# CONTENTS

1 **pyeee Quickstart** 1

1.1 About 1
1.2 Quick usage guide 1
  - Simple Python function 1
  - Python function with extra parameters 2
  - Function wrappers 3
1.3 Installation 3
1.4 License 3
1.5 Contributing to pyeee 3
1.6 Indices and tables 3

2 **pyeee User Guide** 5

2.1 Elementary Effects 5
  - Simple Python functions 5
  - Exclude parameters from calculations 6
  - Parallel model evaluation 7
  - Python functions with additional parameters 8
  - Sampling parameters with other distributions than the uniform distribution 9
2.2 Efficient/Sequential Elementary Effects 10
2.3 External computer models 11
  - Simple executables 11
  - Exclude parameters from screening 12
  - Additional arguments for external executables 13
  - Further arguments of wrappers 13
  - Parallel processing of external executables 14

3 **pyeee Installation** 17

3.1 Manual install 17
3.2 Local install 17
3.3 Dependencies 18

4 **pyeee API** 19

4.1 pyeee.eee 20
4.2 pyeee.screening 23
4.3 pyeee.morris 26
4.4 pyeee.general_functions 30
4.5 pyeee.sa_test_functions 34
4.6 pyeee.utils 37
4.7 pyeee.function_wrapper 38
4.8 pyeee.std_io 42
4.9 pyeee.tee 47
**1.1 About**

`pyeee` is a Python library for performing parameter screening of computational models. It uses Morris’ method of Elementary Effects and also its extension of Efficient or Sequential Elementary Effects published by:


`pyeee` can be used with Python functions but wrappers are provided to use it with external executables as well. Function evaluation can be distributed with Python’s multiprocessing or via MPI.

The complete documentation for `pyeee` is available from Read The Docs.

http://pyeee.readthedocs.org/en/latest/

**1.2 Quick usage guide**

**Simple Python function**

Consider the Ishigami-Homma function:

\[ y = \sin(x_0) + a \sin(x_1)^2 + b x_1^4 \sin(x_0). \]

Taking \( a = b = 1 \) gives:

```python
import numpy as np
def ishigami(x):
    return np.sin(x[0]) + np.sin(x[1])**2 + x[2]**4 * np.sin(x[0])
```

The three parameters \( x_0, x_1, x_2 \) follow uniform distributions between \(-\pi\) and \(+\pi\).

Morris’ Elementary Effects can then be calculated like:

```
npars = 3
# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
```
ub = np.ones(npars) * np.pi

# Elementary Effects
from pyeee import ee
np.random.seed(seed=1023)  # for reproducibility of examples
out = ee(ishigami1, lb, ub, 10)

which gives the Elementary Effects ($\mu^*$):

```python
# mu*
print("{:.1f} {:.1f} {:.1f}".format(*out[:,0]))
# gives: 173.1 0.6 61.7
```

Sequential Elementary Effects distinguish between informative and uninformative parameters using several times Morris’ Elementary Effects:

```python
# screen
from pyeee import eee
np.random.seed(seed=1021)  # for reproducibility of examples
out = eee(ishigami1, lb, ub)
```

which returns a logical ndarray with True for the informative parameters and False for the uninformative parameters:

```python
print(out)
# gives: [ True False True]
```

### Python function with extra parameters

The function for `pyeee` must be of the form `func(x)`. Use Python’s `functools.partial` from the `functools` module to pass other function parameters.

For example pass the parameters $a$ and $b$ to the Ishigami-Homma function:

```python
from functools import partial
def ishigami(x, a, b):
    return np.sin(x[0]) + a * np.sin(x[1])**2 + b * x[2]**4 * np.sin(x[0])

def call_func_ab(func, a, b, x):
    return func(x, a, b)

# Partialise function with fixed parameters a and b
a = 0.5
b = 2.0
func = partial(call_func_ab, ishigami, a, b)
npars = 3

# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
ub = np.ones(npars) * np.pi

# Elementary Effects
np.random.seed(seed=1021)  # for reproducibility of examples
out = ee(func, lb, ub, 10)
```

`partial` passes $a$ and $b$ to the function `call_func_ab` already during definition so that `pyeee` can then simply call it as `func(x)`, so that $x$ is passed to `call_func_ab` as well.
Function wrappers

`pyeee` provides wrappers to use with `partial`.

```python
from pyeee.utils import func_wrapper
args = [a, b]
kwargs = {}
func = partial(func_wrapper, ishigami, args, kwargs)

# screen
np.random.seed(seed=1021)  # for reproducibility of examples
out = eee(func, lb, ub)
```

There are wrappers to use with Python functions with or without masking parameters, as well as wrappers for external executables.

1.3 Installation

The easiest way to install is via `pip`:

```bash
pip install pyeee
```

See the installation instructions for more information.

1.4 License

`pyeee` is distributed under the MIT License. See the LICENSE file for details.

Copyright (c) 2014-2019 Matthias Cuntz, Juliane Mai

The project structure is based on a template[https://github.com/MuellerSeb/template] provided by [Sebastian Müller][https://github.com/MuellerSeb].

1.5 Contributing to pyeee

Users are welcome to submit bug reports, feature requests, and code contributions to this project through GitHub. More information is available in the Contributing guidelines.

1.6 Indices and tables

- genindex
- modindex
**CHAPTER 2**

**PYEEE USER GUIDE**

pyeee is a Python library for performing parameter screening of computational models. It uses Morris’ method of Elementary Effects (EE) and also its extension of Efficient or Sequential Elementary Effects (EEE or SEE) published by:


The numerical models are simply passed to functions `ee()` and `eee()` to perform Elementary Effects or Efficient/Sequential Elementary Effects, respectively.

The numerical models must be callable as `func(x)`. Use `functools.partial()` from Python’s standard library to make any function callable as `func(x)`. pyeee provides wrapper functions to help with this process.

### 2.1 Elementary Effects

**Simple Python functions**

Consider the Ishigami-Homma function:

\[ y = \sin(x_0) + a \sin(x_1)^2 + b x_2^4 \sin(x_0). \]

Taking \( a = b = 1 \) gives:

```python
import numpy as np

# Ishigami-Homma function a=b=1
def ishigami1(x):
    return np.sin(x[0]) + np.sin(x[1])**2 + x[2]**4 * np.sin(x[0])
```

The three parameters \( x_0, x_1, x_2 \) follow uniform distributions between \(-\pi\) and \(+\pi\).

Elementary Effects can be calculated, using 20 trajectories, as follows:

```python
from pyeee import ee

# function
func = ishigami1
npars = 3

# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
ub = np.ones(npars) * (np.pi)

# Elementary Effects
effects = ee(func, lb, ub, trajectories=20)
```

(continues on next page)
ub = np.ones(npars) * np.pi

# Elementary Effects
np.random.seed(seed=1023)  # for reproducibility of examples
out = ee(func, lb, ub, 20)

**ee()** returns a (npars,3) ndarray with:

1. (npars,0) the means of the absolute elementary effects over all trajectories ($\mu^*$)
2. (npars,1) the means of the elementary effects over all nt trajectories ($\mu$)
3. (npars,2) the standard deviations of the elementary effects over all trajectories ($\sigma$)


```python
# mu*
p = [1.0] + [2.0, 3.0]
print("{:.1f} {:.1f} {:.1f}").format(*out[:,0])
# gives: 212.4 0.6 102.8
```

The numerical model `func`, lower parameter boundaries `lb`, upper parameter boundaries `ub`, and the number of trajectories `nt` are mandatory arguments to `ee()`. Further optional arguments relevant to Elementary Effects are:

- `nsteps`: int - Number of steps along one trajectory (default: 6)
- `ntotal`: int - Total number of trajectories to check for the nt most different trajectories (default: `max(nt**2, 10*nt)`)  

Note that the functions `ee()` and `screening()` are identical.

### Exclude parameters from calculations

`ee()` offers the possibility to mask some model parameters so that they will not be changed during calculation of Elementary Effects. Initial values $x_0$ must be given that will be taken where $mask[0]=False$, i.e. $mask$ could be called an include-mask. Note that the size of $x_0$ must be the size of $lb, ub$ and $mask$, i.e. one has to give initial values even if an element is included in the screening, which means $mask[i]==True$.

For example, if one want to exclude the second parameter $x_1$ of the above Ishigami-Homma function in the calculation of the Elementary Effects:

```python
# function
mask = np.ones(npars, dtype=bool)  # True -> include
mask[1] = False  # False -> exclude

# initial values
x0 = np.ones(npars) * 0.5

# Elementary Effects
np.random.seed(seed=1024)  # for reproducibility of examples
out = ee(func, lb, ub, 10, x0=x0, mask=mask, nsteps=8, ntotal=100)

print("{:.1f} {:.1f} {:.1f}").format(*out[:,0])
# gives: 114.8 0.0 26.6
```
Parallel model evaluation

The numerical model \texttt{func} will be evaluated \(nt*(npars+1)\) times, with \(npars\) the number of parameters of the computational model. Multiprocessing can be used for parallel function evaluation. Setting \texttt{processes=nprocs} evaluates \(nprocs\) parameter sets in parallel:

```python
# Elementary Effects using 4 parallel processes
np.random.seed(seed=1024) # for reproducibility of examples
out = ee(func, lb, ub, 10, x0=x0, mask=mask, nsteps=8, ntotal=100, processes=4)
```

\texttt{pyeee} uses the package \texttt{schwimmbad} for parallelisation. \texttt{schwimmbad} provides a uniform interface to parallel processing pools and enables switching easily between local development (e.g. serial processing or multiprocessing) and deployment on a cluster or supercomputer (via e.g. MPI or JobLib).

Consider the following code in a script (e.g. \texttt{eeetest.py}):

```python
# File: eeetest.py
# get number of processes
import sys
if len(sys.argv) > 1:
    nprocs = int(sys.argv[1])
else:
    nprocs = 1

# Ishigami-Homma function a=b=1
import numpy as np
def ishigami1(x):
    return np.sin(x[0]) + np.sin(x[1])**2 + x[2]**4 * np.sin(x[0])

# mpi4py is an optional dependency of pyeee
try:
    from mpi4py import MPI
    comm = MPI.COMM_WORLD
    csize = comm.Get_size()
    crank = comm.Get_rank()
except ImportError:
    comm = None
    csize = 1
    crank = 0

from pyeee import ee

# function
func = ishigami1
npars = 3

# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
ub = np.ones(npars) * np.pi

# choose the serial or parallel pool
import schwimmbad
ipool = schwimmbad.choose_pool(mpi=False if csize==1 else True, processes=nprocs)

# Elementary Effects
np.random.seed(seed=1023) # for reproducibility of examples
out = ee(func, lb, ub, 20, processes=nprocs, pool=ipool)

if crank == 0:
    (continues on next page)
```
print("{:.1f} {:.1f} {:.1f}".format(*out[:,0]))
ipool.close()

This script can be run serially, i.e. that all function evaluations are done one after the other:

```
python eeetest.py
```

or

```
python eeetest.py 1
```

It can use Python’s `multiprocessing` module, e.g. with 4 parallel processes:

```
python eeetest.py 4
```

or use the Message Passing Interface (MPI), e.g. with 4 parallel processes:

```
mpiexec -n 4 python eeetest.py 4
```

Note that `mpi4py` must be installed for the latter.

**Python functions with additional parameters**

The function for `pyeee` must be of the form `func(x)`. Use Python’s `functools.partial` to pass other function parameters.

For example pass the parameters `a` and `b` to the Ishigami-Homma function. One needs a wrapper function that takes the function and its parameters as arguments. The variable parameters of the screening must be the last argument, i.e. `x` of `func(x)`:

```python
from functools import partial
def ishigami(x, a, b):
    return np.sin(x[0]) + a * np.sin(x[1])**2 + b * x[2]**4 * np.sin(x[0])
def call_func_ab(func, a, b, x):
    return func(x, a, b)
```

The parameters `a` and `b` are fixed parameters during screening. They are hence already passed to `call_func_ab` with `functools.partial` before start of the screening.

```
# Partialise function with fixed parameters a and b
a = 0.5
b = 2.0
func = partial(call_func_ab, ishigami, a, b)
out = ee(func, lb, ub, 10)
```

When `func` is called as `func(x)`, the call of `call_func_ab` is finished and `x`, `a` and `b` are passed to `ishigami`.

`pyeee` provides wrapper functions to work with `functools.partial`. `call_func_ab` can be replaced by the wrapper function of `pyeee`: `func_wrapper()`:

```python
from pyeee.utils import func_wrapper
arg = [a, b]
kwarg = {}
func = partial(func_wrapper, ishigami, arg, kwarg)
out = ee(func, lb, ub, 10)
```
where all arguments of the function but the first one must be given as a list and keyword arguments as a dictionary. The function wrapper finally passes x, arg and kwarg to func(x, *arg, **kwarg).

pyeee provides also a wrapper function to work with masks as above. To exclude the second parameter \( x_1 \) from screening of the Ishigami-Homma function, \( x_0 \) and mask must be given to \texttt{func\_mask\_wrapper()}. Then Elementary Effects will be calculated only for the remaining parameters, between \( \text{lb[mask]} \) and \( \text{ub[mask]} \). All other non-masked parameters will be taken as \( x_0 \). Remember that mask is an include-mask, i.e. all mask==True will be screened and all mask==False will not be screened.

```python
from pyeee.utils import func_mask_wrapper
func = partial(func_mask_wrapper, ishigami, x0, mask, arg, kwarg)
out = ee(func, lb[mask], ub[mask], 10)
```

### Sampling parameters with other distributions than the uniform distribution

Morris’ method of Elementary Effects samples parameters along trajectories through the possible parameter space. It assumes uniformly distributed parameters between a lower bound and an upper bound.

pyeee allows sampling parameters from other than uniform distributions. For example, a parameter \( p \) might have been determined by repeated experiments. One can hence determine the mean parameter \( \overline{p} \) and calculate the error of the mean \( \epsilon_p \). This error of the mean is actually the standard deviation of the distribution of the mean. One would thus sample a normal distribution with mean \( \overline{p} \) and a standard deviation \( \epsilon_p \) for the parameter \( p \) for determining Morris’ Elementary Effects.

pyeee allows all distributions of \texttt{scipy.stats}, given with the keyword \texttt{dist}. The parameter of the distributions are given as tuples with the keyword \texttt{distparam}. The lower and upper bounds change their meaning if \texttt{dist} is given for a parameter: pyeee samples uniformly the Percent Point Function (ppf) of the distribution between lower and upper bound. The percent point function is the inverse of the Cumulative Distribution Function (cdf). Lower and upper bound must hence be between 0 and 1. Note the percent point functions of most continuous distributions will be infinite at the limits 0 and 1.

The three parameters \( x_0, x_1, x_2 \) of the Ishigami-Homma function follow uniform distributions between \(-\pi\) and \(+\pi\). Say that \( x_1 \) follows a Gaussian distribution around the mean 0 with a standard deviation of 1.81. We want to sample between three standard deviations, which includes about 99.7% of the total distribution. This means that the lower bound would be 0.0015 and the upper bound 0.9985.

```python
import scipy.stats as stats
dist = [None, stats.normal, stats.uniform]
distparam = [None, (0., 1.81), (-np.pi, 2.*np.pi)]
lb = [-np.pi, 0.0015, 0.]
ub = [np.pi, 0.9985, 1.]
out = ee(func, lb, ub, 20, dist=dist, distparam=distparam)
```

This shows that

1. one has to give a distribution for each parameter;
2. distributions are given as \texttt{scipy.stats} distribution objects;
3. if \texttt{dist} is None, pyeee assumes a uniform distribution and samples between lower and upper bound;
4. (almost) all \texttt{scipy.stats} distributions take the keywords \texttt{loc} and \texttt{scale}. Their meaning is \textbf{NOT} mean and standard deviation in most distributions. For the uniform distribution \texttt{scipy.stats.uniform}, \texttt{loc} is the lower limit and \texttt{loc+scale} the upper limit. This means the combination \texttt{dist=\text{None}, lb=a, ub=b} corresponds to \texttt{dist=\text{scipy.stats.uniform}, distparam=[a,b-a], lb=0, ub=1}.

Note also that

5. if \texttt{distparam==\text{None}}, \texttt{loc=0} and \texttt{scale=1} will be taken;
6. \texttt{loc} and \texttt{scale} are implemented as keywords in \texttt{scipy.stats}. Other parameters such as the shape parameter of the gamma distribution \texttt{scipy.stats.gamma} must hence be given first, e.g. \texttt{(shape,loc, scale)}.

### 2.1. Elementary Effects
Remember that Morris’ method of Elementary Effects assumes uniformly distributed parameters and that other distributions are an extension of the original method.

2.2 Efficient/Sequential Elementary Effects

Morris’ method of Elementary Effects is not a full sensitivity analysis. The sensitivity measures of Elementary Effects are rather used for preliminary screening for noninformative model parameters for a given model output, so that fewer parameters are needed during a full sensitivity analysis or during model optimisation.

The numerical model `func` will be evaluated `nt*(npars+1)` times for calculating Elementary Effects. The user chooses the number of trajectories `nt`. A large number of `nt` might be computationally expensive and a small number might miss areas of the parameter space, where certain parameters become sensitive. Typical values for `nt` in the literature are on the order of tens to hundreds. This means that the method of Elementary Effects needs between 500 and 5000 model evaluations for a model with 50 parameters.

The extension of Efficient or Sequential Elementary Effects can be used if one uses Elementary Effects only to distinguish between sensitive (informative) and insensitive (noninformative) model parameters. It follows the idea: if one knows that a model is sensitive to a certain parameter, this parameter does not have to be included anymore in the further analysis. If a parameter has a large Elementary Effect in one trajectory it will most probably be influential. So one does not have to calculate another Elementary Effect for this parameter and it can be discarded from further trajectories.

The method starts hence with a limited number of trajectories `ntfirst` for all model parameters, i.e. it performs `ntfirst*(npars+1)` model evaluations. Further trajectories are sampled, calculating Elementary Effects, but without the parameters that were already found sensitive. This means that subsequent trajectories need less and less function evaluations. The algorithm ends if a subsequent trajectory did not yield any sensitive parameters anymore. A last `ntlast` trajectories are finally sampled, and Elementary Effects calculated, to assure a large sample for little sensitive parameters.

The call of `eee()` (or the identical function `see()`) is very similar to standard Elementary effects `ee()`:

```python
from pyeee.utils import func_wrapper
def ishigami(x, a, b):
    return np.sin(x[0]) + a * np.sin(x[1])**2 + b * x[2]**4 * np.sin(x[0])
arg = [a, b]
kwarg = {}
func = partial(func_wrapper, ishigami, arg, kwarg)
npars = 3
# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
ub = np.ones(npars) * np.pi

# Sequential Elementary Effects
from pyeee import eee
np.random.seed(seed=1025)  # for reproducibility of examples
out = eee(func, lb, ub, ntfirst=10, ntlast=5, nsteps=6, processes=4)
print(out)
# gives: [ True False  True]
```
out = eee(func, lb[mask], ub[mask])
# update mask
mask[mask] = mask[mask] & out

The numerical model `func` might return several outputs per model run, e.g. a time series. The Morris’ sensitivity
measures are calculated hence for each output, e.g. each point in time. Efficient/Sequential Elementary Effects
`eee()` can either take the arithmetic mean of all $\mu^*$ or a weighted mean $\mu^*$, weighted by $\sigma$. The keyword
`weight=False` is probably appropriate if each single output is equally important. An example is river runoff
where high flows might be floods and low flows might be droughts. One might want that the computer model
reproduces both circumstances. An example for `weight=True` are fluxes to and from the atmosphere such as
evapotranspiration. The atmosphere is more strongly influenced by larger fluxes so that sensitivity measures
during periods of little atmosphere exchange are less interesting. Cuntz et al. (2015) argued that weighting by
standard deviation $\sigma$ is equivalent to flux weighting because parameter variations yield larger variances for large
fluxes than for small fluxes in most computer models.

`eee()` offers the same parallel mechanism as `ee()`, using the keywords `processes` and `pool`, which is again a
`schwimmbad` pool object.

`eee()` also offers the possibility to sample parameters from different distributions of `scipy.stats` with the
keywords `dist` and `distparam`.

One can give a `plotfile` name to check the initial fit to the `ntfirst` Elementary Effects.

```python
# Sequential Elementary Effects using all parameters and keywords
out = eee(func, lb, ub,
x0=x0, mask=mask, ntfirst=10, ntlast=10, nsteps=6, weight=True,
processes=4, seed=1025,
plotfile='ishigami.png', logfile='ishigami.log')
```

Note that `matplotlib` must be installed to produce the `plotfile`.

### 2.3 External computer models

`pyeee` provides wrapper functions to work with external executables. `pyeee` writes the sampled parameter sets
into files that can be read by the external program. The program writes its result to a file that will then be read by
`pyeee` in return. The processing steps are:

```python
parameterwriter(parameterfile, x)
err = subprocess.check_output(exe)
obj = objectivereader(objectivefile)
os.remove(parameterfile)
os.remove(objectivefile)
```

That means `pyeee` needs to have a function `parameterwriter` that writes the parameter file `parameterfile` needed
by the executable `exe`. It then needs to have a function `objectivereader` for reading the output file `objectivefile` of
`exe`, reading or calculating the objective value used by Elementary Effects.

### Simple executables

Consider for simplicity an external Python program (e.g. `ishiexe.py`) that calculates the Ishigami-Homma function
with $a = b = 1$, reading in the three parameters $x_0, x_1, x_2$ from a `parameterfile = params.txt` and writing its
output into an `objectivefile = obj.txt`:

```python
# File: ishiexe.py
# Ishigami-Homma function a=b=1
```
```python
import numpy as np
def ishigami1(x):
    return np.sin(x[0]) + np.sin(x[1])**2 + x[2]**4 * np.sin(x[0])

# read parameters
from pyeee.utils import standard_parameter_reader
pfile = 'params.txt'
x = standard_parameter_reader(pfile)

# calc function
y = ishigami1(x)

# write objective
ofile = 'obj.txt'
ff = open(ofile, 'w')
print(y, file=ff)
ff.close()
```

This program can be called on the command line with:

```
python ishiexe.py
```

The external program can be used in pyeee with `functools.partial` and the wrapper function `exe_wrapper()`:

```python
from functools import partial
from pyeee.utils import exe_wrapper, standard_parameter_writer, standard_objective_reader,
ishi = ['python', 'ishiexe.py']
parameterfile = 'params.txt'
objectivefile = 'obj.txt'
func = partial(exe_wrapper, ishi, parameterfile, standard_parameter_writer,
               objectivefile, standard_objective_reader, ())
npars = 3
lb = np.ones(npars) * (-np.pi)
ub = np.ones(npars) * np.pi

from pyeee import ee
out = ee(func, lb, ub, 10)
```

`standard_parameter_reader()` and `standard_parameter_writer()` are convenience functions that read and write one parameter per line in a file without a header. The function `standard_objective_reader()` simply reads one value from a file without header. The empty directory at the end will be explained below at Further arguments of wrappers.

One can easily imagine to replace the python program `ishiexe.py` by any compiled executable from C, Fortran or alike.

**Exclude parameters from screening**

Similar to `func_mask_wrapper()`, there is also a wrapper to work with masks and external executables: `exe_mask_wrapper()`. To exclude the second parameter $x_1$ from screening of the Ishigami-Homma function again, $x_0$ and `mask` must be given to `exe_mask_wrapper()` as well. Remember that `mask` is an include-mask, i.e. all `mask==True` will be screened and all `mask==False` will not be screened:

```python
mask = np.ones(npars, dtype=bool) # True -> include
mask[1] = False # False -> exclude
x0 = np.ones(npars) * 0.5
```

(continues on next page)
Additional arguments for external executables

Further arguments to the external executable can be given simply by adding it to the call string. For example, if \(a\) and \(b\) were command line arguments to `ishiexe.py`, they could simply be given in the function name:

```python
ishi = ['python3', 'ishiexe.py', '-a str(a)', '-b str(b)']
```

Further arguments of wrappers

The user can pass further arguments to `exe_wrapper()` and `exe_mask_wrapper()` via a dictionary at the end of the call. Setting the key `shell` to `True` passes `shell=True` to `subprocess.check_output()`, which makes `subprocess.check_output()` open a shell for running the external executable. Note that the `args` in `subprocess` must be a string if `shell=True` and a list if `shell=False`. Setting the key `debug` to `True` uses `subprocess.check_call()` so that any output of the external executable will be written to the screen (precisely `subprocess.STDOUT`). This especially prints out also any errors that might have occurred during execution:

```python
ishi = 'python ishiexe.py'
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_writer,
               objectivefile, standard_objective_reader,
               {'shell':True, 'debug':True})
out = ee(func, lb, ub, 10)
```

This mechanism allows passing also additional arguments and keyword arguments to the parameter writer. Setting `pargs` to a list of arguments and `pkwargs` to a dictionary with keyword arguments passes them to the parameter writer as:

```python
parameterwriter(parameterfile, x, *pargs, **pkwargs)
```

Say an external program uses a `parameterfile` that has five informations per line: 1. identifier, 2. current parameter value, 3. minimum parameter value, 4. maximum parameter value, 5. parameter mask, e.g.:

```
# value min max mask
1 0.5 -3.1415 3.1415 1
2 0.0 -3.1415 3.1415 0
3 1.0 -3.1415 3.1415 1
```

One can use `standard_parameter_reader_bounds_mask()` in this case. Parameter bounds and mask can be passed via `pargs`:

```python
from pyee.utils import standard_parameter_reader_bounds_mask
ishi = ['python', 'ishiexe.py']
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_reader_bounds_mask,
               objectivefile, standard_objective_reader,
               {'pargs':[lb,ub,mask]})
out = ee(func, lb, ub, 10)
```

Or in case of exclusion of \(x_1\):
Another common case is that the parameters are given in the form \texttt{parameter = value}, e.g. in Fortran namelists. \texttt{pyeee} provides a function that searches parameter names on the left-hand-side of an equal sign and replaces the values on the right-hand-side of the equal sign with the sampled parameter values. The parameterfile might look like:

```bash
&params
 x0 = 0.5
 x1 = 0.0
 x2 = 1.0
/
```

The function \texttt{sub_names_params_files()} (which is identical to \texttt{sub_names_params_files_ignorecase()}) can be used and parameter names are passed via \texttt{pargs}:

```python
from pyeee.utils import sub_names_params_files
pnames = ['x0', 'x1', 'x2']
func = partial(exe_wrapper, ishi, parameterfile, sub_names_params_files, objectivefile, standard_objective_reader, ('pargs':[pnames], 'pid':True))
out = ee(func, lb, ub, 10)
```

\texttt{parameterfile} can be a list of parameterfiles in case of \texttt{sub_names_params_files()}, \texttt{pid} will be explained in the next section. Note that \texttt{pargs} is set to [\texttt{pnames}]. Setting `pargs':\texttt{pnames} would give *\texttt{pnames} to the parameterwriter, that means each parameter name as an individual argument, which would be wrong because it wants to have a list of parameter names. The docstring of \texttt{exe_wrapper()} states:

```python
Wrapper function for external programs using a parameterwriter with the interface:
   parameterwriter(parameterfile, x, *pargs, **pkwargs)
or if pid==True:
   parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)
```

And the definition of \texttt{sub_names_params_files()} is:

```python
def sub_names_params_files_ignorecase(files, pid, params, names):
```

so *\texttt{pargs} passes *[\texttt{pnames}] that means \texttt{pnames} as argument after the parameters to \texttt{sub_names_params_files()}. Excluding \texttt{x1} would then be achieved by simply excluding \texttt{x1} from \texttt{pnames}:

```python
from pyeee.utils import sub_names_params_files
pnames = ['x0', 'x2']
func = partial(exe_wrapper, ishi, parameterfile, sub_names_params_files, objectivefile, standard_objective_reader, ('pargs':[pnames], 'pid':True))
out = ee(func, lb, ub, 10)
```

### Parallel processing of external executables

Elementary Effects run the computational model $nt*(npars+1)$ times. All model runs are independent and can be executed at the same time if computing resources permit. Even simple personal computers have computing
cores nowadays. If the computational model is run several times in the same directory at the same time, all model runs would read the same parameter file and overwrite the output of each other.

`exe_wrapper()` concatenates an individual integer number to the function string (or list, see `subprocess`), adds the integer to call of `parameterwrite` and appends the number to the `objectivefile`, like:

```python
pid = str(randint())
parameterwriter(parameterfile, pid, x, *pargs, **pkwars)
err = subprocess.check_output([func, pid])
obj = objectivereader(objectivefile+'.'+pid)
os.remove(parameterfile+'.'+pid)
os.remove(objectivefile+'.'+pid)
```

The `parameterwriter` is supposed to write `parameterfile+`'+ipid`

`ishiexe.py` would then need to read the number from the command line:

```python
# File: ishiexel.py

# read pid if given
import sys
pid = None
if len(sys.argv) > 1:
    pid = sys.argv[1]

# Ishigami-Homma function a=b=1
import numpy as np
def ishigami1(x):
    return np.sin(x[0]) + np.sin(x[1])**2 + x[2]**4 * np.sin(x[0])

# read parameters
from pyee.utils import standard_parameter_reader
pfile = 'params.txt'
if pid is not None:
    pfile = pfile+'.'+pid
x = standard_parameter_reader(pfile)

# calc function
y = ishigami1(x)

# write objective
ofile = 'obj.txt'
if pid is not None:
    ofile = ofile+'.'+pid
ff = open(ofile, 'w')
print(y, file=ff)
ff.close()
```

`exe_wrapper()` would then be used with `pid=True` and one can use several parallel processes:

```python
from pyee.utils import exe_wrapper, standard_parameter_writer, standard_objective_reader
ishi = ['python3', 'ishiexel.py']
parameterfile = 'params.txt'
objectivefile = 'obj.txt'
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_writer,
               objectivefile, standard_objective_reader, {'pid':True})
npars = 3
lb = np.ones(npars) * (-np.pi)
ub = np.ones(npars) * np.pi
out = ee(func, lb, ub, 10, processes=8)
```

Note that `sub_names_params_files()` writes `parameterfile+`'+ipid` and does not work with `pid=False`.
If you cannot change your computational model, you can use, for example, a bash script that launches each model run in a separate directory, like:

```
#!/bin/bash
# File: ishiexe.sh
# get pid
pid=$1

# make individual run directory
mkdir tmp.$(pid)

# run in individual directory
cp ishiexe.py tmp.$(pid)/
mv params.txt.$(pid) tmp.$(pid)/params.txt
cd tmp.$(pid)
python ishiexe.py

# make output available to pyee
mv obj.txt ../obj.txt.$(pid)

# clean up
cd..
rmdir tmp.$(pid)
```

which would then be used:

```python
from functools import partial
from pyee.utils import exe_wrapper, standard_parameter_writer, standard_objective_reader
ishi = './ishiexe.sh'
parameterfile = 'params.txt'
objectivefile = 'obj.txt'
func = partial(exe_wrapper, ishi, parameterfile, standard_parameter_writer, objectivefile, standard_objective_reader, {'pid':True, 'shell':True})
npars = 3
lb = np.ones(npars) * (-np.pi)
ub = np.ones(npars) * np.pi
from pyee import ee
out = ee(func, lb, ub, 10, processes=8)
```

Such a script could be written in Python as well, of course, if the bash shell is not available, e.g. on Windows.

That’s all Folks!
The easiest way to install is via pip:

```
pip install pyeee
```

### 3.1 Manual install

The latest version of *pyeee* can be installed from source:

```
git clone https://github.com/mcuntz/pyeee.git
cd pyeee
pip install .
```

### 3.2 Local install

Users without install privileges can append the `--user` flag to `pip` either while installing from Python Package Index (PyPI):

```
pip install pyeee --user
```

or from the top *pyeee* directory:

```
git clone https://github.com/mcuntz/pyeee.git
cd pyeee
pip install . --user
```

If `pip` is not available, then `setup.py` can still be used:

```
python setup.py install --user
```

When using `setup.py` locally, it might be that one needs to append `--prefix=` to the command:

```
python setup.py install --user --prefix=
```
3.3 Dependencies

`pyeee` uses the packages `numpy`, `scipy` and `schimmbad`. They are all available in PyPI and `pip` should install them automatically. Installations via `setup.py` might need to install the three dependencies first.
4.1 pyeee.eee

eee (func, *args, **kwargs)


Note, the input function must be callable as func(x).

Parameters

- **func** (callable) – Python function callable as func(x) with x the function parameters.
- **lb** (array_like) – Lower bounds of parameters or lower fraction of percent point function ppf if distribution given.
  Be aware that the percent point function ppf of most continuous distributions is infinite at 0.
- **ub** (array_like) – Upper bounds of parameters or upper fraction of percent point function ppf if distribution given.
  Be aware that the percent point function ppf of most continuous distributions is infinite at 1.
- **x0** (array_like, optional) – Parameter values used with mask==0.
- **mask** (array_like, optional) – Include (1,True) or exclude (0,False) parameters in screening (default: include all parameters).
- **ntfirst** (int, optional) – Number of trajectories in first step of sequential elementary effects (default: 5).
- **ntlast** (int, optional) – Number of trajectories in last step of sequential elementary effects (default: 5).
- **nsteps** (int, optional) – Number of intervals for each trajectory (default: 6)
- **dist** (list, optional) – List of None or scipy.stats distribution objects for each factor having the method ppf, Percent Point Function (Inverse of CDF) (default: None)
  If None, the uniform distribution will be sampled from lower bound LB to upper bound UB.
  If dist is scipy.stats.uniform, the ppf will be sampled from the lower fraction given in LB and the upper fraction in UB. The sampling interval is then given by the parameters loc=lower and scale=interval=upper-lower in distparam. This means dist=None, LB=a, UB=b corresponds to LB=0, UB=1, dist=scipy.stats.uniform, distparam=[a,b-a]
- **distparam** (list, optional) – List with tuples with parameters as required for dist (default: (0,1)).
  All distributions of scipy.stats have location and scale parameters, at least. loc and scale are implemented as keyword arguments in scipy.stats. Other parameters such as the shape parameter of the gamma distribution must hence be given first, e.g. (shape,loc,scale) for the gamma distribution.
  distparam is ignored if dist is None.
  The percent point function ppf is called like this: dist(*distparam).pdf(x)
- **weight** (boolean, optional) – If False, use the arithmetic mean mu* for each parameter if function has multiple outputs, such as the mean mu* of each time step of a time series (default).
  If True, return weighted mean mu*, weighted by sd.
• **seed** *(int or array_like)* – Seed for numpy’s random number generator (default: None).

• **processes** *(int, optional)* – The number of processes to use to evaluate objective function and constraints (default: 1).

• **pool** *(schwimmbad pool object, optional)* – Generic map function used from module schwimmbad, which provides, serial, multiprocessor, and MPI mapping functions (default: None).

  The pool is chosen with:

  schwimmbad.choose_pool(mpi=True/False, processes=processes).

  The pool will be chosen automatically if pool is None.

  MPI pools can only be opened and closed once. If you want to use screening several times in one program, then you have to choose the pool, pass it to eee, and later close the pool in the calling program.

• **verbose** *(int, optional)* – Print progress report during execution if verbose>0 (default: 0).

• **logfile** *(File handle or logfile name)* – File name of possible log file (default: None = no logfile will be written).

• **plotfile** *(Plot file name)* – File name of possible plot file with fit of logistic function to mu* of first trajectories (default: None = no plot produced).

**Returns**  
mask – *(len(lb),)* Mask with 1=informative and 0=uninformative model parameters, to be used with ‘&’ on input mask.

**Return type**  
ndarray

**See also:**

screening() : Elementary Effects, same as

ee() : Elementary Effects

**Examples**

```python
>>> from functools import partial
>>> import numpy as np
>>> import scipy.stats as stats
>>> from pyeee.functions import G
>>> from pyeee.utils import func_wrapper

>>> seed = 1234
>>> np.random.seed(seed=seed)

>>> func = G

>>> npars = 6

>>> params = [78., 12., 0.5, 2., 97., 33.] # G

>>> arg = [params]

>>> kwarg = {}

>>> obj = partial(func_wrapper, func, arg, kwarg)

>>> lb = np.zeros(npars)

>>> ub = np.ones(npars)

>>> ntfirst = 10

>>> ntlast = 5

>>> nsteps = 6

>>> out = eee(obj, lb, ub, mask=None, ntfirst=ntfirst, ntlast=ntlast, nsteps=nsteps, processes=4)

>>> print(np.where(out)[0] + 1)
[2 3 4 6]
```
**see** (*func, *args, **kwargs*)

Wrapper function for *eee()*. 
4.2 pyeee.screening

screening (func, lb, ub, nt, x0=None, mask=None, nsteps=6, ntotal=None, dist=None, distparam=None, seed=None, processes=1, pool=None, verbose=0)
Parameter screening using Morris method of Elementary Effects.

Note, the input function must be callable as func(x).

Parameters

- **func (callable)** – Python function callable as func(x) with x the function parameters.
- **lb (array_like)** – Lower bounds of parameters or lower fraction of percent point function ppf if distribution given.
  Be aware that the percent point function ppf of most continuous distributions is infinite at 0.
- **ub (array_like)** – Upper bounds of parameters or upper fraction of percent point function ppf if distribution given.
  Be aware that the percent point function ppf of most continuous distributions is infinite at 1.
- **nt (int)** – Number of trajectories used for screening.
- **x0 (array_like, optional)** – Parameter values used with mask==0.
- **mask (array_like, optional)** – Include (1, True) or exclude (0, False) parameters in screening (default: include all parameters).
- **nsteps (int, optional)** – Number of steps along one trajectory (default: 6)
- **ntotal (int, optional)** – Total number of sampled trajectories to select the nt most different trajectories. If None: max(nt**2, 10*nt) (default: None)
- **dist (list, optional)** – List of None or scipy.stats distribution objects for each factor having the method ppf, Percent Point Function (Inverse of CDF) (default: None)
  If None, the uniform distribution will be sampled from lower bound LB to upper bound UB.
  If dist is scipy.stats.uniform, the ppf will be sampled from the lower fraction given in LB and the upper fraction in UB. The sampling interval is then given by the parameters loc=lower and scale=interval=upper-lower in distparam. This means dist=None, LB=a, UB=b corresponds to LB=0, UB=1, dist=scipy.stats.uniform, distparam=[a, b-a]
- **distparam (list, optional)** – List with tuples with parameters as required for dist (default: (0,1)).
  All distributions of scipy.stats have location and scale parameters, at least. loc and scale are implemented as keyword arguments in scipy.stats. Other parameters such as the shape parameter of the gamma distribution must hence be given first, e.g. (shape, loc, scale) for the gamma distribution.
  distparam is ignored if dist is None.
  The percent point function ppf is called like this: dist(*distparam).ppf(x)
- **seed (int or array_like)** – Seed for numpy’s random number generator (default: None).
- **processes (int, optional)** – The number of processes to use to evaluate objective function and constraints (default: 1).
- **pool (schwimmbad pool object, optional)** – Generic map function used from module schwimmbad, which provides, serial, multiprocessor, and MPI mapping functions (default: None).
The pool is chosen with:

```
schwimmbad.choose_pool(mpi=True/False, processes=processes).
```

The pool will be chosen automatically if pool is None.

MPI pools can only be opened and closed once. If you want to use screening several times in one program, then you have to choose the pool, pass it to screening, and later close the pool in the calling program.

- **verbose** *(int, optional)* – Print progress report during execution if verbose>0 (default: 0).

Returns

if nt>1:

2D-array - (nparameter,3) with per parameter

1. mean of absolute elementary effects over all nt trajectories (mu*)
2. mean of elementary effects over all nt trajectories (mu)
3. standard deviation of elementary effects over all nt trajectories (sigma)

else:

2D-array - (nparameter,3) with per parameter

1. absolute elementary effect of each parameter
2. elementary effect of each parameter
3. zeros

**Return type** *(nparameter,3) ndarray*

See also:

- `eee()` : Efficient Elementary Effects, same as
- `see()` : Sequential Elementary Effects

Examples

```python
>>> from functools import partial
>>> import numpy as np
>>> from pyeee.utils import func_wrapper
>>> from pyeee.functions import fmorris
>>> seed = 1023
>>> np.random.seed(seed=seed)
>>> npars = 20
>>> beta0 = 0.
>>> beta1 = np.random.standard_normal(npars)
>>> beta2 = np.random.standard_normal((npars,npars))
>>> beta3 = np.zeros((npars,npars,npars))
>>> beta3[:5,:5,:5] = -10.
>>> beta4 = np.zeros((npars,npars,npars,npars))
>>> beta4[:,:4,:4,:4] = 5.
>>> arg = [beta0, beta1, beta2, beta3, beta4]
>>> kwarg = {}
>>> func = partial(func_wrapper, fmorris, arg, kwarg)
>>> lb = np.zeros(npars)
>>> ub = np.ones(npars)
>>> nt = 20
>>> ntotal = 10*nt
```

(continues on next page)


```python
>>> nsteps = 6
>>> verbose = 0
>>> out = screening(func, lb, ub, nt, x0=None, mask=None, nsteps=nsteps,
                 ntotal=ntotal, processes=4, verbose=verbose)
>>> print(out[0:3,0])
[60.7012889 67.33372626 48.46673528]
```

**ee(**\*args, **kwargs\*)**

Wrapper function for `screening()`.
4.3 pyeee.morris

`morris_sampling` \((nparam, LB, UB, nt, nsteps=6, ntotal=None, dist=None, distparam=None, GroupMat=np.array([], dtype=float64), Diagnostic=0)\)

Sample trajectories in parameter space.

Optimisation in the choice of trajectories for Morris experiment, that means elementary effects.

**Parameters**

- `nparam` \((int)\) – Number of parameters / factors
- `LB` \((array_like)\) – \((nparam,)\) Lower bound of the uniform distribution for each parameter / factor or lower fraction of percent point function ppf if distribution given.
  
  Be aware that the percent point function ppf of most continuous distributions is infinite at 0.
- `UB` \((array_like)\) – \((nparam,)\) Upper bound of the uniform distribution for each parameter / factor or upper fraction of percent point function ppf if distribution given.
  
  Be aware that the percent point function ppf of most continuous distributions is infinite at 1.
- `nt` \((int)\) – Final number of optimal trajectories
- `nsteps` \((int, optional)\) – Number of levels, i.e. intervals in trajectories (default: 6)
- `ntotal` \((int, optional)\) – Total number of sampled trajectories. If None: ntotal=10*nt (default: None)
- `dist` \((list, optional)\) – List of None or scipy.stats distribution objects for each factor having the method ppf, Percent Point Function (Inverse of CDF) (default: None)
  
  If None, the uniform distribution will be sampled from lower bound LB to upper bound UB.
  
  If dist is scipy.stats.uniform, the ppf will be sampled from the lower fraction given in LB and the upper fraction in UB. The sampling interval is then given by the parameters loc=lower and scale=interval=upper-lower in distparam. This means dist=None, LB=a, UB=b corresponds to LB=0, UB=1, dist=scipy.stats.uniform, distparam=[a,b-a]
- `distparam` \((list, optional)\) – List with tuples with parameters as required for dist (default: (0,1)).
  
  All distributions of scipy.stats have location and scale parameters, at least. loc and scale are implemented as keyword arguments in scipy.stats. Other parameters such as the shape parameter of the gamma distribution must hence be given first, e.g. (shape,loc, scale) for the gamma distribution.
  
  distparam is ignored if dist is None.
  
  The percent point function ppf is called like this: dist(*distparam).ppf(x)
- `GroupMat` \((ndarray, optional)\) – \((nparam,nrquanp\) Matrix describing the groups. (default: np.array([],
  
  Each column represents a group. The elements of each column are zero if the parameter / factor is not in the group, otherwise it is 1.
- `Diagnostic` \((int, optional)\) – 1: plot the histograms and compute the efficiency of the sampling or not,
  
  0: otherwise (default)

**Returns**

- `traj` – list [OptMatrix, OptOutVec] with Optmatrix((nparam+1)*nt,nparam) and OptOutVec(nparam*nt)
Return type: list

References


Examples

```python
>>> import numpy as np
>>> seed = 1023
>>> np.random.seed(seed=seed)
>>> npara = 10
>>> x0 = np.ones(npara) * 0.5
>>> lb = np.zeros(npara)
>>> ub = np.ones(npara)
>>> mask = np.ones(npara, dtype=np.bool)
>>> mask[5::2] = False
>>> nmask = np.sum(mask)
>>> nt = npara
>>> ntotal = max(nt**2, 10*nt)
>>> nsteps = 6
>>> tmatrix, tvec = morris_sampling(nmask, lb[mask], ub[mask], nt,
                                           nsteps=nsteps, ntotal=ntotal, Diagnostic=False)
>>> # Set input vector to trajectories and masked elements = x0
>>> x = np.tile(x0, tvec.size).reshape(tvec.size, npara) # default to x0
>>> x[:, mask] = tmatrix # replaced unmasked with trajectory values
>>> print(x[0,:])
[0.6 0.4 0.8 0.6 0.6 0.5 0.4 0.5 0. 0.5]
```

```python
>>> import scipy.stats as stats
>>> seed = 1023
>>> np.random.seed(seed=seed)
>>> npara = 10
>>> x0 = np.ones(npara) * 0.5
>>> lb = np.zeros(npara)
>>> ub = np.ones(npara)
>>> dist = [ stats.uniform for i in range(npara) ]
>>> dpara = [ (lb[i],ub[i]-lb[i]) for i in range(npara) ]
>>> mask = np.ones(npara, dtype=np.bool)
>>> mask[5::2] = False
>>> nmask = np.sum(mask)
>>> nt = npara
>>> ntotal = max(nt**2, 10*nt)
>>> nsteps = 6
>>> tmatrix, tvec = morris_sampling(nmask, lb[mask], ub[mask], nt,
                                           nsteps=nsteps, ntotal=ntotal, dist=dist, distparam=dpara, Diagnostic=False)
>>> # Set input vector to trajectories and masked elements = x0
>>> x = np.tile(x0, tvec.size).reshape(tvec.size, npara) # default to x0
>>> x[:, mask] = tmatrix # replaced unmasked with trajectory values
>>> print(x[0,:])
[0.6 0.4 0.8 0.6 0.6 0.5 0.4 0.5 0. 0.5]
```

elementary_effects(nparam, OptMatrix, OptOutVec, Output, nsteps=4, Group=[], Diagnostic=False)

Compute the Morris measures given the Morris sample matrix, the output values and the group matrix.

Parameters

- `nparam` (int) – Number of parameters / factors list [OptMatrix, OptOutVec] with OptMatrix((nparam+1)*nt,nparam) and OptOutVec(nparam*nt)
**OptMatrix** *(ndarray)* – \(((nparam+1)*nt,nparam)\) Matrix of the Morris sampled trajectories from morris_sampling

**OptOutVec** *(ndarray)* – \((nparam*nt,)\) Matrix with the parameter / factor changings from morris_sampling

**Output** *(ndarray)* – \(((nparam+1)*nt,)\) Matrix of the output values of each point of each trajectory

**nsteps** *(int, optional)* – Number of levels, i.e. intervals in trajectories (default: 4)

**Group** *(ndarray, optional)* – \((nparam,NumGroups)\) Matrix describing the groups. (default: [ ])

Each column represents a group. The elements of each column are zero if the parameter / factor is not in the group, otherwise it is 1.

**Diagnostic** *(boolean, optional)* – True: print out diagnostics
False: otherwise (default)

**Returns**

\(SA, OutMatrix\) – \(SA(nparam*Output.shape[1],N)\) individual sensitivity measures

OutMatrix\((nparam*Output.shape[1], 3) = [Mu*, Mu, StDev]\) Morris Measures

It gives the three measures of each parameter / factor for each output.

**Return type** list of ndarrays

**References**


**Examples**

```python
>>> import numpy as np
>>> seed = 1023
>>> np.random.seed(seed=seed)
>>> npara = 10
>>> x0 = np.ones(npara) *0.5
>>> lb = np.zeros(npara)
>>> ub = np.ones(npara)
>>> mask = np.ones(npara, dtype=np.bool)
>>> mask[5::2] = False
>>> nmask = np.sum(mask)
>>> nt = npara
>>> ntotal = max(nt**2, 10*nt)
>>> nsteps = 6
>>> tmat, tvec = morris_sampling(nmask, lb[mask], ub[mask], nt, nbatches=-ntotal-nntotal, nsteps=nsteps, Diagnostic=False)
>>> # Set input vector to trajectories and masked elements = x0
>>> x = np.tile(x0, tvec.size).reshape(tvec.size, npara)
# default to x0
>>> x[:,:,mask] = tmat
# replaced unmasked with trajectory values
>>> func = np.sum
>>> fx = np.array(list(map(func, x)))
>>> out = np.zeros((npara,3))
>>> sa, res = elementary_effects(nmask, tmat, tvec, fx, nsteps=nsteps, Diagnostic=False)
>>> out[mask,:] = res
```
>>> print(out[:,0])
[1. 1. 1. 1. 0. 1. 0. 1. 0.]

4.3. pyeee.morris
4.4 pyee général_functions

**curvature** *(x, dfunc, d2func, *args, **kwargs)*

Curvature of function:

\[
f''/(1+f'^2)^{3/2}
\]

**Parameters**

- *x* *(array_like)* – Independent variable to evaluate curvature
- *dfunc* *(callable)* – Function giving first derivative of function \( f' \), to be called \( dfunc(x, *args, **kwargs) \)
- *d2func* *(callable)* – Function giving second derivative of function \( f'' \), to be called \( d2func(x, *args, **kwargs) \)
- *args* *(iterable)* – Arguments passed to *dfunc* and *d2func*
- *kwargs* *(dict)* – Keyword arguments passed to *dfunc* and *d2func*

**Returns**

Curvature of function \( f \) at *x*

**Return type**

float or ndarray

**logistic** *(x, L, k, x0)*

Logistic function:

\[
L/(1+exp(-k(x-x0)))
\]

**Parameters**

- *x* *(array_like)* – Independent variable to evaluate logistic function
- *L* *(float)* – Maximum of logistic function
- *k* *(float)* – Steepness of logistic function
- *x0* *(float)* – Inflection point of logistic function

**Returns**

Logistic function at *x* with maximum \( L \), steepness \( k \) and inflection point \( x0 \)

**Return type**

float or ndarray

**logistic_p** *(x, p)*

Wrapper function for *logistic(): logistic(x, *p).*

**dlogistic** *(x, L, k, x0)*

First derivative of logistic function:

\[
L(1+exp(-k(x-x0)))
\]

which is

\[
k.L/(2(cosh(k(x-x0))+1))
\]

**Parameters**

- *x* *(array_like)* – Independent variable to evaluate derivative of logistic function
- *L* *(float)* – Maximum of logistic function
- *k* *(float)* – Steepness of logistic function
- *x0* *(float)* – Inflection point of logistic function

**Returns**

First derivative of logistic function at *x* with maximum \( L \), steepness \( k \) and inflection point \( x0 \)

**Return type**

float or ndarray
dlogistic_p(x, p)
Wrapper function for dlogistic(): dlogistic(x, *p).

d2logistic(x, L, k, x0)
Second derivative of logistic function:

\[ L/(1+\exp(-k(x-x0))) \]

which is

\[-k^2L\sinh(k(x-x0))/(2(\cosh(k(x-x0))+1)^2)\]

Parameters

- x (array_like) – Independent variable to evaluate derivative of logistic function
- L (float) – Maximum of logistic function
- k (float) – Steepness of logistic function
- x0 (float) – Inflection point of logistic function

Returns Second derivative of logistic function at x with maximum L, steepness k and inflection point x0

Return type float or ndarray

d2logistic_p(x, p)
Wrapper function for d2logistic(): d2logistic(x, *p).

logistic_offset(x, L, k, x0, a)
Logistic function with offset:

\[ L/(1+\exp(-k(x-x0))) + a \]

Parameters

- x (array_like) – Independent variable to evaluate logistic function
- L (float) – Maximum of logistic function
- k (float) – Steepness of logistic function
- x0 (float) – Inflection point of logistic function
- a (float) – Offset of logistic function

Returns Logistic function at x with maximum L, steepness k, inflection point x0 and offset a

Return type float or ndarray

logistic_offset_p(x, p)
Wrapper function for logistic_offset(): logistic_offset(x, *p).

dlogistic_offset(x, L, k, x0, a)
First derivative of logistic function with offset:

\[ L/(1+\exp(-k(x-x0))) + a \]

which is

\[ kL/(2(cosh(k(x-x0))+1)) \]

Parameters

- x (array_like) – Independent variable to evaluate derivative of logistic function
- L (float) – Maximum of logistic function
- k (float) – Steepness of logistic function
- x0 (float) – Inflection point of logistic function
• **a** (*float*) – Offset of logistic function

**Returns** First derivative of logistic function with offset at \( x \) with maximum \( L \), steepness \( k \), inflection point \( x_0 \), and offset \( a \)

**Return type** float or ndarray

```python
def dlogistic_offset_p(x, p):
    return dlogistic_offset(x, *p).
```

```python
def d2logistic_offset(x, L, k, x0, a):
    return L/(1+exp(-k*(x-x0))) + a
```

which is

\[-k^2 . L . \sinh(k(x-x0)) / (2(\cosh(k(x-x0))+1)^2)\]

**Parameters**

- **\( x \)** (*array_like*) – Independent variable to evaluate derivative of logistic function
- **\( L \)** (*float*) – Maximum of logistic function
- **\( k \)** (*float*) – Steepness of logistic function
- **\( x_0 \)** (*float*) – Inflection point of logistic function
- **\( a \)** (*float*) – Offset of logistic function

**Returns** Second derivative of logistic function at \( x \) with maximum \( L \), steepness \( k \), inflection point \( x_0 \), and offset \( a \)

**Return type** float or ndarray

```python
def d2logistic_offset_p(x, p):
    return d2logistic_offset(x, *p).
```

```python
def logistic2_offset(x, L1, k1, x01, L2, k2, x02, a):
    return L1/(1+exp(-k1*(x-x01))) - L2/(1+exp(-k2*(x-x02))) + a
```

**Parameters**

- **\( x \)** (*array_like*) – Independent variable to evaluate logistic function
- **\( L_1 \)** (*float*) – Maximum of first logistic function
- **\( k_1 \)** (*float*) – Steepness of first logistic function
- **\( x_{01} \)** (*float*) – Inflection point of first logistic function
- **\( L_2 \)** (*float*) – Maximum of second logistic function
- **\( k_2 \)** (*float*) – Steepness of second logistic function
- **\( x_{02} \)** (*float*) – Inflection point of second logistic function
- **\( a \)** (*float*) – Offset of double logistic function

**Returns** Double Logistic function at \( x \)

**Return type** float or ndarray

```python
def logistic2_offset_p(x, p):
    return logistic2_offset(x, *p).
```

```python
def dlogistic2_offset(x, L1, k1, x01, L2, k2, x02, a):
    return 32 Chapter 4. pyee API
```
\[ \frac{L_1}{1+\exp(-k_1(x-x_{01}))} - \frac{L_2}{1+\exp(-k_2(x-x_{02}))} + a \]

which is

\[ k_1 \cdot L_1 / (2(\cosh(k_1(x-x_{01}))+1)) - k_2 \cdot L_2 / (2(\cosh(k_2(x-x_{02}))+1)) \]

**Parameters**

- \( x \) (array_like) – Independent variable to evaluate logistic function
- \( L_1 \) (float) – Maximum of first logistic function
- \( k_1 \) (float) – Steepness of first logistic function
- \( x_{01} \) (float) – Inflection point of first logistic function
- \( L_2 \) (float) – Maximum of second logistic function
- \( k_2 \) (float) – Steepness of second logistic function
- \( x_{02} \) (float) – Inflection point of second logistic function
- \( a \) (float) – Offset of double logistic function

**Returns** First derivative of double logistic function with offset at \( x \)

**Return type** float or ndarray

```python
dlogistic2_offset_p(x, p)
```
Wrapper function for `dlogistic2_offset`: `dlogistic2_offset(x, *p)`.

**d2logistic2_offset**

Second derivative of double logistic function with offset:

\[ \frac{L_1}{1+\exp(-k_1(x-x_{01}))} - \frac{L_2}{1+\exp(-k_2(x-x_{02}))} + a \]

which is

\[ -k_1^2 \cdot L_1 \cdot \sinh(k_1(x-x_{01}))/ (2(\cosh(k_1(x-x_{01}))+1)^2) + k_2^2 \cdot L_2 \cdot \sinh(k_2(x-x_{02}))/ (2(\cosh(k_2(x-x_{02}))+1)^2) \]

**Parameters**

- \( x \) (array_like) – Independent variable to evaluate logistic function
- \( L_1 \) (float) – Maximum of first logistic function
- \( k_1 \) (float) – Steepness of first logistic function
- \( x_{01} \) (float) – Inflection point of first logistic function
- \( L_2 \) (float) – Maximum of second logistic function
- \( k_2 \) (float) – Steepness of second logistic function
- \( x_{02} \) (float) – Inflection point of second logistic function
- \( a \) (float) – Offset of double logistic function

**Returns** Second derivative of double logistic function with offset at \( x \)

**Return type** float or ndarray

```python
d2logistic2_offset_p(x, p)
```
Wrapper function for `d2logistic2_offset`: `d2logistic2_offset(x, *p)`.
### 4.5 pyeee.sa_test_functions

**B\((X)\)**


**Parameters**

\(X\ (array\_like) – (nX,)\) or \((nX,npoints)\) array of floats

**Returns**

B – float or \((npoints,)\) floats of B function values at \(X\)

**Return type**

float or ndarray

**g\((X, a)\)**

G-function


**Parameters**

\(X\ (array\_like) – (nX,)\) or \((nX,npoints)\) array of floats

\(a\ (array\_like) – (nX,)\) array of floats

**Returns**

G – float or \((npoints,)\) floats of G function values at \(X\) with parameters \(a\)

**Return type**

float or ndarray

**G\((X, a)\)**

G-function


**Parameters**

\(X\ (array\_like) – (nX,)\) or \((nX,npoints)\) array of floats

\(a\ (array\_like) – (nX,)\) array of floats

**Returns**

g – float or \((npoints,)\) floats of G function values at \(X\) with parameters \(a\)

**Return type**

float or ndarray

**Gstar\((X, alpha, delta, a)\)**


**Parameters**

\(X\ (array\_like) – (nX,)\) or \((nX,npoints)\) array of floats

\(alpha\ (array\_like) – (nX,)\) array of floats

\(delta\ (array\_like) – (nX,)\) array of floats

\(a\ (array\_like) – (nX,)\) array of floats

**Returns**

G* – float or \((npoints,)\) floats of G* function values at \(X\) with parameters \(alpha, delta\) and \(a\)

**Return type**

float or ndarray

**K\((X)\)**


**Parameters**

\(X\ (array\_like) – (nX,)\) or \((nX,npoints)\) array of floats

**Returns**

K – float or \((npoints,)\) floats of K function values at \(X\)

**Return type**

float or ndarray

**bratley\(*args\)**

**Parameters**

\(X\) (array_like) – (nX,) or (nX,npoints) array of floats

**Returns**

`bratley` – float or (npoints,) floats of \(K\) function values at \(X\)

**Return type**

float or ndarray

**fmorris**

\((X, \beta_0, \beta_1, \beta_2, \beta_3, \beta_4)\)

Morris-function, Morris (1991) Technometrics 33, 161-174

**Parameters**

- \(X\) (array_like) – (20,) or (20,npoints) array of floats
- \(\beta_0\) (float) – float
- \(\beta_1\) (array_like) – (20,) array of floats
- \(\beta_2\) (array_like) – (20,20) array of floats
- \(\beta_3\) (array_like) – (20,20,20) array of floats
- \(\beta_4\) (array_like) – (20,20,20,20) array of floats

**Returns**

`fmorris` – float or (npoints,) floats of Morris function values at \(X\) with parameters \(\beta_0-\beta_4\)

**Return type**

float or ndarray

**morris**

\("*args"\)

Morris-function, Morris (1991) Technometrics 33, 161-174

**Parameters**

- \(X\) (array_like) – (20,) or (20,npoints) array of floats
- \(\beta_0\) (float) – float
- \(\beta_1\) (array_like) – (20,) array of floats
- \(\beta_2\) (array_like) – (20,20) array of floats
- \(\beta_3\) (array_like) – (20,20,20) array of floats
- \(\beta_4\) (array_like) – (20,20,20,20) array of floats

**Returns**

`morris` – float or (npoints,) floats of Morris function values at \(X\) with parameters \(\beta_0-\beta_4\)

**Return type**

float or ndarray

**oakley_ohagan**

\((X)\)


**Parameters**

\(X\) (array_like) – (15,) or (15,npoints) array of floats

**Returns**

`oakley_ohagan` – float or (npoints,) floats of Oakley and O’Hagan function values at \(X\)

**Return type**

float or ndarray

**ishigami_homma**

\((X, a, b)\)

Ishigami and Homma (1990), given by Saltelli et al. (2008, page 179)

**Parameters**

- \(X\) (array_like) – (3,) or (3,npoints) array of floats
- \(a\) (array_like) – float or (npoints,) array of floats
- \(b\) (array_like) – float or (npoints,) array of floats

**Returns**

`ishigami_homma` – float or (npoints,) floats of Ishigami and Homma function values at \(X\) with parameters \(a\) and \(b\)

**Return type**

float or ndarray
linear \( (X, a, b) \)
Linear test function to test PAWN method:
\[
Y = a \times X + b
\]

Parameters
- \( X \) (array_like) – (1,) or (1,npoints) array of floats
- \( a \) (array_like) – float or (npoints,) array of floats
- \( b \) (array_like) – float or (npoints,) array of floats

Returns linear – float or (npoints,) floats of linear function values at \( X \) with parameters \( a \) and \( b \)

Return type float or ndarray

product \( (X) \)
Product test function to test PAWN method:
\[
Y = X[0] \times X[1]
\]

Parameters \( X \) (array_like) – (2,) or (2,npoints) array of floats

Returns product – float or (npoints,) floats of product function values at \( X \)

Return type float or ndarray

ratio \( (X) \)
Ratio test function:
\[
Y = X[0] / X[1]
\]

Simple nonlinear model proposed by Liu et al. (2006):

Used by Pianosi & Wagener, Environmental Modelling & Software (2015)

Parameters \( X \) (array_like) – (2,) or (2,npoints) array of floats

Returns ratio – float or (npoints,) floats of ratio function values at \( X \)

Return type float or ndarray

ishigami_homma_easy \( (X) \)
Simplified Ishigami and Homma function to test PAWN method:
\[
Y = \sin(X[0]) + X[1]
\]
with \( X[0],X[1] \sim \text{Uniform[-Pi, Pi]} \)

Parameters \( X \) (array_like) – (2,) or (2,npoints) array of floats

Returns ishigami_homma_easy – float or (npoints,) floats of simplified Ishigami and Homma function values at \( X \)

Return type float or ndarray
4.6 pyeee.utils

**cost_square** *(p, func, x, y)*

General cost function for least square optimising *func(x,p)* vs *y*.

**Parameters**

- **p** *(array_like)* – Parameters of *func*.
- **func** *(callable)* – Python function callable as *func(x,p)*.
- **x** *(array_like)* – Independent *func* parameters.
- **y** *(array_like)* – Target values for *func(x,p)*.

**Returns**  
- **B** – Squared sum of *y* vs. *func(x,p)*: `sum((y-func(x,p))**2)`

**Return type**  
- float or ndarray
4.7 pyeee.function_wrapper

**exe_wrapper** *(func, parameterfile, parameterwriter, objectivefile, objectivereader, kwarg, x)*

Wrapper function for external programs using a *parameterwriter* with the interface:

```
parameterwriter(parameterfile, x, *pargs, **pkwargs)
```

or if `pid==True`:

```
parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)
```

Examples of *parameterwriter* with `pid==True` are: *standard_parameter_writer* or *sub_ia_params_files*.

To be used with `functools.partial`:

```
obj = partial(exe_wrapper, func, parameterfile, parameterwriter, objectivefile, objectivereader, 
{'shell':bool, 'debug':bool, 'pid':bool, 'pargs':list, 'pkwargs':dict})
```

This allows then calling *obj* with only the non-masked parameters:

```
fx = obj(x)
```

which translates to:

```
parameterwriter(parameterfile, x, *pargs, **pkwargs)

e = subprocess.check_output(func, stderr=subprocess.STDOUT, shell=shell)

obj = objectivereader(objectivefile)
```

or if `pid==True` to:

```
parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)

e = subprocess.check_output(func, stderr=subprocess.STDOUT, shell=shell)

obj = objectivereader(objectivefile+'.'+pid)
```

**Parameters**

- **func** *(string or list of strings)* – External program to launch by subprocess
- **parameterfile** *(string)* – Filename of parameter file
- **parameterwriter** *(callable)* – Python function writing the *parameterfile*, called as:

```
parameterwriter(parameterfile, x, *pargs, **pkwargs)
```

or if `pid==True` as:

```
parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)
```

- **objectivefile** *(string)* – Filename of file with objective values written by external executable
- **objectivereader** *(callable)* – Python function for reading objective value from *objectivefile*
- **kwarg** *(dict)* – Dictionary with keyword arguments for *exe_wrapper*. Possible arguments are:
  - **shell** *(bool)*

  If True, *subprocess* opens shell for external executable
  - **debug** *(bool)*

  If True, model output is displayed while executable is running
pid (bool)
    If True, append ‘.RandomNumber’ to parameterfile and objectivefile for parallel calls of func
pars (iterable)
    List of arguments of parameterwriters.
pkwargs (dict)
    Dictionary with keyword arguments of parameterwriters.

Returns
    Objective value calculated by the external executable func or via the objectivereader
Return type
    float

exe_mask_wrapper (func, x0, mask, parameterfile, parameterwriter, objectivefile, objectivereader, kwarg, x)
    Wrapper function for external programs using a mask and a parameterwriter with the interface:
        parameterwriter(parameterfile, x, *pargs, **pkwargs)
    or if pid==True:
        parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)
    where x are the masked parameters (include mask).
Examples of parameterwriter with pid==True are: standard_parameter_writer_bounds_mask or sub_ja_params_files.
To be used with functools.partial:
    obj = partial(exe_mask_wrapper, func, x0, mask, parameterfile, parameterwriter, objectivefile, objectivereader, {'shell':bool, 'debug':bool, 'pid':bool, 'pargs':list, 'pkwargs':dict})
This allows then calling obj with only the non-masked parameters:
    fx = obj(x)
which translates to:
    xx = np.copy(x0)
    xx[mask] = x
    parameterwriter(parameterfile, xx, *pargs, **pkwargs)
    err = subprocess.check_output(func, stderr=subprocess.STDOUT, shell=shell)
    obj = objectivereader(objectivefile)
or if pid==True to:
    xx = np.copy(x0)
    xx[mask] = x
    parameterwriter(parameterfile, pid, xx, *pargs, **pkwargs)
    err = subprocess.check_output(func, stderr=subprocess.STDOUT, shell=shell)
    obj = objectivereader(objectivefile+’.’+pid)

Parameters
    • func (string or list of strings) – External program to launch by subprocess
    • x0 (array_like) – Initial values of parameters and fixed values of masked parameters
• **mask** *(array_like)* – Mask to include (1) or exclude (0) parameter from parameter-writer

• **parameterfile** *(string)* – Filename of parameter file

• **parameterwriter** *(callable)* – Python function writing the parameterfile, called as:

  
  
  parameterwriter(parameterfile, x, *pargs, **pkwargs)

  or if pid=True:

  
  
  parameterwriter(parameterfile, pid, x, *pargs, **pkwargs)

• **objectivefile** *(string)* – Filename of file with objective values written by external executable

• **objectivereader** *(callable)* – Python function for reading objective value from objectivefile

• **kwarg** *(dict)* – Dictionary with keyword arguments for exe_mask_wrapper. Possible arguments are:

  
  shell *(bool)*

  If True, `subprocess` opens shell for external executable

  debug *(bool)*

  If True, model output is displayed while executable is running

  pid *(bool)*

  If True, append `.RandomNumber` to parameterfile and objectivefile for parallel calls of `func`

  pargs *(iterable)*

  List of arguments of parameterwriters.

  pkwargs *(dict)*

  Dictionary with keyword arguments of parameterwriters.

**Returns** Objective value calculated by the external executable `func` or via the `objectivereader`

**Return type** float

---

**func_wrapper**(func, arg, kwarg, x)

Wrapper function for Python function. To be used with partial:

  
  
  `obj = partial(func_wrapper, func, arg, kwarg)`

  
  
  This allows then calling obj with only the non-masked parameters:

  
  
  `fx = obj(x)`

  
  which translates to:

  
  
  `fx = func(x, *arg, **kwarg)`

**Parameters**

• **func** *(callable)* – Python function to be called `func(x, *arg, **kwarg)`

• **arg** *(iterable)* – Arguments passed to `func`

• **kwarg** *(dictionary)* – Keyword arguments passed to `func`

**Returns** Objective value calculated by `func`

**Return type** float
func_mask_wrapper (func, x0, mask, arg, kwarg, x)

Wrapper function for Python function using a mask. To be used with partial:

    obj = partial(func_mask_wrapper, func, x0, mask, arg, kwarg)

This allows then calling obj with only the non-masked parameters:

    fx = obj(x)

which translates to:

    xx = np.copy(x0)
    xx[mask] = x
    fx = func(xx, *arg, **kwarg)

Parameters

- **func** (callable) – Python function to be called \( \text{func}(x, *\text{arg}, **\text{kwarg}) \)
- **x0** (array_like) – Fixed values of masked parameters
- **mask** (array_like) – Mask to use \( x0 \) values (‘mask[i]=1’) or use new parameters \( x \) (‘mask[i]=0’) in call of function
- **arg** (iterable) – Arguments passed to \( \text{func} \)
- **kwarg** (dictionary) – Keyword arguments passed to \( \text{func} \)

Returns  Objective value calculated by \( \text{func} \)

Return type  float
4.8 pyeee.std_io

**sub_ja_params_files** *(files, pid, params)*

Substitute #JA????# with parameter value in several files, i.e.

* #JA0000# with params[0]
* #JA0001# with params[1]
* ...

**Parameters**

- `files` *(list)* – List with file names in which #JA????# will be replaced.
- `pid` *(int)* – Output files will be input files suffixed by .pid
- `params` *(iterable)* – Parameter values to replace #JA????# patterns.

  * params[0] will replace #JA0000#
  * params[1] will replace #JA0001#
  * ...

**Returns**

No return value but output files with names of the input files suffixed by .pid, in which all #JA????# patterns were replaced by params elements.

**Return type** None

**Examples**

```python
>>> sub_ja_params_files([file1, file2], 1234, [0, 1, 2, 3])
```

**sub_names_params_files** *(args, **kwargs)*

Wrapper for **sub_names_params_files_ignorecase**.

**sub_names_params_files_case** *(files, pid, params, names)*

Substitute name = .* with `name = parameter` value in several files, i.e.

```
    names[i] = params[i]
```

*Note, names are case sensitive.*

**Parameters**

- `files` *(list)* – List with file names in which #JA????# will be replaced.
- `pid` *(int)* – Output files will be input files suffixed by .pid
- `params` *(iterable)* – Parameter values to be given to variables on the right of = sign

  * Parameter values to replace #JA????# patterns.

  * Variable in names[0] will be assigned value in params[0]
  * Variable in names[1] will be assigned value in params[1]
  * ...

  * `names` *(iterable)* – Variable names on left of = sign in files

**Returns**

No return value but output files with names of the input files suffixed by .pid, in which all variables given in names are assigned the values in given in params.

**Return type** None
Examples

```python
>>> sub_names_params_files_case([file1, file2], 1234, [0, 1, 2, 3], ['param1',
                                             'param2', 'param3', 'param4'])
```

```python
standard_objective_reader(filename)
```

Standard objective reader.

The standard objective reader (if objectivereader=None) reads a single value from a file without header, comment line or similar.

That means for example:

```
0.0123456789e-02
```

Parameters `filename` (string) – Filename of with objective value

Returns Single number read from filename

Return type float

Examples

```python
>>> subprocess.call(model)
```

```python
>>> obj = standard_objective_reader(filename)
```

4.8. pyeee.std_io
Lines starting with # will be excluded.
That means a standard parameter file might look like:

```plaintext
#par
3.000000000000000e-01
2.300000000000000e-01
1.440000000000000e+01
3.000000000000000e-01
...
```

Parameters

- `filename (string)` – Filename with parameter values
- `Returns` Parameter values
- `Return type` ndarray

Examples

```python
>>> params = standard_parameter_reader(paramfile)
```

`standard_parameter_writer (filename, pid, params=None)`

Standard parameter writer.

The standard parameter writer writes a file containing 1 line per parameter with the parameter value.
All values will be written in IEEE double precision: {:.14e}.
That means:

```
3.000000000000000e-01
2.300000000000000e-01
1.440000000000000e+01
3.000000000000000e-01
...
```

Parameters

- `filename (string)` – Output filename with parameter values
- `pid (int)` – Output file will be filename.pid
- `params (iterable)` – Parameter values

If standard_parameter_writer is called with two arguments, then the second argument will be params.

- `Returns` No return value but output file written: filename.pid
- `Return type` None

Examples

```python
>>> randst = np.random.RandomState()
>>> pid = str(randst.randint(2147483647))
>>> params = sample_parameter(pis, pmin, pmax, pmask)
>>> standard_parameter_writer(paramfile, pid, params)
```
standard_parameter_reader_bounds_mask (filename)

Read standard parameter file with parameter bounds and mask.

The standard parameter file is a space separated file containing 1 line per parameter with the following columns:

identifier, current parameter value, minimum parameter value, maximum parameter value, parameter mask
(1: include, 0: exclude).

Lines starting with # will be excluded.

That means a standard parameter file might look like:

```
# value min max mask
1 3.000000000000000e-01 0.000000000000000e+00 1.000000000000000e+00 1
2 2.300000000000000e-01 -1.000000000000000e+00 1.000000000000000e+00 1
3 1.440000000000000e+01 9.000000000000000e+00 2.000000000000000e+01 1
4 3.000000000000000e-01 0.000000000000000e+00 1.000000000000000e+00 0
```

Parameters

- **filename** *(string)* – Filename with parameter values

Returns

- List with ndarrays of
  - ids - identifier
  - params - parameter values
  - pmin - minimum parameter value
  - pmax - maximum parameter value
  - mask - parameter mask (1: include, 0: exclude from optimisation)

Examples

```
>>> ids, params, pmin, pmax, pmask = standard_parameter_reader_bounds_mask(filename)
```

standard_parameter_writer_bounds_mask (filename, pid, params, pmin, pmax, mask)

Standard parameter writer with parameter bounds and mask.

The standard parameter writer writes a space separated file containing 1 header line (# value min max mask) plus 1 line per parameter with the following columns:

consecutive parameter number, current parameter value, minimum parameter value, maximum parameter value, parameter mask (1: include, 0: exclude).

All values will be written in IEEE double precision: {:.14e}.

That means:

```
# value min max mask
1 3.000000000000000e-01 0.000000000000000e+00 1.000000000000000e+00 1
2 2.300000000000000e-01 -1.000000000000000e+00 1.000000000000000e+00 1
3 1.440000000000000e+01 9.000000000000000e+00 2.000000000000000e+01 1
4 3.000000000000000e-01 0.000000000000000e+00 1.000000000000000e+00 0
```
Parameters

- **filename** *(string)* – Output filename with parameter values
- **pid** *(int)* – Output file will be filename.pid if pid is not None
- **params** *(iterable)* – Parameter values
- **pmin** *(iterable)* – Minimum parameter values
- **pmax** *(iterable)* – Maximum parameter values
- **mask** *(iterable)* – Parameter mask (1: include, 0: exclude from optimisation)

Returns
No return value but output file written: filename.pid

Return type
None

Examples

```python
>>> randst = np.random.RandomState()
>>> pid = str(randst.randint(2147483647))
>>> params = sample_parameter(pis, pmin, pmax, pmask)
>>> standard_parameter_writer_bounds_mask(paramfile, pid, params, pmin, pmax, pmask)
```

**standard_time_series_reader** *(filename)*

Standard reader for time series.

The standard time series reader reads a time series of arbitrary length from a file without header, comment line or similar.

That means for example:

```
0.0123456789e-02
0.1234567890e-02
0.2345678900e-02
0.3456789000e-02
0.4567890000e-02
...
```

Parameters

- **filename** *(string)* – Filename of with time series values

Returns

- **timeseries** – ndarray with values of each line in filename

Return type
ndarray

Examples

```python
>>> subprocess.call(model)
>>> ts = standard_time_series_reader(filename)
```

**standard_timeseries_reader** *(filename)*

Wrapper for **standard_time_series_reader**
4.9 pyeee.tee

TEE (*args, **kwargs)
Prints arguments on screen and in file, like Unix/Linux tee utility.

Calls print function twice, once with the keyword file and once without, i.e. prints on sys.stdout.

Parameters

- *args (iterable) – All arguments of the print function; will be passed to the print function.
- **kwargs (dict) – All keyword arguments of the print function; will be passed to the print function.
- file (object) – The file argument must be an object with a write(string) method. If it is not present or None, *args will be printed on sys.stdout only. Since printed arguments are converted to text strings, print() cannot be used with binary mode file objects.arguments.

Returns

If file is given and not None, then print will be called with *args and **kwargs and a second time with the file keyword argument removed, so that *args will be written to sys.stdout. If file is not given or None, *args will only be written to sys.stdout, i.e. tee is a simple wrapper of the print function.

Return type None

Examples

```python
>>> st = 'Output on screen and in log file'
>>> ff = 'tee_log.txt'

>>> f = open(ff, 'w')
>>> tee(st, file=f)
Output on screen and in log file
>>> f.close()

>>> f = open(ff, 'r')
>>> test = f.readline()
>>> f.close()
>>> test = test[:-1] # rm trailing newline character
>>> if test == st:
...     print('Yes')
... else:
...     print('No')
Yes

>>> import os
>>> os.remove(ff)
>>> f=None

>>> st = 'Output only on screen'
>>> tee(st, file=f)
Output only on screen
```
p
pyeee, 19
pyeee.eee, 20
pyeee.functions.general_functions, 30
pyeee.functions.sa_test_functions, 34
pyeee.functions.utils, 37
pyeee.morris_method, 26
pyeee.screening, 23
pyeee.utils.function_wrapper, 38
pyeee.utils.std_io, 42
pyeee.utils.tee, 47
INDEX

B
B() (in module pyeee.functions.sa_test_functions), 34
bratley() (in module pyeee.functions.sa_test_functions), 34

C
cost_square() (in module pyeee.functions.utils), 37
curvature() (in module pyeee.functions.general_functions), 30

d
D
D2logistic() (in module pyeee.functions.general_functions), 31
d2logistic2_offset() (in module pyeee.functions.general_functions), 33
d2logistic2_offset_p() (in module pyeee.functions.general_functions), 33
d2logistic_offset() (in module pyeee.functions.general_functions), 33
d2logistic_offset_p() (in module pyeee.functions.general_functions), 32
d2logistic_p() (in module pyeee.functions.general_functions), 31
dlogistic() (in module pyeee.functions.general_functions), 30
dlogistic2_offset() (in module pyeee.functions.general_functions), 32
dlogistic2_offset_p() (in module pyeee.functions.general_functions), 33
dlogistic_offset() (in module pyeee.functions.general_functions), 33
dlogistic_offset_p() (in module pyeee.functions.general_functions), 32
dlogistic_p() (in module pyeee.functions.general_functions), 31

E
ee() (in module pyeee.screening), 25
eee() (in module pyeee.eee), 20
elementary_effects() (in module pyeee.morris_method), 27
exe_mask_wrapper() (in module pyeee.utils.function_wrapper), 39
exe_wrapper() (in module pyeee.utils.function_wrapper), 38

F
fmorris() (in module pyeee.functions.sa_test_functions), 35
func_mask_wrapper() (in module pyeee.utils.function_wrapper), 40
func_wrapper() (in module pyeee.utils.function_wrapper), 40

G
G() (in module pyeee.functions.sa_test_functions), 34
g() (in module pyeee.functions.sa_test_functions), 34
Gstar() (in module pyeee.functions.sa_test_functions), 34

I
ishigami_homma() (in module pyeee.functions.sa_test_functions), 35
ishigami_homma_easy() (in module pyeee.functions.sa_test_functions), 36

K
K() (in module pyeee.functions.sa_test_functions), 34

L
linear() (in module pyeee.functions.sa_test_functions), 36
logistic() (in module pyeee.functions.sa_test_functions), 30
logistic2_offset() (in module pyeee.functions.general_functions), 32
logistic2_offset_p() (in module pyeee.functions.general_functions), 32
logistic_offset() (in module pyeee.functions.general_functions), 31
logistic_offset_p() (in module pyeee.functions.general_functions), 31
logistic_p() (in module pyeee.functions.general_functions), 31
M
morris() (in module pyeee.functions.sa_test_functions), 35
morris_sampling() (in module pyeee.morris_method), 26

O
oakley_ohagan() (in module pyeee.functions.sa_test_functions), 35

P
product() (in module pyeee.functions.sa_test_functions), 36
pyeee (module), 19
pyeee.eee (module), 20
pyeee.functions.general_functions (module), 30
pyeee.functions.sa_test_functions (module), 34
pyeee.functions.utils (module), 37
pyeee.morris_method (module), 26
pyeee.screening (module), 23
pyeee.utils.function_wrapper (module), 38
pyeee.utils.std_io (module), 42
pyeee.utils.tee (module), 47

R
ratio() (in module pyeee.functions.sa_test_functions), 36

S
screening() (in module pyeee.screening), 23
see() (in module pyeee.eee), 21
standard_objective_reader() (in module pyeee.utils.std_io), 43
standard_parameter_reader() (in module pyeee.utils.std_io), 43
standard_parameter_reader_bounds_mask() (in module pyeee.utils.std_io), 44
standard_parameter_writer() (in module pyeee.utils.std_io), 44
standard_parameter_writer_bounds_mask() (in module pyeee.utils.std_io), 45
standard_time_series_reader() (in module pyeee.utils.std_io), 46
standard_timeseries_reader() (in module pyeee.utils.std_io), 46
sub_ja_params_files() (in module pyeee.utils.std_io), 42
sub_names_params_files() (in module pyeee.utils.std_io), 42
sub_names_params_files_case() (in module pyeee.utils.std_io), 42
sub_names_params_files_ignorecase() (in module pyeee.utils.std_io), 43

T
tee() (in module pyeee.utils.tee), 47