pyeee Documentation

Release 3.0

Matthias Cuntz

Oct 18, 2021

CONTENTS

CHAPTER 1

QUICKSTART

pyeee: A Python library for parameter screening of computational models using Morris' method of Elementary Effects or its extension of Efficient or Sequential Elementary Effects by Cuntz, Mai et al. (Water Res Research, 2015).

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:

Cuntz M, Mai J *et al.* (2015) Computationally inexpensive identification of noninformative model parameters by sequential screening *Water Resources Research* 51, 6417-6441, doi:10.1002/2015WR016907.

pyeee can be used with Python functions and external programs, using for example the package partialwrap. 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: $y = \sin(x_0) + a\ \sin(x_1)^2 + b\, x_2^4 \sin(x_0)$.

Taking $a = b = 1$ gives:

```
import numpy as np
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 $+\pi$.

Morris' Elementary Effects can then be calculated like:

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

(continues on next page)

(continued from previous page)

```
ub = np.ones(npars) * np.pi

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

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

```
# 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:

```
# 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:

```
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:

```
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)
```

Figuratively speaking, *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

We recommend to use the package `partialwrap`, which provides wrappers to use with partial.

```
from partialwrap import function_wrapper
args = [a, b]
kwargs = {}
func = partial(function_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 programs.

1.3 Installation

The easiest way to install is via *pip*:

```
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.

The project structure is based on a template provided by Sebastian Müller .

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

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:

Cuntz, Mai *et al.* (2015) Computationally inexpensive identification of noninformative model parameters by sequential screening, *Water Resources Research* 51, 6417-6441, doi:10.1002/2015WR016907.

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)*. One can use the package `partialwrap` to use external programs with `functools.partial()` and hence `pyeee`.

The package uses several functions of the JAMS Python package

> https://github.com/mcuntz/jams_python

The JAMS package and hesseflux are synchronised irregularly.

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:

```
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:

```
from pyjams import ee

# function
```

(continues on next page)

(continued from previous page)

```
func  = ishigami1
npars = 3

# lower boundaries
lb = np.ones(npars) * (-np.pi)
# upper boundaries
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 (μ)
3. (npars,2) the standard deviations of the elementary effects over all trajectories (σ)

For Elementary Effects and its sensititvity measures, see https://en.wikipedia.org/wiki/Elementary_effects_method, or

Saltelli *et al.* (2007) Global Sensitivity Analysis. The Primer, John Wiley & Sons Ltd, Chichester, UK, ISBN: 978-0470-059-975, doi:10.1002/9780470725184.

```
# mu*
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. Inital values *x0* must be given that will be taken where *mask==False*, i.e. *mask* could be called an include-mask. Note that the size of *x0* 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 wants to exclude the second parameter x_1 of the above Ishigami-Homma function in the calculation of the Elementary Effects:

```
# 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 *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 *processes=nprocs* evaluates *nprocs* parameter sets in parallel:

```
# 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)
```

`pyeee` uses the package `schwimmbad` for parallelisation. `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. *eeetest.py*):

```
# 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 pyjams 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)

(continued from previous 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)*:

```
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*.

We recommend the package `partialwrap` that provides wrapper functions to work with `functools.partial()`. *call_func_ab* can be replaced by the wrapper function of `partialwrap`: `function_wrapper()`:

```
from partialwrap import function_wrapper
arg   = [a, b]
kwarg = {}
func  = partial(function_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 `dict`. The function wrapper finally passes *x*, *arg* and *kwarg* to *func(x, *arg, **kwarg)*.

`partialwrap` 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, *x0* and *mask* must be given to `function_mask_wrapper()`. Then Elementary Effects will be calculated only for the remaining parameters, between *lb[mask]* and *ub[mask]*. All other non-masked parameters will be taken as *x0*. Remember that *mask* is an include-mask, i.e. all *mask==True* will be screened and all *mask==False* will not be screened.

```
from partialwrap import function_mask_wrapper
func = partial(function_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 ϵ_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 ϵ_p for the parameter p for determining Morris' Elementary Effects.

`pyeee` allows all distributions of `scipy.stats`, given with the keyword *dist*. The parameter of the distributions are given as tuples with the keyword *distparam*. The lower and upper bounds change their meaning if *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 plus or minus three standard deviations, which includes about 99.7% of the total distribution. This means that the lower bound would be 0.0015 (0.003/2.) and the upper bound 0.9985.

```
import scipy.stats as stats
dist      = [None, stats.norm, 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 `scipy.stats` distribution objects;
3. if *dist* is None, `pyeee` assumes a uniform distribution and samples between lower and upper bound;
4. (almost) all `scipy.stats` distributions take the keywords *loc* and *scale*. Their meaning is *NOT* mean and standard deviation in most distributions. For the uniform distribution `scipy.stats.uniform`, *loc* is the lower limit and *loc+scale* the upper limit. This means the combination *dist=None*, *lb=a*, *ub=b* corresponds to *dist=scipy.stats.uniform*, *distparam=[a,b-a]*, *lb=0*, *ub=1*.

Note also that

5. if *distparam* is None, *loc=0* and *scale=1* will be taken;
6. *loc* and *scale* are implemented as keywords in `scipy.stats`. Other parameters such as for example the shape parameter of the gamma distribution `scipy.stats.gamma` must hence be given first, i.e. *(shape,loc,scale)*.

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 sensititvity 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 has to be included anymore in the further screening. 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, to minimize the possibility that the parameters are sensitive in a small part of the parameter space, which was missed due to a little sample.

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

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

from partialwrap import function_wrapper
arg   = [a, b]
kwarg = {}
unc   = partial(function_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]
```

`eee()` returns an include-mask, being *True* for sensitive parameters and *False* for noninformative parameters. The mask can be combined by *logical_and* with an incoming mask.

Note if you use `function_mask_wrapper()`, *out* has the dimension of the *mask==True* elements:

```
from partialwrap import function_mask_wrapper
func = partial(function_mask_wrapper, ishigami, x0, mask, arg, kwarg)
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 σ. 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 σ 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 :func: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.

```
# 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

Note that this section is pretty much a repetition of the User Guide **of** `partialwrap`**, which is not limited to be used with** `pyeee` **but can be used with any package that calls functions in the form** *func(x)*. **The notions of** `partialwrap` **might be better explained in its** user guide.

`partialwrap` provides wrapper functions to work with external executables. `partialwrap` 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 `partialwrap` in return. The processing steps are:

```
parameterwriter(parameterfile, parameters)
err = subprocess.check_output(exe)
obj = outputreader(outputfile)
os.remove(parameterfile)
os.remove(outputfile)
```

That means `partialwrap` needs to have a function *parameterwriter* that writes the parameter file *parameterfile* needed by the executable *exe*. It then needs to have a function *outputreader* for reading the output file *outputfile* 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 *outputfile* = *obj.txt*:

```
# File: ishiexe.py

# 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 partialwrap import standard_parameter_reader
pfile = 'params.txt'
x = standard_parameter_reader(pfile)

# calc function
y = ishigami1(x)

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

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()`:

```
from functools import partial
from partialwrap import exe_wrapper, standard_parameter_writer, standard_output_
↪reader

ishi = ['python', 'ishiexe.py']
parameterfile = 'params.txt'
outputfile = 'obj.txt'
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_writer,
               outputfile, standard_output_reader, {})

npars = 3
lb = np.ones(npars) * (-np.pi)
ub = np.ones(npars) * np.pi

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

`standard_parameter_reader()` and *~partialwrap.standard_parameter_writer* are convenience functions that read and write one parameter per line in a file without a header. The function `standard_output_reader()` simply reads one value from a file without header. The empty dictionary 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 `function_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, *x0* 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:

```
mask    = np.ones(npars, dtype=bool) # True  -> include
mask[1] = False                      # False -> exclude
x0      = np.ones(npars) * 0.5
func = partial(exe_mask_wrapper, ishi, x0, mask,
               parameterfile, standard_parameter_writer,
               outputfile, standard_output_reader, {})
out = ee(func, lb[mask], ub[mask], 10)
```

x_1 will then always be the second element of *x0*.

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:

```
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 occured during execution:

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

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

```
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*:

```
from partialwrap import standard_parameter_reader_bounds_mask

ishi = ['python', 'ishiexe.py']
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_reader_bounds_mask,
               outputfile, standard_output_reader,
```

(continues on next page)

(continued from previous page)

```
                {'pargs':[lb,ub,mask]})
out  = ee(func, lb, ub, 10)
```

Or in case of exclusion of x_1:

```
from partialwrap import standard_parameter_reader_bounds_mask
func = partial(exe_mask_wrapper, ishi, x0, mask,
               parameterfile, standard_parameter_reader_bounds_mask,
               outputfile, standard_output_reader,
               {'pargs':[lb,ub,mask]})
out  = ee(func, lb[mask], ub[mask], 10)
```

Another common case is that the parameters are given in the form *parameter* = *value*, e.g. in Fortran namelists. `partialwrap` 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:

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

The function `sub_params_names()` (which is identical to `sub_params_names_ignorecase()`) can be used and parameter names are passed via *pargs*:

```
from partialwrap import sub_params_names

pnames = ['x0', 'x1', 'x2']
func = partial(exe_wrapper, ishi,
               parameterfile, sub_params_names,
               outputfile, standard_output_reader,
               {'pargs':[pnames], 'pid':True})
out = ee(func, lb, ub, 10)
```

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

```
Wrapper function for external programs using a `parameterwriter` and `outputreader`
with the interfaces:
    `parameterwriter(parameterfile, x, *pargs, **pkwargs)`
    `outputreader(outputfile, *oargs, **okwargs)`
or if `pid==True`:
    `parameterwriter(parameterfile, x, *pargs, pid=pid, **pkwargs)`
    `outputreader(outputfile, *oargs, pid=pid, **okwargs)`
```

And the definition of `sub_params_names()` is:

```
def sub_params_names_ignorecase(files, params, names, pid=None):
```

This means that **pargs* passes **[pnames]*, which is *pnames*, as an argument after the parameters *x* to `sub_params_names()`.

Excluding x_1 would then be achieved by simply excluding *x1* from *pnames*:

```
from partialwrap import sub_params_names

pnames = ['x0', 'x2']
func = partial(exe_wrapper, ishi,
```

(continues on next page)

(continued from previous page)

```
                parameterfile, sub_params_names,
                outputfile, standard_output_reader,
                {'pargs':[pnames], 'pid':True})
out   = ee(func, lb[mask], ub[mask], 10)
```

Parallel processing of external executables

Elementary Effects run the computational model *nt*(npars+1)* times. All model runs are independent and can be executated at the same time if computing ressources permit. Even simple personal computers have several 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 of *outputreader*, like:

```
pid = str(randst.randint())
parameterwriter(parameterfile, x, *pargs, pid=pid, **pkwargs)
err = subprocess.check_output([func, pid])
obj = outputreader(outputfile, *oargs, pid=pid, **okwargs)
os.remove(parameterfile+'.'+pid)
os.remove(outputfile+'.'+pid)
```

The *parameterwriter* is assumed to write *parameterfile.pid* and the external model is assumed to write *outputfile.pid*. Only these filenames are cleaned up by `exe_wrapper()`. If different filenames are used, the user has to clean up herself.

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

```
# File: ishiexe1.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 partialwrap import standard_parameter_reader
pfile = 'params.txt'
x = standard_parameter_reader(pfile, pid=pid)

# calc function
y = ishigami1(x)

# write objective
ofile = 'obj.txt'
if pid:
    ofile = ofile+'.'+pid
with open(ofile, 'w') as ff:
    print(y, file=ff)
```

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

```
from partialwrap import exe_wrapper, standard_parameter_writer, standard_output_
↪reader

ishi = ['python3', 'ishiexe1.py']
parameterfile = 'params.txt'
outputfile    = 'obj.txt'
func = partial(exe_wrapper, ishi,
               parameterfile, standard_parameter_writer,
               outputfile, standard_output_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)
```

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 pyeee
mv obj.txt ../obj.txt.${pid}

# clean up
cd ..
rm -r tmp.${pid}
```

which would then be used:

```
from functools import partial
from partialwrap import exe_wrapper, standard_parameter_writer, standard_output_
↪reader

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

The User Guide of `partialwrap` gives a similar script written in Python, which could be used if the bash shell is not available, for example on Windows.

That's all Folks!

CHAPTER 3

INSTALLATION

The easiest way to install `pyeee` 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 proper privileges can append the *–user* flag to `pip` either while installing from the 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`, `schwimmbad`, `pyjams`. They are all available in PyPI and `pip` should install them automatically. Installations via *setup.py* might need to install the three dependencies first.

CHAPTER 4

PYEEE API

4.1 Purpose

pyeee provides parameter screening of computational models using the Morris method of elementary effects or the extension of Efficient/Sequential Elementary Effects of Cuntz, Mai et al. (Water Res Research, 2015).

The package uses several functions of the JAMS Python package https://github.com/mcuntz/jams_python The JAMS package and hesseflux are synchronised irregularly.

4.2 Subpackages

eee(func, *args, **kwargs)	Parameter screening using Efficient/Sequential Elementary Effects of Cuntz, Mai et al.
version	Provide version number for pyeee library.

4.3 pyeee.eee

Function eee for Efficient/Sequential Elementary Effects, an extension of Morris' method of Elementary Effects by Cuntz, Mai et al. (Water Res Research, 2015).

This function was written by Matthias Cuntz while at Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement (INRAE), Nancy, France.

The following functions are provided

see(func, *args, **kwargs)	Wrapper function for *eee()*.
eee(func, *args, **kwargs)	Parameter screening using Efficient/Sequential Elementary Effects of Cuntz, Mai et al.

History

- Written Nov 2017 by Matthias Cuntz (mc (at) macu (dot) de)
- Added *weight* option, Jan 2018, Matthias Cuntz
- Added *plotfile* and made docstring sphinx compatible option, Jan 2018, Matthias Cuntz
- x0 optional; added verbose keyword; distinguish iterable and array_like parameter types, Jan 2020, Matthias Cuntz
- Rename ntsteps to nsteps to be consistent with screening/ee; and check if logfile is string rather thean checking for file handle, Feb 2020, Matthias Cuntz
- Sample not only from uniform distribution but allow all distributions of scipy.stats, Mar 2020, Matthias Cuntz
- Use pyjams package, Oct 2021, Matthias Cuntz
- Make flake8 compatible, Oct 2021, Matthias Cuntz

eee (*func*, **args*, ***kwargs*)

Parameter screening using Efficient/Sequential Elementary Effects of Cuntz, Mai et al. (Water Res Research, 2015).

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).ppf(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, optinal*) – The number of processes to use to evaluate objective function and constraints (default: 1).
- **pool** (*schwimmbad* pool object, optinal) – 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

```
>>> from functools import partial
>>> import numpy as np
>>> import scipy.stats as stats
>>> from pyjams.functions import G
>>> from partialwrap import function_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(function_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()*.

PYTHON MODULE INDEX

p

INDEX