111548
#  b  0.179003  0.145714
#  x  0.664727  0.241977,
#                 S2   S2_conf
#  (a, b) -0.022070  0.185510
```

(continues on next page)

(continued from previous page)

```
#  (a, x) -0.010781  0.186743
#  (b, x) -0.014616  0.279925]
```

Use of the core SALib functions equivalent to the previous example are shown below:

```
problem = {
    'names': ['a', 'b', 'x'],
    'bounds': [
        [-1, 0],
        [-1, 0],
        [-1, 1],
    ],
    'num_vars': 3
}

X = saltelli.sample(problem, 64)
Y = np.empty(params.shape[0])
for i in range(params.shape[0]):
    Y[i] = wrapped_linear(params[i, :])

res = sobol.analyze(problem, Y)
res.to_df()

# [         ST   ST_conf
#  a  0.165854  0.054096
#  b  0.165854  0.053200
#  x  0.665366  0.192756,
#           S1   S1_conf
#  a  0.167805  0.121550
#  b  0.167805  0.125178
#  x  0.665366  0.230872,
#                     S2   S2_conf
#  (a, b) -2.775558e-17  0.180493
#  (a, x) -3.902439e-03  0.202343
#  (b, x) -3.902439e-03  0.232957]
```

2.5.1 Parallel evaluation and analysis

Here we expand on some technical details that enable parallel evaluation and analysis. We noted earlier that the model being "wrapped" is also passed in as an argument. This is to facilitate parallel evaluation, as the arguments to the wrapper are passed on to workers. The approach works be using Python's mutable default argument behavior.

A further consideration is that imported modules/packages are not made available to workers in cases where functions are defined in the same file SALib is used in. Running the previous example with `.evaluate(wrapped_linear, nprocs=2)` will fail with `NameError:  name 'np' is not defined`.

The quick fix is to re-import the required packages within the model function itself:

```
def wrapped_linear(X: np.ndarray, func=linear) -> np.ndarray:
    import numpy as np  # re-import necessary packages

    N, D = X.shape
    results = np.empty(N)
```

(continues on next page)

(continued from previous page)

```
    for i in range(N):
        a, b, x = X[i, :]
        results[i] = func(a, b, x)

    return results
```

This can, however, get unwieldy for complicated models. The recommended best practice is to separate implementation (i.e., model definitions) from its use. Simply moving the model functions into a separate file is enough for this example, such that the project structure is something like:

```
project_directory
|-- model_definition.py
└── analysis.py
```

Tip

Project structure

The project structure shown above is for example purposes only. It is highly recommended that a standardized directory structure, such as https://github.com/drivendata/cookiecutter-data-science, be adopted to improve usability and reproducibility.

Here, `model_definitions.py` holds the model definitions:

```
import numpy as np


def linear(a: float, b: float, x: float) -> float:
    return a + b * x


def wrapped_linear(X: np.ndarray, func=linear) -> np.ndarray:
    N, D = X.shape
    results = np.empty(N)
    for i in range(N):
        a, b, x = X[i, :]
        results[i] = func(a, b, x)

    return results
```

and `analysis.py` contains use of SALib:

```
from model_definition import wrapped_linear


sp = ProblemSpec({
    'names': ['a', 'b', 'x'],
    'bounds': [
        [-1, 0],
        [-1, 0],
        [-1, 1],
    ],
```

(continues on next page)

(continued from previous page)

```
})

(
    sp.sample_sobol(2**6)
    .evaluate(wrapped_linear, nprocs=2)
    .analyze_sobol(nprocs=2)
)
```

Note

Multi-processing

Some interactive Python consoles, including earlier versions of IPython, may appear to hang on Windows when utilizing parallel evaluation and analysis. In such cases, the recommended workaround is to wrap use of SALib with a `__main__` check to ensure it is only run in the top-level environment.

```
if __name__ == "__main__":
    (
        sp.sample_sobol(2**6)
        .evaluate(wrapped_linear, nprocs=2)
        .analyze_sobol(nprocs=2)
    )
```

2.6 Frequently Asked Questions

2.6.1 Q. How do I wrap my model?

See this guide on *wrapping models*.

2.6.2 Q. Which technique can I use for pre-existing results?

DMIM, RBD-FAST, PAWN and HDMR methods are "given-data" approaches and can be independently applied.

2.6.3 Q. How do I get the sensitivity results?

Call the `.to_df()` method if you would like Pandas DataFrames.

If using the SALib Interface, see *SALib Interface Basics*.

2.6.4 Q. How can I plot my results?

SALib provides some basic plotting functionality. See the subsection "Basic Plotting" in *SALib Interface Basics*

See also, these examples.

2.6.5 Q. Why does the Sobol' method implemented in SALib normalize outputs?

Estimates of Sobol' indices can be biased in cases where model outputs are non-centered. We have opted to normalize outputs with the standard deviation.

See the discussion here.

In practice, non-normalized model outputs are still usable but requires larger sample sizes for the indices to converge.

CHAPTER

THREE

FOR DEVELOPERS

3.1 Concise API Reference

This page documents the sensitivity analysis methods supported by SALib.

3.1.1 FAST - Fourier Amplitude Sensitivity Test

SALib.sample.fast_sampler.**sample**(*problem*, *N*, *M=4*, *seed: int | Generator | None = None*)

Generate model inputs for extended Fourier Amplitude Sensitivity Test.

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)
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. Cukier, R.I., Fortuin, C.M., Shuler, K.E., Petschek, A.G., Schaibly, J.H., 1973. Study of the sensitivity of coupled reaction systems to uncertainties in rate coefficients. I theory. Journal of Chemical Physics 59, 3873-3878. https://doi.org/10.1063/1.1680571
2. Saltelli, A., S. Tarantola, and K. P.-S. Chan (1999). A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output. Technometrics, 41(1):39-56, doi:10.1080/00401706.1999.10485594.

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

Perform extended Fourier Amplitude Sensitivity Test 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()*

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. Cukier, R. I., C. M. Fortuin, K. E. Shuler, A. G. Petschek, and J. H. Schaibly (1973). Study of the sensitivity of coupled reaction systems to uncertainties in rate coefficients. J. Chem. Phys., 59(8):3873-3878 doi:10.1063/1.1680571
2. Saltelli, A., S. Tarantola, and K. P.-S. Chan (1999). A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output. Technometrics, 41(1):39-56, doi:10.1080/00401706.1999.10485594.
3. Pujol, G. (2006) fast99 - R *sensitivity* package https://github.com/cran/sensitivity/blob/master/R/fast99.R

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

`SALib.sample.latin.sample`(*problem*, *N*, *seed: int | Generator | None = 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
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. **McKay, M.D., Beckman, R.J., Conover, W.J., 1979.**
A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. Technometrics 21, 239-245. https://doi.org/10.2307/1268522

2. **Iman, R.L., Helton, J.C., Campbell, J.E., 1981.**
   An Approach to Sensitivity Analysis of Computer Models: Part I—Introduction, Input Variable Selection and Preliminary Variable Assessment. Journal of Quality Technology 13, 174-183. https://doi.org/10.1080/00224065.1981.11978748

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

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. S. Tarantola, D. Gatelli and T. Mara (2006) Random Balance Designs for the Estimation of First Order Global Sensitivity Indices, Reliability Engineering and System Safety, 91:6, 717-727 https://doi.org/10.1016/j.ress.2005.06.003
2. **Elmar Plischke (2010)**
   An effective algorithm for computing global sensitivity indices (EASI), Reliability Engineering & System Safety, 95:4, 354-360. doi:10.1016/j.ress.2009.11.005
3. **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
4. **Jeanne Goffart, Mickael Rabouille & Nathan Mendes (2015)**
   Uncertainty and sensitivity analysis applied to hygrothermal simulation of a brick building in a hot and humid climate, Journal of Building Performance Simulation. doi:10.1080/19401493.2015.1112430

3.1.3 Method of Morris

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

Generate model inputs using the Method of Morris.

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

- Vanilla Morris (see [1]) when `optimal_trajectories` is `None`/`False` and `local_optimization` is `False`
- **Optimised trajectories when `optimal_trajectories=True` using**
  Campolongo's enhancements (see [2]) and optionally Ruano's enhancement (see [3]) when `local_optimization=True`
- Morris with groups when the problem definition specifies groups of parameters

Results from these model inputs are intended to be used with *SALib.analyze.morris.analyze()*.

Notes

Campolongo et al., [2] introduces an optimal trajectories approach which attempts to maximize the parameter space scanned for a given number of trajectories (where *optimal_trajectories* $\in 2, ..., N$). The approach accomplishes this aim by randomly generating a high number of possible trajectories (500 to 1000 in [2]) and selecting a subset of `r` trajectories which have the highest spread in parameter space. The `r` variable in [2] corresponds to the `optimal_trajectories` parameter here.

Calculating all possible combinations of trajectories can be computationally expensive. 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 levels from a pool of 100 samples with this "brute force" approach.

Ruano et al., [3] proposed an alternative approach with an iterative process that maximizes the distance between subgroups of generated trajectories, from which the final set of trajectories are selected, again maximizing the distance between each. The approach is not guaranteed to produce the most optimal spread of trajectories, but are at least locally maximized and significantly reduce the time taken to select trajectories. With `local_optimization = True` (which is default), it is possible to go higher than the previously suggested 4 levels from a pool of 100 samples.

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** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

Returns

sample_morris – Array 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, 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.

Return type

np.ndarray

References

1. Morris, M.D., 1991. Factorial Sampling Plans for Preliminary Computational Experiments. Technometrics 33, 161-174. https://doi.org/10.1080/00401706.1991.10484804
2. Campolongo, F., Cariboni, J., & Saltelli, A. 2007. An effective screening design for sensitivity analysis of large models. Environmental Modelling & Software, 22(10), 1509-1518. https://doi.org/10.1016/j.envsoft.2006.10.004
3. Ruano, M.V., Ribes, J., Seco, A., Ferrer, J., 2012. An improved sampling strategy based on trajectory design for application of the Morris method to systems with many input factors. Environmental Modelling & Software 37, 103-109. https://doi.org/10.1016/j.envsoft.2012.03.008

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

Perform Morris Analysis on model outputs.

Returns a result set with keys `mu`, `mu_star`, `sigma`, and `mu_star_conf`, where each entry corresponds to the parameters defined in the problem spec or parameter file.

- `mu` metric indicates the mean of the distribution
- `mu_star` metric indicates the mean of the distribution of absolute values
- `sigma` is the standard deviation of the distribution

When `scaled` is True, the elementary effects are scaled by the ratio of standard deviation of `X` and `Y` according to [3]. When using this option it is important to ensure that `X` contains the actual values passed into the model, as the elementary effects are divided by the step calculated from `X` rather than using *delta* which is calculated from the number of levels used in the sample. This could be the case if you perform post-processing on the values before passing them to the model.

Scaled elementary effects are useful when comparing different model outputs with each other when the input and output parameters have different scales. The ranking between the ordinary elementary effects and the scaled should be the same.

Notes

When applied with groups, the `mu` metric is less reliable as the effect from parameters within a group become averaged out.

The `mu_star` metric avoids this issue as it indicates the mean of the absolute values. If the direction of effects is important, Campolongo et al., [2] suggest comparing `mu_star` with `mu`. If `mu` is low and `mu_star` is high, then the effects are of different signs.

`sigma` is used as an indicator of interactions between parameters, or groups of parameters.

Compatible with:

morris : `SALib.sample.morris.sample()`

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)
```

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
- **scaled** (*bool, default=False*) – If True, the elementary effects are scaled by the ratio of standard deviation of X and Y according to [3]
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 100)
- **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, None, np.random.Generator}*) – 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

1. Morris, M. (1991). Factorial Sampling Plans for Preliminary Computational Experiments. Technometrics, 33(2):161-174, doi:10.1080/00401706.1991.10484804.
2. Campolongo, F., J. Cariboni, and A. Saltelli (2007). An effective screening design for sensitivity analysis of large models. Environmental Modelling & Software, 22(10):1509-1518, doi:10.1016/j.envsoft.2006.10.004.
3. Sin and Gearney (2009) Improving the Morris Method for Sensitivity Analysis by Scaling the Elementary Effects. 19th European Symposium on Computer Aided Process Engineering ESCAPE19:925-930
4. **Moret et al. (2017)**
   Characterization of input uncertainties in strategic energy planning models. Applied Energy, Volume 202, 15 September 2017, Pages 597-617 https://doi.org/10.1016/j.apenergy.2017.05.106

3.1.4 Sobol' Sensitivity Analysis

SALib.sample.saltelli.**sample**(*problem: Dict*, *N: int*, *calc_second_order: bool = True*, *skip_values: 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()*.

Deprecated since version 1.4.6.

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 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

1. **Campolongo, F., Saltelli, A., Cariboni, J., 2011.**
   From screening to quantitative sensitivity analysis. A unified approach. Computer Physics Communications 182, 978-988. https://doi.org/10.1016/j.cpc.2010.12.039
2. **Owen, A. B., 2020.**
   On dropping the first Sobol' point. arXiv:2008.08051 [cs, math, stat]. Available at: http://arxiv.org/abs/2008.08051 (Accessed: 20 April 2021).
3. **Saltelli, A., 2002.**
   Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications 145, 280-297. https://doi.org/10.1016/S0010-4655(02)00280-1
4. **Sobol', I.M., 2001.**
   Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, The Second IMACS Seminar on Monte Carlo Methods 55, 271-280. https://doi.org/10.1016/S0378-4754(00)00270-6

5. **Discussion:** https://github.com/scipy/scipy/pull/10844
   https://github.com/scipy/scipy/pull/10844#issuecomment-672186615 https://github.com/scipy/scipy/pull/10844#issuecomment-673029539

SALib.sample.sobol.**sample**(*problem: Dict*, *N: int*, ** (Keyword-only parameters separator (PEP 3102))*, *calc_second_order: bool = True*, *scramble: bool = True*, *skip_values: int = 0*, *seed: int | Generator | None = 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. The general approach is described in [1].

Returns a NumPy matrix containing the model inputs using Saltelli's sampling scheme.

Saltelli's scheme reduces the number of required model runs from `N(2D+1)` to `N(D+1)` (see [2]).

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 [3]__), which can be avoided by scrambling the sequence.

Another option, not recommended and available for educational purposes, is to use the *skip_values* parameter. Skipping values reportedly improves the uniformity of samples. But, it has been shown that naively skipping values may reduce accuracy, increasing the number of samples needed to achieve convergence (see Owen [4]__).

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, optional*) – Calculate second-order sensitivities. Default is True.
- **scramble** (*bool, optional*) – If True, use LMS+shift scrambling. Otherwise, no scrambling is done. Default is True.
- **skip_values** (*int, optional*) – Number of points in Sobol' sequence to skip, ideally a value of base 2. It's recommended not to change this value and use *scramble* instead. *scramble* and *skip_values* can be used together. Default is 0.
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. Sobol', I.M., 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, The Second IMACS Seminar on Monte Carlo Methods 55, 271-280. https://doi.org/10.1016/S0378-4754(00)00270-6
2. Saltelli, A. (2002). Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications, 145(2), 280-297. https://doi.org/10.1016/S0010-4655(02)00280-1

3. Campolongo, F., Saltelli, A., Cariboni, J., 2011. From screening to quantitative sensitivity analysis. A unified approach. Computer Physics Communications 182, 978-988. https://doi.org/10.1016/j.cpc.2010.12.039

4. Owen, A. B., 2020. On dropping the first Sobol' point. arXiv:2008.08051 [cs, math, stat]. Available at: http://arxiv.org/abs/2008.08051 (Accessed: 20 April 2021).

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'.

There are several approaches to estimating sensitivity indices. The general approach is described in [1]. The implementation offered here follows [2] for first and total order indices, whereas estimation of second order sensitivities follows [3]. A noteworthy point is the improvement to reduce error rates in sensitivity estimation is introduced in [4].

Notes

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

Examples

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

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)
- **parallel** (*bool*) – Perform analysis in parallel if True
- **n_processors** (*int*) – Number of parallel processes (only used if parallel is True)
- **keep_resamples** (*bool*) – Whether or not to store intermediate resampling results (default False)
- **seed** (*int*) – Seed to generate a random number

References

1. Sobol, I. M. (2001). Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, 55(1-3):271-280, doi:10.1016/S0378-4754(00)00270-6.

2. Saltelli, A., P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola (2010). Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index. Computer Physics Communications, 181(2):259-270, doi:10.1016/j.cpc.2009.09.018.

3. Saltelli, A. (2002). Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications, 145(2):280-297 doi:10.1016/S0010-4655(02)00280-1.

4. Sobol', I. M., Tarantola, S., Gatelli, D., Kucherenko, S. S., & Mauntz, W. (2007). Estimating the approximation error when fixing unessential factors in global sensitivity analysis. Reliability Engineering & System Safety, 92(7), 957-960. https://doi.org/10.1016/j.ress.2006.07.001

3.1.5 Delta Moment-Independent Measure

SALib.analyze.delta.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *seed: int = None*, *y_resamples: int = None*, *method: str = 'all'*) → Dict

Perform Delta Moment-Independent Analysis on model outputs.

Returns a dictionary with keys 'delta', 'delta_conf', 'S1', and 'S1_conf' (first-order sobol indices), 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

Examples

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

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 100)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **y_resamples** (*int, optional*) – Number of samples to use when resampling (bootstrap) (default None)
- **method** (*{"all", "delta", "sobol"}, optional*) – Whether to compute "delta", "sobol" or both ("all") indices (default "all")

References

1. **Borgonovo, E. (2007). "A new uncertainty importance measure."**
Reliability Engineering & System Safety, 92(6):771-784, doi:10.1016/j.ress.2006.04.015.

2. **Plischke, E., E. Borgonovo, and C. L. Smith (2013). "Global** sensitivity measures from given data." European Journal of Operational Research, 226(3):536-550, doi:10.1016/j.ejor.2012.11.047.

3.1.6 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()*

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. **Sobol, I. M. and S. Kucherenko (2009). "Derivative based global** sensitivity measures and their link with global sensitivity indices." Mathematics and Computers in Simulation, 79(10):3009-3017, doi:10.1016/j.matcom.2009.01.023.

3.1.7 Fractional Factorial

SALib.sample.ff.**sample**(*problem*, *seed: int | Generator | None = 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
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

Returns
sample

Return type
`numpy.array`

References

1. Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. http://doi.org/10.1002/9780470725184

`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()*

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)
```

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
- **seed** (*int*) – Seed to generate a random number

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

Return type
dict

References

1. **Saltelli, A., Ratto, M., Andres, T., Campolongo, F.,**
Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. http://doi.org/10.1002/9780470725184

3.1.8 PAWN Sensitivity Analysis

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

Performs PAWN sensitivity analysis.

The PAWN method [1] is a moment-independent approach to Global Sensitivity Analysis (GSA). It is described as producing robust results at relatively low sample sizes (see [2]) for the purpose of factor ranking and screening.

The distribution of model outputs is examined rather than their variation as is typical in other common GSA approaches. The PAWN method further distinguishes itself from other moment-independent approaches by characterizing outputs by their cumulative distribution function (CDF) as opposed to their probability distribution function. As the CDF for a given random variable is typically normally distributed, PAWN can be more appropriately applied when outputs are highly-skewed or multi-modal, for which variance-based methods may produce unreliable results.

PAWN characterizes the relationship between inputs and outputs by quantifying the variation in the output distributions after conditioning an input. A factor is deemed non-influential if distributions coincide at all `S` conditioning intervals. The Kolmogorov-Smirnov statistic is used as a measure of distance between the distributions.

This implementation reports the PAWN index at the min, mean, median, and max across the slides/conditioning intervals as well as the coefficient of variation (`CV`) and standard deviation (`stdev`). The median value is the typically reported value. As the `CV` is (standard deviation / mean), it indicates the level of variability across the slides, with values closer to zero indicating lower variation.

Notes

Compatible with:
all samplers

This implementation ignores all NaNs.

When applied to grouped factors, the analysis is conducted on each factor individually, and the mean of their results are reported.

Examples

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

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; the conditioning intervals (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results

References

1. **Pianosi, F., Wagener, T., 2015.**
   A simple and efficient method for global sensitivity analysis based on cumulative distribution functions. Environmental Modelling & Software 67, 1-11. https://doi.org/10.1016/j.envsoft.2015.01.004

2. **Pianosi, F., Wagener, T., 2018.**
   Distribution-based sensitivity analysis from a generic input-output sample. Environmental Modelling & Software 108, 197-207. https://doi.org/10.1016/j.envsoft.2018.07.019

3. **Baroni, G., Francke, T., 2020.**
   An effective strategy for combining variance- and distribution-based global sensitivity analysis. Environmental Modelling & Software, 134, 104851. https://doi.org/10.1016/j.envsoft.2020.104851

4. **Baroni, G., Francke, T., 2020.**
   GSA-cvd Combining variance- and distribution-based global sensitivity analysis https://github.com/baronig/GSA-cvd

3.1.9 High-Dimensional Model Representation

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

Compute global sensitivity indices using the meta-modeling technique known as High-Dimensional Model Representation (HDMR).

HDMR itself is not a sensitivity analysis method but a surrogate modeling approach. It constructs a map of relationship between sets of high dimensional inputs and output system variables [1]. This I/O relation can be constructed using different basis functions (orthonormal polynomials, splines, etc.). The model decomposition can be expressed as

$$\widehat{y} = \sum_{u \subseteq \{1,2,\ldots,d\}} f_u$$

where u represents any subset including an empty set.

HDMR becomes extremely useful when the computational cost of obtaining sufficient Monte Carlo samples are prohibitive, as may be the case with Sobol's method. It uses least-square regression to reduce the required number of samples and thus the number of function (model) evaluations. Another advantage of this method is that it can account for correlation among the model input. Unlike other variance-based methods, the main effects are the combination of structural (uncorrelated) and correlated contributions.

This method 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

Notes

Compatible with:
all samplers

Sets an *emulate* method allowing re-use of the emulator.

Examples

```
sp = ProblemSpec({
    'names': ['X1', 'X2', 'X3'],
    'bounds': [[-np.pi, np.pi]] * 3,
    # 'groups': ['A', 'B', 'A'],
    'outputs': ['Y']
})

(sp.sample_saltelli(2048)
    .evaluate(Ishigami.evaluate)
    .analyze_hdmr()
)

sp.emulate()
```

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** (*{int, bool, None}*) – Seed to generate a random number

Returns

Si – Sa : Uncorrelated contribution of a term

Sa_conf : Confidence interval of Sa

Sb : Correlated contribution of a term

Sb_conf : Confidence interval of Sb

S

[Total contribution of a particular term] Sum of Sa and Sb, representing first/second/third order sensitivity indices

S_conf : Confidence interval of S

ST : Total contribution of a particular dimension/parameter

ST_conf : Confidence interval of ST

select : Number of selection (F-Test)

Em
[Emulator result set] C1: First order coefficient C2: Second order coefficient C3: Third Order coefficient

Return type
ResultDict,

References

1. Rabitz, H. and Aliş, Ö.F., "General foundations of high dimensional model representations", Journal of Mathematical Chemistry 25, 197-233 (1999) https://doi.org/10.1023/A:1019188517934
2. Genyuan Li, H. Rabitz, P.E. Yelvington, O.O. Oluwole, F. Bacon, C.E. Kolb, and J. Schoendorf, "Global Sensitivity Analysis for Systems with Independent and/or Correlated Inputs", Journal of Physical Chemistry A, Vol. 114 (19), pp. 6022 - 6032, 2010, https://doi.org/10.1021/jp9096919

3.1.10 Regional Sensitivity Analysis

SALib.analyze.rsa.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *bins: int = 20*, *target: str = 'Y'*, *print_to_console: bool = False*, *seed: int = None*)

Perform Regional Sensitivity Analysis (RSA), also known as Monte Carlo Filtering.

In a usual RSA, a desirable region of output space is defined. Outputs which fall within this region is categorized as being "behavioral" (B), and those outside are described as being "non-behavioral" ($\bar{B}$). The input factors are also partitioned into behavioral and non-behavioral subsets, such that $f(X_i|B) \rightarrow (Y|B)$ and $f(X_i|\bar{B}) \rightarrow (Y|\bar{B})$. The distribution between the two sub-samples are compared for each factor. The greater the difference between the two distributions, the more important the given factor is in driving model outputs.

The approach implemented in SALib partitions factor or output space into b bins (default: 20) according to their percentile values. Output space is targeted for analysis by default (`target="Y"`), such that $(Y|b_i)$ is mapped back to $(X_i|b_i)$. In other words, we treat outputs falling within a given bin (b_i) corresponding to their inputs as behavioral, and those outside the bin as non-behavioral. This aids in answering the question "Which X_i contributes most toward a given range of outputs?". Factor space can also be assessed (`target="X"`), such that $f(X_i|b_i) \rightarrow (Y|b_i)$ and $f(X_i|b_{\sim i}) \rightarrow (Y|b_{\sim i})$. This aids in answering the question "where in factor space are outputs most sensitive to?"

The two-sample Cramér-von Mises (CvM) test is used to compare distributions. Results of the analysis indicate sensitivity across factor/output space. As the Cramér-von Mises criterion ranges from 0 to ∞, a value of zero will indicates the two distributions being compared are identical, with larger values indicating greater differences.

Notes

Compatible with:
all samplers

When applied to grouped factors, the analysis is conducted on each factor individually, and the mean of the results for a group are reported.

Increasing the value of `bins` increases the granularity of the analysis (across factor space), but necessitates larger sample sizes.

This analysis will produce NaNs, indicating areas of factor space that did not have any samples, or for which the outputs were constant.

Analysis results are normalized against the maximum value such that 1.0 indicates the greatest sensitivity.

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
- **bins** (*int*) – The number of bins to use (default: 20)
- **target** (*str*) – Assess factor space ("X") or output space ("Y") (default: "Y")
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results Unused, but defined to maintain compatibility.

References

1. **Hornberger, G. M., and R. C. Spear. 1981.** Approach to the preliminary analysis of environmental systems. Journal of Environmental Management 12:1. https://www.osti.gov/biblio/6396608-approach-preliminary-analysis-environmental-systems
2. **Pianosi, F., K. Beven, J. Freer, J. W. Hall, J. Rougier, D. B. Stephenson, and** T. Wagener. 2016. Sensitivity analysis of environmental models: A systematic review with practical workflow. Environmental Modelling & Software 79:214-232. https://dx.doi.org/10.1016/j.envsoft.2016.02.008
3. **Saltelli, A., M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli,** M. Saisana, and S. Tarantola. 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. https://dx.doi.org/10.1002/9780470725184 Accessible at: http://www.andreasaltelli.eu/file/repository/Primer_Corrected_2022.pdf

3.1.11 Discrepancy Sensitivity Indices

SALib.analyze.discrepancy.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *method: str = 'WD'*, *print_to_console: bool = False*, *seed: int = None*)

Discrepancy indices.

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.ndarray*) – An array of model inputs and outputs.
- **Y** (*numpy.ndarray*) – An array of model inputs and outputs.
- **method** (*{"WD", "CD", "MD", "L2-star"}*) – Type of discrepancy. Refer to *scipy.stats.qmc.discrepancy* for more details. Default is "WD".
- **print_to_console** (*bool, optional*) – Print results directly to console (default False)
- **seed** (*int, optional*) – Seed value to ensure deterministic results Unused, but defined to maintain compatibility with other functions.

Notes

Compatible with:
all samplers

Based on 2D sub projections of `[Xi,Y]`, the discrepancy of each sample is calculated which gives a value for all `Xi`. This information is used as a measure of sensitivity.

Discrepancy analysis is very fast and is visually explainable. Considering two variables `X1` and `X2`, `X1` is more influential than `X2` when the scatterplot of `X1` against `Y` displays a more discernible shape than the scatterplot of `X2` against `Y`.

For the method to work properly, the input parameter space need to be uniformly covered as the quality of the measure depends on the value of the discrepancy. Taking a 2D sub projection, if the distribution of sample along `Xi` is not uniform, it will have an impact on the discrepancy, the value will increase, i.e. the importance of this parameter would be inflated.

References

1. A. Puy, P.T. Roy and A. Saltelli. 2023. Discrepancy measures for sensitivity analysis. https://arxiv.org/abs/2206.13470

2. A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. https://dx.doi.org/10.1002/9780470725184 Accessible at: http://www.andreasaltelli.eu/file/repository/Primer_Corrected_2022.pdf

Examples

```
>>> import numpy as np
>>> from SALib.sample import latin
>>> from SALib.analyze import discrepancy
>>> from SALib.test_functions import Ishigami
```

```
>>> problem = {
...   'num_vars': 3,
...   'names': ['x1', 'x2', 'x3'],
...   'bounds': [[-np.pi, np.pi]]*3
... }
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = discrepancy.analyze(problem, X, Y, print_to_console=True)
```

3.2 Developers Guide

3.2.1 Running tests

To run tests, run the following command from the SALib project directory.

```
$ pytest
```

3.2.2 Building documentation locally

Notes here assumes you are at the root of the project directory.

Install dependencies for documentation

```
$ pip install -e .[doc]
```

On *nix

```
$ cd docs
$ make html
```

On Windows

In a command prompt

```
> cd docs
> sphinx-build . ./html
```

3.2.3 Prior to submitting a PR

Run the below to catch any formatting issues.

```
# pre-commit install

pre-commit run --all-files
```

3.3 Changelog

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

3.3.1 [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)

3.3.2 [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)

3.3.3 [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)

3.3.4 [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)

3.3.5 [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

3.3.6 [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

3.3.7 [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.

3.3.8 [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.

3.3.9 [0.6.3]

Parallel option for Sobol method

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

3.3.10 [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

3.3.11 [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

3.3.12 [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.

3.3.13 [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.

3.3.14 [0.3.0]

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

3.3.15 [0.2.0]

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

3.3.16 [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.

3.4 SALib

3.4.1 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: ndarray*, *Y: ndarray*, *num_resamples: int = 100*, *conf_level: float = 0.95*, *print_to_console: bool = False*, *seed: int = None*, *y_resamples: int = None*, *method: str = 'all'*) → Dict

Perform Delta Moment-Independent Analysis on model outputs.

Returns a dictionary with keys ‘delta’, ‘delta_conf’, ‘S1’, and ‘S1_conf’ (first-order sobol indices), 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

Examples

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

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 100)
- **conf_level** (*float*) – The confidence interval level (default 0.95)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **y_resamples** (*int, optional*) – Number of samples to use when resampling (bootstrap) (default None)
- **method** (*{"all", "delta", "sobol"}, optional*) – Whether to compute "delta", "sobol" or both ("all") indices (default "all")

References

1. **Borgonovo, E. (2007). "A new uncertainty importance measure."**
   Reliability Engineering & System Safety, 92(6):771-784, doi:10.1016/j.ress.2006.04.015.
2. **Plischke, E., E. Borgonovo, and C. L. Smith (2013). "Global**
   sensitivity measures from given data." European Journal of Operational Research, 226(3):536-550, doi:10.1016/j.ejor.2012.11.047.

SALib.analyze.delta.**bias_reduced_delta**(*Y*, *Ygrid*, *X*, *m*, *num_resamples*, *conf_level*, *y_resamples*)

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*, *y_resamples*)

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()*

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. **Sobol, I. M. and S. Kucherenko (2009). "Derivative based global** sensitivity measures and their link with global sensitivity indices." Mathematics and Computers in Simulation, 79(10):3009-3017, doi:10.1016/j.matcom.2009.01.023.

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.discrepancy module

SALib.analyze.discrepancy.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *method: str = 'WD'*, *print_to_console: bool = False*, *seed: int = None*)

Discrepancy indices.

Parameters

- **problem** (*dict*) – The problem definition
- **X** (*numpy.ndarray*) – An array of model inputs and outputs.
- **Y** (*numpy.ndarray*) – An array of model inputs and outputs.
- **method** (*{"WD", "CD", "MD", "L2-star"}*) – Type of discrepancy. Refer to *scipy.stats.qmc.discrepancy* for more details. Default is "WD".
- **print_to_console** (*bool, optional*) – Print results directly to console (default False)
- **seed** (*int, optional*) – Seed value to ensure deterministic results Unused, but defined to maintain compatibility with other functions.

Notes

Compatible with:
all samplers

Based on 2D sub projections of `[Xi,Y]`, the discrepancy of each sample is calculated which gives a value for all `Xi`. This information is used as a measure of sensitivity.

Discrepancy analysis is very fast and is visually explainable. Considering two variables `X1` and `X2`, `X1` is more influential than `X2` when the scatterplot of `X1` against `Y` displays a more discernible shape than the scatterplot of `X2` against `Y`.

For the method to work properly, the input parameter space need to be uniformly covered as the quality of the measure depends on the value of the discrepancy. Taking a 2D sub projection, if the distribution of sample along `Xi` is not uniform, it will have an impact on the discrepancy, the value will increase, i.e. the importance of this parameter would be inflated.

References

1. A. Puy, P.T. Roy and A. Saltelli. 2023. Discrepancy measures for sensitivity analysis. https://arxiv.org/abs/2206.13470

2. A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. https://dx.doi.org/10.1002/9780470725184 Accessible at: http://www.andreasaltelli.eu/file/repository/Primer_Corrected_2022.pdf

Examples

```
>>> import numpy as np
>>> from SALib.sample import latin
>>> from SALib.analyze import discrepancy
>>> from SALib.test_functions import Ishigami
```

```
>>> problem = {
...   'num_vars': 3,
...   'names': ['x1', 'x2', 'x3'],
...   'bounds': [[-np.pi, np.pi]]*3
... }
>>> X = latin.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = discrepancy.analyze(problem, X, Y, print_to_console=True)
```

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

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

SALib.analyze.enhanced_hdmr module

SALib.analyze.enhanced_hdmr.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *max_order: int = 2*, *poly_order: int = 3*, *bootstrap: int = 20*, *subset: int = None*, *max_iter: int = 100*, *l2_penalty: float = 0.01*, *alpha: float = 0.95*, *extended_base: bool = True*, *print_to_console: bool = False*, *return_emulator: bool = False*, *seed: int = None*) → Dict

Compute global sensitivity indices using the meta-modeling technique known as High-Dimensional Model Representation (HDMR).

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

SALib.analyze.enhanced_hdmr.**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*)

Perform extended Fourier Amplitude Sensitivity Test 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()*

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. Cukier, R. I., C. M. Fortuin, K. E. Shuler, A. G. Petschek, and J. H. Schaibly (1973). Study of the sensitivity of coupled reaction systems to uncertainties in rate coefficients. J. Chem. Phys., 59(8):3873-3878 doi:10.1063/1.1680571
2. Saltelli, A., S. Tarantola, and K. P.-S. Chan (1999). A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output. Technometrics, 41(1):39-56, doi:10.1080/00401706.1999.10485594.

3. Pujol, G. (2006) fast99 - R *sensitivity* package https://github.com/cran/sensitivity/blob/master/R/fast99.R

SALib.analyze.fast.**bootstrap**(*Y: ndarray*, *M: int*, *resamples: int*, *conf_level: float*)

Compute CIs.

Infers `N` from results of sub-sample `Y` and re-estimates omega () for the above `N`.

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

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

Add method specific options to CLI parser.

Parameters
parser (*argparse object*)

Return type
Updated argparse object

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

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()*

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)
```

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
- **seed** (*int*) – Seed to generate a random number

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

Return type
dict

References

1. **Saltelli, A., Ratto, M., Andres, T., Campolongo, F.,**
Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. http://doi.org/10.1002/9780470725184

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: ndarray*, *Y: ndarray*, *maxorder: int = 2*, *maxiter: int = 100*, *m: int = 2*, *K: int = 20*, *R: int = None*, *alpha: float = 0.95*, *lambdax: float = 0.01*, *print_to_console: bool = False*, *seed: int | bool | None = None*) → Dict
Compute global sensitivity indices using the meta-modeling technique known as High-Dimensional Model Representation (HDMR).

HDMR itself is not a sensitivity analysis method but a surrogate modeling approach. It constructs a map of relationship between sets of high dimensional inputs and output system variables [1]. This I/O relation can be constructed using different basis functions (orthonormal polynomials, splines, etc.). The model decomposition can be expressed as

$$\widehat{y} = \sum_{u \subseteq \{1,2,\ldots,d\}} f_u$$

where u represents any subset including an empty set.

HDMR becomes extremely useful when the computational cost of obtaining sufficient Monte Carlo samples are prohibitive, as may be the case with Sobol's method. It uses least-square regression to reduce the required

number of samples and thus the number of function (model) evaluations. Another advantage of this method is that it can account for correlation among the model input. Unlike other variance-based methods, the main effects are the combination of structural (uncorrelated) and correlated contributions.

This method 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

Notes

Compatible with:
all samplers

Sets an *emulate* method allowing re-use of the emulator.

Examples

```
sp = ProblemSpec({
    'names': ['X1', 'X2', 'X3'],
    'bounds': [[-np.pi, np.pi]] * 3,
    # 'groups': ['A', 'B', 'A'],
    'outputs': ['Y']
})

(sp.sample_saltelli(2048)
    .evaluate(Ishigami.evaluate)
    .analyze_hdmr()
)

sp.emulate()
```

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** (*{int, bool, None}*) – Seed to generate a random number

Returns

Si – Sa : Uncorrelated contribution of a term

Sa_conf : Confidence interval of Sa

Sb : Correlated contribution of a term

Sb_conf : Confidence interval of Sb

S
[Total contribution of a particular term] Sum of Sa and Sb, representing first/second/third order sensitivity indices

S_conf : Confidence interval of S

ST : Total contribution of a particular dimension/parameter

ST_conf : Confidence interval of ST

select : Number of selection (F-Test)

Em
[Emulator result set] C1: First order coefficient C2: Second order coefficient C3: Third Order coefficient

Return type
ResultDict,

References

1. Rabitz, H. and Aliş, Ö.F., "General foundations of high dimensional model representations", Journal of Mathematical Chemistry 25, 197-233 (1999) https://doi.org/10.1023/A:1019188517934
2. Genyuan Li, H. Rabitz, P.E. Yelvington, O.O. Oluwole, F. Bacon, C.E. Kolb, and J. Schoendorf, "Global Sensitivity Analysis for Systems with Independent and/or Correlated Inputs", Journal of Physical Chemistry A, Vol. 114 (19), pp. 6022 - 6032, 2010, https://doi.org/10.1021/jp9096919

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

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

SALib.analyze.morris module

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

Perform Morris Analysis on model outputs.

Returns a result set with keys `mu`, `mu_star`, `sigma`, and `mu_star_conf`, where each entry corresponds to the parameters defined in the problem spec or parameter file.

- `mu` metric indicates the mean of the distribution
- `mu_star` metric indicates the mean of the distribution of absolute values
- `sigma` is the standard deviation of the distribution

When `scaled` is True, the elementary effects are scaled by the ratio of standard deviation of `X` and `Y` according to [3]. When using this option it is important to ensure that `X` contains the actual values passed into the model, as the elementary effects are divided by the step calculated from `X` rather than using *delta* which is calculated from

the number of levels used in the sample. This could be the case if you perform post-processing on the values before passing them to the model.

Scaled elementary effects are useful when comparing different model outputs with each other when the input and output parameters have different scales. The ranking between the ordinary elementary effects and the scaled should be the same.

Notes

When applied with groups, the `mu` metric is less reliable as the effect from parameters within a group become averaged out.

The `mu_star` metric avoids this issue as it indicates the mean of the absolute values. If the direction of effects is important, Campolongo et al., [2] suggest comparing `mu_star` with `mu`. If `mu` is low and `mu_star` is high, then the effects are of different signs.

`sigma` is used as an indicator of interactions between parameters, or groups of parameters.

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

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)
```

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
- **scaled** (*bool, default=False*) – If True, the elementary effects are scaled by the ratio of standard deviation of X and Y according to [3]
- **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 100)
- **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, None, np.random.Generator}*) – 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

1. Morris, M. (1991). Factorial Sampling Plans for Preliminary Computational Experiments. Technometrics, 33(2):161-174, doi:10.1080/00401706.1991.10484804.
2. Campolongo, F., J. Cariboni, and A. Saltelli (2007). An effective screening design for sensitivity analysis of large models. Environmental Modelling & Software, 22(10):1509-1518, doi:10.1016/j.envsoft.2006.10.004.
3. Sin and Gearney (2009) Improving the Morris Method for Sensitivity Analysis by Scaling the Elementary Effects. 19th European Symposium on Computer Aided Process Engineering ESCAPE19:925-930
4. **Moret et al. (2017)**
   Characterization of input uncertainties in strategic energy planning models. Applied Energy, Volume 202, 15 September 2017, Pages 597-617 https://doi.org/10.1016/j.apenergy.2017.05.106

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

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

SALib.analyze.pawn module

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

Performs PAWN sensitivity analysis.

The PAWN method [1] is a moment-independent approach to Global Sensitivity Analysis (GSA). It is described as producing robust results at relatively low sample sizes (see [2]) for the purpose of factor ranking and screening.

The distribution of model outputs is examined rather than their variation as is typical in other common GSA approaches. The PAWN method further distinguishes itself from other moment-independent approaches by characterizing outputs by their cumulative distribution function (CDF) as opposed to their probability distribution function. As the CDF for a given random variable is typically normally distributed, PAWN can be more appropriately applied when outputs are highly-skewed or multi-modal, for which variance-based methods may produce unreliable results.

PAWN characterizes the relationship between inputs and outputs by quantifying the variation in the output distributions after conditioning an input. A factor is deemed non-influential if distributions coincide at all `S` conditioning intervals. The Kolmogorov-Smirnov statistic is used as a measure of distance between the distributions.

This implementation reports the PAWN index at the min, mean, median, and max across the slides/conditioning intervals as well as the coefficient of variation (`CV`) and standard deviation (`stdev`). The median value is the typically reported value. As the `CV` is (standard deviation / mean), it indicates the level of variability across the slides, with values closer to zero indicating lower variation.

Notes

Compatible with:
all samplers

This implementation ignores all NaNs.

When applied to grouped factors, the analysis is conducted on each factor individually, and the mean of their results are reported.

Examples

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

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; the conditioning intervals (default 10)
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results

References

1. **Pianosi, F., Wagener, T., 2015.**
   A simple and efficient method for global sensitivity analysis based on cumulative distribution functions. Environmental Modelling & Software 67, 1-11. https://doi.org/10.1016/j.envsoft.2015.01.004
2. **Pianosi, F., Wagener, T., 2018.**
   Distribution-based sensitivity analysis from a generic input-output sample. Environmental Modelling & Software 108, 197-207. https://doi.org/10.1016/j.envsoft.2018.07.019
3. **Baroni, G., Francke, T., 2020.**
   An effective strategy for combining variance- and distribution-based global sensitivity analysis. Environmental Modelling & Software, 134, 104851. https://doi.org/10.1016/j.envsoft.2020.104851
4. **Baroni, G., Francke, T., 2020.**
   GSA-cvd Combining variance- and distribution-based global sensitivity analysis https://github.com/baronig/GSA-cvd

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: int | Generator | None = 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

Examples

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

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)
- **seed** (*int*) – Seed to generate a random number

References

1. S. Tarantola, D. Gatelli and T. Mara (2006) Random Balance Designs for the Estimation of First Order Global Sensitivity Indices, Reliability Engineering and System Safety, 91:6, 717-727 https://doi.org/10.1016/j.ress.2005.06.003
2. **Elmar Plischke (2010)**
   An effective algorithm for computing global sensitivity indices (EASI), Reliability Engineering & System Safety, 95:4, 354-360. doi:10.1016/j.ress.2009.11.005
3. **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
4. **Jeanne Goffart, Mickael Rabouille & Nathan Mendes (2015)**
   Uncertainty and sensitivity analysis applied to hygrothermal simulation of a brick building in a hot and humid climate, Journal of Building Performance Simulation. doi:10.1080/19401493.2015.1112430

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

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 sensitivity indices (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.rsa module

SALib.analyze.rsa.**analyze**(*problem: Dict*, *X: ndarray*, *Y: ndarray*, *bins: int = 20*, *target: str = 'Y'*, *print_to_console: bool = False*, *seed: int = None*)

Perform Regional Sensitivity Analysis (RSA), also known as Monte Carlo Filtering.

In a usual RSA, a desirable region of output space is defined. Outputs which fall within this region is categorized as being "behavioral" (B), and those outside are described as being "non-behavioral" ($\bar{B}$). The input factors are also partitioned into behavioral and non-behavioral subsets, such that $f(X_i|B) \rightarrow (Y|B)$ and $f(X_i|\bar{B}) \rightarrow (Y|\bar{B})$. The distribution between the two sub-samples are compared for each factor. The greater the difference between the two distributions, the more important the given factor is in driving model outputs.

The approach implemented in SALib partitions factor or output space into b bins (default: 20) according to their percentile values. Output space is targeted for analysis by default (`target="Y"`), such that $(Y|b_i)$ is mapped back to $(X_i|b_i)$. In other words, we treat outputs falling within a given bin (b_i) corresponding to their inputs as behavioral, and those outside the bin as non-behavioral. This aids in answering the question "Which X_i contributes most toward a given range of outputs?". Factor space can also be assessed (`target="X"`), such that $f(X_i|b_i) \rightarrow (Y|b_i)$ and $f(X_i|b_{\sim i}) \rightarrow (Y|b_{\sim i})$. This aids in answering the question "where in factor space are outputs most sensitive to?"

The two-sample Cramér-von Mises (CvM) test is used to compare distributions. Results of the analysis indicate sensitivity across factor/output space. As the Cramér-von Mises criterion ranges from 0 to ∞, a value of zero will indicates the two distributions being compared are identical, with larger values indicating greater differences.

Notes

Compatible with:
all samplers

When applied to grouped factors, the analysis is conducted on each factor individually, and the mean of the results for a group are reported.

Increasing the value of `bins` increases the granularity of the analysis (across factor space), but necessitates larger sample sizes.

This analysis will produce NaNs, indicating areas of factor space that did not have any samples, or for which the outputs were constant.

Analysis results are normalized against the maximum value such that 1.0 indicates the greatest sensitivity.

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
- **bins** (*int*) – The number of bins to use (default: 20)
- **target** (*str*) – Assess factor space ("X") or output space ("Y") (default: "Y")
- **print_to_console** (*bool*) – Print results directly to console (default False)
- **seed** (*int*) – Seed value to ensure deterministic results Unused, but defined to maintain compatibility.

References

1. **Hornberger, G. M., and R. C. Spear. 1981.** Approach to the preliminary analysis of environmental systems. Journal of Environmental Management 12:1. https://www.osti.gov/biblio/6396608-approach-preliminary-analysis-environmental-systems
2. **Pianosi, F., K. Beven, J. Freer, J. W. Hall, J. Rougier, D. B. Stephenson, and** T. Wagener. 2016. Sensitivity analysis of environmental models: A systematic review with practical workflow. Environmental Modelling & Software 79:214-232. https://dx.doi.org/10.1016/j.envsoft.2016.02.008
3. **Saltelli, A., M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli,** M. Saisana, and S. Tarantola. 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. https://dx.doi.org/10.1002/9780470725184 Accessible at: http://www.andreasaltelli.eu/file/repository/Primer_Corrected_2022.pdf

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

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

SALib.analyze.rsa.**plot**(*self*, *factors=None*)

Plotting for Regional Sensitivity Analysis.

Overrides the plot method attached to the SALib problem spec.

SALib.analyze.rsa.**rsa**(*X: ndarray*, *y: ndarray*, *bins: int = 10*, *target='X'*) → ndarray

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

Conversion to Pandas DataFrame specific to Regional Sensitivity Analysis results.

Overrides the conversion method attached to the SALib problem spec.

Return type
Pandas DataFrame

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.

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)
```

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.

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'.

There are several approaches to estimating sensitivity indices. The general approach is described in [1]. The implementation offered here follows [2] for first and total order indices, whereas estimation of second order sensitivities follows [3]. A noteworthy point is the improvement to reduce error rates in sensitivity estimation is introduced in [4].

Notes

Compatible with:

saltelli : *SALib.sample.saltelli.sample()* *sobol* : *SALib.sample.sobol.sample()*

Examples

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

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)
- **parallel** (*bool*) – Perform analysis in parallel if True
- **n_processors** (*int*) – Number of parallel processes (only used if parallel is True)
- **keep_resamples** (*bool*) – Whether or not to store intermediate resampling results (default False)
- **seed** (*int*) – Seed to generate a random number

References

1. Sobol, I. M. (2001). Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, 55(1-3):271-280, doi:10.1016/S0378-4754(00)00270-6.
2. Saltelli, A., P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola (2010). Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index. Computer Physics Communications, 181(2):259-270, doi:10.1016/j.cpc.2009.09.018.
3. Saltelli, A. (2002). Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications, 145(2):280-297 doi:10.1016/S0010-4655(02)00280-1.
4. Sobol', I. M., Tarantola, S., Gatelli, D., Kucherenko, S. S., & Mauntz, W. (2007). Estimating the approximation error when fixing unessential factors in global sensitivity analysis. Reliability Engineering & System Safety, 92(7), 957-960. https://doi.org/10.1016/j.ress.2006.07.001

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

Examples

```
>>> 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.

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)
```

Parameters

Si_df (*)

Returns

*** ax**

Return type

matplotlib axes object

SALib.plotting.hdmr module

Created on Dec 20, 2019

@author: @sahin-abdullah

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

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

SALib.plotting.heatmap module

SALib.plotting.heatmap.**heatmap**(*sp: Dict*, *metric: str*, *index: str*, *title: str = None*, *ax=None*)

Plot a heatmap of the target metric.

Parameters

- **sp** (*object, SALib ProblemSpec*)
- **metric** (str, metric to plot. Defaults to first metric/result output if *None*.)
- **index** (str, sensitivity indices to plot ('S1', 'ST', etc). Displays all if *None*.)
- **title** (*str, plot title to use*)

- **ax** (*axes object, matplotlib axes object to assign figure to.*)

Returns
ax

Return type
matplotlib axes object

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: *Strategy*

Implements the brute force optimisation strategy

Methods

brute_force_most_distant(input_sample, ...)	Use brute force method to find most distant trajectories
check_input_sample(input_sample, num_params, ...)	Check the *input_sample* is valid
compile_output(input_sample, num_samples, ...)	Picks the trajectories from the input
compute_distance(m, l)	Compute distance between two trajectories
compute_distance_matrix(input_sample, ...[, ...])	Computes the distance between each and every trajectory
find_maximum(scores, N, k_choices)	Finds the *k_choices* maximum scores from *scores*
find_most_distant(input_sample, num_samples, ...)	Finds the 'k_choices' most distant choices from the 'num_samples' trajectories contained in 'input_sample'
mappable(combos, pairwise, distance_matrix)	Obtains scores from the distance_matrix for each pairwise combination held in the combos array
nth(iterable, n[, default])	Returns the nth item or a default value
run_checks(number_samples, k_choices)	Runs checks on *k_choices*
sample(input_sample, num_samples, ...[, ...])	Computes the optimum k_choices of trajectories from the input_sample.

grouper

brute_force_most_distant(*input_sample: ndarray*, *num_samples: int*, *num_params: int*, *k_choices: int*, *num_groups: 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

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*)

Return type
list

find_most_distant(*input_sample: ndarray*, *num_samples: int*, *num_params: int*, *k_choices: int*, *num_groups: int = None*) → 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

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: *Strategy*

Implements the local optimisation algorithm using the Strategy interface

Methods

add_indices(indices, distance_matrix)	Adds extra indices for the combinatorial problem.
check_input_sample(input_sample, num_params, ...)	Check the *input_sample* is valid
compile_output(input_sample, num_samples, ...)	Picks the trajectories from the input
compute_distance(m, l)	Compute distance between two trajectories
compute_distance_matrix(input_sample, ...[, ...])	Computes the distance between each and every trajectory
find_local_maximum(input_sample, N, ...[, ...])	Find the most different trajectories in the input sample using a local approach

continues on next page

Table 2 – continued from previous page

get_max_sum_ind(indices_list, distances, i, m)	Get the indices that belong to the maximum distance in *distances*
run_checks(number_samples, k_choices)	Runs checks on *k_choices*
sample(input_sample, num_samples, ...[, ...])	Computes the optimum k_choices of trajectories from the input_sample.
sum_distances(indices, distance_matrix)	Calculate combinatorial distance between a select group of trajectories, indicated by indices

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

Adds extra indices for the combinatorial problem.

Parameters

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

Examples

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

find_local_maximum(*input_sample: ndarray*, *N: int*, *num_params: int*, *k_choices: int*, *num_groups: 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

Return type
list

get_max_sum_ind(*indices_list: List[Tuple]*, *distances: ndarray*, *i: str | int*, *m: 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*)

Return type
list

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

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

Parameters

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

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

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

Generate model inputs using the Method of Morris.

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

- Vanilla Morris (see [1]) when `optimal_trajectories` is `None`/`False` and `local_optimization` is `False`
- **Optimised trajectories when `optimal_trajectories=True` using**
  Campolongo's enhancements (see [2]) and optionally Ruano's enhancement (see [3]) when `local_optimization=True`
- Morris with groups when the problem definition specifies groups of parameters

Results from these model inputs are intended to be used with *SALib.analyze.morris.analyze()*.

Notes

Campolongo et al., [2] introduces an optimal trajectories approach which attempts to maximize the parameter space scanned for a given number of trajectories (where *optimal_trajectories* $\in 2, ..., N$). The approach accomplishes this aim by randomly generating a high number of possible trajectories (500 to 1000 in [2]) and selecting a subset of `r` trajectories which have the highest spread in parameter space. The `r` variable in [2] corresponds to the `optimal_trajectories` parameter here.

Calculating all possible combinations of trajectories can be computationally expensive. 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 levels from a pool of 100 samples with this "brute force" approach.

Ruano et al., [3] proposed an alternative approach with an iterative process that maximizes the distance between subgroups of generated trajectories, from which the final set of trajectories are selected, again maximizing the distance between each. The approach is not guaranteed to produce the most optimal spread of trajectories, but are at least locally maximized and significantly reduce the time taken to select trajectories. With `local_optimization = True` (which is default), it is possible to go higher than the previously suggested 4 levels from a pool of 100 samples.

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** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

Returns

sample_morris – Array 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, 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.

Return type

np.ndarray

References

1. Morris, M.D., 1991. Factorial Sampling Plans for Preliminary Computational Experiments. Technometrics 33, 161-174. https://doi.org/10.1080/00401706.1991.10484804
2. Campolongo, F., Cariboni, J., & Saltelli, A. 2007. An effective screening design for sensitivity analysis of large models. Environmental Modelling & Software, 22(10), 1509-1518. https://doi.org/10.1016/j.envsoft.2006.10.004
3. Ruano, M.V., Ribes, J., Seco, A., Ferrer, J., 2012. An improved sampling strategy based on trajectory design for application of the Morris method to systems with many input factors. Environmental Modelling & Software 37, 103-109. https://doi.org/10.1016/j.envsoft.2012.03.008

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*)

Methods

sample(input_sample, num_samples, ...)	Computes the optimum k_choices of trajectories from the input_sample.

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.

Methods

check_input_sample(input_sample, num_params, ...)	Check the *input_sample* is valid
compile_output(input_sample, num_samples, ...)	Picks the trajectories from the input
compute_distance(m, l)	Compute distance between two trajectories
compute_distance_matrix(input_sample, ...[, ...])	Computes the distance between each and every trajectory
run_checks(number_samples, k_choices)	Runs checks on *k_choices*
sample(input_sample, num_samples, ...[, ...])	Computes the optimum k_choices of trajectories from the input_sample.

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*)

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

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

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

Return type
argparse object

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

Add common sampling options to CLI parser.

Parameters
parser (*argparse object*)

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*)

Return type
Updated argparse object

SALib.sample.fast_sampler.**sample**(*problem*, *N*, *M=4*, *seed: int | Generator | None = None*)

Generate model inputs for extended Fourier Amplitude Sensitivity Test.

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)
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. Cukier, R.I., Fortuin, C.M., Shuler, K.E., Petschek, A.G., Schaibly, J.H., 1973. Study of the sensitivity of coupled reaction systems to uncertainties in rate coefficients. I theory. Journal of Chemical Physics 59, 3873-3878. https://doi.org/10.1063/1.1680571
2. Saltelli, A., S. Tarantola, and K. P.-S. Chan (1999). A Quantitative Model-Independent Method for Global Sensitivity Analysis of Model Output. Technometrics, 41(1):39-56, doi:10.1080/00401706.1999.10485594.

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: int | Generator | None = 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
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

Returns
sample

Return type
`numpy.array`

References

1. Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. http://doi.org/10.1002/9780470725184

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*)

Return type
Updated argparse object

SALib.sample.finite_diff.**sample**(*problem: Dict*, *N: int*, *delta: float = 0.01*, *seed: int | Generator | None = None*, *skip_values: int = 1024*) → 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** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.
- **skip_values** (*int*) – How many values of the Sobol sequence to skip

Returns
np.array

Return type
DGSM sequence

References

1. Sobol', I.M., Kucherenko, S., 2009. Derivative based global sensitivity measures and their link with global sensitivity indices. Mathematics and Computers in Simulation 79, 3009-3017. https://doi.org/10.1016/j.matcom.2009.01.023
2. Sobol', I.M., Kucherenko, S., 2010. Derivative based global sensitivity measures. Procedia - Social and Behavioral Sciences 2, 7745-7746. https://doi.org/10.1016/j.sbspro.2010.05.208

SALib.sample.latin module

SALib.sample.latin.**cli_action**(*args*)

Run sampling method

Parameters
args (*argparse namespace*)

SALib.sample.latin.**sample**(*problem*, *N*, *seed: int | Generator | None = 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
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. **McKay, M.D., Beckman, R.J., Conover, W.J., 1979.**
   A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. Technometrics 21, 239-245. https://doi.org/10.2307/1268522
2. **Iman, R.L., Helton, J.C., Campbell, J.E., 1981.**
   An Approach to Sensitivity Analysis of Computer Models: Part I—Introduction, Input Variable Selection and Preliminary Variable Assessment. Journal of Quality Technology 13, 174-183. https://doi.org/10.1080/00224065.1981.11978748

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*)

Return type
Updated argparse object

`SALib.sample.saltelli.`**sample**(*problem: Dict*, *N: int*, *calc_second_order: bool = True*, *skip_values: 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()*.

Deprecated since version 1.4.6.

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 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

1. **Campolongo, F., Saltelli, A., Cariboni, J., 2011.**
   From screening to quantitative sensitivity analysis. A unified approach. Computer Physics Communications 182, 978-988. https://doi.org/10.1016/j.cpc.2010.12.039
2. **Owen, A. B., 2020.**
   On dropping the first Sobol' point. arXiv:2008.08051 [cs, math, stat]. Available at: http://arxiv.org/abs/2008.08051 (Accessed: 20 April 2021).
3. **Saltelli, A., 2002.**
   Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications 145, 280-297. https://doi.org/10.1016/S0010-4655(02)00280-1
4. **Sobol', I.M., 2001.**
   Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, The Second IMACS Seminar on Monte Carlo Methods 55, 271-280. https://doi.org/10.1016/S0378-4754(00)00270-6
5. **Discussion: https://github.com/scipy/scipy/pull/10844**
   https://github.com/scipy/scipy/pull/10844#issuecomment-672186615 https://github.com/scipy/scipy/pull/10844#issuecomment-673029539

SALib.sample.sobol module

`SALib.sample.sobol.cli_action(args)`

Run sampling method

Parameters
args (*argparse namespace*)

`SALib.sample.sobol.cli_parse(parser)`

Add method specific options to CLI parser.

Parameters
parser (*argparse object*)

Return type
Updated argparse object

SALib.sample.sobol.**sample**(*problem: Dict*, *N: int*, ***, *calc_second_order: bool = True*, *scramble: bool = True*, *skip_values: int = 0*, *seed: int | Generator | None = 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. The general approach is described in [1].

Returns a NumPy matrix containing the model inputs using Saltelli's sampling scheme.

Saltelli's scheme reduces the number of required model runs from `N(2D+1)` to `N(D+1)` (see [2]).

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 [3]__), which can be avoided by scrambling the sequence.

Another option, not recommended and available for educational purposes, is to use the *skip_values* parameter. Skipping values reportedly improves the uniformity of samples. But, it has been shown that naively skipping values may reduce accuracy, increasing the number of samples needed to achieve convergence (see Owen [4]__).

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, optional*) – Calculate second-order sensitivities. Default is True.
- **scramble** (*bool, optional*) – If True, use LMS+shift scrambling. Otherwise, no scrambling is done. Default is True.
- **skip_values** (*int, optional*) – Number of points in Sobol' sequence to skip, ideally a value of base 2. It's recommended not to change this value and use *scramble* instead. *scramble* and *skip_values* can be used together. Default is 0.
- **seed** ({None, int, *numpy.random.Generator*}, optional) – If *seed* is None the *numpy.random.Generator* generator is used. If *seed* is an int, a new `Generator` instance is used, seeded with *seed*. If *seed* is already a `Generator` instance then that instance is used. Default is None.

References

1. Sobol', I.M., 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation, The Second IMACS Seminar on Monte Carlo Methods 55, 271-280. https://doi.org/10.1016/S0378-4754(00)00270-6
2. Saltelli, A. (2002). Making best use of model evaluations to compute sensitivity indices. Computer Physics Communications, 145(2), 280-297. https://doi.org/10.1016/S0010-4655(02)00280-1
3. Campolongo, F., Saltelli, A., Cariboni, J., 2011. From screening to quantitative sensitivity analysis. A unified approach. Computer Physics Communications 182, 978-988. https://doi.org/10.1016/j.cpc.2010.12.039
4. Owen, A. B., 2020. On dropping the first Sobol' point. arXiv:2008.08051 [cs, math, stat]. Available at: http://arxiv.org/abs/2008.08051 (Accessed: 20 April 2021).

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: ndarray*, *A: float = 7.0*, *B: float = 0.1*) → 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: 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: ndarray*, *seed=101*) → 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]

Return type

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

SALib.test_functions.lake_problem.**lake_problem**(*X: float | array*, *a: float | array = 0.1*, *q: float | array = 2.0*, *b: float | array = 0.42*, *eps: float | 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]

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: ndarray*, *A: ndarray*, *M: ndarray*) → 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.

samples

Type
np.array, of generated samples

results

Type
np.array, of evaluations (i.e., model outputs)

analysis

Type
np.array or dict, of sensitivity indices

Attributes

analysis
results
samples

Methods

analyze(func, *args, **kwargs)	Analyze sampled results using given function.
analyze_parallel(func, *args[, nprocs])	Analyze sampled results using the given function in parallel.
clear(/)	Remove all items from the dict.
copy(/)	Return a shallow copy of the dict.
evaluate(func, *args, **kwargs)	Evaluate a given model.
evaluate_distributed(func, *args[, nprocs, ...])	Distribute model evaluation across a cluster.
evaluate_parallel(func, *args[, nprocs])	Evaluate model locally in parallel.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.

continues on next page

Table 5 – continued from previous page

heatmap([metric, index, title, ax])	Plot results as a heatmap.
items(/)	Return a set-like object providing a view on the dict's items.
keys(/)	Return a set-like object providing a view on the dict's keys.
plot(**kwargs)	Plot results as a bar chart.
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
sample(func, *args, **kwargs)	Create sample using given function.
set_results(results)	Set previously available model results.
set_samples(samples)	Set previous samples used.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
to_df()	Convert results to Pandas DataFrame.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E.keys(): D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values(/)	Return an object providing a view on the dict's values.

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

heatmap(*metric: str = None*, *index: str = None*, *title: str = None*, *ax=None*)

Plot results as a heatmap.

Parameters

- **metric** (str or None, name of output to analyze (display all if *None*))
- **index** (*str or None, name of index to plot, dependent on what*) – analysis was conducted (ST, S1, etc; displays all if *None*)
- **title** (str, title of plot to use (defaults to the same as *metric*))
- **ax** (*axes object, matplotlib axes object to use for plot.*) – Creates a new figure if not provided.

Returns
ax

Return type
matplotlib axes object

plot(***kwargs*)

Plot results as a bar chart.

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: ndarray*)

Set previously available model results.

set_samples(*samples: 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

Methods

clear(/)	Remove all items from the dict.
copy(/)	Return a shallow copy of the dict.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items(/)	Return a set-like object providing a view on the dict's items.
keys(/)	Return a set-like object providing a view on the dict's keys.
plot([ax])	Create bar chart of results
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
to_df()	Convert dict structure into Pandas DataFrame.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E.keys(): D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values(/)	Return an object providing a view on the dict's values.

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.**handle_seed**(*seed: Generator | BitGenerator | SeedSequence | int | Sequence[int] | None*) → Generator

Set (or create) a random number generator.

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

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.**handle_seed**(*seed: Generator | BitGenerator | SeedSequence | int | Sequence[int] | None*) → Generator

Set (or create) a random number generator.

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: 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

Return type
np.ndarray, scaled samples

Module contents

class SALib.**ProblemSpec**(**args*, ***kwargs*)

Bases: dict

Dictionary-like object representing an SALib Problem specification.

samples

Type
np.array, of generated samples

results

Type
np.array, of evaluations (i.e., model outputs)

analysis

Type
np.array or dict, of sensitivity indices

Attributes

analysis
results
samples

Methods

analyze(func, *args, **kwargs)	Analyze sampled results using given function.
analyze_parallel(func, *args[, nprocs])	Analyze sampled results using the given function in parallel.
clear(/)	Remove all items from the dict.
copy(/)	Return a shallow copy of the dict.
evaluate(func, *args, **kwargs)	Evaluate a given model.
evaluate_distributed(func, *args[, nprocs, ...])	Distribute model evaluation across a cluster.
evaluate_parallel(func, *args[, nprocs])	Evaluate model locally in parallel.
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
heatmap([metric, index, title, ax])	Plot results as a heatmap.
items(/)	Return a set-like object providing a view on the dict's items.
keys(/)	Return a set-like object providing a view on the dict's keys.
plot(**kwargs)	Plot results as a bar chart.
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
sample(func, *args, **kwargs)	Create sample using given function.
set_results(results)	Set previously available model results.
set_samples(samples)	Set previous samples used.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
to_df()	Convert results to Pandas DataFrame.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E.keys(): D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

continues on next page

Table 7 – continued from previous page

values(/)	Return an object providing a view on the dict's values.

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

heatmap(*metric: str = None*, *index: str = None*, *title: str = None*, *ax=None*)

Plot results as a heatmap.

Parameters

- **metric** (str or None, name of output to analyze (display all if *None*))

- **index** (*str or None, name of index to plot, dependent on what*) – analysis was conducted (ST, S1, etc; displays all if *None*)
- **title** (str, title of plot to use (defaults to the same as *metric*))
- **ax** (*axes object, matplotlib axes object to use for plot.*) – Creates a new figure if not provided.

Returns
ax

Return type
matplotlib axes object

plot(***kwargs*)

Plot results as a bar chart.

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: ndarray*)

Set previously available model results.

set_samples(*samples: ndarray*)

Set previous samples used.

to_df()

Convert results to Pandas DataFrame.

CHAPTER

FOUR

OTHER INFO

4.1 License

The MIT License (MIT)

4.2 Developers

- Jon Herman
- Will Usher
- Chris Mutel
- Bernardo Trindade
- Dave Hadka
- Matt Woodruff
- Fernando Rios
- Dan Hyams
- xantares
- Abdullah Sahin <sahina@uci.edu>
- Takuya Iwanaga
- Pamphile T. Roy

4.3 How to cite SAlib

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

> Iwanaga, T., Usher, W., & Herman, J. (2022). Toward SALib 2.0: Advancing the accessibility and interpretability of global sensitivity analyses. Socio-Environmental Systems Modelling, 4, 18155. doi:10.18174/sesmo.18155
>
> 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

If you use BibTeX, cite using the following entries:

```
@article{Iwanaga2022,
  title = {Toward {SALib} 2.0: {Advancing} the accessibility and interpretability of␣
↪global sensitivity analyses},
  volume = {4},
  url = {https://sesmo.org/article/view/18155},
  doi = {10.18174/sesmo.18155},
  journal = {Socio-Environmental Systems Modelling},
  author = {Iwanaga, Takuya and Usher, William and Herman, Jonathan},
  month = may,
  year = {2022},
  pages = {18155},
}

@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}
}
```

4.4 Projects that use SALib

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

4.4.1 Software

- The City Energy Analyst
- pynoddy
- savvy
- rhodium
- pySur
- EMA workbench

- Brain/Circulation Model Developer
- DAE Tools
- agentpy
- uncertainpy
- CLIMADA
- parsac

4.4.2 Blogs

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

4.4.3 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.

BIBLIOGRAPHY

[2] Elmar Plischke (2010) "An effective algorithm for computing global sensitivity indices (EASI) Reliability Engineering & System Safety", 95:4, 354-360. doi:10.1016/j.ress.2009.11.005

[1] Ishigami, T., Homma, T., 1990. An importance quantification technique in uncertainty analysis for computer models. Proceedings. First International Symposium on Uncertainty Modeling and Analysis. https://doi.org/10.1109/ISUMA.1990.151285

[2] Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis: The Primer. Wiley, West Sussex, U.K. https://dx.doi.org/10.1002/9780470725184

[1] Saltelli, A., Annoni, P., Azzini, I., Campolongo, F., Ratto, M., Tarantola, S., 2010. Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index. Computer Physics Communications 181, 259–270. https://doi.org/10.1016/j.cpc.2009.09.018

[1] Hadka, D., Herman, J., Reed, P., Keller, K., (2015). "An open source framework for many-objective robust decision making." Environmental Modelling & Software 74, 114–129. doi:10.1016/j.envsoft.2015.07.014

[2] Singh, R., Reed, P., Keller, K., (2015). "Many-objective robust decision making for managing an ecosystem with a deeply uncertain threshold response." Ecology and Society 20. doi:10.5751/ES-07687-200312

[1] Hadka, D., Herman, J., Reed, P., Keller, K., (2015). "An open source framework for many-objective robust decision making." Environmental Modelling & Software 74, 114-129. doi:10.1016/j.envsoft.2015.07.014

[2] Kwakkel, J.H, (2017). "The Exploratory Modeling Workbench: An open source toolkit for exploratory modeling, scenario discovery, and (multi-objective) robust decision making." Environmental Modelling & Software 96, 239-250. doi:10.1016/j.envsoft.2017.06.054

[3] Singh, R., Reed, P., Keller, K., (2015). "Many-objective robust decision making for managing an ecosystem with a deeply uncertain threshold response." Ecology and Society 20. doi:10.5751/ES-07687-200312

[1] Genyuan Li, H. Rabitz, P.E. Yelvington, O.O. Oluwole, F. Bacon, C.E. Kolb, and J. Schoendorf, "Global Sensitivity Analysis for Systems with Independent and/or Correlated Inputs", Journal of Physical Chemistry A, Vol. 114 (19), pp. 6022 - 6032, 2010, https://doi.org/10.1021/jp9096919

[1] Genyuan Li, H. Rabitz, P.E. Yelvington, O.O. Oluwole, F. Bacon, C.E. Kolb, and J. Schoendorf, "Global Sensitivity Analysis for Systems with Independent and/or Correlated Inputs", Journal of Physical Chemistry A, Vol. 114 (19), pp. 6022 - 6032, 2010, https://doi.org/10.1021/jp9096919

[1] Oakley, J.E., O'Hagan, A., 2004. Probabilistic sensitivity analysis of complex models: a Bayesian approach. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 66, 751–769. https://doi.org/10.1111/j.1467-9868.2004.05304.x

PYTHON MODULE INDEX

S

INDEX

E

F

G

H

I

L

M

N

P

R

S

T

U