SALib Documentation

Release 1.3.11.post0.dev64+g2689340

Jon Herman, Will Usher and others

Jul 13, 2020
## Contents

1 Supported Methods ........................................... 3
  1.1 Getting Started ............................................ 3
  1.2 Basics .................................................... 5
  1.3 Concise API Reference ...................................... 7
  1.4 License .................................................... 13
  1.5 Developers ............................................... 13
  1.6 Projects that use SALib .................................... 14
  1.7 Changelog .................................................. 15
  1.8 SALib ...................................................... 18
  1.9 Indices and tables ......................................... 38

Python Module Index ............................................ 39

Index ............................................................ 41
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 1

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](https://hal.archives-ouvertes.fr/hal-01065897/file/Tarantola06RESS_HAL.pdf), [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])

1.1 Getting Started

1.1.1 Installing SALib

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

```
pip install SALib
```

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

```
git clone https://github.com/SALib/SALib.git
cd SALib
python setup.py develop
```
1.1.2 Installing Prerequisite Software

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

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

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

1.1.3 Testing Installation

To test your installation of SALib, run the following command

```
pytest
```

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

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

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

# Generate samples
param_values = saltelli.sample(problem, 1000)

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

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

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

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

Chapter 1. Supported Methods
1.2 Basics

1.2.1 What is Sensitivity Analysis?

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

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

1.2.2 What is SALib?

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

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

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

1.2.3 An Example

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

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

Importing SALib

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

Defining the Model Inputs

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

```python
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:

```python
param_values = saltelli.sample(problem, 1000)
```

Here, \texttt{param_values} is a NumPy matrix. If we run \texttt{param_values.shape}, we see that the matrix is 8000 by 3. The Saltelli sampler generated 8000 samples. The Saltelli sampler generates \( N \times (2D + 2) \) samples, where in this example \( N \) is 1000 (the argument we supplied) and \( D \) is 3 (the number of model inputs). The keyword argument \texttt{calc_second_order=False} will exclude second-order indices, resulting in a smaller sample matrix with \( N \times (D + 2) \) 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:

```python
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:

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

Each line in \texttt{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:

```python
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:
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.

```python
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.

```python
print Si['S1']
[ 0.30644324  0.44776661 -0.00104936 ]
```

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

```python
print Si['ST']
[ 0.56013728  0.4387225  0.24284474]
```

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:

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

```python
x1-x2: 0.0155279
x1-x3: 0.25484902
x2-x3: -0.00995392
```

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.

### 1.3 Concise API Reference

This page documents the sensitivity analysis methods supported by SALib.

#### 1.3.1 FAST - Fourier Amplitude Sensitivity Test

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

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

Returns a NumPy matrix containing the model inputs required by the 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)

SALib.analyze.fast.analyze(problem, Y, M=4, print_to_console=False, seed=None)

Performs the Fourier Amplitude Sensitivity Test (FAST) 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.

**Parameters**

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

**References**

**Examples**

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

### 1.3.2 RBD-FAST - Random Balance Designs Fourier Amplitude Sensitivity Test

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

Generate model inputs using Latin hypercube sampling (LHS).

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

**Parameters**

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

SALib.analyze.rbd_fast.analyze(problem, X, Y, M=10, print_to_console=False, seed=None)

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

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

**Parameters**

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

### References

### Examples

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

#### 1.3.3 Method of Morris

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

Generate model inputs using the Method of Morris. The resulting matrix has 
\((G+1) \times T\) rows and \(D\) columns, where \(D\) is the number of parameters, \(G\) is the number of groups (if no groups are selected, the number of parameters). \(T\) is the number of trajectories \(N\), or `optimal_trajectories` if selected. These model inputs are intended to be used with `SALib.analyze.morris.analyze()`.

**Parameters**

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

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

**Return type** `numpy.ndarray`

SALib.analyze.morris.analyze *(problem, X, Y, num_resamples=100, conf_level=0.95, print_to_console=False, num_levels=4, seed=None)*

Perform Morris Analysis on model outputs.

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

**Parameters**

- **problem** *(dict)* – The problem definition
- **X** *(numpy.matrix)* – The NumPy matrix containing the model inputs of dtype=float
- **Y** *(numpy.array)* – The NumPy array containing the model outputs of dtype=float
• **num_resamples** (*int*) – The number of resamples used to compute the confidence intervals (default 1000)

• **conf_level** (*float*) – The confidence interval level (default 0.95)

• **print_to_console** (*bool*) – Print results directly to console (default False)

• **num_levels** (*int*) – The number of grid levels, must be identical to the value passed to SALib.sample.morris (default 4)

Returns

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

• **mu** - the mean elementary effect

• **mu_star** - the absolute of the mean elementary effect

• **sigma** - the standard deviation of the elementary effect

• **mu_star_conf** - the bootstrapped confidence interval

• **names** - the names of the parameters

Return type  **dict**

References

Examples

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

### 1.3.4 Sobol Sensitivity Analysis

SALib.sample.saltelli.sample(*problem*, *N*, *calc_second_order=True*, *seed=None*,
*skip_values=1000*)

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

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().

Parameters

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

• **N** (*int*) – The number of samples to generate

• **calc_second_order** (*bool*) – Calculate second-order sensitivities (default True)

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*, *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’.

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)

References

Examples

```python
>>> X = saltelli.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
```

1.3.5 Delta Moment-Independent Measure

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

Generate model inputs using Latin hypercube sampling (LHS).

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

Parameters

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

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

Perform Delta Moment-Independent Analysis on model outputs.

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

Parameters

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

### Examples

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

#### 1.3.6 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.

**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)

#### 1.3.7 Fractional Factorial

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

Generates model inputs using a fractional factorial sample

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

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

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

**Parameters**

- `problem` (dict) – The problem definition

**Return**

sample

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.

Parameters

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

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

**Return type** *dict*

### Examples

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

### 1.4 License

The MIT License (MIT)

Copyright (c) 2013-2017 Jon Herman, Will Usher, and others.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 1.5 Developers

- Jon Herman <jdherman8@gmail.com>
- Will Usher <william.usher@ouce.ox.ac.uk>
- Chris Mutel
- Bernardo Trindade
1.6 Projects that use SALib

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


If you use Bibtex, cite using the following entry:

```latex
@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} }
```

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

1.6.1 Software

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

1.6.2 Blogs

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

- PyData Presentation on SALib

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

Many thanks for using SALib.

1.7 Changelog

1.7.1 Version 1.3

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

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

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

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

1.7.2 Version 1.1.0

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

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

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

Development Features:
• Implemented automatic deployment to PyPi
• Fixed a bug preventing automatic deployment to PyPi upon tagging a branch
• Removed postgres from travis config

1.7.3 Version 1.0.0

Release of our stable version of SALIB to coincide with an submission to JOSS:
• Added a paper for submission to the Journal of Open-source Software
• Updated back-end for documentation on read-the-docs
• Updated the back-end for version introspection using PyScaffold, rather than versioneer
• Updated the Travis-CI scripts
• Moved the tests out of the SALib package and migrated to using pytest

1.7.4 Version 0.7.1

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

1.7.5 Version 0.7.0

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

1.7.6 Version 0.6.3

Parallel option for Sobol method
• New option to run analyze.sobol function in parallel using multiprocessing
1.7.7 Version 0.6.2

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

Some minor updates are included:

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

1.7.8 Version 0.6.0

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

1.7.9 Version 0.5.0

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

1.7.10 Version 0.4.0

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

1.7.11 Version 0.3.0

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

1.7.12 Version 0.2.0

Improvements to Morris sampling methods (support for group sampling, and optimized trajectories). Much better test coverage, and fixed Python 3 compatibility.
1.7.13 Version 0.1.0

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

1.8 SALib

1.8.1 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[KT, VT], X: numpy.array, Y: numpy.array, num_resamples: int = 100, conf_level: float = 0.95, print_to_console: bool = False, seed: int = None) → Dict[KT, VT]

Perform Delta Moment-Independent Analysis on model outputs.

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

Parameters

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

References

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

SALib.analyze.delta.bias_reduced_delta(Y, Ygrid, X, m, num_resamples, conf_level)
         Plischke et al. 2013 bias reduction technique (eqn 30)
SALib.analyze.delta.calc_delta(Y, Ygrid, X, m)
         Plischke et al. (2013) delta index estimator (eqn 26) for d_hat.
SALib.analyze.delta.cli_action(args)
SALib.analyze.delta.cli_parse(parser)
SALib.analyze.delta.sobol_first(Y, X, m)
SALib.analyze.delta.sobol_first_conf(Y, X, m, num_resamples, conf_level)

SALib.analyze.dgsm module

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

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

Parameters

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

References

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 Documentation, Release 1.3.11.post0.dev64+g2689340

SALib.analyze.fast module

SALib.analyze.fast.analyze(problem, Y, M=4, print_to_console=False, seed=None)
Performs the Fourier Amplitude Sensitivity Test (FAST) 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.

Parameters

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

References

Examples

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

SALib.analyze.fast.cli_action(args)
SALib.analyze.fast.compute_first_order(outputs, N, M, omega)
SALib.analyze.fast.compute_total_order(outputs, N, omega)

SALib.analyze.ff module

Created on 30 Jun 2015
@author: will2

SALib.analyze.ff.analyze(problem, X, Y, second_order=False, print_to_console=False, seed=None)
Perform a fractional factorial analysis
Returns a dictionary with keys ‘ME’ (main effect) and ‘IE’ (interaction effect). The techniques bulks out the number of parameters with dummy parameters to the nearest 2**n. Any results involving dummy parameters could indicate a problem with the model runs.

Parameters

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

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

Examples

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

SALib.analyze.ff.cli_action(args)
SALib.analyze.ff.cli_parse(parser)
SALib.analyze.ff.interactions(problem, Y, print_to_console=False)
Computes the second order effects

Parameters

- `problem (dict)` – The problem definition
- `Y (numpy.array)` – The NumPy array containing the model outputs
- `print_to_console (bool, default=False)` – Print results directly to console

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.morris module

SALib.analyze.morris.analyze(problem, X, Y, num_resamples=100, conf_level=0.95, print_to_console=False, num_levels=4, seed=None)
Perform Morris Analysis on model outputs.

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

Parameters

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

**Returns**

\( \text{Si} \) – A dictionary of sensitivity indices containing the following entries.

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

**Return type**  *dict*

**References**

**Examples**

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

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

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

SALib.analyze.morris.compute_elementary_effects(*model_inputs*, *model_outputs*, *trajectory_size*, *delta*)

**Parameters**

• *model_inputs* (matrix of inputs to the model under analysis.) – x-by-r where x is the number of variables and r is the number of rows (a function of x and num_trajectories)
• *model_outputs* – an r-length vector of model outputs
• *trajectory_size* – a scalar indicating the number of rows in a trajectory
• *delta* (*float*) – scaling factor computed from num_levels

**Returns**  *ee* – Elementary Effects for each parameter

**Return type**  *np.array*

SALib.analyze.morris.compute_grouped_metric(*ungrouped_metric*, *group_matrix*)

Computes the mean value for the groups of parameter values in the argument ungrouped_metric

SALib.analyze.morris.compute_grouped_sigma(*ungrouped_sigma*, *group_matrix*)

Returns sigma for the groups of parameter values in the argument ungrouped_metric where the group consists of no more than one parameter

SALib.analyze.morris.compute_mu_star_confidence(*ee*, *num_trajectories*, *num_resamples*, *conf_level*)

Uses bootstrapping where the elementary effects are resampled with replacement to produce a histogram of resampled mu_star metrics. This resample is used to produce a confidence interval.

SALib.analyze.morris.get_decreased_values(*op_vec*, *up*, *lo*)
SALib Documentation, Release 1.3.11.post0.dev64+g2689340

SALib.analyze.morris.get_increased_values(op_vec, up, lo)

SALib.analyze.rbd_fast module

SALib.analyze.rbd_fast.analyze(problem, X, Y, M=10, print_to_console=False, seed=None)
Performs the Random Balanced Design - Fourier Amplitude Sensitivity Test (RBD-FAST) on model outputs.
Returns a dictionary with keys ‘S1’, where each entry is a list of size D (the number of parameters) containing
the indices in the same order as the parameter file.

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

References

Examples

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

SALib.analyze.rbd_fast.cli_action(args)
SALib.analyze.rbd_fast.cli_parse(parser)
SALib.analyze.rbd_fast.compute_first_order.permute_outputs(X, Y)
  Permute the output according to one of the inputs as in [2]

References

SALib.analyze.rbd_fast.unskew_S1(SI, M, N)
Unskew the sensivity indice (Jean-Yves Tissot, Clémence Prieur (2012) “Bias correction for the estimation of
sensitivity indices based on random balance designs.”, Reliability Engineering and System Safety, Elsevier, 107,
205-213. doi:10.1016/j.ress.2012.06.010)

SALib.analyze.sobol module

SALib.analyze.sobol.Si_list_to_dict(S_list, D, calc_second_order)
SALib.analyze.sobol.Si_to_pandas_dict(S_dict)
  Convert Si information into Pandas DataFrame compatible dict.
  Parameters S_dict (ResultDict) – Sobol sensitivity indices
See also:

\texttt{Si\_list\_to\_dict()}

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

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

**Examples**

```python
>>> X = saltelli.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
>>> T_Si, first_Si, (idx, second_Si) = sobol.Si_to_pandas_dict(Si, problem)
```

```
SALib.analyze.sobol.analyze(problem, Y, calc_second_order=True, num_resamples=100,
conf_level=0.95, print_to_console=False, parallel=False,
n_processors=None, 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’.

**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)

**References**

**Examples**

```python
>>> X = saltelli.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, print_to_console=True)
```

```
SALib.analyze.sobol.cli_action(args)
SALib.analyze.sobol.cli_parse(parser)
SALib.analyze.sobol.create_Si_dict(D, calc_second_order)
SALib.analyze.sobol.create_task_list(D, calc_second_order, n_processors)
SALib.analyze.sobol.first_order(A, AB, B)
SALib.analyze.sobol.print_indices(S, problem, calc_second_order)
SALib.analyze.sobol.second_order(A, ABj, ABk, Baj, B)
```
SALib Documentation, Release 1.3.11.post0.dev64+g2689340

SALib.analyze.sobol.\texttt{separate\_output\_values}(Y, D, N, \texttt{calc\_second\_order})

SALib.analyze.sobol.\texttt{sobol\_parallel}(Z, A, AB, BA, B, r, tasks)

SALib.analyze.sobol.\texttt{to\_df}(self)

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

\begin{description}
\item[Returns] List
\item[Return type] of Pandas DataFrames in order of Total, First, Second
\end{description}

SALib.analyze.sobol.\texttt{total\_order}(A, AB, B)

Module contents

SALib.plotting package

Submodules

SALib.plotting.bar module

SALib.plotting.bar.\texttt{plot}(Si_df, \texttt{ax=None})

Create bar chart of results

\begin{description}
\item[Parameters] \texttt{Si\_df} (\texttt{*}) –
\item[Returns] \texttt{* ax}
\item[Return type] matplotlib axes object
\end{description}

Examples

\begin{verbatim}
>>> from SALib.plotting.bar import plot as barplot
>>> from SALib.test_functions import Ishigami

>>> X = saltelli.sample(problem, 1000)
>>> Y = Ishigami.evaluate(X)
>>> Si = sobol.analyze(problem, Y, \texttt{print\_to\_console=False})
>>> Si\_df = Si.to\_df()
>>> barplot(Si\_df)
\end{verbatim}

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:

\begin{verbatim}
import matplotlib.pyplot as plt
Si = morris.analyze(problem, param_values, Y, conf_level=0.95,
print_to_console=False, num_levels=10)
p = morris.horizontal_bar_plot(Si)
\end{verbatim}

(continues on next page)
# set plot style etc.

```python
fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker':'x'})
p.show()
```

SALib.plotting.morris.covariance_plot(ax, Si, param_dict, unit="")
Plots \( \mu^* \) against sigma or the 95% confidence interval

SALib.plotting.morris.horizontal_bar_plot(ax, Si, param_dict, sortby='mu_star', unit="")
Updates a matplotlib axes instance with a horizontal bar plot
of \( \mu^* \), with error bars representing \( \mu^*_\text{conf} \)

SALib.plotting.morris.sample_histograms(fig, input_sample, problem, param_dict)
Plots a set of subplots of histograms of the input sample

Module contents

**SALib.sample package**

Subpackages

**SALib.sample.morris package**

Submodules

**SALib.sample.morris.brute module**

class SALib.sample.morris.brute.BruteForce
   Bases: SALib.sample.morris.strategy.Strategy
   Implements the brute force optimisation strategy

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

Parameters

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

Returns

Return type **list**

find_maximum(scores, N, k_choices)
Finds the \( k\_choices \) maximum scores from \( scores \)
Parameters

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

Returns

Return type: list

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

Finds the ‘k_choices’ most distant choices from the ‘num_samples’ trajectories contained in ‘input_sample’

Parameters

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

Returns

Return type: numpy.ndarray

**static grouper** (*n, iterable*)

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

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

Parameters

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

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

Returns the nth item or a default value

Parameters

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

**SALib.sample.morris.local module**

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

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

Implements the local optimisation algorithm using the Strategy interface

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

Adds extra indices for the combinatorial problem.
Parameters

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

Example

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

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

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

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

Parameters

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

Returns

Return type list

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

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

Parameters

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

Returns

Return type list

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

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

Parameters

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

Returns

Return type numpy.ndarray
Notes

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

SALib.sample.morris.morris module

Generate a sample using the Method of Morris

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

• Vanilla Morris

• Optimised trajectories when `optimal_trajectories=True` (using Campolongo’s enhancements from 2007 and optionally Ruano’s enhancement from 2012; `local_optimization=True`)

• Groups with optimised trajectories when `optimal_trajectories=True` and the problem definition specifies groups (note that `local_optimization` must be `False`)

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

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

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

Generate model inputs using the Method of Morris

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

Parameters

• `problem (dict)` – The problem definition

• `N (int)` – The number of trajectories to generate

• `num_levels (int, default=4)` – The number of grid levels (should be even)

• `optimal_trajectories (int)` – The number of optimal trajectories to sample (between 2 and \(N\))

• `local_optimization (bool, default=True)` – Flag whether to use local optimization according to Ruano et al. (2012) Speeds up the process tremendously for bigger \(N\) and `num_levels`. If set to `False` brute force method is used, unless `gurobipy` is available

• `seed (int)` – Seed to generate a random number

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

Return type `numpy.ndarray`
SALib Documentation, Release 1.3.11.post0.dev64+g2689340

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

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

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

Bases: object

Computes the optimum \( k_{choices} \) of trajectories from the input_sample.

Parameters

- **strategy** (*Strategy*) –

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

  Computes the optimum \( k_{choices} \) of trajectories from the input_sample.

Parameters

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

Returns

An array of optimal trajectories

Return type

*numpy.ndarray*

class SALib.sample.morris.strategy.Strategy

Bases: object

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

static check_input_sample(input_sample, num_params, num_samples)

Check the input_sample is valid

Checks input sample is:

- the correct size
- values between 0 and 1

Parameters

- **input_sample** (*numpy.ndarray*) –
- **num_params** (*int*) –
- **num_samples** (*int*) –
**compile_output** *(input_sample, num_samples, num_params, maximum_combo, num_groups=None)*

Picks the trajectories from the input

**Parameters**

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

**static compute_distance** *(m, l)*

Compute distance between two trajectories

**Returns**

**Return type** numpy.ndarray

**compute_distance_matrix** *(input_sample, num_samples, num_params, num_groups=None, local_optimization=False)*

Computes the distance between each and every trajectory

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

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

**Parameters**

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

**Returns** distance_matrix

**Return type** numpy.ndarray

**static run_checks** *(number_samples, k_choices)*

Runs checks on *k_choices*

**sample** *(input_sample, num_samples, num_params, k_choices, num_groups=None)*

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

**Parameters**

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

**Returns**
Return type: `numpy.ndarray`

Module contents

Submodules

**SALib.sample.common_args module**

SALib.sample.common_args.create(cli_parser=None)

Create CLI parser object.

**Parameters**

- `cli_parser` *(function [optional]*)

**Returns**

**Return type**`argparse object`

SALib.sample.common_args.run_cli(cli_parser, run_sample, known_args=None)

Run sampling with CLI arguments.

**Parameters**

- `cli_parser` *(function)*
- `run_sample` *(function)*
- `known_args` *(list [optional]*)

**Returns**

**Return type**`argparse object`

SALib.sample.common_args.setup(parser)

Add common sampling options to CLI parser.

**Parameters**

- `parser` *(argparse object)*

**Returns**

**Return type**Updated argparse object

**SALib.sample.directions module**

**SALib.sample.fast_sampler module**

SALib.sample.fast_sampler.cli_action(args)

Run sampling method.

**Parameters**

- `args` *(argparse namespace)*

SALib.sample.fast_sampler-cli_parse(parser)

Add method specific options to CLI parser.

**Parameters**

- `parser` *(argparse object)*

**Returns**

**Return type**Updated argparse object
SALib.sample.fast_sampler.sample (problem, N, M=4, seed=None)

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

Returns a NumPy matrix containing the model inputs required by the 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)

SALib.sample.ff module

The sampling implementation of fractional factorial method

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

SALib.sample.ff.cli_action (args)

Run sampling method

Parameters args (argparse namespace) –

SALib.sample.ff.extend_bounds (problem)

Extends the problem bounds to the nearest power of two

Parameters problem (dict) – The problem definition

SALib.sample.ff.find_smallest (num_vars)

Find the smallest exponent of two that is greater than the number of variables

Parameters num_vars (int) – Number of variables

Returns x – Smallest exponent of two greater than num_vars

Return type int

SALib.sample.ff.generate_contrast (problem)

Generates the raw sample from the problem file

Parameters problem (dict) – The problem definition

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

Generates model inputs using a fractional factorial sample

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

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

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

Parameters problem (dict) – The problem definition

Returns sample

Return type numpy.array
**SALib.sample.finite_diff module**

SALib.sample.finite_diff.cli_action(args)

Run sampling method

Parameters

args (argparse namespace) –

SALib.sample.finite_diff.cli_parse(parser)

Add method specific options to CLI parser.

Parameters

parser (argparse object) –

Returns

Return type Updated argparse object

SALib.sample.finite_diff.sample(problem, N, delta=0.01, seed=None)

Generate matrix of samples for derivative-based global sensitivity measure (dgsm). Start from a QMC (sobol) sequence and finite difference with delta % steps

Parameters

- **problem** (dict) – SALib problem specification
- **N** (int) – number of samples
- **delta** (float) – Finite difference step size (percent)
- **seed** (int or None) – random seed value

Returns np.array

Return type DGSM sequence

**SALib.sample.latin module**

SALib.sample.latin.cli_action(args)

Run sampling method

Parameters

args (argparse namespace) –

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

Generate model inputs using Latin hypercube sampling (LHS).

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

Parameters

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

**SALib.sample.saltelli module**

SALib.sample.saltelli.cli_action(args)

Run sampling method

Parameters

args (argparse namespace) –

SALib.sample.saltelli.cli_parse(parser)

Add method specific options to CLI parser.
Parameters parser(argparse object) –

Returns

Return type Updated argparse object

SALib.sample.saltelli.sample(problem, N, calc_second_order=True, seed=None, skip_values=1000)

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

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().

Parameters

- problem(dict) – The problem definition
- N(int) – The number of samples to generate
- calc_second_order(bool) – Calculate second-order sensitivities (default True)

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(values)

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)

Module contents

SALib.util package

Submodules

SALib.util.results module

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

Bases: dict

Dictionary holding analysis results.

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

plot()

Create bar chart of results

to_df()

Convert dict structure into Pandas DataFrame.
Module contents

A set of utility functions

SALib.util.scale_samples (params: numpy.ndarray, bounds: List[T])
    Rescale samples in 0-to-1 range to arbitrary bounds

    Parameters
    • params (numpy.ndarray) – numpy array of dimensions num_params-by-N, where N
      is the number of samples
    • bounds (list) – list of lists of dimensions num_params-by-2

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 be-
      tween columns

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

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

    plot ()
        Create bar chart of results
    to_df ()
        Convert dict structure into Pandas DataFrame.

SALib.util_avail_approaches (pkg)
    Create list of available modules.

    Parameters pkg (module) – module to inspect

    Returns method – A list of available submodules

    Return type list
Module contents

1.9 Indices and tables

- genindex
- modindex
- search
SALib, 38
SALib.analyze, 25
SALib.analyze.common_args, 18
SALib.analyze.delta, 18
SALib.analyze.dgsm, 19
SALib.analyze.fast, 20
SALib.analyze.ff, 20
SALib.analyze.morris, 21
SALib.analyze.rbd_fast, 23
SALib.analyze.sobol, 23
SALib.plotting, 26
SALib.plotting.bar, 25
SALib.plotting.morris, 25
SALib.sample, 35
SALib.sample.common_args, 32
SALib.sample.directions, 32
SALib.sample.fast_sampler, 32
SALib.sample.ff, 33
SALib.sample.finite_diff, 34
SALib.sample.latin, 34
SALib.sample.morris, 32
SALib.sample.morris.brute, 26
SALib.sample.morris.local, 27
SALib.sample.morris.morris, 29
SALib.sample.morris.strategy, 30
SALib.sample.saltelli, 34
SALib.sample.sobol_sequence, 35
SALib.scripts, 36
SALib.scripts.salib, 35
SALib.test_functions, 36
SALib.test_functions.Ishigami, 36
SALib.test_functions.Sobol_G, 36
SALib.util, 37
SALib.util.results, 36
Index

A
add_indices() (SALib.sample.morris.local.LocalOptimisation method), 27
analyze() (in module SALib.analyze.delta), 18
analyze() (in module SALib.analyze.dgsm), 19
analyze() (in module SALib.analyze.fast), 20
analyze() (in module SALib.analyze.ff), 20
analyze() (in module SALib.analyze.morris), 21
analyze() (in module SALib.analyze.rbd_fast), 23
analyze() (in module SALib.analyze.sobol), 24
avail_approaches() (in module SALib.util), 37

B
bias_reduced_delta() (in module SALib.analyze.delta), 19
brute_force_most_distant() (SALib.sample.morris.brute.BruteForce method), 26
BruteForce (class in SALib.sample.morris.brute), 26

C
calc_delta() (in module SALib.analyze.delta), 19
calc_dgsm() (in module SALib.analyze.dgsm), 19
calc_vi_mean() (in module SALib.analyze.dgsm), 19
calc_vi_stats() (in module SALib.analyze.dgsm), 19
check_input_sample()
   (SALib.sample.morris.strategy.Strategy static method), 30
cli_action() (in module SALib.sample.morris.strategy.Strategy method), 30
cli_action() (in module SALib.sample.morris.strategy.Strategy static method), 31
compute_distance()
   (SALib.sample.morris.strategy.Strategy static method), 31
compute_distance_matrix()
   (SALib.sample.morris.strategy.Strategy static method), 31
compute_elementary_effects() (in module SALib.analyze.morris), 22
compute_first_order() (in module SALib.analyze.fast), 20
compute_first_order() (in module SALib.analyze.rbd_fast), 23
compute_grouped_metric() (in module SALib.analyze.morris), 22
compute_grouped_sigma() (in module SALib.analyze.morris), 22
compute_mu_star_confidence() (in module SALib.analyze.morris), 22
covariance_plot() (in module SALib.plotting.morris), 26
create() (in module SALib.analyze.common_args), 18
create() (in module SALib.sample.common_args), 32
cli_action() (in module SALib.sample.finite_diff), 34
cli_action() (in module SALib.sample.latim), 34
cli_action() (in module SALib.sample.saltelli), 34
cli_parse() (in module SALib.analyze.delta), 19
cli_parse() (in module SALib.analyze.dgsm), 19
cli_parse() (in module SALib.analyze.morris), 22
cli_parse() (in module SALib.analyze.rbd_fast), 23
cli_parse() (in module SALib.analyze.sobol), 24
cli_parse() (in module SALib.sample.fast_sampler), 32
cli_parse() (in module SALib.sample.finite_diff), 34
cli_parse() (in module SALib.sample.saltelli), 34
cli_action() (in module SALib.analyze.delta), 19
cli_action() (in module SALib.analyze.dgsm), 19
cli_action() (in module SALib.analyze.ff), 20
cli_action() (in module SALib.analyze.morris), 21
cli_action() (in module SALib.analyze.rbd_fast), 23
cli_action() (in module SALib.analyze.sobol), 24
cli_action() (in module SALib.sample.fast_sampler), 32
cli_action() (in module SALib.sample.finite_diff), 33