pyPESTO Documentation

Release 0.1.0

The pyPESTO developers

Jun 17, 2020

USER'S GUIDE

Version: 0.1.0
Source code: https://github.com/icb-dcm/pypesto

CHAPTER

ONE

INSTALL AND UPGRADE

1.1 Requirements

This package requires Python 3.6 or later. It is tested on Linux using Travis continuous integration.

1.1.1 I cannot use my system's Python distribution, what now?

Several Python distributions can co-exist on a single system. If you don't have access to a recent Python version via your system's package manager (this might be the case for old operating systems), it is recommended to install the latest version of the Anaconda Python 3 distribution.

Also, there is the possibility to use multiple virtual environments via:

```
python3 -m virtualenv ENV_NAME
source ENV_NAME/bin/activate
```

where ENV_NAME denotes an individual environment name, if you do not want to mess up the system environment.

1.2 Install from PIP

The package can be installed from the Python Package Index PyPI via pip:

```
pip3 install pypesto
```

1.3 Install from GIT

If you want the bleeding edge version, install directly from github:

```
pip3 install git+https://github.com/icb-dcm/pypesto.git
```

If you need to have access to the source code, you can download it via:

```
git clone https://github.com/icb-dcm/pypesto.git
```

and then install from the local repository via:

```
cd pypesto
pip3 install .
```

1.4 Upgrade

If you want to upgrade from an existing previous version, replace `install` by `install --upgrade` in the above commands.

1.5 Install optional packages

- This package includes multiple comfort methods simplyfing its use for parameter estimation for models generated using the toolbox amici. To use AMICI, install it via pip:

```
pip3 install amici
```

- This package inherently supports optimization using the dlib toolbox. To use it, install dlib via:

```
pip3 install dlib
```

CHAPTER

TWO

EXAMPLES

The following examples cover typical use cases and should help get a better idea of how to use this package:

2.1 Rosenbrock banana

Here, we perform optimization for the Rosenbrock banana function, which does not require an AMICI model. In particular, we try several ways of specifying derivative information.

[1]:
```
import pypesto
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline
```

2.1.1 Define the objective and problem

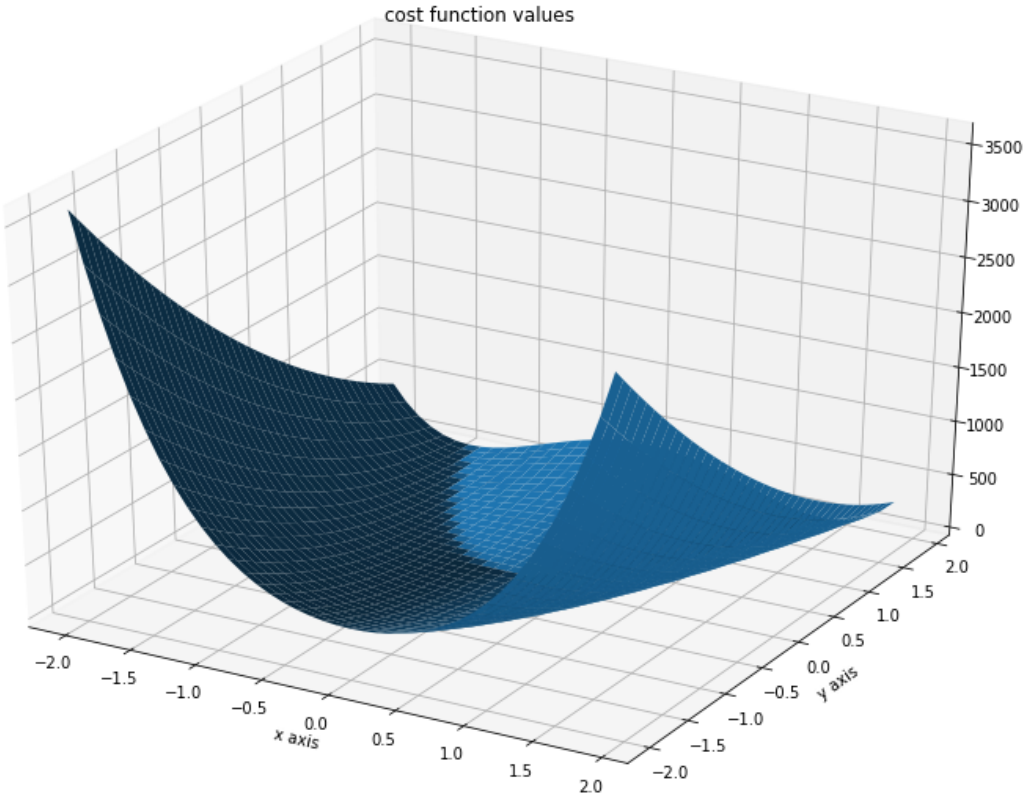
[2]:
```
# first type of objective
objective1 = pypesto.Objective(fun=sp.optimize.rosen,
                               grad=sp.optimize.rosen_der,
                               hess=sp.optimize.rosen_hess)

# second type of objective
def rosen2(x):
    return sp.optimize.rosen(x), sp.optimize.rosen_der(x), sp.optimize.rosen_hess(x)
objective2 = pypesto.Objective(fun=rosen2, grad=True, hess=True)

dim_full = 10
lb = -5 * np.ones((dim_full, 1))
ub = 5 * np.ones((dim_full, 1))

problem1 = pypesto.Problem(objective=objective1, lb=lb, ub=ub)
problem2 = pypesto.Problem(objective=objective2, lb=lb, ub=ub)
```

Illustration

[3]:
```
x = np.arange(-2, 2, 0.1)
y = np.arange(-2, 2, 0.1)
x, y = np.meshgrid(x, y)
z = np.zeros_like(x)
for j in range(0, x.shape[0]):
    for k in range(0, x.shape[1]):
        z[j,k] = objective1([x[j,k], y[j,k]], (0,))
```

[4]:
```
fig = plt.figure()
fig.set_size_inches(*(14,10))
ax = plt.axes(projection='3d')
ax.plot_surface(X=x, Y=y, Z=z)
plt.xlabel('x axis')
plt.ylabel('y axis')
ax.set_title('cost function values')
```

[4]:
```
Text(0.5, 0.92, 'cost function values')
```



2.1.2 Run optimization

[5]:
```
%%time

# create different optimizers
optimizer_bfgs = pypesto.ScipyOptimizer(method='l-bfgs-b')
optimizer_tnc = pypesto.ScipyOptimizer(method='TNC')
optimizer_dogleg = pypesto.ScipyOptimizer(method='dogleg')

# set number of starts
n_starts = 20

# save optimizer trace
history_options = pypesto.HistoryOptions(trace_record=True)

# Run optimizaitons for different optimzers
result1_bfgs = pypesto.minimize(
    problem=problem1, optimizer=optimizer_bfgs,
    n_starts=n_starts, history_options=history_options)
result1_tnc = pypesto.minimize(
    problem=problem1, optimizer=optimizer_tnc,
    n_starts=n_starts, history_options=history_options)
result1_dogleg = pypesto.minimize(
    problem=problem1, optimizer=optimizer_dogleg,
    n_starts=n_starts, history_options=history_options)

# Optimize second type of objective
result2 = pypesto.minimize(problem=problem2, optimizer=optimizer_tnc, n_starts=n_
↪starts)
```

```
Function values from history and optimizer do not match: 0.2816653821914888, 0.
↪5320236837009729
Parameters obtained from history and optimizer do not match: [0.98858861 0.98891374 0.
↪98766938 0.98161879 0.9652898  0.93879652
 0.88306125 0.77229682 0.59072196 0.34463656], [0.98496582 0.99181314 0.98908854 0.
↪97887796 0.9637969  0.92698863
 0.85833682 0.7281339  0.5100925  0.23030355]
Function values from history and optimizer do not match: 2.4964418432638706, 3.
↪2754756228142554
Parameters obtained from history and optimizer do not match: [9.82288496e-01 9.
↪66985425e-01 9.38419566e-01 8.78054852e-01
 7.69913490e-01 5.86275734e-01 3.34232174e-01 1.02020372e-01
 1.12463415e-02 8.92327778e-05], [ 9.77936483e-01  9.58866835e-01  9.18622429e-01  8.
↪45125973e-01
  6.98308810e-01  4.68636934e-01  1.81161241e-01  1.56538059e-02
  1.25320748e-02 -3.42245429e-05]
Function values from history and optimizer do not match: 5.2801527327005155, 6.
↪006584566765655
Parameters obtained from history and optimizer do not match: [ 8.97044955e-01  8.
↪02352694e-01  6.38595532e-01  3.92952520e-01
  1.39352250e-01  1.46386081e-02  1.03838418e-02  1.04224904e-02
  9.87858840e-03 -3.10151313e-05], [8.53850657e-01 7.25968721e-01 5.02444501e-01 2.
↪29720701e-01
 2.91937619e-02 1.04279105e-02 1.00588034e-02 9.91318272e-03
 1.01923977e-02 2.48830744e-04]
Function values from history and optimizer do not match: 1.9692004336022721, 2.
↪7115798760594214
Parameters obtained from history and optimizer do not match: [ 0.99019497  0.97774137␣
↪ 0.9561126   0.91685931  0.84524646  0.70666505
```

(continues on next page)

(continued from previous page)

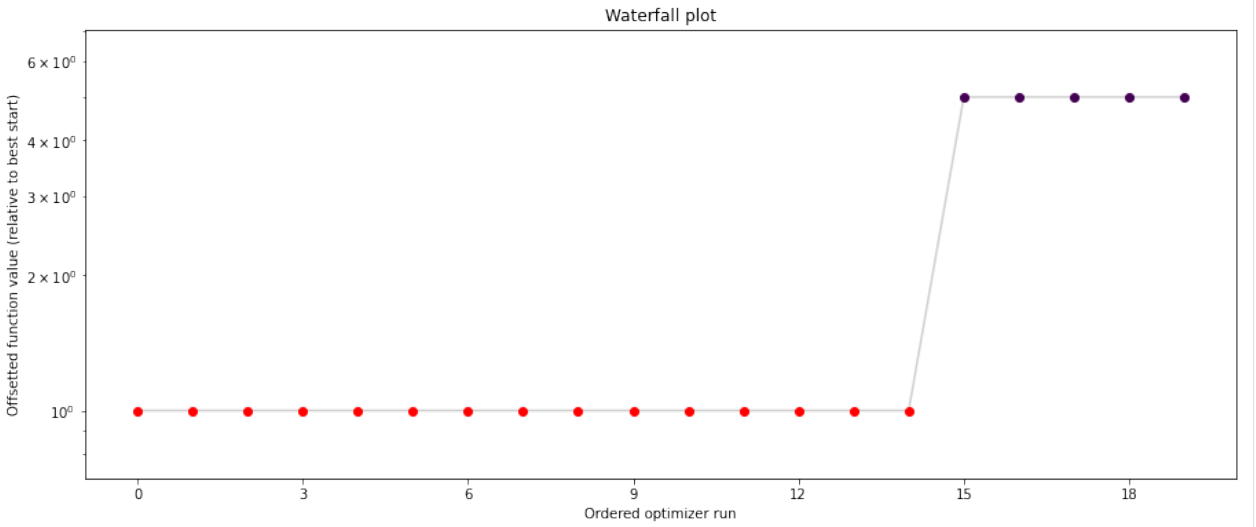
```
  0.49175856  0.23072529  0.03408112 -0.00353803], [ 0.99027014  0.97935022  0.
↪95186067  0.89016816  0.78796339  0.60782056
  0.3439489   0.082957    0.01236261 -0.0090733 ]
Function values from history and optimizer do not match: 8.168889493703478, 8.
↪725825462780923
Parameters obtained from history and optimizer do not match: [-0.94088519  0.89001051␣
↪  0.79111519  0.61462856  0.36442     0.11619655
  0.01427664  0.01024192  0.00981114  0.0030506 ], [-0.9055118   0.82811563  0.
↪68759974  0.46924951  0.18965995  0.0303
  0.00988695  0.00686135  0.01131552  0.01232474]
Function values from history and optimizer do not match: 5.952343722253962, 6.
↪6703631037412485
Parameters obtained from history and optimizer do not match: [ 8.43835312e-01  7.
↪07958350e-01  4.88024405e-01  2.15876140e-01
  3.61965507e-02  1.06449763e-02  1.02201791e-02  1.01343248e-02
  1.00194415e-02 -4.24013443e-05], [ 7.73014395e-01  5.76458083e-01  3.06751296e-01 ␣
↪7.23116662e-02
  1.11101457e-02  1.04925729e-02  1.02198284e-02  9.31800098e-03
  1.00303338e-02 -3.35414249e-04]
Function values from history and optimizer do not match: 2.9695656384271043, 3.
↪1184191036428834
Parameters obtained from history and optimizer do not match: [9.87034254e-01 9.
↪72306804e-01 9.49987341e-01 8.93423068e-01
 8.17286050e-01 6.40113467e-01 3.50303066e-01 9.21040023e-02
 1.61619582e-02 1.07301208e-05], [0.97539345 0.96038658 0.92103141 0.84071849 0.
↪69173518 0.46411457
 0.22535018 0.02137437 0.0140465  0.00187475]
Function values from history and optimizer do not match: 5.243703633002998, 5.
↪815927741761821
Parameters obtained from history and optimizer do not match: [-0.98596483  0.98274868␣
↪  0.97442973  0.9514248   0.90830696  0.82066704
  0.6645958   0.42265816  0.16754891  0.00843015], [-9.83776132e-01  9.76891471e-01 ␣
↪9.59822778e-01  9.25805546e-01
  8.61313435e-01  7.38255806e-01  5.36125583e-01  2.68900058e-01
  4.63857306e-02  5.55478747e-04]
Function values from history and optimizer do not match: 5.568452686104578, 6.
↪240940426777619
Parameters obtained from history and optimizer do not match: [-0.98409971  0.97981689␣
↪  0.96708475  0.93994779  0.882228    0.77067159
  0.58158721  0.3234338   0.08784476  0.00137899], [-9.80621630e-01  9.73220286e-01 ␣
↪9.49144994e-01  8.99631345e-01
  8.13904700e-01  6.72270174e-01  4.38458930e-01  1.59095121e-01
  1.36813582e-02  5.45995408e-04]
Function values from history and optimizer do not match: 0.036324365386529986, 0.
↪09483254249425768
Parameters obtained from history and optimizer do not match: [0.99822004 0.99707243 0.
↪99610909 0.99417929 0.99002028 0.98113164
 0.96153381 0.92444914 0.85179403 0.71878419], [0.99793405 0.9950185  0.99388555 0.
↪99040659 0.98235824 0.966224
 0.93616269 0.87811711 0.77111827 0.58106498]
Function values from history and optimizer do not match: 0.6471617322900747, 1.
↪0298609819176106
Parameters obtained from history and optimizer do not match: [0.99617777 0.9935325  0.
↪98584858 0.97134588 0.94515948 0.88609812
 0.78352677 0.60468994 0.36020607 0.12888943], [0.99486306 0.99041659 0.98198154 0.
↪96653722 0.93323387 0.86752936
 0.75275173 0.55276676 0.2795121  0.04110074]
```

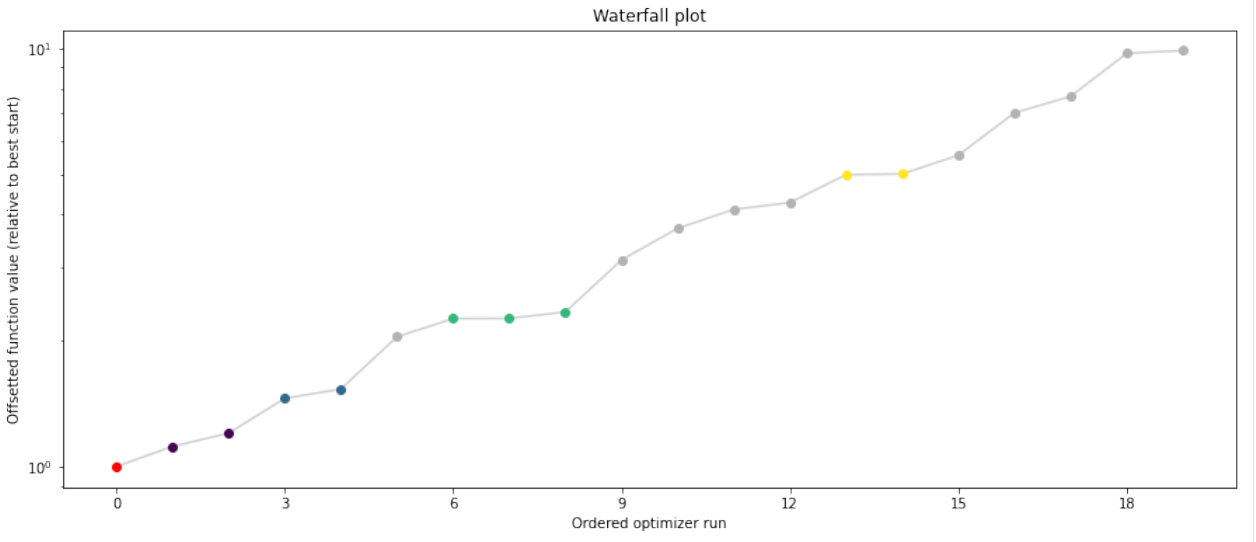
```
CPU times: user 4.73 s, sys: 150 ms, total: 4.88 s
Wall time: 4.85 s
```

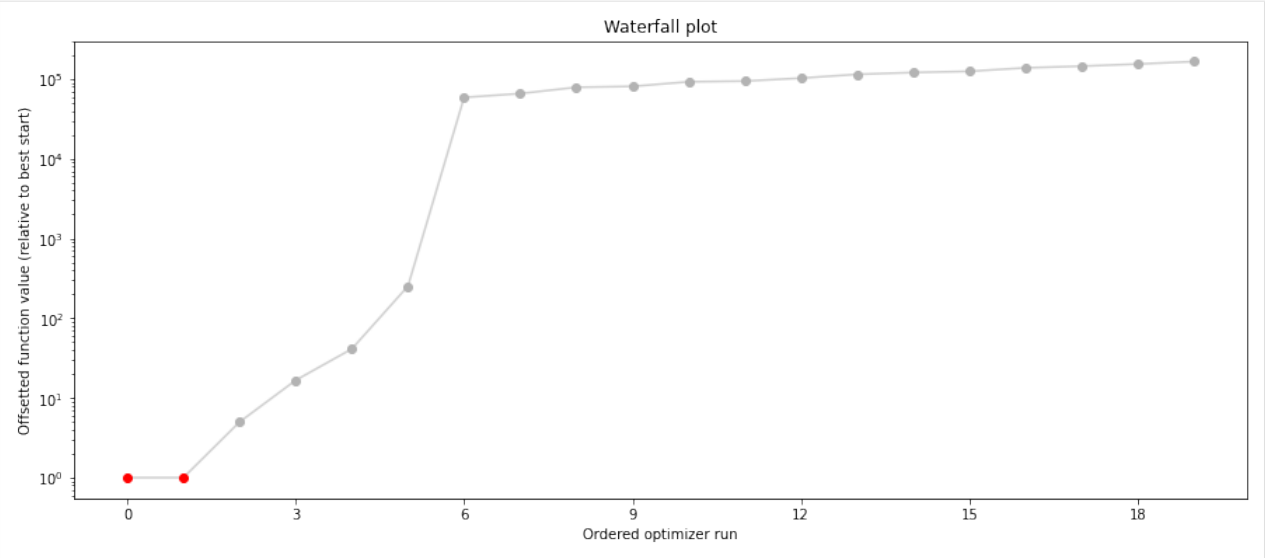
Visualize and compare optimization results

[6]:
```
import pypesto.visualize

# plot separated waterfalls
pypesto.visualize.waterfall(result1_bfgs, size=(15,6))
pypesto.visualize.waterfall(result1_tnc, size=(15,6))
pypesto.visualize.waterfall(result1_dogleg, size=(15,6))
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x126cfd6a0>
```



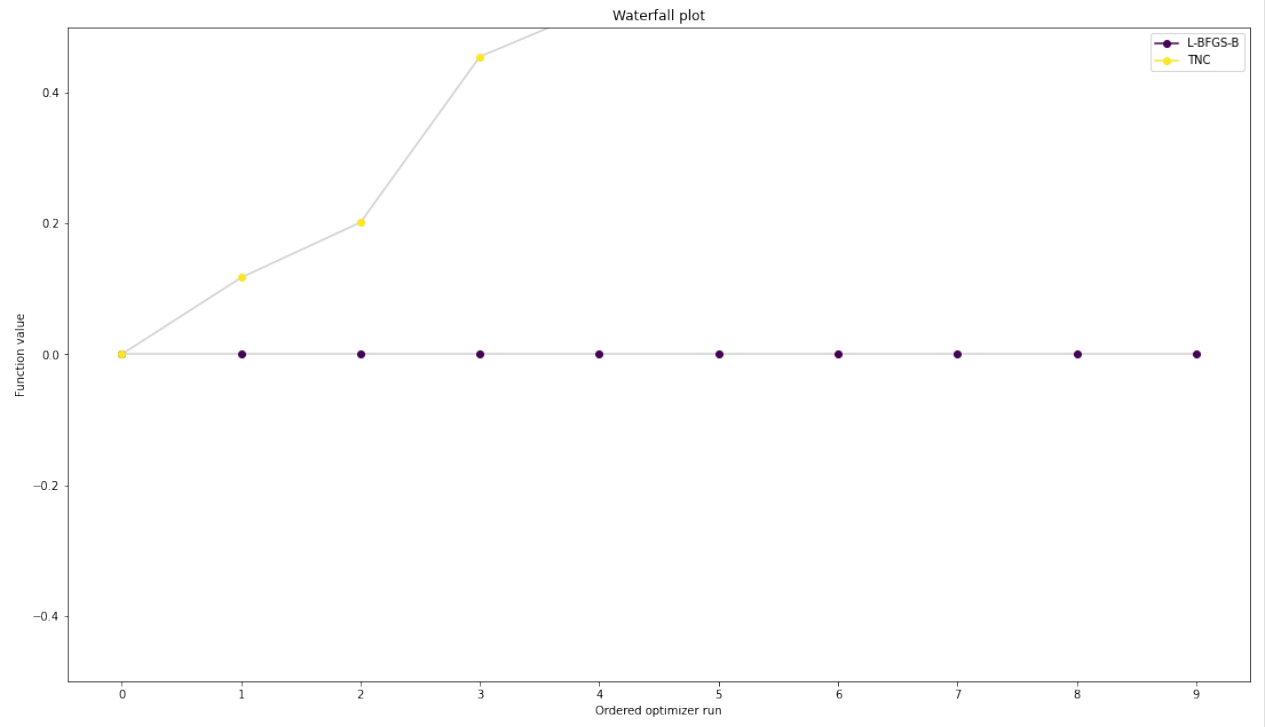




We can now have a closer look, which method perfomred better: Let's first compare bfgs and TNC, since both methods gave good results. How does the fine convergence look like?

[7]:
```
# plot one list of waterfalls
pypesto.visualize.waterfall([result1_bfgs, result1_tnc],
                            legends=['L-BFGS-B', 'TNC'],
                            start_indices=10,
                            scale_y='lin')
```

[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x126ddca00>
```



[8]:
```
# retrieve second optimum
```

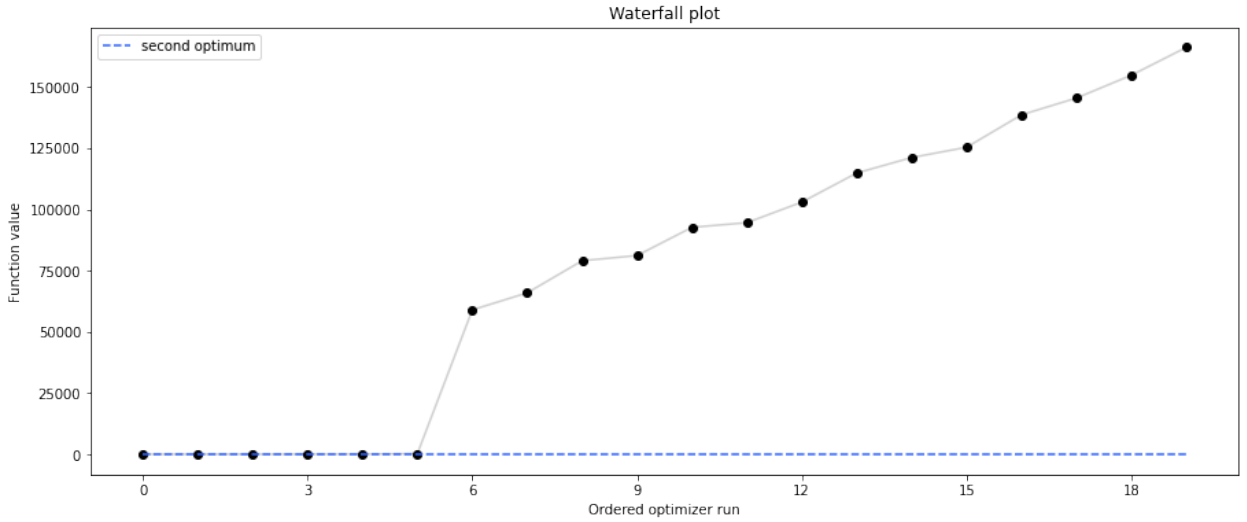
(continues on next page)

(continued from previous page)

```
all_x = result1_bfgs.optimize_result.get_for_key('x')
all_fval = result1_bfgs.optimize_result.get_for_key('fval')
x = all_x[19]
fval = all_fval[19]
print('Second optimum at: ' + str(fval))

# create a reference point from it
ref = {'x': x, 'fval': fval, 'color': [
    0.2, 0.4, 1., 1.], 'legend': 'second optimum'}
ref = pypesto.visualize.create_references(ref)

# new waterfall plot with reference point for second optimum
pypesto.visualize.waterfall(result1_dogleg, size=(15,6),
                            scale_y='lin', y_limits=[-1, 101],
                            reference=ref, colors=[0., 0., 0., 1.])
```

```
Second optimum at: 3.986579113118511
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1275cbfd0>
```

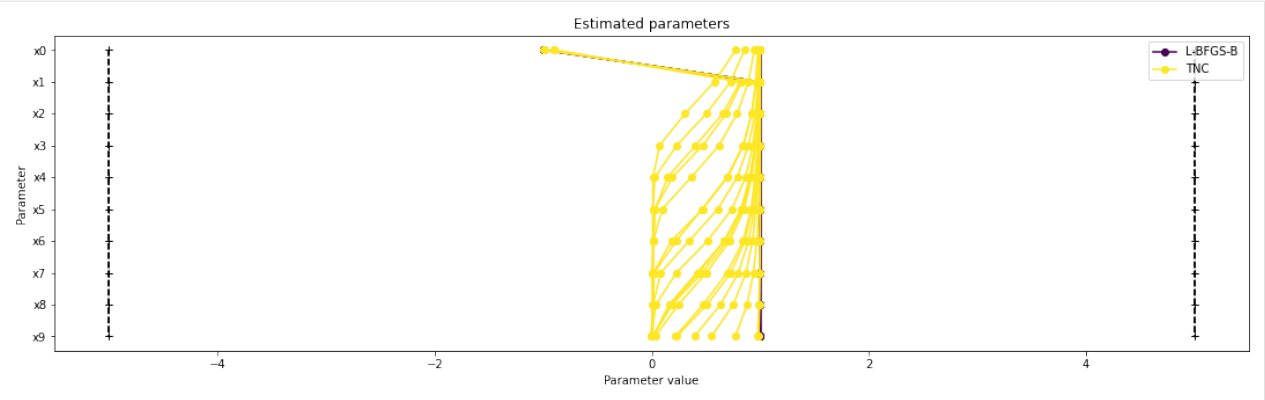


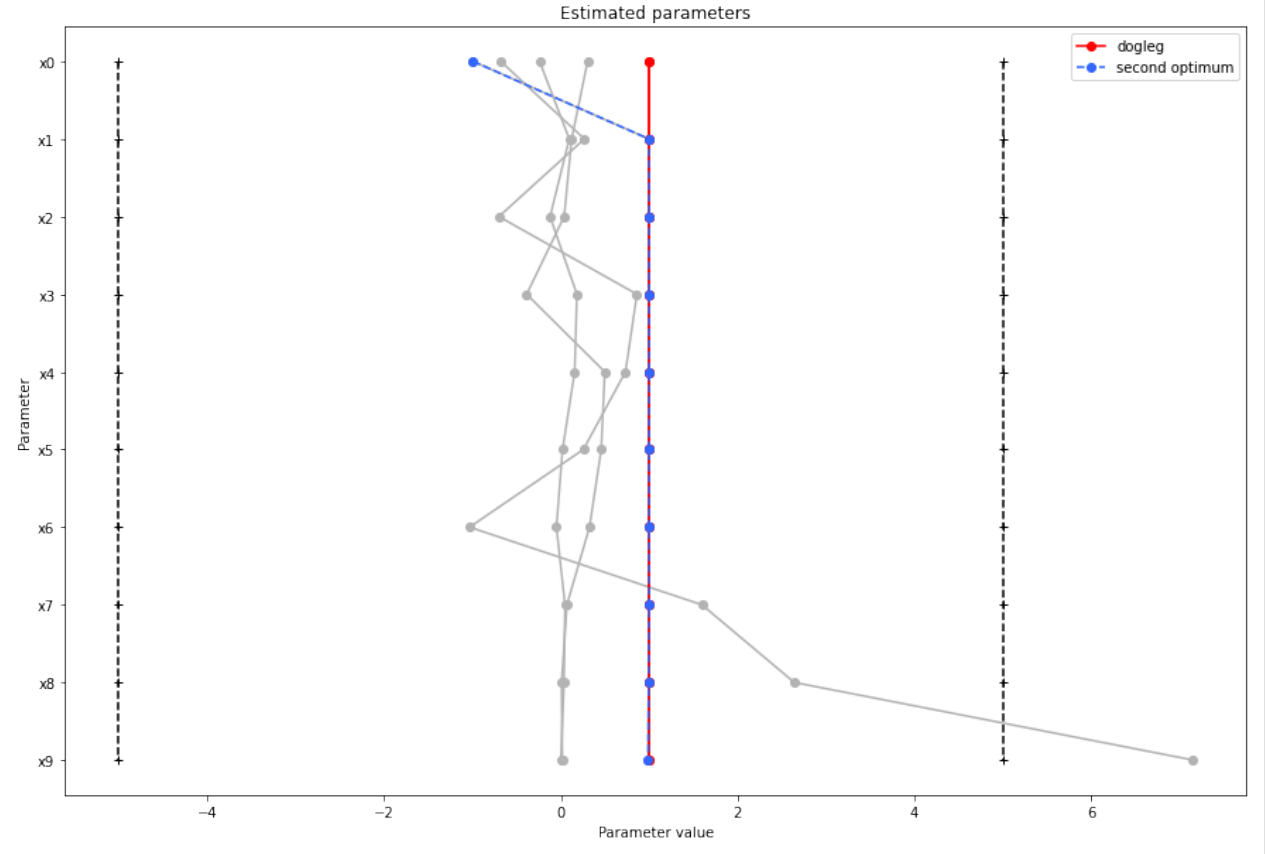
2.1.3 Visualize parameters

There seems to be a second local optimum. We want to see whether it was also found by the dogleg method

[9]:
```
pypesto.visualize.parameters([result1_bfgs, result1_tnc],
                             legends=['L-BFGS-B', 'TNC'],
                             balance_alpha=False)
pypesto.visualize.parameters(result1_dogleg,
                              legends='dogleg',
                              reference=ref,
                              size=(15,10),
                              start_indices=[0, 1, 2, 3, 4, 5],
                              balance_alpha=False)
```

[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1280b3df0>
```





If the result needs to be examined in more detail, it can easily be exported as a pandas.DataFrame:

[10]:
```
df = result1_tnc.optimize_result.as_dataframe(
    ['fval', 'n_fval', 'n_grad', 'n_hess', 'n_res', 'n_sres', 'time'])
df.head()
```

[10]:

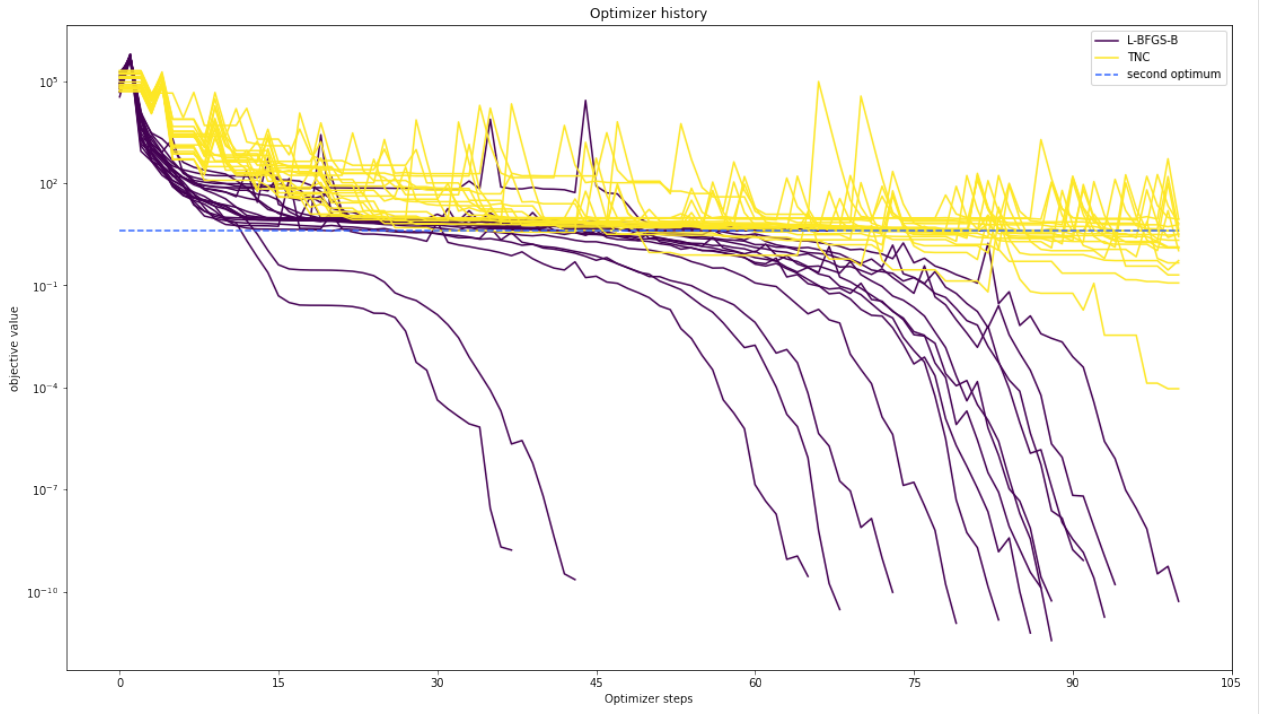
	fval	n_fval	n_grad	n_hess	n_res	n_sres	time
0	0.000092	101	101	0	0	0	0.075175
1	0.117351	101	101	0	0	0	0.055569
2	0.201761	101	101	0	0	0	0.089948
3	0.455242	101	101	0	0	0	0.086691
4	0.532024	101	101	0	0	0	0.061193

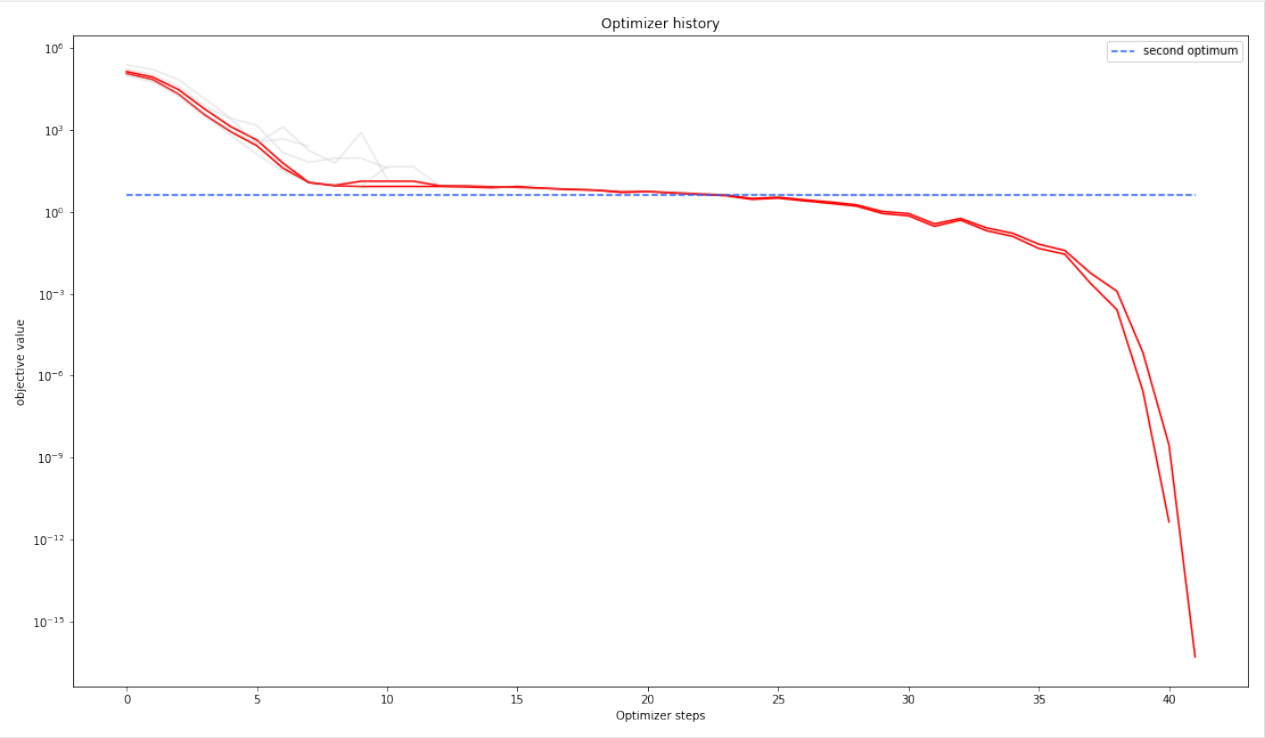
Optimizer history

Let's compare optimzer progress over time.

[11]:
```
# plot one list of waterfalls
pypesto.visualize.optimizer_history([result1_bfgs, result1_tnc],
                                    legends=['L-BFGS-B', 'TNC'],
                                    reference=ref)
# plot one list of waterfalls
pypesto.visualize.optimizer_history(result1_dogleg,
                                    reference=ref)
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x128611c70>
```

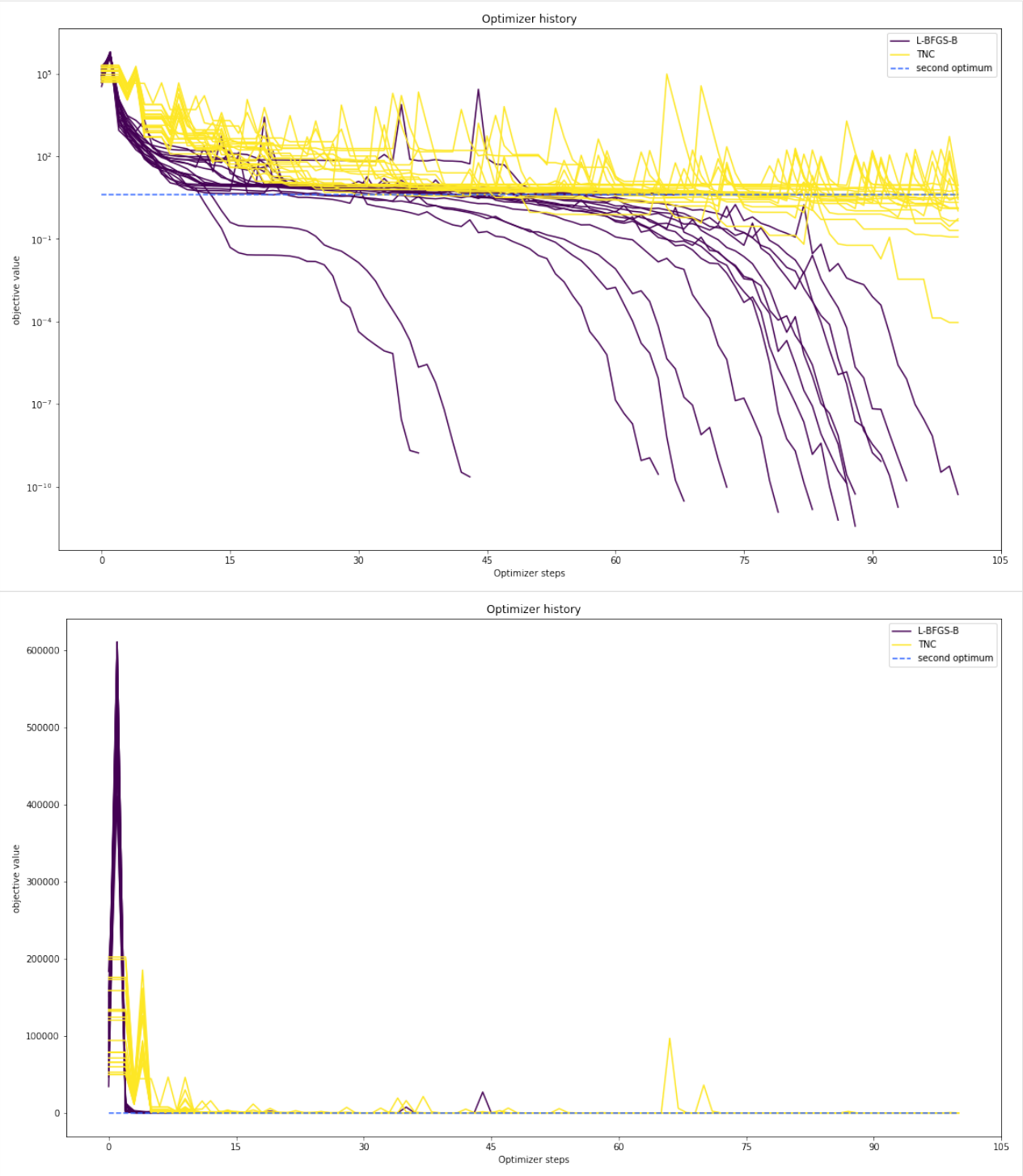




We can also visualize this usign other scalings or offsets...

[12]:
```
# plot one list of waterfalls
pypesto.visualize.optimizer_history([result1_bfgs, result1_tnc],
                                    legends=['L-BFGS-B', 'TNC'],
                                    reference=ref,
                                    offset_y=0.)

# plot one list of waterfalls
pypesto.visualize.optimizer_history([result1_bfgs, result1_tnc],
                                    legends=['L-BFGS-B', 'TNC'],
                                    reference=ref,
                                    scale_y='lin',
                                    y_limits=[-1., 11.])
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x129249d90>
```


Optimizer history
L-BFGS-B
TNC
second optimum
objective value
Optimizer steps
Optimizer history
L-BFGS-B
TNC
second optimum
objective value
Optimizer steps

2.1.4 Compute profiles

The profiling routine needs a problem, a results object and an optimizer.

Moreover it accepts an index of integer (profile_index), whether or not a profile should be computed.

Finally, an integer (result_index) can be passed, in order to specify the local optimum, from which profiling should be started.

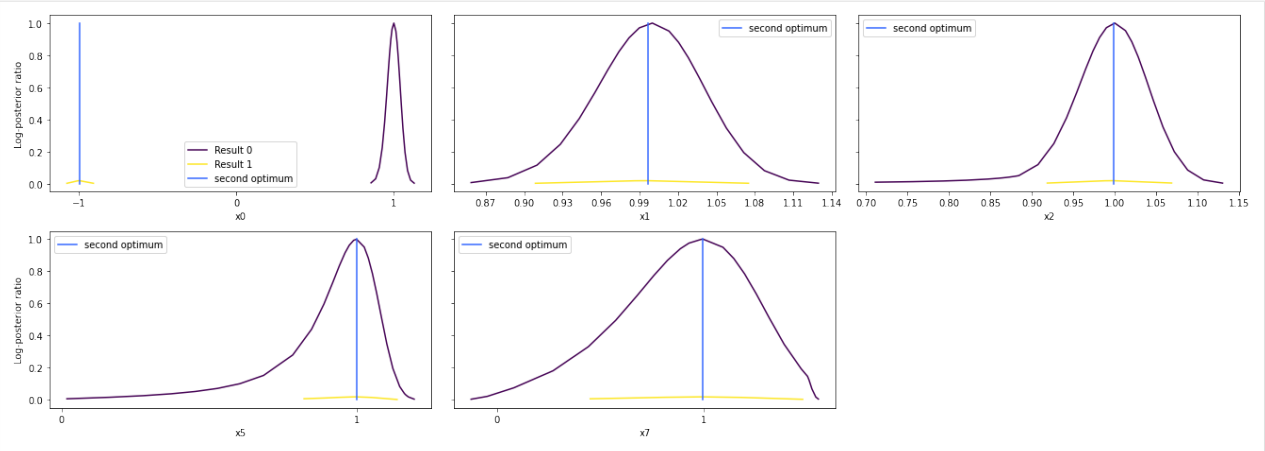
[13]:
```
%%time

# compute profiles
profile_options = pypesto.ProfileOptions(min_step_size=0.0005,
    delta_ratio_max=0.05,
    default_step_size=0.005,
    ratio_min=0.01)

result1_bfgs = pypesto.parameter_profile(
    problem=problem1,
    result=result1_bfgs,
    optimizer=optimizer_bfgs,
    profile_index=np.array([1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0]),
    result_index=0,
    profile_options=profile_options)

# compute profiles from second optimum
result1_bfgs = pypesto.parameter_profile(
    problem=problem1,
    result=result1_bfgs,
    optimizer=optimizer_bfgs,
    profile_index=np.array([1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0]),
    result_index=19,
    profile_options=profile_options)
```

```
CPU times: user 2.81 s, sys: 17.2 ms, total: 2.83 s
Wall time: 2.84 s
```

Visualize and analyze results

pypesto offers easy-to-use visualization routines:

[14]:
```
# specify the parameters, for which profiles should be computed
ax = pypesto.visualize.profiles(result1_bfgs, profile_indices = [0,1,2,5,7],
                                reference=ref, profile_list_ids=[0, 1])
```



Approximate profiles

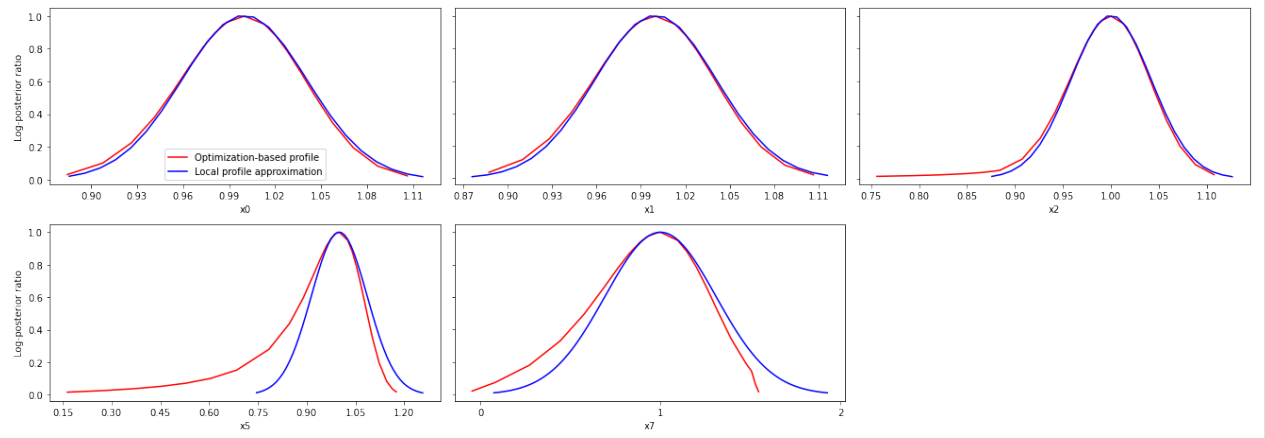
When computing the profiles is computationally too demanding, it is possible to employ to at least consider a normal approximation with covariance matrix given by the Hessian or FIM at the optimal parameters.

[15]:
```
%%time

result1_tnc = pypesto.profile.approximate_parameter_profile(
    problem=problem1,
    result=result1_bfgs,
    profile_index=np.array([1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0]),
    result_index=0,
    n_steps=1000)
```

```
CPU times: user 27.8 ms, sys: 5.92 ms, total: 33.7 ms
Wall time: 20.9 ms
```

These approximate profiles require at most one additional function evaluation, can however yield substantial approximation errors:

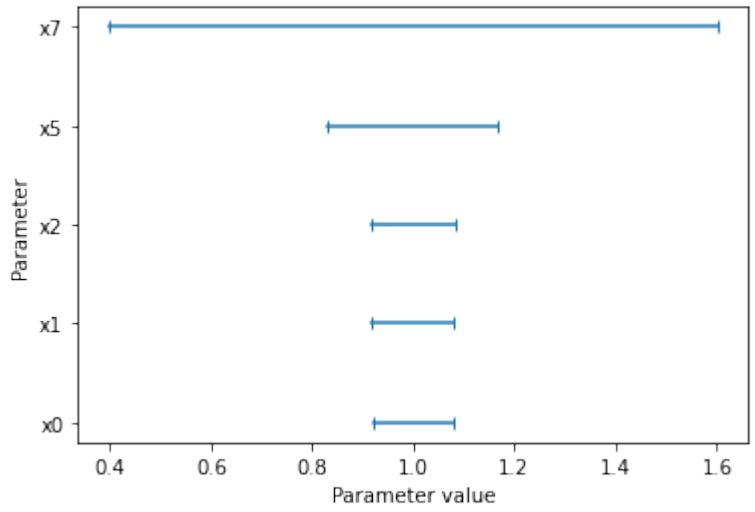
[16]:
```
axes = pypesto.visualize.profiles(
    result1_bfgs, profile_indices = [0,1,2,5,7], profile_list_ids=[0, 2],
    ratio_min=0.01, colors=[(1,0,0,1), (0,0,1,1)],
    legends=["Optimization-based profile", "Local profile approximation"])
```



We can also plot approximate confidence intervals based on profiles:

[17]:
```
pypesto.visualize.profile_cis(result1_bfgs, confidence_level=0.95, profile_list=2)
```

[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x126ba6a30>
```



2.2 Conversion reaction

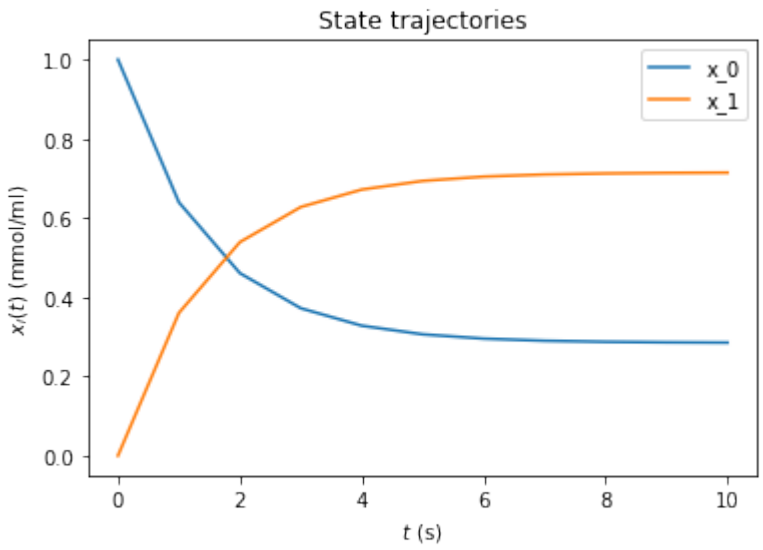
[1]:
```
import importlib
import os
import sys
import numpy as np
import amici
import amici.plotting
import pypesto

# sbml file we want to import
sbml_file = 'conversion_reaction/model_conversion_reaction.xml'
# name of the model that will also be the name of the python module
model_name = 'model_conversion_reaction'
# directory to which the generated model code is written
model_output_dir = 'tmp/' + model_name
```

2.2.1 Compile AMICI model

[2]:
```
# import sbml model, compile and generate amici module
sbml_importer = amici.SbmlImporter(sbml_file)
sbml_importer.sbml2amici(model_name,
                         model_output_dir,
                         verbose=False)
```

2.2.2 Load AMICI model

[3]:
```
# load amici module (the usual starting point later for the analysis)
sys.path.insert(0, os.path.abspath(model_output_dir))
model_module = importlib.import_module(model_name)
model = model_module.getModel()
model.requireSensitivitiesForAllParameters()
model.setTimepoints(amici.DoubleVector(np.linspace(0, 10, 11)))
model.setParameterScale(amici.ParameterScaling_log10)
model.setParameters(amici.DoubleVector([-0.3,-0.7]))
solver = model.getSolver()
solver.setSensitivityMethod(amici.SensitivityMethod_forward)
solver.setSensitivityOrder(amici.SensitivityOrder_first)

# how to run amici now:
rdata = amici.runAmiciSimulation(model, solver, None)
amici.plotting.plotStateTrajectories(rdata)
edata = amici.ExpData(rdata, 0.2, 0.0)
```



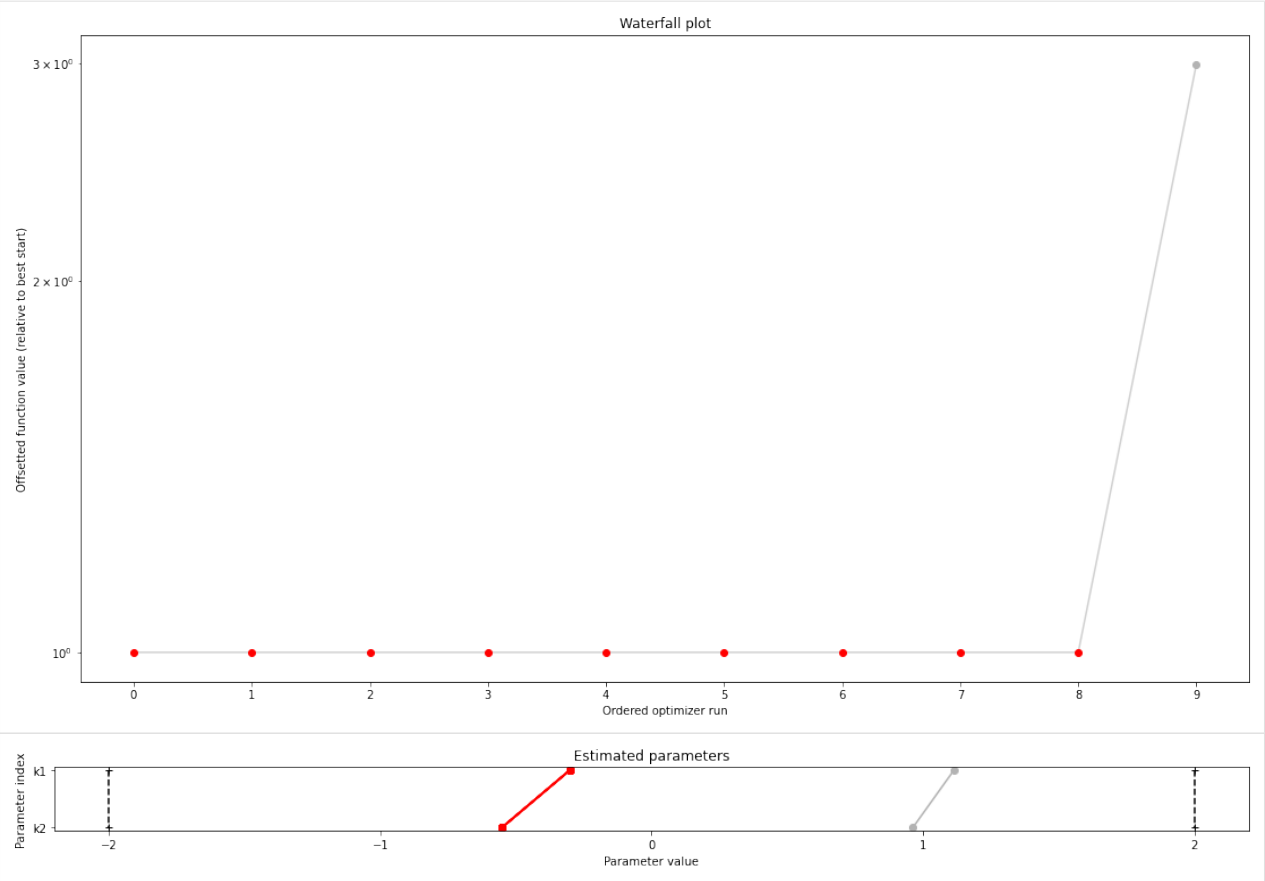
2.2.3 Optimize

[4]:
```
# create objective function from amici model
# pesto.AmiciObjective is derived from pesto.Objective,
# the general pesto objective function class
objective = pypesto.AmiciObjective(model, solver, [edata], 1)

# create optimizer object which contains all information for doing the optimization
optimizer = pypesto.ScipyOptimizer(method='ls_trf')

#optimizer.solver = 'bfgs|meigo'
# if select meigo -> also set default values in solver_options
#optimizer.options = {'maxiter': 1000, 'disp': True} # = pesto.default_options_meigo()
#optimizer.startpoints = []
#optimizer.startpoint_method = 'lhs|uniform|something|function'
#optimizer.n_starts = 100

# see PestoOptions.m for more required options here
# returns OptimizationResult, see parameters.MS for what to return
# list of final optim results foreach multistart, times, hess, grad,
# flags, meta information (which optimizer -> optimizer.get_repr())

# create problem object containing all information on the problem to be solved
problem = pypesto.Problem(objective=objective,
                          lb=[-2,-2], ub=[2,2])

# maybe lb, ub = inf
# other constraints: kwargs, class pesto.Constraints
# constraints on pams, states, esp. pesto.AmiciConstraints (e.g. pam1 + pam2<= const)
# if optimizer cannot handle -> error
# maybe also scaling / transformation of parameters encoded here

# do the optimization
result = pypesto.minimize(problem=problem,
                          optimizer=optimizer,
                          n_starts=10)
# optimize is a function since it does not need an internal memory,
# just takes input and returns output in the form of a Result object
# 'result' parameter: e.g. some results from somewhere -> pick best start points
```

2.2.4 Visualize

[5]:
```
# waterfall, parameter space, scatter plots, fits to data
# different functions for different plotting types
import pypesto.visualize

pypesto.visualize.waterfall(result)
pypesto.visualize.parameters(result)
```

[5]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f48849e1150>
```



2.2.5 Data storage

[6]:
```
# result = pypesto.storage.load('db_file.db')
```

2.2.6 Profiles

[7]:
```
# there are three main parts: optimize, profile, sample. the overall structure of␣
↪profiles and sampling
# will be similar to optimizer like above.
# we intend to only have just one result object which can be reused everywhere, but␣
↪the problem of how to
# not have one huge class but
# maybe simplified views on it for optimization, profiles and sampling is still to be␣
↪solved

# profiler = pypesto.Profiler()

# result = pypesto.profile(problem, profiler, result=None)
# possibly pass result object from optimization to get good parameter guesses
```

2.2.7 Sampling

[8]:
```
# sampler = pypesto.Sampler()

# result = pypesto.sample(problem, sampler, result=None)
```

[9]:
```
# open: how to parallelize. the idea is to use methods similar to those in pyabc for working on clusters.
# one way would be to specify an additional 'engine' object passed to optimize(), profile(), sample(),
# which in the default setting just does a for loop, but can also be customized.
```

2.3 Fixed parameters

In this notebook we will show how to use fixed parameters. Therefore, we employ our Rosenbrock example. We define two problems, where for the first problem all parameters are optimized, and for the second we fix some of them to specified values.

2.3.1 Define problem

[1]:
```
import pypesto
import pypesto.visualize
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

%matplotlib inline
```

[2]:
```
objective = pypesto.Objective(fun=sp.optimize.rosen,
                              grad=sp.optimize.rosen_der,
                              hess=sp.optimize.rosen_hess)

dim_full = 5
lb = -2 * np.ones((dim_full,1))
ub = 2 * np.ones((dim_full,1))

problem1 = pypesto.Problem(objective=objective, lb=lb, ub=ub)

x_fixed_indices = [1, 3]
x_fixed_vals = [1, 1]
problem2 = pypesto.Problem(objective=objective, lb=lb, ub=ub,
                           x_fixed_indices=x_fixed_indices,
                           x_fixed_vals=x_fixed_vals)
```

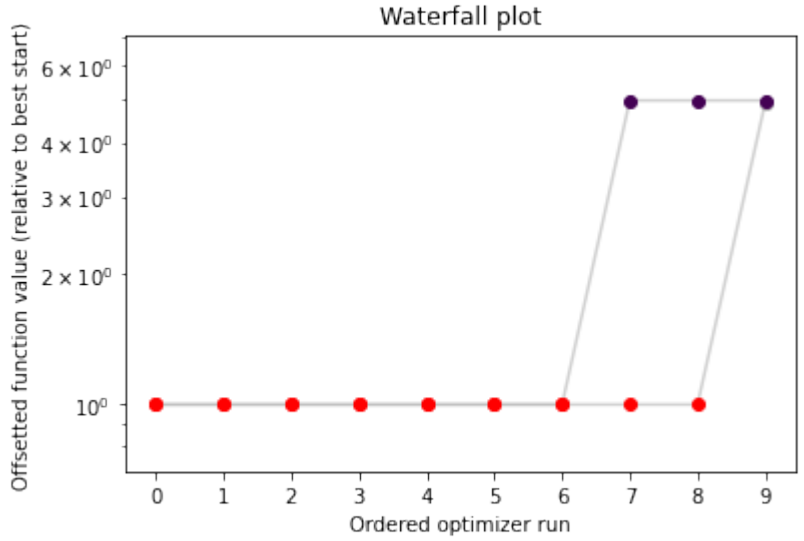
2.3.2 Optimize

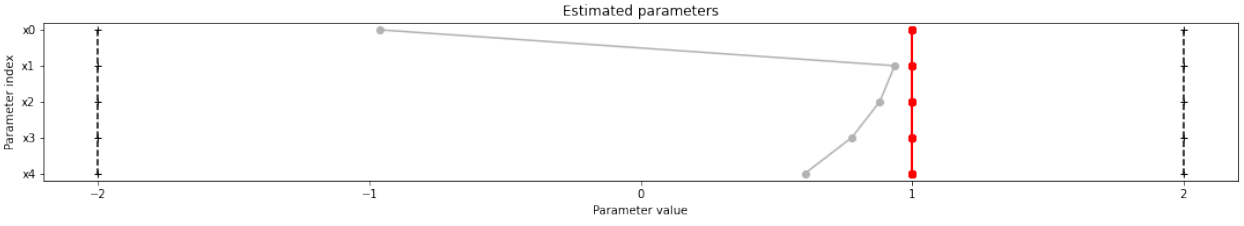
[3]:
```
optimizer = pypesto.ScipyOptimizer()
n_starts = 10

result1 = pypesto.minimize(problem=problem1, optimizer=optimizer,
                           n_starts=n_starts)
result2 = pypesto.minimize(problem=problem2, optimizer=optimizer,
                           n_starts=n_starts)
```

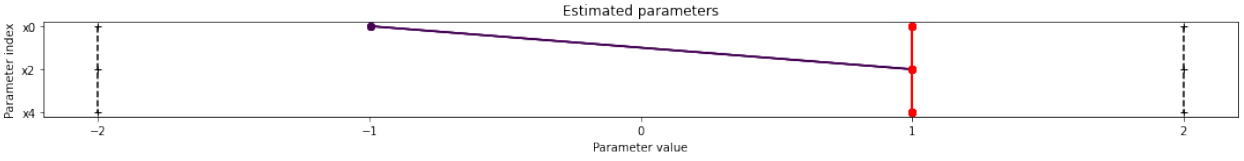
2.3.3 Visualize

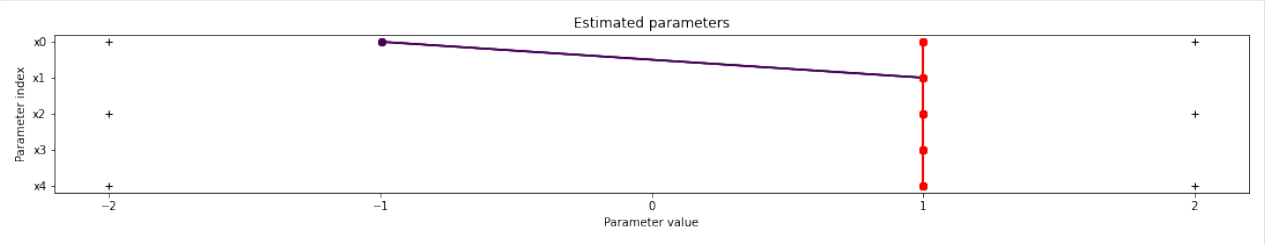
[4]:
```
fig, ax = plt.subplots()
pypesto.visualize.waterfall(result1, ax)
pypesto.visualize.waterfall(result2, ax)
pypesto.visualize.parameters(result1)
pypesto.visualize.parameters(result2)
pypesto.visualize.parameters(result2, free_indices_only=False)
```

[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe98fe12650>
```









[5]:
```
result1.optimize_result.as_dataframe(['fval', 'x', 'grad'])
```

[5]:

	fval	x	\
0	4.689854e-14	[0.9999999815635604, 0.9999999601303594, 0.999...	
1	5.590260e-14	[0.9999999788673749, 0.9999999710531022, 0.999...	
2	3.799999e-13	[0.9999999894531796, 0.9999999867856568, 1.000...	
3	4.110300e-13	[1.0000000452200102, 1.0000000894442573, 1.000...	
4	6.110801e-13	[0.9999999640563018, 0.9999999933489752, 0.999...	
5	1.231774e-12	[1.0000000581023145, 1.000000113974549, 1.0000...	
6	6.918491e-12	[1.0000000005661727, 1.0000001076896814, 1.000...	
7	2.552430e-11	[0.9999999192005172, 1.0000000369812814, 1.000...	
8	2.855055e-11	[0.9999995268339968, 0.9999992696810054, 0.999...	
9	3.930839e+00	[-0.9620508371994185, 0.9357391790840979, 0.88...	

	grad
0	[1.1618317897690609e-06, 1.2817350405303142e-0...
1	[-5.3696059283362676e-06, -3.17862832097996e-0...
2	[-3.1728126266944613e-06, -1.6951548081005115e...
3	[4.887460487692825e-07, 3.7999227735179252e-06...
4	[-2.6166434580847274e-05, 8.102944813282283e-0...
5	[1.0082380613547457e-06, 1.8071346123806702e-0...
6	[-4.262180210516511e-05, 7.700439519900304e-05...
7	[-7.959368869249685e-05, -8.749066771695052e-0...
8	[-8.735140622961871e-05, 4.373181014008344e-05...
9	[5.2019939965397555e-05, 2.44790688288532e-05,...

[6]:
```
result2.optimize_result.as_dataframe(['fval', 'x', 'grad'])
```

[6]:

	fval	x	\
0	3.757636e-21	[0.9999999999987625, 1.0, 1.0000000000008613, ...	
1	3.403187e-16	[1.000000000296693, 1.0, 0.9999999993226122, 1...	
2	1.078998e-15	[0.9999999994275696, 1.0, 0.9999999986532816, ...	
3	1.986582e-15	[0.9999999981059375, 1.0, 0.9999999990468542, ...	
4	1.777511e-14	[1.0000000013724608, 1.0, 0.9999999967022452, ...	
5	2.344376e-14	[0.9999999959141576, 1.0, 0.9999999996553852, ...	
6	2.096868e-13	[0.9999999984695723, 1.0, 1.0000000066163943, ...	
7	3.989975e+00	[-0.9949747468749881, 1.0, 0.9999999997811084,...	
8	3.989975e+00	[-0.9949747444243423, 1.0, 1.000000008470282, ...	
9	3.989975e+00	[-0.9949747299382655, 1.0, 1.0000000336260297,...	

	grad
0	[-9.92438575763029e-10, nan, 8.630336445497983...
1	[2.3794769983646036e-07, nan, -6.7874258776674...
2	[-4.5908920971880386e-07, nan, -1.349411838817...
3	[-1.5190380919959983e-06, nan, -9.550520753635...
4	[1.100713572463121e-06, nan, -3.30435034796972...
5	[-3.2768456252559055e-06, nan, -3.453040106231...
6	[-1.2274029922568326e-06, nan, 6.6296271449766...

(continues on next page)

(continued from previous page)

```
7  [-3.507845658390352e-08, nan, -2.1932935463983...
8  [1.900857428349667e-06, nan, 8.487222652784575...
9  [1.334441878420023e-05, nan, 3.369328314657676...
```

2.4 AMICI Python example "Boehm"

This is an example using the model [boehm_ProteomeRes2014.xml] model to demonstrate and test SBML import and AMICI Python interface.

[1]:
```
import libsbml
import importlib
import amici
import pypesto
import os
import sys
import numpy as np
import matplotlib.pyplot as plt

# temporarily add the simulate file
sys.path.insert(0, 'boehm_JProteomeRes2014')

from benchmark_import import DataProvider

# sbml file
sbml_file = 'boehm_JProteomeRes2014/boehm_JProteomeRes2014.xml'

# name of the model that will also be the name of the python module
model_name = 'boehm_JProteomeRes2014'

# output directory
model_output_dir = 'tmp/' + model_name
```

2.4.1 The example model

Here we use `libsbml` to show the reactions and species described by the model (this is independent of AMICI).

[2]:
```
sbml_reader = libsbml.SBMLReader()
sbml_doc = sbml_reader.readSBML(os.path.abspath(sbml_file))
sbml_model = sbml_doc.getModel()
dir(sbml_doc)
print(os.path.abspath(sbml_file))
print('Species: ', [s.getId() for s in sbml_model.getListOfSpecies()])


print('\nReactions:')
for reaction in sbml_model.getListOfReactions():
    reactants = ' + '.join(['%s %s'%(int(r.getStoichiometry()) if r.
↪getStoichiometry() > 1 else '', r.getSpecies()) for r in reaction.
↪getListOfReactants()])
    products  = ' + '.join(['%s %s'%(int(r.getStoichiometry()) if r.
↪getStoichiometry() > 1 else '', r.getSpecies()) for r in reaction.
↪getListOfProducts()])
```

(continues on next page)

(continued from previous page)

```
    reversible = '<' if reaction.getReversible() else ''
    print('%3s: %10s %1s->%10s\t\t[%s]' % (reaction.getId(),
                         reactants,
                         reversible,
                          products,
                         libsbml.formulaToL3String(reaction.getKineticLaw().
↪getMath())))
```

```
/home/yannik/pypesto/doc/example/boehm_JProteomeRes2014/boehm_JProteomeRes2014.xml
Species:  ['STAT5A', 'STAT5B', 'pApB', 'pApA', 'pBpB', 'nucpApA', 'nucpApB', 'nucpBpB
↪']

Reactions:
v1_v_0:   2 STAT5A  ->      pApA              [cyt * BaF3_Epo * STAT5A^2 * k_phos]
v2_v_1:  STAT5A +  STAT5B  ->      pApB               [cyt * BaF3_Epo * STAT5A *␣
↪STAT5B * k_phos]
v3_v_2:   2 STAT5B  ->      pBpB              [cyt * BaF3_Epo * STAT5B^2 * k_phos]
v4_v_3:       pApA  ->   nucpApA              [cyt * k_imp_homo * pApA]
v5_v_4:       pApB  ->   nucpApB              [cyt * k_imp_hetero * pApB]
v6_v_5:       pBpB  ->   nucpBpB              [cyt * k_imp_homo * pBpB]
v7_v_6:    nucpApA  ->  2 STAT5A              [nuc * k_exp_homo * nucpApA]
v8_v_7:    nucpApB  -> STAT5A +  STAT5B               [nuc * k_exp_hetero * nucpApB]
v9_v_8:    nucpBpB  ->  2 STAT5B              [nuc * k_exp_homo * nucpBpB]
```

2.4.2 Importing an SBML model, compiling and generating an AMICI module

Before we can use AMICI to simulate our model, the SBML model needs to be translated to C++ code. This is done by `amici.SbmlImporter`.

[3]:
```
# Create an SbmlImporter instance for our SBML model
sbml_importer = amici.SbmlImporter(sbml_file)
```

In this example, we want to specify fixed parameters, observables and a σ parameter. Unfortunately, the latter two are not part of the SBML standard. However, they can be provided to `amici.SbmlImporter.sbml2amici` as demonstrated in the following.

Constant parameters

Constant parameters, i.e. parameters with respect to which no sensitivities are to be computed (these are often parameters specifying a certain experimental condition) are provided as a list of parameter names.

[4]:
```
constantParameters = {'ratio', 'specC17'}
```

Observables

We used SBML's `AssignmentRule <http://sbml.org/Software/libSBML/5.13.0/docs//python-api/classlibsbml_1_1_rule.html>`__ as a non-standard way to specify *Model outputs* within the SBML file. These rules need to be removed prior to the model import (AMICI does at this time not support these Rules). This can be easily done using `amici.assignmentRules2observables()`.

In this example, we introduced parameters named `observable_*` as targets of the observable AssignmentRules. Where applicable we have `observable_*_sigma` parameters for σ parameters (see below).

[5]:
```
# Retrieve model output names and formulae from AssignmentRules and remove the␣
↪respective rules
observables = amici.assignmentRules2observables(
        sbml_importer.sbml, # the libsbml model object
        filter_function=lambda variable: variable.getId().startswith('observable_')␣
↪and not variable.getId().endswith('_sigma')
    )
print('Observables:', observables)
```

```
Observables: {'observable_pSTAT5A_rel': {'name': '', 'formula': '(100 * pApB + 200 *␣
↪pApA * specC17) / (pApB + STAT5A * specC17 + 2 * pApA * specC17)'}, 'observable_
↪pSTAT5B_rel': {'name': '', 'formula': '-(100 * pApB - 200 * pBpB * (specC17 - 1)) /␣
↪(STAT5B * (specC17 - 1) - pApB + 2 * pBpB * (specC17 - 1))'}, 'observable_rSTAT5A_
↪rel': {'name': '', 'formula': '(100 * pApB + 100 * STAT5A * specC17 + 200 * pApA *␣
↪specC17) / (2 * pApB + STAT5A * specC17 + 2 * pApA * specC17 - STAT5B * (specC17 -␣
↪1) - 2 * pBpB * (specC17 - 1))'}}
```

σ parameters

To specify measurement noise as a parameter, we simply provide a dictionary with (preexisting) parameter names as keys and a list of observable names as values to indicate which sigma parameter is to be used for which observable.

[6]:
```
sigma_vals = ['sd_pSTAT5A_rel', 'sd_pSTAT5B_rel', 'sd_rSTAT5A_rel']
observable_names = observables.keys()
sigmas = dict(zip(list(observable_names), sigma_vals))
print(sigmas)
```

```
{'observable_pSTAT5A_rel': 'sd_pSTAT5A_rel', 'observable_pSTAT5B_rel': 'sd_pSTAT5B_rel
↪', 'observable_rSTAT5A_rel': 'sd_rSTAT5A_rel'}
```

Generating the module

Now we can generate the python module for our model. `amici.SbmlImporter.sbml2amici` will symbolically derive the sensitivity equations, generate C++ code for model simulation, and assemble the python module.

[7]:
```
sbml_importer.sbml2amici(model_name,
                         model_output_dir,
                         verbose=False,
                         observables=observables,
                         constantParameters=constantParameters,
                         sigmas=sigmas
  )
```

Importing the module and loading the model

If everything went well, we need to add the previously selected model output directory to our PYTHON_PATH and are then ready to load newly generated model:

[8]:
```
sys.path.insert(0, os.path.abspath(model_output_dir))
model_module = importlib.import_module(model_name)
```

And get an instance of our model from which we can retrieve information such as parameter names:

[9]:
```
model = model_module.getModel()

print("Model parameters:", list(model.getParameterIds()))
print("Model outputs:   ", list(model.getObservableIds()))
print("Model states:    ", list(model.getStateIds()))
```

```
Model parameters: ['Epo_degradation_BaF3', 'k_exp_hetero', 'k_exp_homo', 'k_imp_hetero
↪', 'k_imp_homo', 'k_phos', 'sd_pSTAT5A_rel', 'sd_pSTAT5B_rel', 'sd_rSTAT5A_rel']
Model outputs:    ['observable_pSTAT5A_rel', 'observable_pSTAT5B_rel', 'observable_
↪rSTAT5A_rel']
Model states:     ['STAT5A', 'STAT5B', 'pApB', 'pApA', 'pBpB', 'nucpApA', 'nucpApB',
↪'nucpBpB']
```

2.4.3 Running simulations and analyzing results

After importing the model, we can run simulations using `amici.runAmiciSimulation`. This requires a `Model` instance and a `Solver` instance. Optionally you can provide measurements inside an `ExpData` instance, as shown later in this notebook.

[10]:
```
h5_file = 'boehm_JProteomeRes2014/data_boehm_JProteomeRes2014.h5'
dp = DataProvider(h5_file)
```

[11]:
```
# set timepoints for which we want to simulate the model
timepoints = amici.DoubleVector(dp.get_timepoints())
model.setTimepoints(timepoints)

# set fixed parameters for which we want to simulate the model
model.setFixedParameters(amici.DoubleVector(np.array([0.693, 0.107])))

# set parameters to optimal values found in the benchmark collection
model.setParameterScale(2)
model.setParameters(amici.DoubleVector(np.array([-1.568917588,
-4.999704894,
-2.209698782,
-1.786006548,
4.990114009,
4.197735488,
0.585755271,
0.818982819,
0.498684404
])))

# Create solver instance
solver = model.getSolver()
```

(continues on next page)

(continued from previous page)

```
# Run simulation using model parameters from the benchmark collection and default␣
↪solver options
rdata = amici.runAmiciSimulation(model, solver)
```

[12]:
```
# Create edata
edata = amici.ExpData(rdata, 1.0, 0)

# set observed data
edata.setObservedData(amici.DoubleVector(dp.get_measurements()[0][:, 0]), 0)
edata.setObservedData(amici.DoubleVector(dp.get_measurements()[0][:, 1]), 1)
edata.setObservedData(amici.DoubleVector(dp.get_measurements()[0][:, 2]), 2)

# set standard deviations to optimal values found in the benchmark collection
edata.setObservedDataStdDev(amici.DoubleVector(np.array(16*[10**0.585755271])), 0)
edata.setObservedDataStdDev(amici.DoubleVector(np.array(16*[10**0.818982819])), 1)
edata.setObservedDataStdDev(amici.DoubleVector(np.array(16*[10**0.498684404])), 2)
```

[13]:
```
rdata = amici.runAmiciSimulation(model, solver, edata)

print('Chi2 value reported in benchmark collection: 47.9765479')
print('chi2 value using AMICI:')
print(rdata['chi2'])
```

```
Chi2 value reported in benchmark collection: 47.9765479
chi2 value using AMICI:
47.97654266893465
```

2.4.4 Run optimization using pyPESTO

[14]:
```
# create objective function from amici model
# pesto.AmiciObjective is derived from pesto.Objective,
# the general pesto objective function class

model.requireSensitivitiesForAllParameters()


solver.setSensitivityMethod(amici.SensitivityMethod_forward)
solver.setSensitivityOrder(amici.SensitivityOrder_first)


objective = pypesto.AmiciObjective(model, solver, [edata], 1)
```

[15]:
```
# create optimizer object which contains all information for doing the optimization
optimizer = pypesto.ScipyOptimizer()

optimizer.solver = 'bfgs'
```

[16]:
```
# create problem object containing all information on the problem to be solved
x_names = ['x' + str(j) for j in range(0, 9)]
problem = pypesto.Problem(objective=objective,
                          lb=-5*np.ones((9)), ub=5*np.ones((9)),
                          x_names=x_names)
```

[17]:

```
# do the optimization
result = pypesto.minimize(problem=problem,
                          optimizer=optimizer,
                          n_starts=10) # 200
```

```
[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 221.821 and h = 3.00478e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 221.821149:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 221.821 and h = 3.00478e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 221.821149:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 221.821 and h = 3.00478e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 221.821149:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 147.199 and h = 2.90261e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 147.198629:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 147.199 and h = 2.90261e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 147.198629:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 147.199 and h = 2.90261e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 147.198629:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 198 and h = 2.97875e-05, the error test failed repeatedly or with␣
↪|h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 197.999609:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 197.697 and h = 2.98464e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 197.696730:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 197.697 and h = 2.98464e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 197.696730:
AMICI failed to integrate the forward problem
```

(continues on next page)

(continued from previous page)

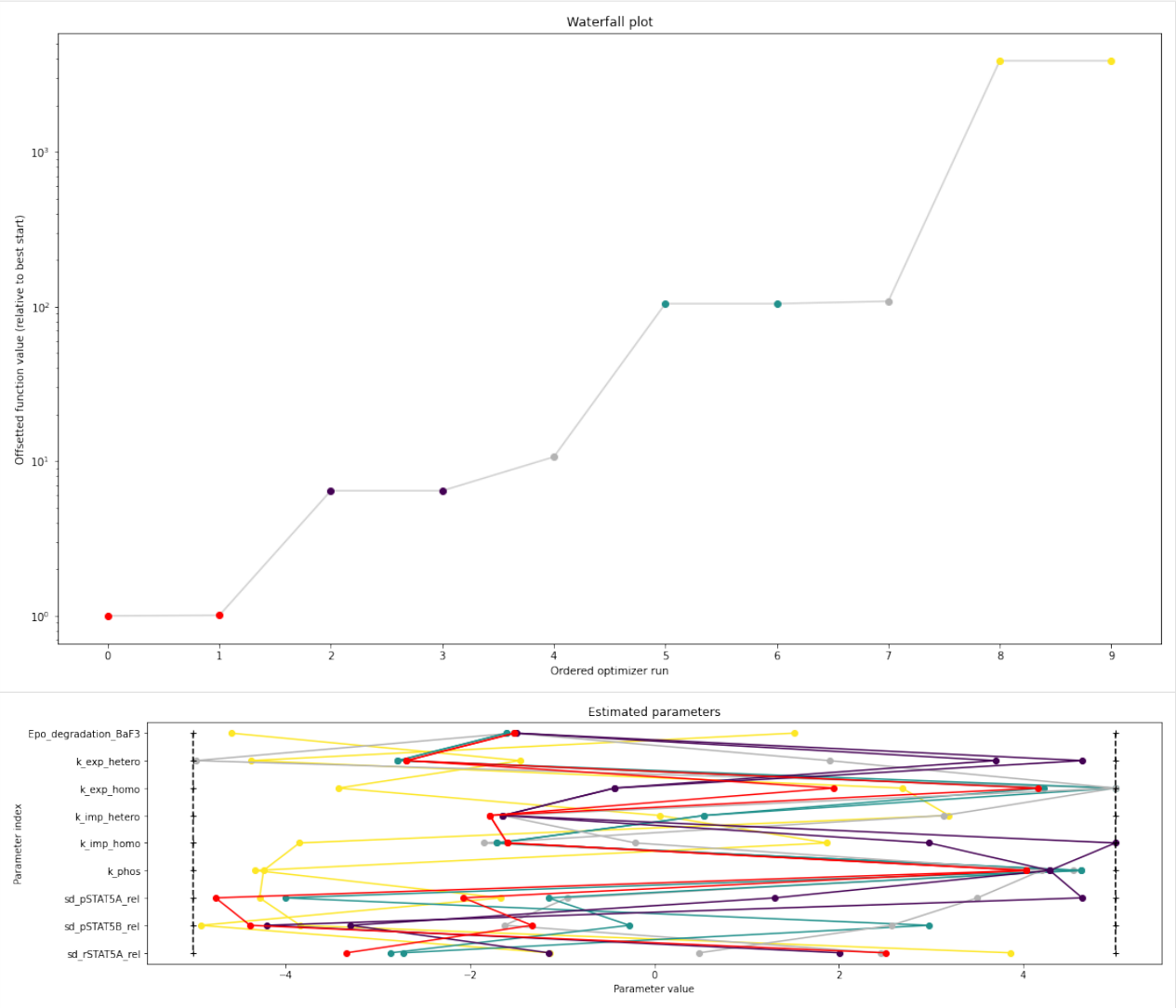
```
[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 197.697 and h = 2.98464e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 197.696730:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 66.4603 and h = 6.88533e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 66.460272:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 66.3735 and h = 8.78908e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 66.373478:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 85.8974 and h = 2.05376e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 85.897359:
AMICI failed to integrate the forward problem
```

2.4.5 Visualization

Create waterfall and parameter plot

[18]:
```
# waterfall, parameter space,
import pypesto.visualize

pypesto.visualize.waterfall(result)
pypesto.visualize.parameters(result)
```

[18]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3bc5881c10>
```



2.5 Model import using the Petab format

In this notebook, we illustrate how to use pyPESTO together with PEtab and AMICI. We employ models from the benchmark collection, which we first download:

[1]:
```
import pypesto
import amici
import petab

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

%matplotlib inline

!git clone --depth 1 https://github.com/Benchmarking-Initiative/Benchmark-Models-
↪PEtab.git tmp/benchmark-models || (cd tmp/benchmark-models && git pull)
```

(continues on next page)

(continued from previous page)

```
folder_base = "tmp/benchmark-models/Benchmark-Models/"
```

```
fatal: destination path 'tmp/benchmark-models' already exists and is not an empty␣
↪directory.
Already up to date.
```

2.5.1 Import

Manage PEtab model

A PEtab problem comprises all the information on the model, the data and the parameters to perform parameter estimation. We import a model as a `petab.Problem`.

[2]:
```
# a collection of models that can be simulated

#model_name = "Zheng_PNAS2012"
model_name = "Boehm_JProteomeRes2014"
#model_name = "Fujita_SciSignal2010"
#model_name = "Sneyd_PNAS2002"
#model_name = "Borghans_BiophysChem1997"
#model_name = "Elowitz_Nature2000"
#model_name = "Crauste_CellSystems2017"
#model_name = "Lucarelli_CellSystems2018"
#model_name = "Schwen_PONE2014"
#model_name = "Blasi_CellSystems2016"

# the yaml configuration file links to all needed files
yaml_config = os.path.join(folder_base, model_name, model_name + '.yaml')

# create a petab problem
petab_problem = petab.Problem.from_yaml(yaml_config)
```

Import model to AMICI

The model must be imported to pyPESTO and AMICI. Therefore, we create a `pypesto.PetabImporter` from the problem, and create an AMICI model.

[3]:
```
importer = pypesto.PetabImporter(petab_problem)

model = importer.create_model()

# some model properties
print("Model parameters:", list(model.getParameterIds()), '\n')
print("Model const parameters:", list(model.getFixedParameterIds()), '\n')
print("Model outputs:    ", list(model.getObservableIds()), '\n')
print("Model states:     ", list(model.getStateIds()), '\n')
```

```
Model parameters: ['Epo_degradation_BaF3', 'k_exp_hetero', 'k_exp_homo', 'k_imp_hetero
↪', 'k_imp_homo', 'k_phos', 'ratio', 'specC17', 'noiseParameter1_pSTAT5A_rel',
↪'noiseParameter1_pSTAT5B_rel', 'noiseParameter1_rSTAT5A_rel']

Model const parameters: []
```

(continues on next page)

(continued from previous page)

```
Model outputs:    ['pSTAT5A_rel', 'pSTAT5B_rel', 'rSTAT5A_rel']

Model states:     ['STAT5A', 'STAT5B', 'pApB', 'pApA', 'pBpB', 'nucpApA', 'nucpApB',
↪'nucpBpB']
```

Create objective function

To perform parameter estimation, we need to define an objective function, which integrates the model, data, and noise model defined in the PEtab problem.

[4]:
```
import libsbml
converter_config = libsbml.SBMLLocalParameterConverter()\
    .getDefaultProperties()
petab_problem.sbml_document.convert(converter_config)

obj = importer.create_objective()

# for some models, hyperparamters need to be adjusted
#obj.amici_solver.setMaxSteps(10000)
#obj.amici_solver.setRelativeTolerance(1e-7)
#obj.amici_solver.setAbsoluteTolerance(1e-7)
```

We can request variable derivatives via `sensi_orders`, or function values or residuals as specified via `mode`. Passing `return_dict`, we obtain the direct result of the AMICI simulation.

[5]:
```
ret = obj(petab_problem.x_nominal_scaled, mode='mode_fun', sensi_orders=(0,1), return_
↪dict=True)
print(ret)
```

```
{'fval': 138.22199677513575, 'grad': array([ 2.20386015e-02,  5.53227506e-02,  5.
↪78886452e-03,  5.40656415e-03,
       -4.51595809e-05,  7.91163446e-03,  0.00000000e+00,  1.07840959e-02,
        2.40378735e-02,  1.91919657e-02,  0.00000000e+00]), 'hess': array([[ 2.
↪11105595e+03,  5.89390039e-01,  1.07159910e+02,
         2.81393973e+03,  8.94333861e-06, -7.86055092e+02,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 5.89390039e-01,  1.91513744e-03, -1.72774945e-01,
         7.12558479e-01, -3.69774927e-08, -3.20531692e-01,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 1.07159910e+02, -1.72774945e-01,  6.99839693e+01,
         1.61497679e+02,  7.16323554e-06, -8.83572656e+01,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 2.81393973e+03,  7.12558479e-01,  1.61497679e+02,
         3.76058352e+03,  8.40044683e-06, -1.04136909e+03,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 8.94333861e-06, -3.69774927e-08,  7.16323554e-06,
         8.40044683e-06,  2.86438192e-10, -2.24927732e-04,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-7.86055092e+02, -3.20531692e-01, -8.83572656e+01,
```

(continues on next page)

(continued from previous page)

```
        -1.04136909e+03, -2.24927732e-04,  9.29902113e+02,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00]]), 'res': array([], dtype=float64), 'sres':␣
↪array([], shape=(0, 11), dtype=float64), 'rdatas': [<amici.numpy.ReturnDataView␣
↪object at 0x7f7802f86610>]}
```

The problem defined in PEtab also defines the fixing of parameters, and parameter bounds. This information is contained in a `pypesto.Problem`.

[6]:
```
problem = importer.create_problem(obj)
```

In particular, the problem accounts for the fixing of parametes.

[7]:
```
print(problem.x_fixed_indices, problem.x_free_indices)
```

```
[6, 10] [0, 1, 2, 3, 4, 5, 7, 8, 9]
```

The problem creates a copy of he objective function that takes into account the fixed parameters. The objective function is able to calculate function values and derivatives. A finite difference check whether the computed gradient is accurate:

[8]:
```
objective = problem.objective
ret = objective(petab_problem.x_nominal_free_scaled, sensi_orders=(0,1))
print(ret)
```

```
(138.22199677513575, array([ 2.20386015e-02,  5.53227506e-02,  5.78886452e-03,  5.
↪40656415e-03,
       -4.51595809e-05,  7.91163446e-03,  1.07840959e-02,  2.40378735e-02,
        1.91919657e-02]))
```

[9]:
```
eps = 1e-4

def fd(x):
    grad = np.zeros_like(x)
    j = 0
    for i, xi in enumerate(x):
        mask = np.zeros_like(x)
```

(continues on next page)

(continued from previous page)

```
        mask[i] += eps
        valinc, _ = objective(x+mask, sensi_orders=(0,1))
        valdec, _ = objective(x-mask, sensi_orders=(0,1))
        grad[j] = (valinc - valdec) / (2*eps)
        j += 1
    return grad

fdval = fd(petab_problem.x_nominal_free_scaled)
print("fd: ", fdval)
print("l2 difference: ", np.linalg.norm(ret[1] - fdval))
```

```
fd:  [0.02493368 0.05309659 0.00530587 0.01291083 0.00587754 0.01473653
 0.01078279 0.02403657 0.01919066]
l2 difference:  0.012310244824532144
```

In short

All of the previous steps can be shortened by directly creating an importer object and then a problem:

[10]:
```
importer = pypesto.PetabImporter.from_yaml(yaml_config)
problem = importer.create_problem()
```

2.5.2 Run optimization

Given the problem, we can perform optimization. We can specify an optimizer to use, and a parallelization engine to speed things up.

[11]:
```
optimizer = pypesto.ScipyOptimizer()

# engine = pypesto.SingleCoreEngine()
engine = pypesto.MultiProcessEngine()

# do the optimization
result = pypesto.minimize(problem=problem, optimizer=optimizer,
                          n_starts=10, engine=engine)
```

```
Engine set up to use up to 4 processes in total. The number was automatically␣
↪determined and might not be appropriate on some systems.
[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 38.1195 and h = 5.55541e-06, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 38.119511:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 88.9211 and h = 2.14177e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 88.921131:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 88.9211 and h = 2.14177e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 88.921131:
```

(continues on next page)

(continued from previous page)

```
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 88.9211 and h = 2.14177e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 88.921131:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 145.551 and h = 1.32433e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 145.550813:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 145.551 and h = 1.32433e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 145.550813:
AMICI failed to integrate the forward problem

[Warning] AMICI:CVODES:CVode:ERR_FAILURE: AMICI ERROR: in module CVODES in function␣
↪CVode : At t = 145.551 and h = 1.32433e-05, the error test failed repeatedly or␣
↪with |h| = hmin.
[Warning] AMICI:simulation: AMICI forward simulation failed at t = 145.550813:
AMICI failed to integrate the forward problem
```

2.5.3 Visualize

The results are contained in a `pypesto.Result` object. It contains e.g. the optimal function values.

[12]:

```
result.optimize_result.get_for_key('fval')
```

[12]:

```
[138.2219740350346,
 138.22404611978106,
 145.7594099868979,
 147.54397516143254,
 149.58782926326572,
 151.16644923400784,
 154.73312826411254,
 205.61953652493594,
 249.27713115708494,
 249.7459974433355]
```

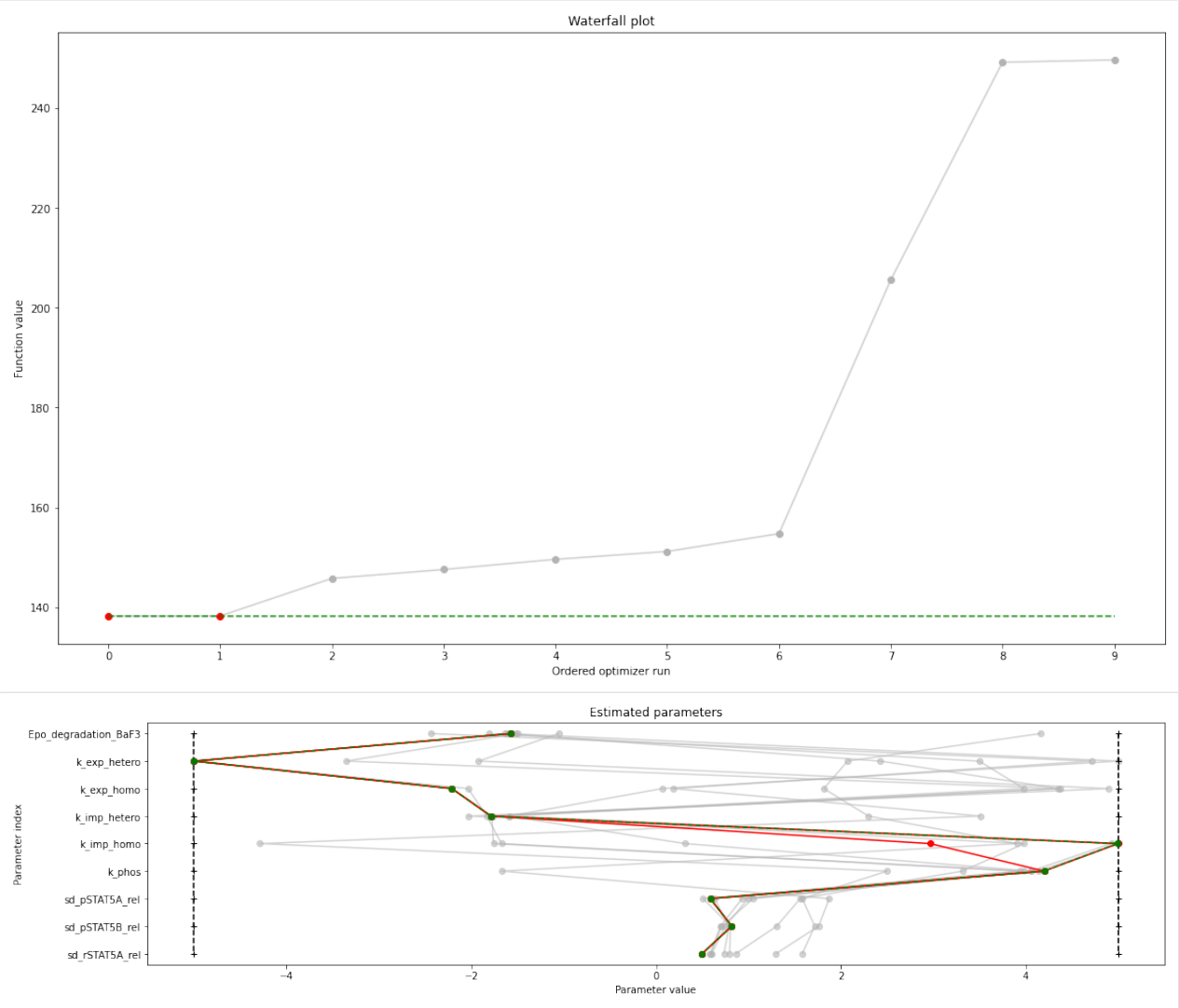
We can use the standard pyPESTO plotting routines to visualize and analyze the results.

[13]:

```
import pypesto.visualize

ref = pypesto.visualize.create_references(x=petab_problem.x_nominal_scaled,␣
↪fval=obj(petab_problem.x_nominal_scaled))

pypesto.visualize.waterfall(result, reference=ref, scale_y='lin')
pypesto.visualize.parameters(result, reference=ref)
```

[13]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7802e43510>
```



2.6 Storage

This notebook illustrates how simulations and results can be saved to file.

[1]:
```
import pypesto
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from pypesto.storage import (save_to_hdf5, read_from_hdf5)
import tempfile

%matplotlib inline
```

2.6.1 Define the objective and problem

[2]:
```
objective = pypesto.Objective(fun=sp.optimize.rosen,
                              grad=sp.optimize.rosen_der,
                              hess=sp.optimize.rosen_hess)

dim_full = 10
lb = -3 * np.ones((dim_full, 1))
ub = 3 * np.ones((dim_full, 1))

problem = pypesto.Problem(objective=objective, lb=lb, ub=ub)

# create optimizers
optimizer = pypesto.ScipyOptimizer(method='l-bfgs-b')

# set number of starts
n_starts = 20
```

2.6.2 Objective function traces

During optimization, it is possible to regularly write the objective function trace to file. This is useful e.g. when runs fail, or for various diagnostics. Currently, pyPESTO can save histories to 3 backends: in-memory, as CSV files, or to HDF5 files.

Memory History

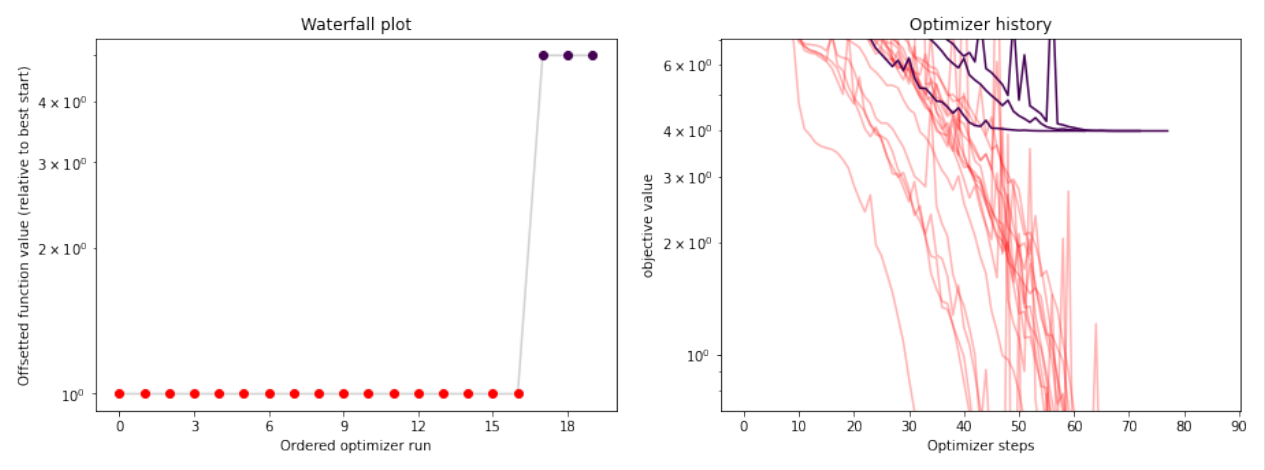
To record the history in-memory, just set `trace_record=True` in the `pypesto.HistoryOptions`. Then, the optimization result contains those histories:

[3]:
```
# record the history
history_options = pypesto.HistoryOptions(trace_record=True)

# Run optimizaitons
result = pypesto.minimize(
    problem=problem, optimizer=optimizer,
    n_starts=n_starts, history_options=history_options)
```

Now, in addition to queries on the result, we can also access the

[4]:
```
print("History type: ", type(result.optimize_result.list[0].history))
# print("Function value trace of best run: ", result.optimize_result.list[0].history.
↪get_fval_trace())

fig, ax = plt.subplots(1, 2)
pypesto.visualize.waterfall(result, ax=ax[0])
pypesto.visualize.optimizer_history(result, ax=ax[1])
fig.set_size_inches((15, 5))
```

```
History type:  <class 'pypesto.objective.history.MemoryHistory'>
```



CSV History

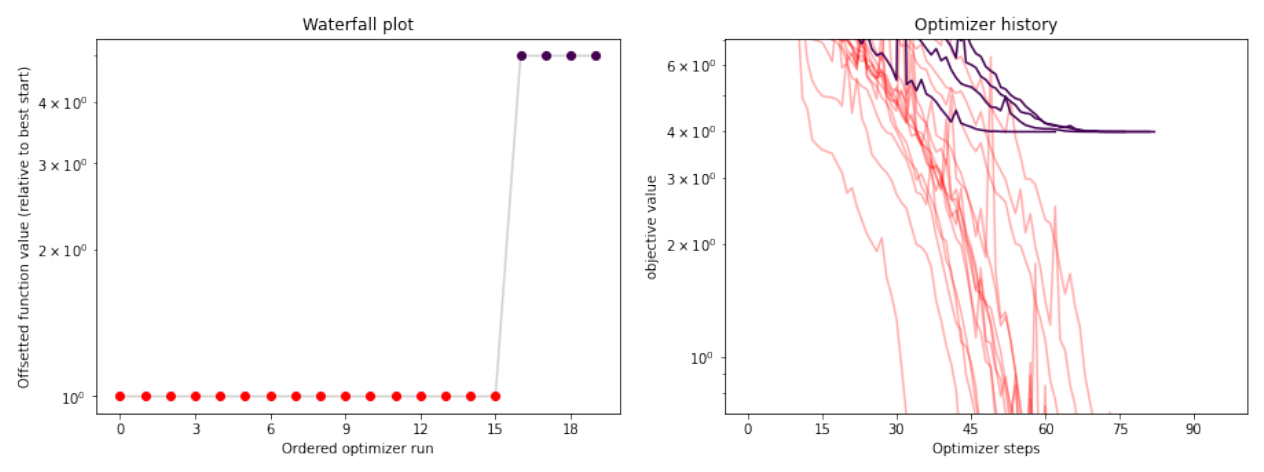
The in-memory storage is however not stored anywhere. To do that, it is possible to store either to CSV or HDF5. This is specified via the `storage_file` option. If it ends in `.csv`, a `pypesto.objective.history.CsvHistory` will be employed; if it ends in `.hdf5` a `pypesto.objective.history.Hdf5History`. Occurrences of the substring `{id}` in the filename are replaced by the multistart id, allowing to maintain a separate file per run (this is necessary for CSV as otherwise runs are overwritten).

[5]:
```
# record the history and store to CSV
history_options = pypesto.HistoryOptions(trace_record=True, storage_file='history_{id}
↪.csv')

# Run optimizaitons
result = pypesto.minimize(
    problem=problem, optimizer=optimizer,
    n_starts=n_starts, history_options=history_options)
```

Note that for this simple cost function, saving to CSV takes a considerable amount of time. This overhead decreases for more costly simulators, e.g. using ODE simulations via AMICI.

[6]:
```
print("History type: ", type(result.optimize_result.list[0].history))
# print("Function value trace of best run: ", result.optimize_result.list[0].history.
↪get_fval_trace())

fig, ax = plt.subplots(1, 2)
pypesto.visualize.waterfall(result, ax=ax[0])
pypesto.visualize.optimizer_history(result, ax=ax[1])
fig.set_size_inches((15, 5))
```

```
History type:  <class 'pypesto.objective.history.CsvHistory'>
```



HDF5 History

TODO: This is not fully implemented yet (it's on the way . . .).

2.6.3 Result storage

Result objects can be stored to HDF5 files. When appliable, this is preferable to just pickling results, which is not guaranteed to be reproducible in the future.

[7]:
```
# Run optimizaitons
result = pypesto.minimize(
    problem=problem, optimizer=optimizer,
    n_starts=n_starts)
```

[8]:
```
result.optimize_result.list[0:2]
```

[8]:
```
[{'id': '16',
  'x': array([0.99999998, 0.99999999, 0.99999999, 1.00000002, 1.00000004,
         1.00000005, 1.00000011, 1.00000019, 1.00000041, 1.00000086]),
  'fval': 1.0840759188627262e-12,
  'grad': array([-1.54076893e-05,  4.88166217e-06, -1.27059470e-05,  6.96391071e-06,
          1.48065820e-05, -1.19529462e-05,  1.62532245e-05, -1.91045260e-05,
         -1.18540309e-05,  9.49378891e-06]),
  'hess': None,
  'res': None,
  'sres': None,
  'n_fval': 68,
  'n_grad': 68,
  'n_hess': 0,
  'n_res': 0,
  'n_sres': 0,
  'x0': array([-2.06905833, -0.19206942, -1.72867234, -2.33804033, -0.71061311,
         -2.98146846, -1.10208315, -0.02865718,  2.40969018, -2.71535605]),
  'fval0': 28216.041116743356,
  'history': <pypesto.objective.history.History at 0x7faa1d10ee90>,
  'exitflag': 0,
  'time': 0.017784595489501953,
  'message': b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'},
```

(continues on next page)

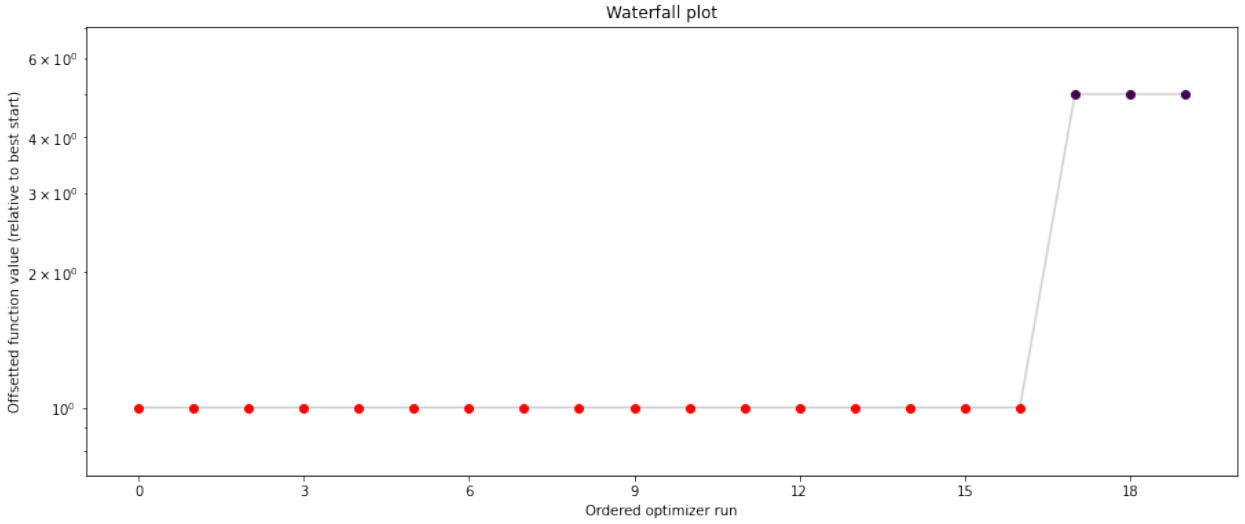
(continued from previous page)

```
{'id': '18',
 'x': array([0.99999998, 0.99999997, 1.00000004, 1.00000003, 1.00000008,
        1.00000005, 1.00000001, 1.00000002, 1.00000016, 1.00000028]),
 'fval': 4.7323128234188734e-12,
 'grad': array([-5.95491904e-06, -3.45232841e-05,  3.78255472e-05, -1.56002304e-05,
         5.19710685e-05,  9.03199095e-06, -1.61464965e-05, -4.44088779e-05,
         3.77395966e-05, -7.65640382e-06]),
 'hess': None,
 'res': None,
 'sres': None,
 'n_fval': 83,
 'n_grad': 83,
 'n_hess': 0,
 'n_res': 0,
 'n_sres': 0,
 'x0': array([ 2.09368986, -1.92757011,  2.17342608,  0.94097953, -1.36443024,
        -0.4115728 ,  2.10149021,  1.64481536,  0.66254531,  0.15108917]),
 'fval0': 8262.214270452409,
 'history': <pypesto.objective.history.History at 0x7faa1d10e090>,
 'exitflag': 0,
 'time': 0.018738746643066406,
 'message': b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'}]
```

As usual, having obtained our result, we can directly perform some plots:

[9]:
```
import pypesto.visualize

# plot waterfalls
pypesto.visualize.waterfall(result, size=(15,6))
```

[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7faa1d8fb550>
```



Save optimization result as HDF5 file

The optimization result can be saved via a `pypesto.OptimizationResultHDF5Writer`.

[10]:
```
fn = tempfile.mktemp(".hdf5")

# Write result
hdf5_writer = save_to_hdf5.OptimizationResultHDF5Writer(fn)
hdf5_writer.write(result)

# Write problem
hdf5_writer = save_to_hdf5.ProblemHDF5Writer(fn)
hdf5_writer.write(problem)
```

Read optimization result from HDF5 file

When reading in the stored result again, we recover the original optimization result:

[11]:
```
# Read result and problem
hdf5_reader = read_from_hdf5.OptimizationResultHDF5Reader(fn)
result = hdf5_reader.read()
```

[12]:
```
result.optimize_result.list[0:2]
```

[12]:
```
[{'id': '16',
  'x': array([0.99999998, 0.99999999, 0.99999999, 1.00000002, 1.00000004,
         1.00000005, 1.00000011, 1.00000019, 1.00000041, 1.00000086]),
  'fval': 1.0840759188627262e-12,
  'grad': array([-1.54076893e-05,  4.88166217e-06, -1.27059470e-05,  6.96391071e-06,
          1.48065820e-05, -1.19529462e-05,  1.62532245e-05, -1.91045260e-05,
         -1.18540309e-05,  9.49378891e-06]),
  'hess': None,
  'res': None,
  'sres': None,
  'n_fval': 68,
  'n_grad': 68,
  'n_hess': 0,
  'n_res': 0,
  'n_sres': 0,
  'x0': array([-2.06905833, -0.19206942, -1.72867234, -2.33804033, -0.71061311,
         -2.98146846, -1.10208315, -0.02865718,  2.40969018, -2.71535605]),
  'fval0': 28216.041116743356,
  'history': None,
  'exitflag': 0,
  'time': 0.017784595489501953,
  'message': b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'},
 {'id': '18',
  'x': array([0.99999998, 0.99999997, 1.00000004, 1.00000003, 1.00000008,
         1.00000005, 1.00000001, 1.00000002, 1.00000016, 1.00000028]),
  'fval': 4.7323128234188734e-12,
  'grad': array([-5.95491904e-06, -3.45232841e-05,  3.78255472e-05, -1.56002304e-05,
          5.19710685e-05,  9.03199095e-06, -1.61464965e-05, -4.44088779e-05,
          3.77395966e-05, -7.65640382e-06]),
  'hess': None,
  'res': None,
  'sres': None,
```

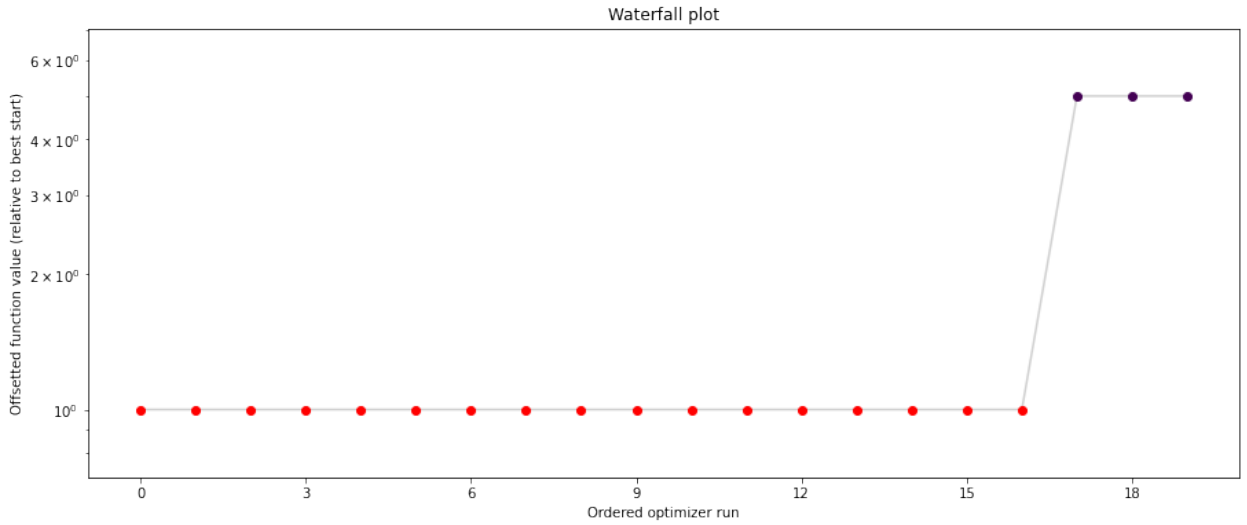
(continues on next page)

(continued from previous page)

```
  'n_fval': 83,
  'n_grad': 83,
  'n_hess': 0,
  'n_res': 0,
  'n_sres': 0,
  'x0': array([ 2.09368986, -1.92757011,  2.17342608,  0.94097953, -1.36443024,
         -0.4115728 ,  2.10149021,  1.64481536,  0.66254531,  0.15108917]),
  'fval0': 8262.214270452409,
  'history': None,
  'exitflag': 0,
  'time': 0.018738746643066406,
  'message': b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'}]
```

[13]:
```
# plot waterfalls
pypesto.visualize.waterfall(result, size=(15,6))
```

[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7faa1db4e950>
```



2.7 A sampler study

In this notebook, we perform a short study of how various samplers implemented in pyPESTO perform.

2.7.1 The pipeline

First, we show a typical workflow, fully integrating the samplers with a PEtab problem, using a toy example of a conversion reaction.

[1]:
```
import pypesto
import petab

# import to petab
petab_problem = petab.Problem.from_yaml(
    "conversion_reaction/conversion_reaction.yaml")
# import to pypesto
```

(continues on next page)

(continued from previous page)

```
importer = pypesto.PetabImporter(petab_problem)
# create problem
problem = importer.create_problem()
```

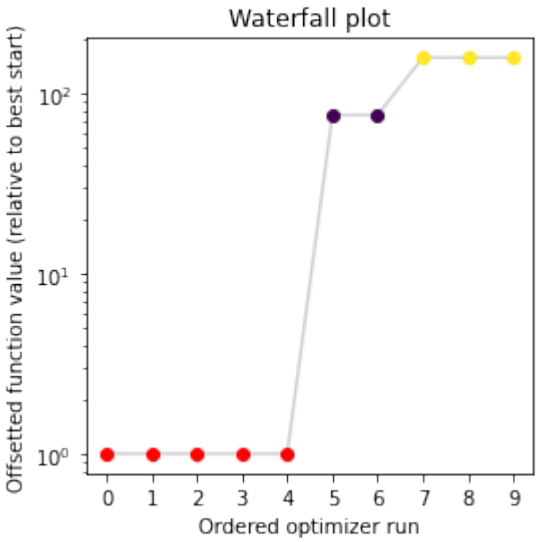
Commonly, as a first step, optimization is performed, in order to find good parameter point estimates.

[2]:
```
%%time
result = pypesto.minimize(problem, n_starts=10)
```

```
Parameters obtained from history and optimizer do not match: [-0.91620777 -9.1644571␣
↪], [-0.9160963  -9.16577932]
```

```
CPU times: user 937 ms, sys: 5.21 ms, total: 943 ms
Wall time: 943 ms
```

[3]:
```
ax = pypesto.visualize.waterfall(result, size=(4,4))
```



Next, we perform sampling. Here, we employ a `pypesto.sample.AdaptiveParallelTemperingSampler` sampler, which runs Markov Chain Monte Carlo (MCMC) chains on different temperatures. For each chain, we employ a `pypesto.sample.AdaptiveMetropolisSampler`. For more on the samplers see below or the API documentation.

[4]:
```
sampler = pypesto.AdaptiveParallelTemperingSampler(
    internal_sampler=pypesto.AdaptiveMetropolisSampler(),
    n_chains=3)
```

For the actual sampling, we call the `pypesto.sample` function. By passing the result object to the function, the previously found global optimum is used as starting point for the MCMC sampling.

[5]:
```
%%time
result = pypesto.sample(problem, n_samples=10000, sampler=sampler, result=result)
```

```
100%|| 10000/10000 [00:30<00:00, 330.85it/s]
```

```
CPU times: user 29.8 s, sys: 445 ms, total: 30.2 s
Wall time: 30.3 s
```

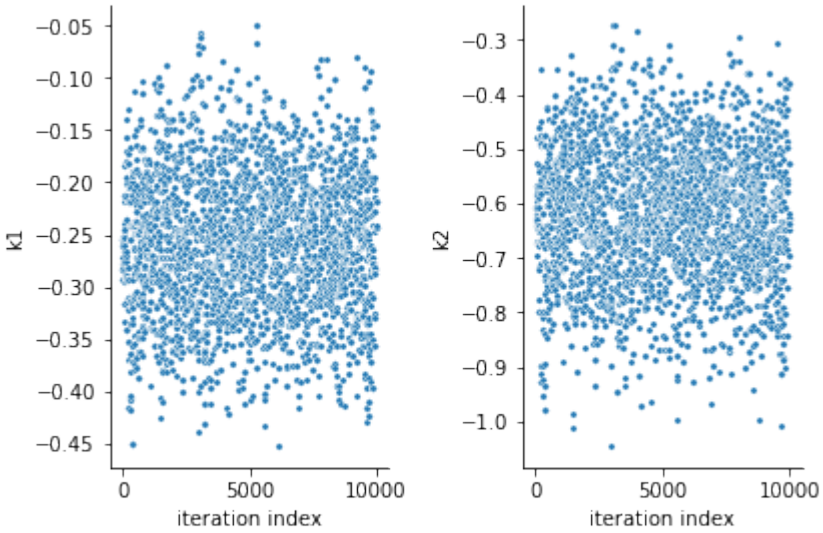
When the sampling is finished, we can analyse our results. A first thing to do is to analyze the sampling burn-in:

[6]:
```
pypesto.geweke_test(result)
```

[6]:
```
0
```

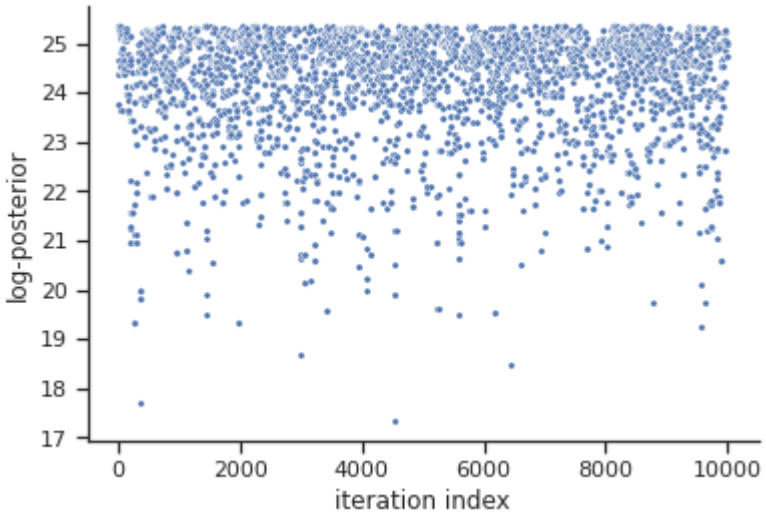
pyPESTO provides functions to analyse both the sampling process as well as the obtained sampling result. Visualizing the traces e.g. allows to detect burn-in phases, or fine-tune hyperparameters. First, the parameter trajectories can be visualized:

[7]:
```
pypesto.geweke_test(result)
ax = pypesto.visualize.sampling_parameters_trace(result, use_problem_bounds=False)
```



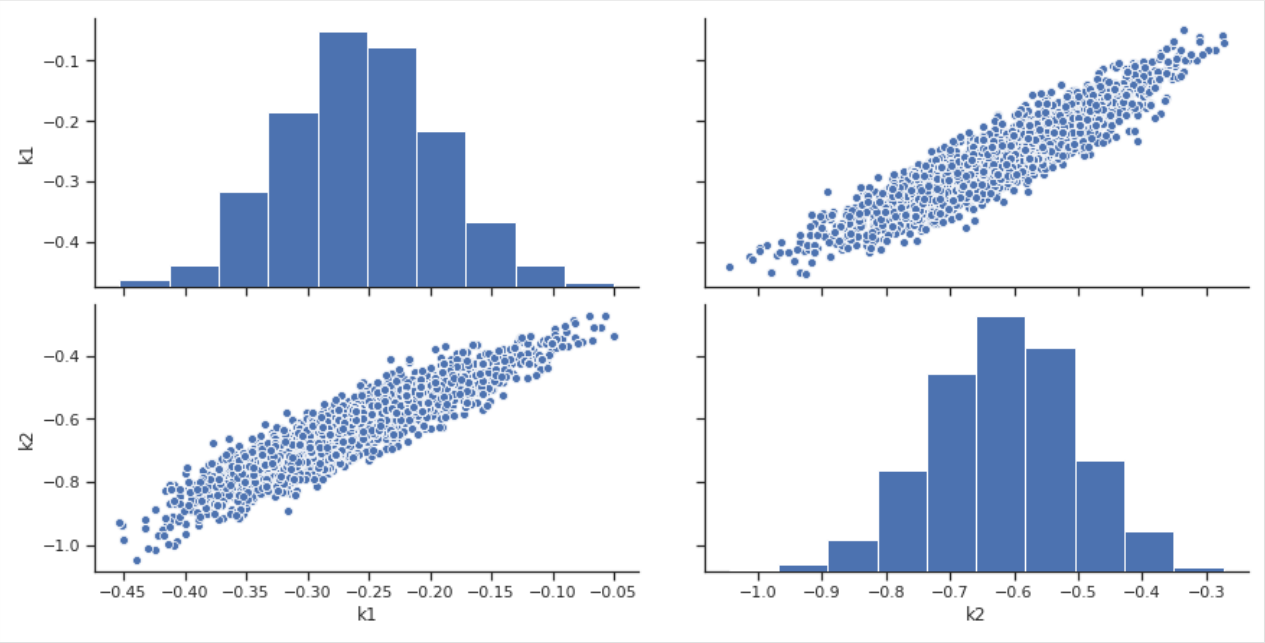
Next, also the log posterior trace can be visualized:

[8]:
```
ax = pypesto.visualize.sampling_fval_trace(result)
```



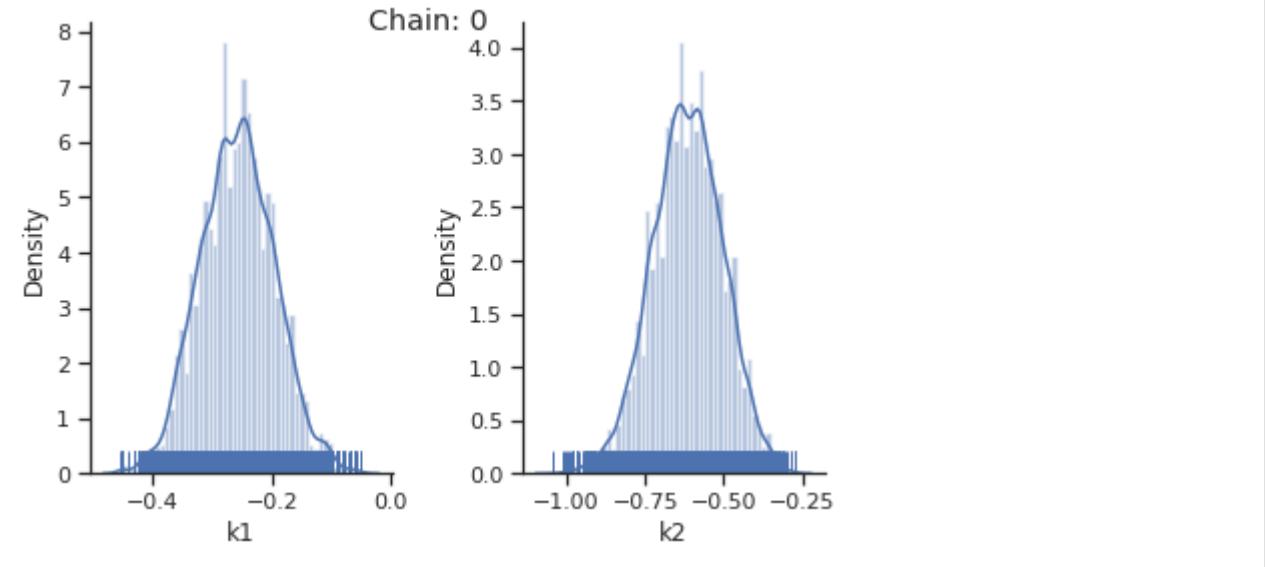
To visualize the result, there are various options. The scatter plot shows histograms of 1-dim parameter marginals and scatter plots of 2-dimensional parameter combinations:

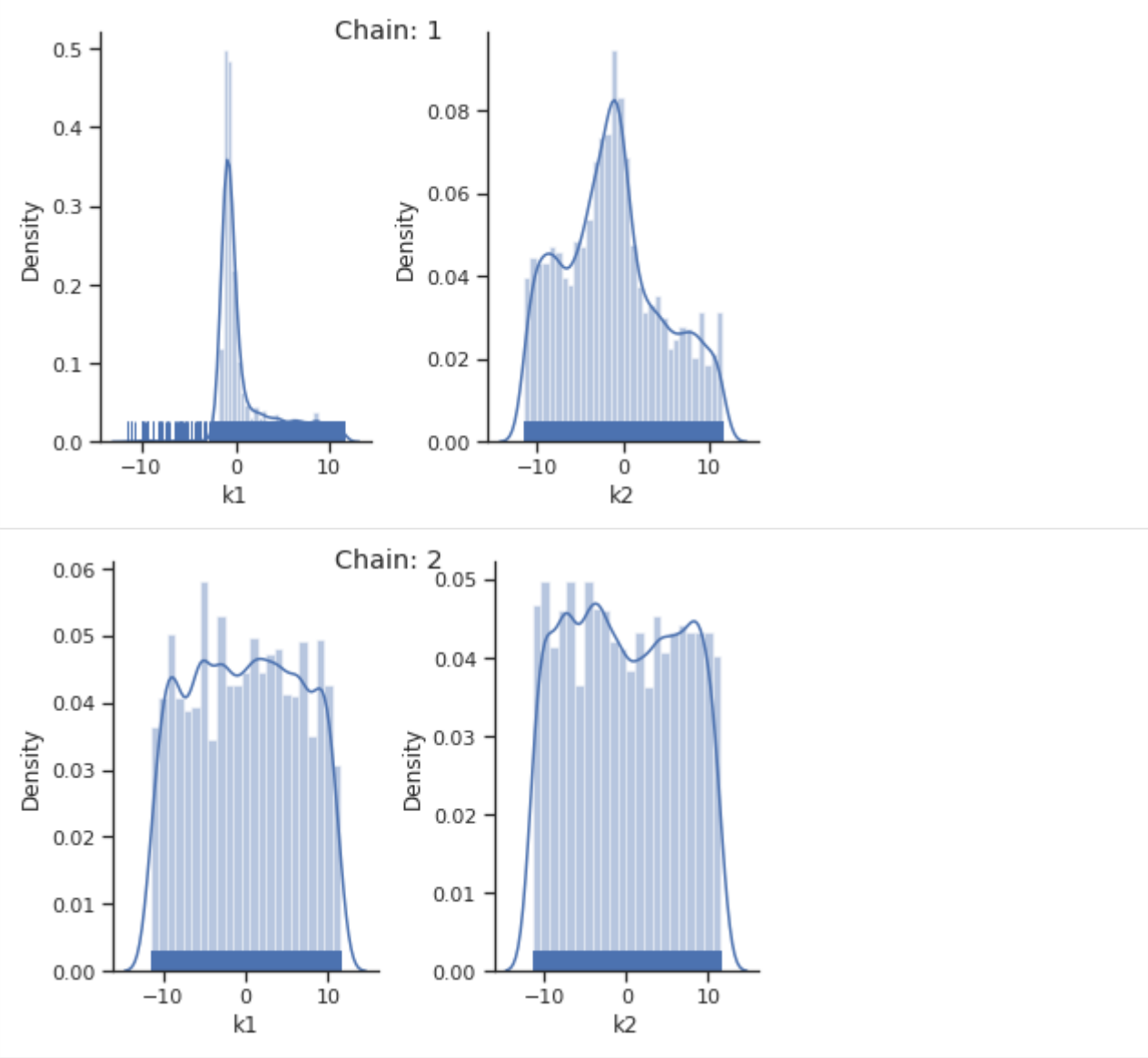
[9]:
```
ax = pypesto.visualize.sampling_scatter(result, size=[13,6])
```



`sampling_1d_marginals` allows to plot e.g. kernel density estimates or histograms (internally using seaborn):

[10]:
```
for i_chain in range(len(result.sample_result.betas)):
    pypesto.visualize.sampling_1d_marginals(
        result, i_chain=i_chain, suptitle=f"Chain: {i_chain}")
```





That's it for the moment on using the sampling pipeline.

2.7.2 1-dim test problem

To compare and test the various implemented samplers, we first study a 1-dimensional test problem of a gaussian mixture density, together with a flat prior.

[11]:
```
import numpy as np
from scipy.stats import multivariate_normal
import seaborn as sns
import pypesto

def density(x):
    return 0.3*multivariate_normal.pdf(x, mean=-1.5, cov=0.1) + \
        0.7*multivariate_normal.pdf(x, mean=2.5, cov=0.2)
```

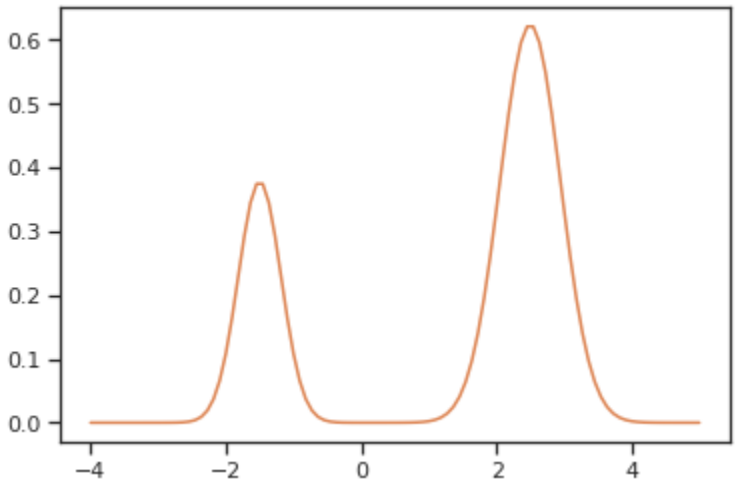
(continues on next page)

(continued from previous page)

```
def nllh(x):
    return - np.log(density(x))

objective = pypesto.Objective(fun=nllh)
problem = pypesto.Problem(
    objective=objective, lb=-4, ub=5, x_names=['x'])
```

The likelihood has two separate modes:

[12]:
```
xs = np.linspace(-4, 5, 100)
ys = [density(x) for x in xs]

ax = sns.lineplot(xs, ys, color='C1')
```



Metropolis sampler

For this problem, let us try out the simplest sampler, the `pypesto.sample.MetropolisSampler`.

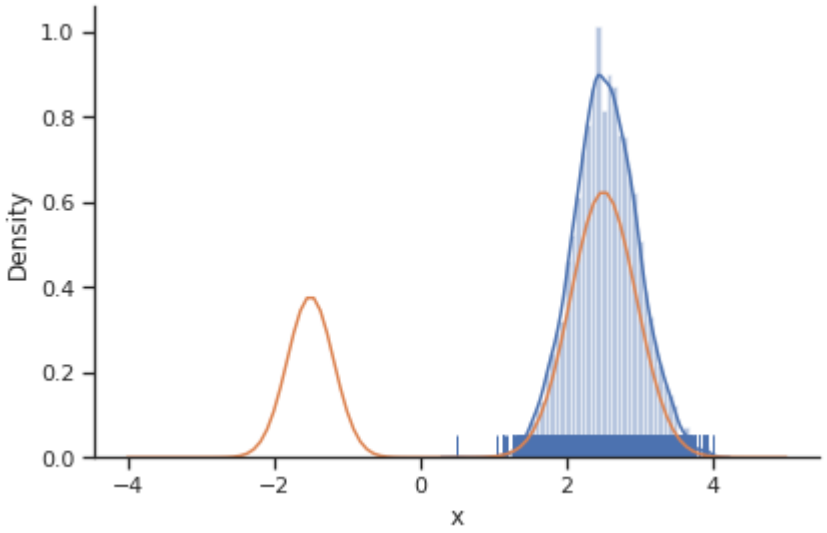
[13]:
```
%%time
sampler = pypesto.MetropolisSampler({'std': 0.5})
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
100%|| 10000/10000 [00:03<00:00, 2644.33it/s]
```

```
CPU times: user 3.75 s, sys: 97.4 ms, total: 3.85 s
Wall time: 3.8 s
```

[14]:
```
pypesto.geweke_test(result)
ax = pypesto.visualize.sampling_1d_marginals(result)
ax[0][0].plot(xs, ys)
```

[14]:
```
[<matplotlib.lines.Line2D at 0x7f7260a7baf0>]
```



The obtained posterior does not accurately represent the distribution, often only capturing one mode. This is because it is hard for the Markov chain to jump between the distribution's two modes. This can be fixed by choosing a higher proposal variation `std`:

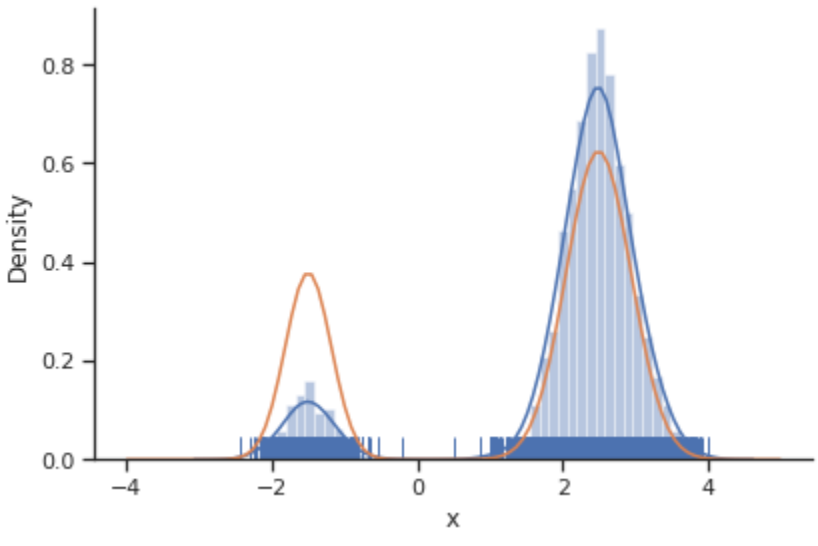
[15]:
```
%%time
sampler = pypesto.MetropolisSampler({'std': 1})
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
100%|| 10000/10000 [00:03<00:00, 3167.94it/s]
```

```
CPU times: user 3.23 s, sys: 113 ms, total: 3.34 s
Wall time: 3.17 s
```

[16]:
```
pypesto.geweke_test(result)
ax = pypesto.visualize.sampling_1d_marginals(result)
ax[0][0].plot(xs, ys)
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7f725abaec10>]
```



In general, MCMC have difficulties exploring multimodel landscapes. One way to overcome this is to used parallel

tempering. There, various chains are run, lifting the densities to different temperatures. At high temperatures, proposed steps are more likely to get accepted and thus jumps between modes more likely.

Parallel tempering sampler

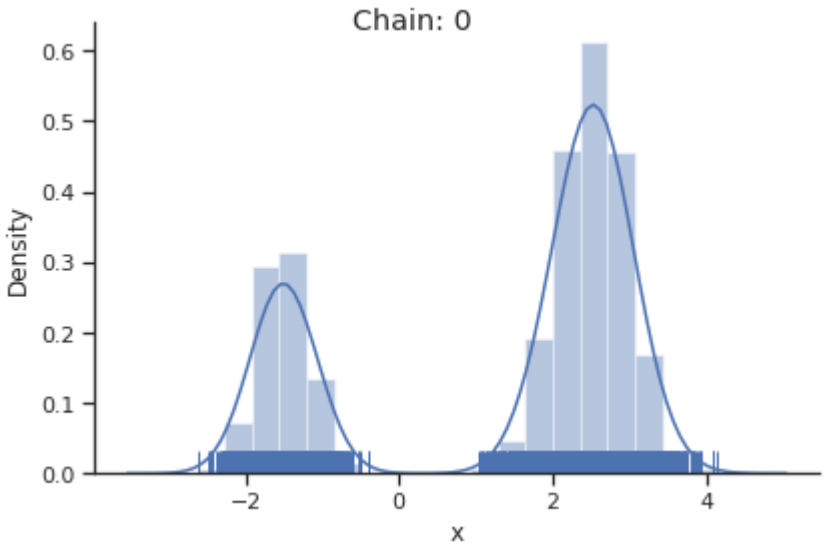
In pyPESTO, the most basic parallel tempering algorithm is the `pypesto.sample.ParallelTemperingSampler`. It takes an `internal_sampler` parameter, to specify what sampler to use for performing sampling the different chains. Further, we can directly specify what inverse temperatures `betas` to use. When not specifying the `betas` explicitly but just the number of chains `n_chains`, an established near-exponential decay scheme is used.

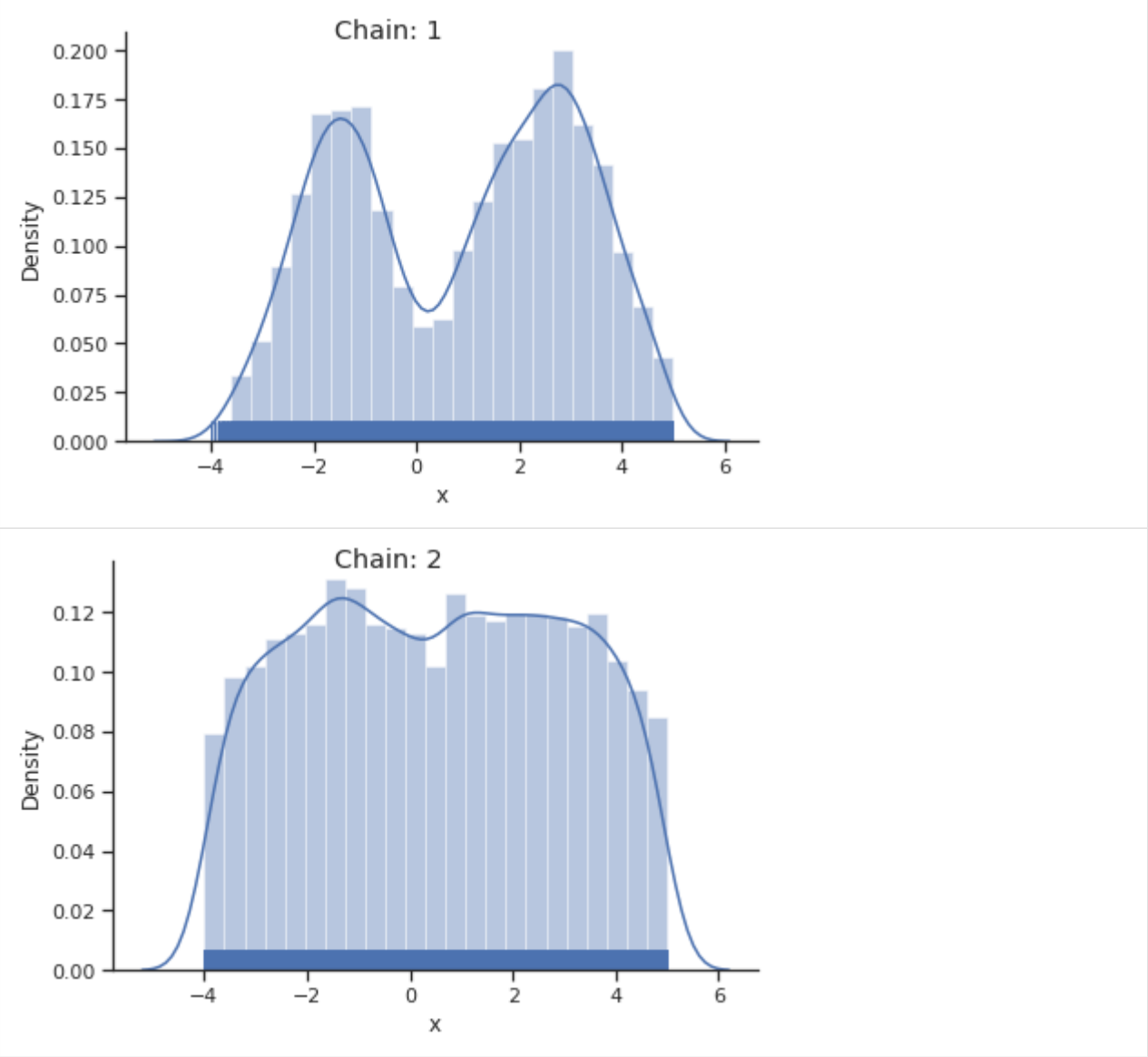
[17]:
```
%%time
sampler = pypesto.ParallelTemperingSampler(
    internal_sampler=pypesto.MetropolisSampler(),
    betas=[1, 1e-1, 1e-2])
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
100%|| 10000/10000 [00:11<00:00, 899.49it/s]
```

```
CPU times: user 11.2 s, sys: 473 ms, total: 11.7 s
Wall time: 11.1 s
```

[18]:
```
pypesto.geweke_test(result)
for i_chain in range(len(result.sample_result.betas)):
    pypesto.visualize.sampling_1d_marginals(
        result, i_chain=i_chain, suptitle=f"Chain: {i_chain}")
```





Of interest is here finally the first chain at index `i_chain=0`, which approximates the posterior well.

Adaptive Metropolis sampler

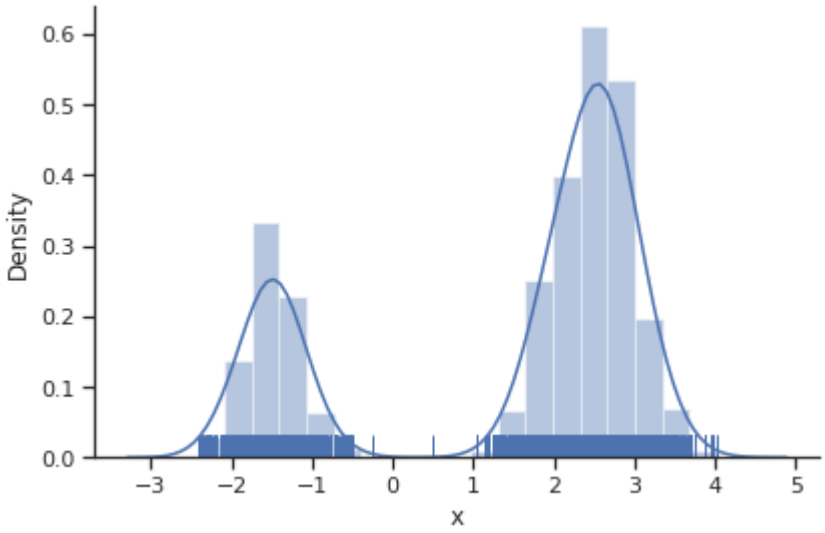
The problem of having to specify the proposal step variation manually can be overcome by using the `pypesto.sample.AdaptiveMetropolisSampler`, which iteratively adjusts the proposal steps to the function landscape.

[19]:
```
%%time
sampler = pypesto.AdaptiveMetropolisSampler()
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
100%|| 10000/10000 [00:04<00:00, 2292.14it/s]
```

```
CPU times: user 4.42 s, sys: 24 ms, total: 4.45 s
Wall time: 4.38 s
```

[20]:
```
pypesto.geweke_test(result)
ax = pypesto.visualize.sampling_1d_marginals(result)
```



Adaptive parallel tempering sampler

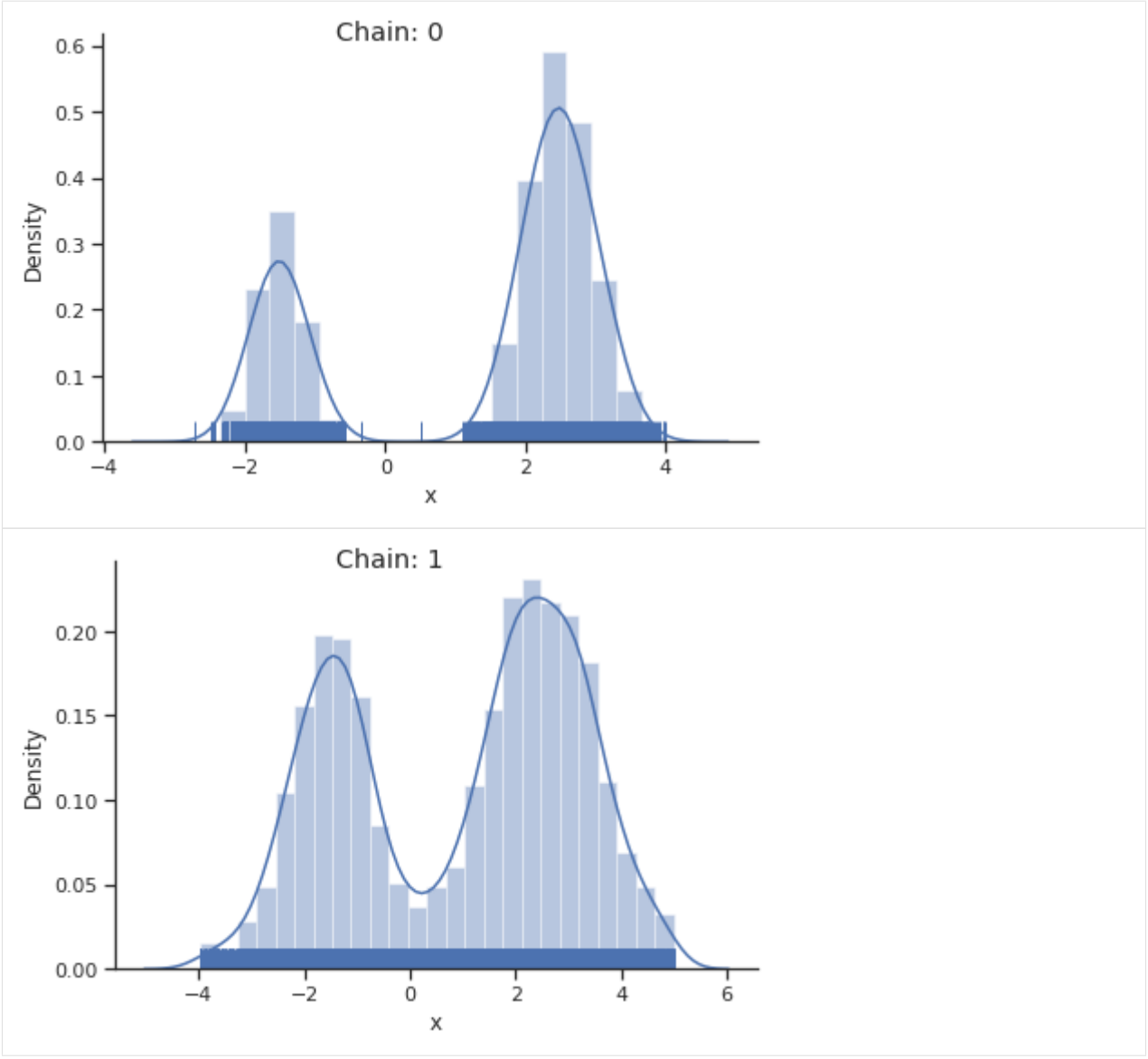
The `pypesto.sample.AdaptiveParallelTemperingSampler` iteratively adjusts the temperatures to obtain good swapping rates between chains.

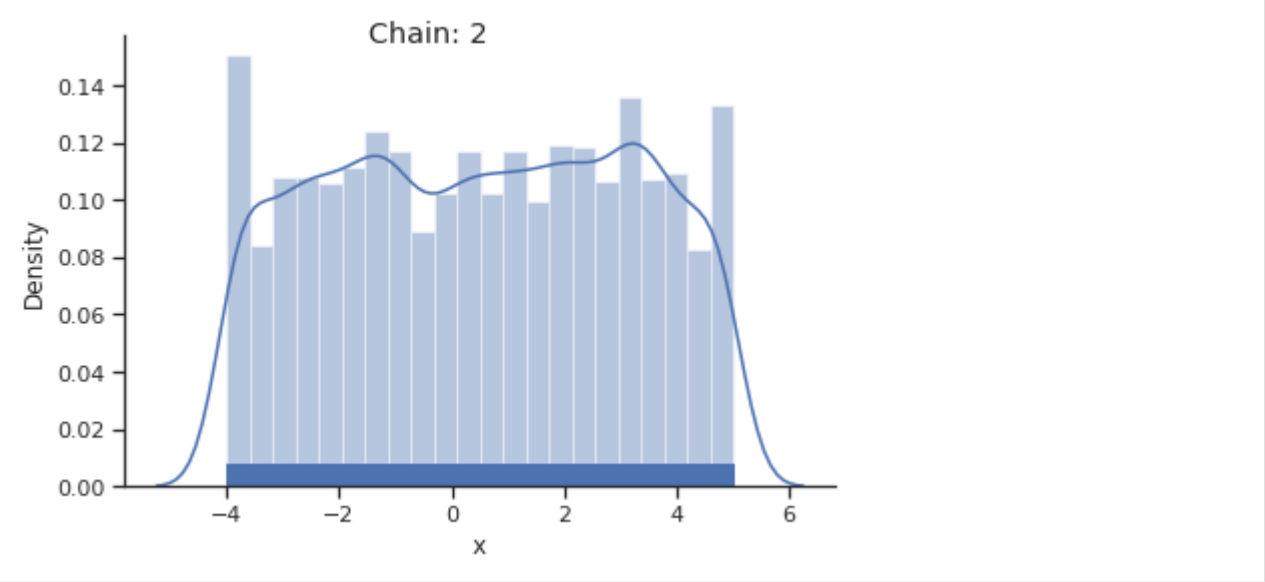
[21]:
```
%%time
sampler = pypesto.AdaptiveParallelTemperingSampler(
    internal_sampler=pypesto.AdaptiveMetropolisSampler(), n_chains=3)
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
100%|| 10000/10000 [00:12<00:00, 803.56it/s]
```

```
CPU times: user 12.5 s, sys: 151 ms, total: 12.7 s
Wall time: 12.5 s
```

[22]:
```
pypesto.geweke_test(result)
for i_chain in range(len(result.sample_result.betas)):
    pypesto.visualize.sampling_1d_marginals(
        result, i_chain=i_chain, suptitle=f"Chain: {i_chain}")
```


Chain: 0
Density
x
0.0
0.1
0.2
0.3
0.4
0.5
0.6
−4
−2
0
2
4
Chain: 1
Density
x
0.00
0.05
0.10
0.15
0.20
−4
−2
0
2
4
6



[23]:
```
result.sample_result.betas
```

[23]:
```
array([1.00000000e+00, 2.20932285e-01, 2.00000000e-05])
```

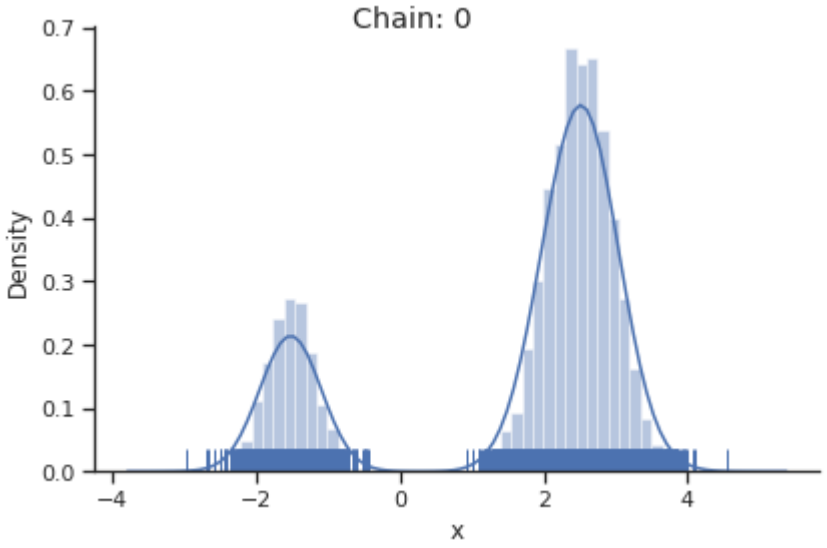
Pymc3 sampler

[24]:
```
%%time
sampler = pypesto.Pymc3Sampler()
result = pypesto.sample(problem, 1e4, sampler, x0=np.array([0.5]))
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Initializing NUTS failed. Falling back to elementwise auto-assignment.
Sequential sampling (1 chains in 1 job)
Slice: [x]
Sampling chain 0, 0 divergences: 100%|| 10500/10500 [00:24<00:00, 422.31it/s]
Only one chain was sampled, this makes it impossible to run some convergence checks
```

```
CPU times: user 28.2 s, sys: 1.1 s, total: 29.3 s
Wall time: 29.5 s
```

[25]:
```
pypesto.geweke_test(result)
for i_chain in range(len(result.sample_result.betas)):
    pypesto.visualize.sampling_1d_marginals(
        result, i_chain=i_chain, suptitle=f"Chain: {i_chain}")
```



If not specified, pymc3 chooses an adequate sampler automatically.

2.7.3 2-dim test problem: Rosenbrock banana

The adaptive parallel tempering sampler with chains running adaptive Metropolis samplers is also able to sample from more challenging posterior distributions. To illustrates this shortly, we use the Rosenbrock function.

[26]:
```
import scipy.optimize as so
import pypesto

# first type of objective
objective = pypesto.Objective(fun=so.rosen)

dim_full = 4
lb = -5 * np.ones((dim_full, 1))
ub = 5 * np.ones((dim_full, 1))

problem = pypesto.Problem(objective=objective, lb=lb, ub=ub)
```

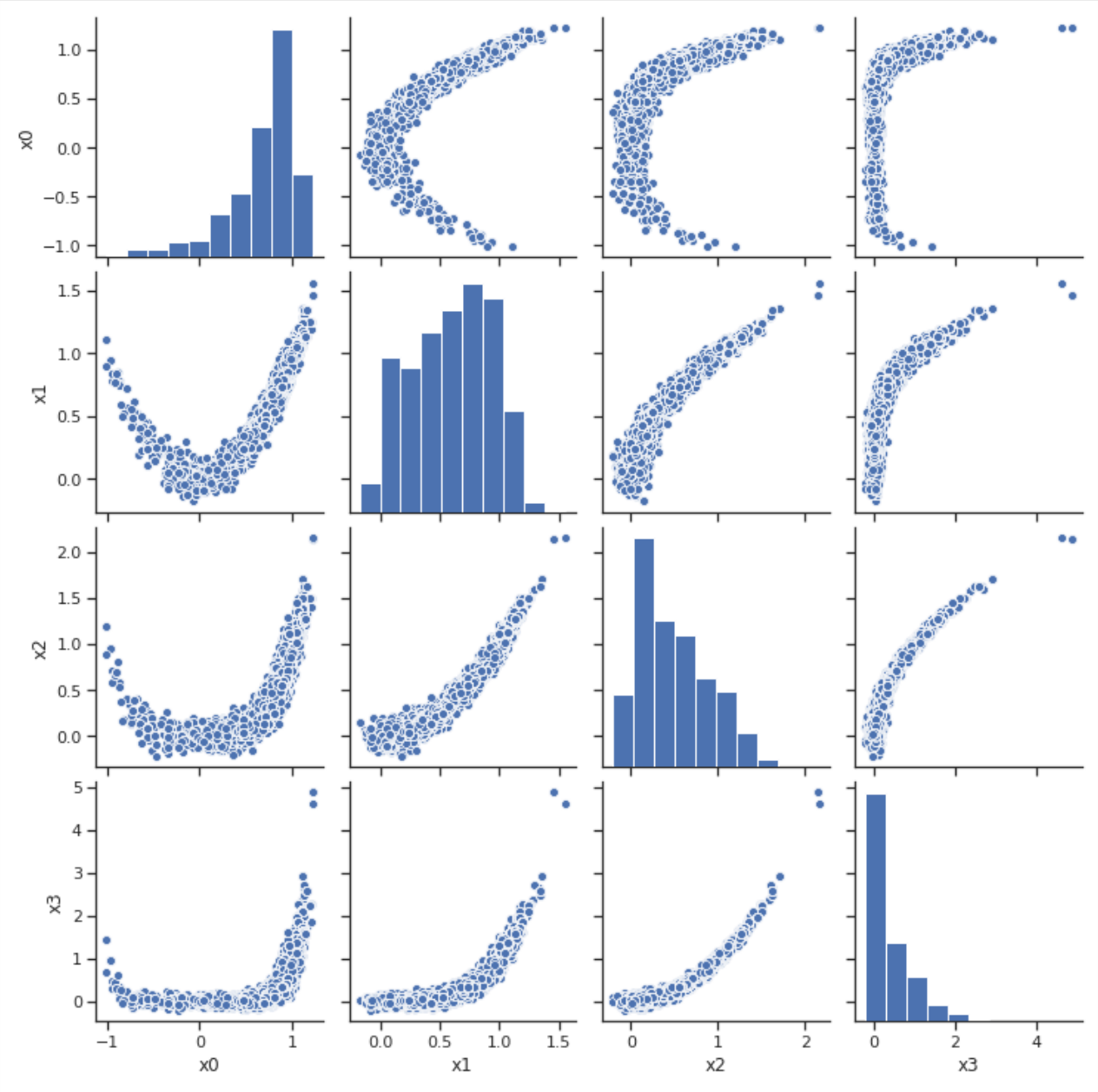
[27]:
```
%%time
sampler = pypesto.AdaptiveParallelTemperingSampler(
    internal_sampler=pypesto.AdaptiveMetropolisSampler(), n_chains=10)
result = pypesto.sample(problem, 1e4, sampler, x0=np.zeros(dim_full))
```

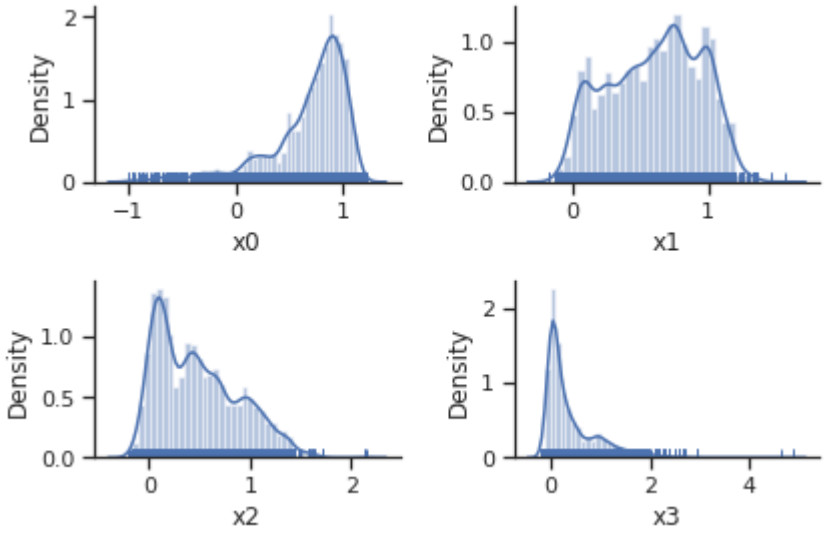
```
100%|| 10000/10000 [00:31<00:00, 316.47it/s]
```

```
CPU times: user 31.7 s, sys: 148 ms, total: 31.9 s
Wall time: 31.7 s
```

[28]:
```
ax = pypesto.visualize.sampling_scatter(result)
ax = pypesto.visualize.sampling_1d_marginals(result)
```

```
Burn in index not found in the results, the full chain will be shown.
You may want to use, e.g., 'pypesto.sampling.geweke_test'.
Burn in index not found in the results, the full chain will be shown.
You may want to use, e.g., 'pypesto.sampling.geweke_test'.
```


x0
x1
x2
x3
1.0
0.5
0.0
−0.5
−1.0
1.5
2.0
4
5
−1
0
1
0.0
0.5
1.0
1.5
2
3



[ ]:

2.8 MCMC sampling diagnostics

In this notebook, we illustrate how to assess the quality of your MCMC samples, e.g. convergence and auto-correlation, in pyPESTO.

2.8.1 The pipeline

First, we load the model and data to generate the MCMC samples from. In this example we show a toy example of a conversion reaction, loaded as a PEtab problem.

[1]:
```
import pypesto
import petab
import numpy as np
import logging

# log diagnostics
logger = logging.getLogger("pypesto.sampling.diagnostics")
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler())

# import to petab
petab_problem = petab.Problem.from_yaml(
    "conversion_reaction/conversion_reaction.yaml")
# import to pypesto
importer = pypesto.PetabImporter(petab_problem)
# create problem
problem = importer.create_problem()
```

Create the sampler object, in this case we will use adaptive parallel tempering with 3 temperatures.

[2]:
```
sampler = pypesto.AdaptiveParallelTemperingSampler(
    internal_sampler=pypesto.AdaptiveMetropolisSampler(),
    n_chains=3)
```

First, we will initiate the MCMC chain at a "random" point in parameter space, e.g. $\theta_{start} = [3, -4]$

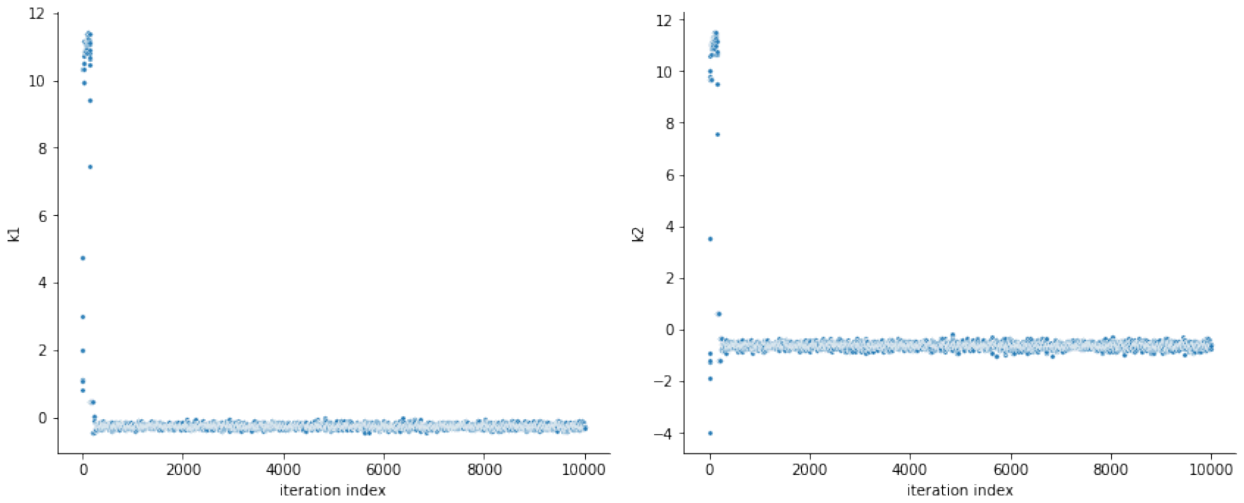
[3]:
```
result = pypesto.sample(problem, n_samples=10000, sampler=sampler, x0=np.array([3,-
↪4]))
elapsed_time = result.sample_result.time
print(f'Elapsed time: {round(elapsed_time,2)}')
```

```
100%|| 10000/10000 [00:28<00:00, 347.59it/s]
```

```
Elapsed time: 28.83
```

[4]:
```
ax = pypesto.visualize.sampling_parameters_trace(result, use_problem_bounds=False,␣
↪size=(12,5))
```

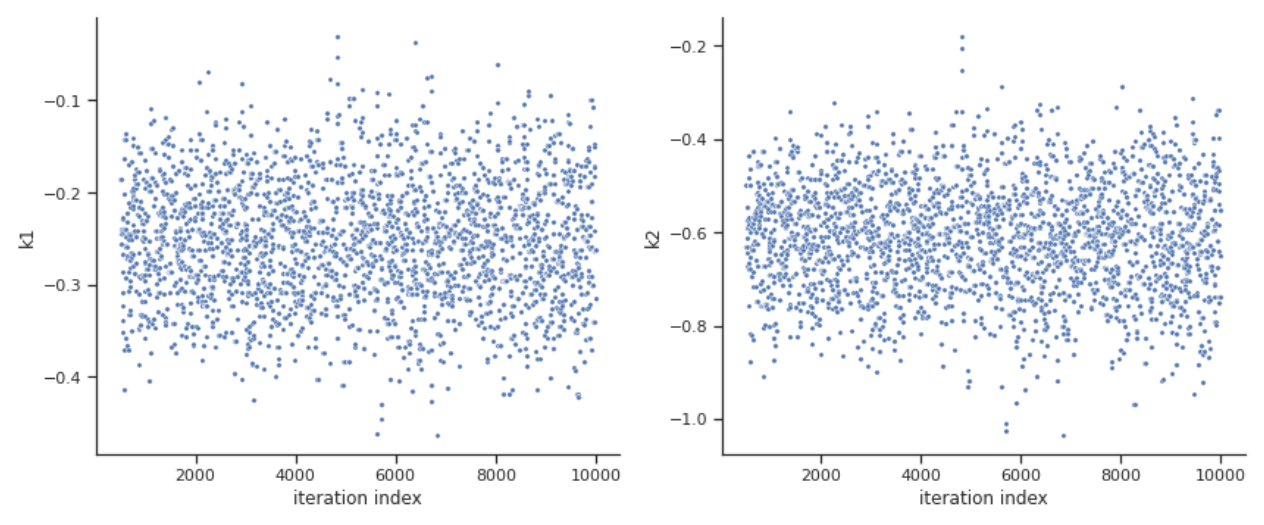
```
Burn in index not found in the results, the full chain will be shown.
You may want to use, e.g., 'pypesto.sampling.geweke_test'.
```



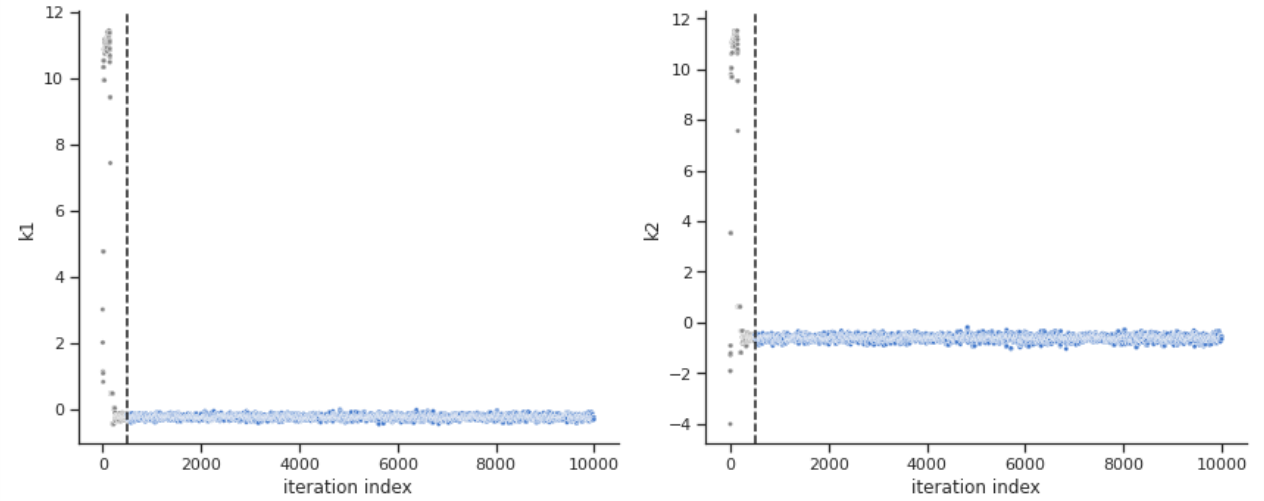
By visualizing the chains, we can see a warm up phase occurring until convergence of the chain is reached. This is commonly known as "burn in" phase and should be discarded. An automatic way to evaluate and find the index of the chain in which the warm up is finished can be done by using the Geweke test.

[5]:
```
pypesto.sampling.geweke_test(result=result)
ax = pypesto.visualize.sampling_parameters_trace(result, use_problem_bounds=False,␣
↪size=(12,5))
```

```
Geweke burn-in index: 500
```



[6]:
```
ax = pypesto.visualize.sampling_parameters_trace(result, use_problem_bounds=False,␣
↪full_trace=True, size=(12,5))
```



Commonly, as a first step, optimization is performed, in order to find good parameter point estimates.

[7]:
```
res = pypesto.minimize(problem, n_starts=10)
```

By passing the result object to the function, the previously found global optimum is used as starting point for the MCMC sampling.

[8]:
```
res = pypesto.sample(problem, n_samples=10000, sampler=sampler, result=res)
elapsed_time = res.sample_result.time
print('Elapsed time: '+str(round(elapsed_time,2)))
```

```
100%|| 10000/10000 [00:33<00:00, 299.21it/s]
```

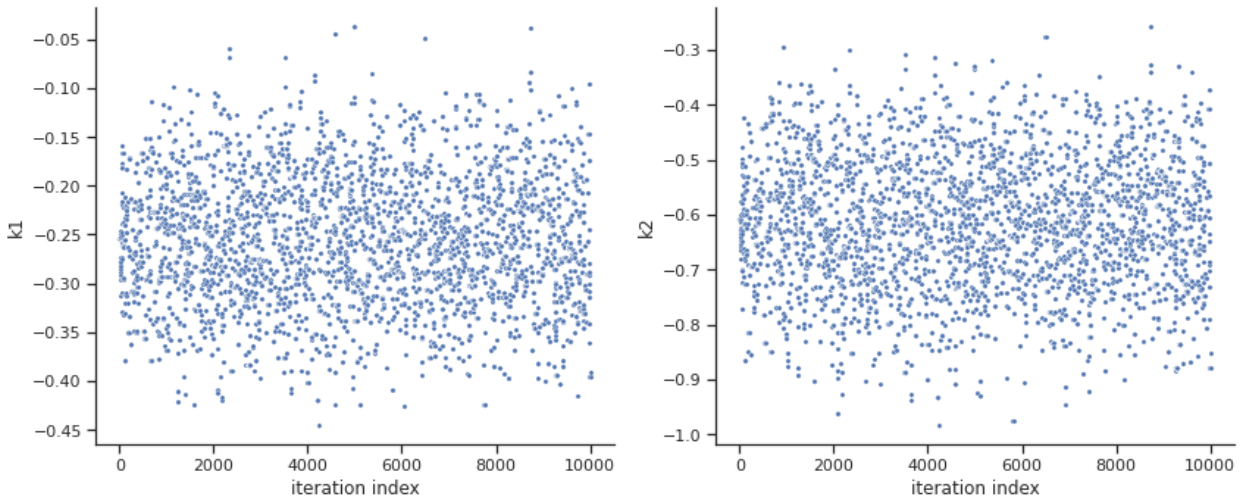
```
Elapsed time: 32.9
```

When the sampling is finished, we can analyse our results. pyPESTO provides functions to analyse both the sampling process as well as the obtained sampling result. Visualizing the traces e.g. allows to detect burn-in phases, or fine-tune

hyperparameters. First, the parameter trajectories can be visualized:

[9]:
```
ax = pypesto.visualize.sampling_parameters_trace(res, use_problem_bounds=False,␣
↪size=(12,5))
```

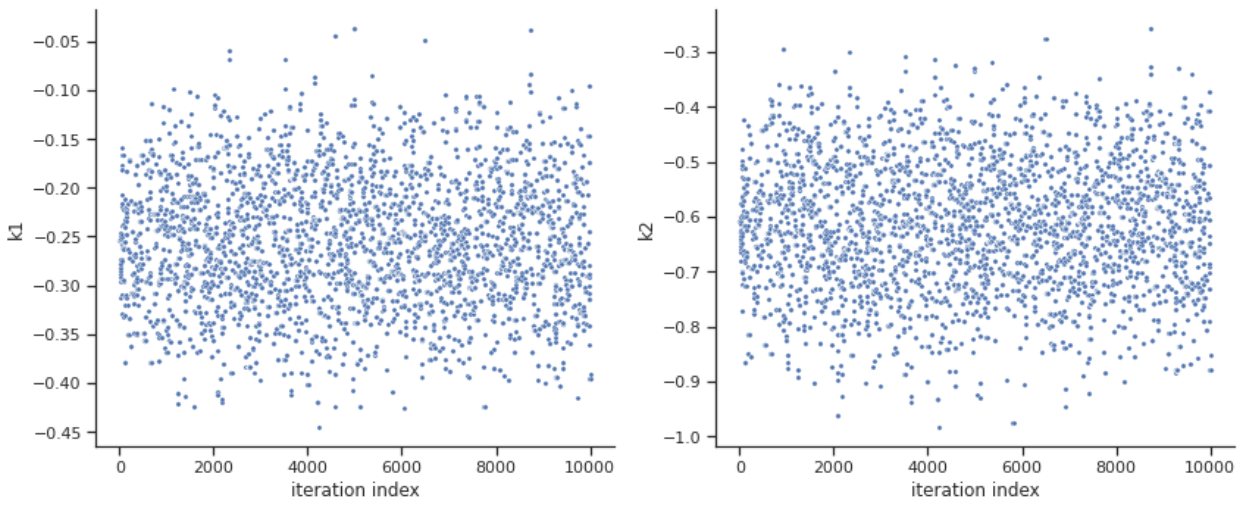
```
Burn in index not found in the results, the full chain will be shown.
You may want to use, e.g., 'pypesto.sampling.geweke_test'.
```



By visual inspection one can see that the chain is already converged from the start. This is already showing the benefit of initiating the chain at the optimal parameter vector. However, this may not be always the case.

[10]:
```
pypesto.sampling.geweke_test(result=res)
ax = pypesto.visualize.sampling_parameters_trace(res, use_problem_bounds=False,␣
↪size=(12,5))
```

```
Geweke burn-in index: 0
```



2.9 Download the examples as notebooks

- `Rosenbrock`
- `Conversion reaction`
- `Fixed parameters`
- `Boehm model`
- `Petab import`
- `Storage`
- `Sampler study`
- `Sampling diagnostics`

Note: Some of the notebooks have extra dependencies.

CHAPTER

THREE

STORAGE

It is important to be able to store analysis results efficiently, easily accessible, and portable across systems. For this aim, pyPESTO allows to store results in efficient, portable HDF5 files. Further, optimization trajectories can be stored using various backends, including HDF5 and CSV.

In the following, describe the file formats. For detailed information on usage, consult the `doc/example/hdf5_storage.ipynb` notebook, and the API documentation for the `pypesto.objective.history` and `pypesto.storage` modules.

3.1 pyPESTO Problem

```
+ /problem/
  - Attributes:
    - filled by objective.get_config()
    - ...

  - lb [float n_par]
  - ub [float n_par]
  - lb_full [float n_par_full]
  - ub_full [float n_par_full]
  - dim [int]
  - dim_full [int]
  - x_fixed_values [float (n_par_full-n_par)]
  - x_fixed_indices [int (n_par_full-n_par)]
  - x_free_indices [int n_par]
  - x_names [str n_par_full]
```

3.2 Parameter estimation

3.2.1 Parameter estimation settings

Parameter estimation settings are saved in `/optimization/settings`.

3.2.2 Parameter estimation results

Parameter estimation results are saved in `/optimization/results/`.

Results per local optimization

Results of the `$n`'th multistart a saved in the format

```
+ /optimization/results/$n/
  - fval: [float]
      Objective function value of best iteration
  - x: [float n_par_full]
      Parameter set of best iteration
  - grad: [float n_par_full]
      Gradient of objective function at point x
  - hess: [float n_par_full x n_par_full]
      Hessian matrix of objective function at point x
  - n_fval: [int]
      Total number of objective function evaluations
  - n_grad: [int]
      Number of gradient evaluations
  - n_hess: [int]
      Number of Hessian evaluations
  - x0: [float n_par_full]
      Initial parameter set
  - fval0: [float]
      Objective function value at starting parameters
  - exitflag: [str] Some exit flag
  - time: [float] Execution time
  - message: [str] Some exit message
```

Trace per local optimization

When objective function call histories are saved to HDF5, they are under `/optimization/results/$n/trace/`.

```
+ /optimization/results/$n/trace/
  - fval: [float n_iter]
      Objective function value of best iteration
  - x: [float n_iter x n_par_full]
      Parameter set of best iteration
  - grad: [float n_iter x n_par_full]
      Gradient of objective function at point x
  - hess: [float n_iter x n_par_full x n_par_full]
      Hessian matrix of objective function at point x
  - time: [float n_iter] Executition time
  - chi2: [float n_iter x ...]
  - schi2: [float n_iter x ...]
```

3.3 Sampling

3.3.1 Sampling results

Sampling results are saved in `/sampling/chains/`.

```
+ /sampling/chains/$n/
```

TODO

3.4 Profiling

TODO

3.4.1 Profiling results

TODO

CHAPTER

FOUR

CONTRIBUTE

4.1 Contribute documentation

To make pypesto easily usable, we are committed to documenting extensively. This involves in particular documenting the functionality of methods and classes, the purpose of single lines of code, and giving usage examples. The documentation is hosted on pypesto.readthedocs.io and updated automatically every time the master branch on github.com/icb-dcm/pypesto is updated. To compile the documentation locally, use:

```
cd doc
make html
```

4.2 Contribute tests

Tests are located in the `test` folder. All files starting with `test_` contain tests and are automatically run on Travis CI. To run them manually, type:

```
python3 -m pytest test
```

or alternatively:

```
python3 -m unittest test
```

You can also run specific tests.

Tests can be written with pytest or the unittest module.

4.2.1 PEP8

We try to respect the PEP8 coding standards. We run flake8 as part of the tests. If flake8 complains, the tests won't pass. You can run it via:

```
./run_flake8.sh
```

in Linux from the base directory, or directly from python. More, you can use the tool autopep8 to automatically fix various coding issues.

4.3 Contribute code

If you start working on a new feature or a fix, if not already done, please create an issue on github shortly describing your plans and assign it to yourself.

To get your code merged, please:

1. create a pull request to develop
2. if not already done in a commit message already, use the pull request description to reference and automatically close the respective issue (see https://help.github.com/articles/closing-issues-using-keywords/)
3. check that all tests on travis pass
4. check that the documentation is up-to-date
5. request a code review

General notes:

- Internally, we use `numpy` for arrays. In particular, vectors are represented as arrays of shape (n,).
- Use informative commmit messages.

CHAPTER

FIVE

DEPLOY

New features and bug fixes are continuously added to the develop branch. On every merge to master, the version number in `pypesto/version.py` should be incremented as described below.

5.1 Versioning scheme

For version numbers, we use `A.B.C`, where

- `C` is increased for bug fixes,
- `B` is increased for new features and minor API breaking changes,
- `A` is increased for major API breaking changes.

5.2 Creating a new release

After new commits have been added to the develop branch, changes can be merged to master and a new version of pyPESTO can be released. Every merge to master should coincide with an incremented version number and a git tag on the respective merge commit.

5.2.1 Merge into master

1. create a pull request from develop to master
2. check that all tests on travis pass
3. check that the documentation is up-to-date
4. adapt the version number in the file `pesto/version.py` (see above)
5. update the release notes in `doc/releasenotes.rst`
6. request a code review
7. merge into the origin master branch

To be able to actually perform the merge, sufficient rights may be required. Also, at least one review is required.

5.2.2 Creating a release on github

After merging into master, create a new release on Github. In the release form:

- specify a tag with the new version as specified in `pesto/version.py`, prefixed with `v` (e.g. `v0.0.1`)
- include the latest additions to `doc/releasenotes.rst` in the release description

Tagging the release commit will automatically trigger deployment of the new version to pypi.

CHAPTER

SIX

DOCUMENTATION

The `doc/` folder contains the files for building the pyPESTO documentation.

6.1 Requirements

The documentation is based on sphinx. Install via

```
pip3 install sphinx
```

Furthermore, the files specified in `../.rtd_pip_reqs.txt` and `../.rtd_apt_reqs.txt` are required. Install via

```
pip3 install --upgrade -r ../.rtd_pip_reqs.txt
```

and

```
cat ../.rtd_apt_reqs.txt | xargs sudo apt install -y
```

respectively.

6.2 Build the documentation

The documentation can be built in different formats, e.g. in html via

```
make html
```

The built documentation can then be found locally in the `_build` sub-directory.

The documentation is built and published automatically on readthedocs.io every time the master branch on github.com is changed. It is recommended to compile and check the documentation manually beforehand.

CHAPTER

SEVEN

OBJECTIVE

class pypesto.objective.**AggregatedObjective**(*objectives: Sequence[pypesto.objective.base.ObjectiveBase]*, *x_names: Sequence[str] = None*)

Bases: pypesto.objective.base.ObjectiveBase

This class aggregates multiple objectives into one objective.

__init__(*objectives: Sequence[pypesto.objective.base.ObjectiveBase]*, *x_names: Sequence[str] = None*)

Constructor.

Parameters

- **objectives** – Sequence of pypesto.ObjectiveBase instances
- **x_names** – Sequence of names of the (optimized) parameters. (Details see documentation of x_names in pypesto.ObjectiveBase)

call_unprocessed(*x*, *sensi_orders*, *mode*) → Dict[str, Union[float, numpy.ndarray, Dict]]

Call objective function without pre- or post-processing and formatting.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns A dict containing the results.

Return type result

check_mode(*mode*) → bool

Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

check_sensi_orders(*sensi_orders*, *mode*) → bool

Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

initialize()
Initialize the objective function. This function is used at the beginning of an analysis, e.g. optimization, and can e.g. reset the objective memory. By default does nothing.

class pypesto.objective.**AmiciCalculator**
Bases: object

Class to perform the actual call to AMICI and obtain requested objective function values.

__call__(*x_dct: Dict*, *sensi_order: int*, *mode: str*, *amici_model: Union[amici.Model, amici.ModelPtr]*, *amici_solver: Union[amici.Solver, amici.SolverPtr]*, *edatas: List[amici.ExpData]*, *n_threads: int*, *x_ids: Sequence[str]*, *parameter_mapping: ParameterMapping*)
Perform the actual AMICI call.

Called within the AmiciObjective.__call__() method.

Parameters

- **x_dct** – Parameters for which to compute function value and derivatives.
- **sensi_order** – Maximum sensitivity order.
- **mode** – Call mode (function value or residual based).
- **amici_model** – The AMICI model.
- **amici_solver** – The AMICI solver.
- **edatas** – The experimental data.
- **n_threads** – Number of threads for AMICI call.
- **x_ids** – Ids of optimization parameters.
- **parameter_mapping** – Mapping of optimization to simulation parameters.

__init__()
Initialize self. See help(type(self)) for accurate signature.

initialize()
Initialize the calculator. Default: Do nothing.

class pypesto.objective.**AmiciObjectBuilder**
Bases: abc.ABC

Allows to build AMICI model, solver, and edatas.

This class is useful for pickling an pypesto.AmiciObjective, which is required in some parallelization schemes. Therefore, this class itself must be picklable.

abstract create_edatas(*model: Union[amici.Model, amici.ModelPtr]*) → Sequence[amici.ExpData]
Create AMICI experimental data.

abstract create_model() → Union[amici.Model, amici.ModelPtr]
Create an AMICI model.

abstract create_solver(*model: Union[amici.Model, amici.ModelPtr]*) → Union[amici.Solver, amici.SolverPtr]
Create an AMICI solver.

class pypesto.objective.**AmiciObjective**(*amici_model: Union[amici.Model, amici.ModelPtr]*, *amici_solver: Union[amici.Solver, amici.SolverPtr]*, *edatas: Union[Sequence[amici.ExpData], amici.ExpData]*, *max_sensi_order: int = None*, *x_ids: Sequence[str] = None*, *x_names: Sequence[str] = None*, *parameter_mapping: ParameterMapping = None*, *guess_steadystate: bool = True*, *n_threads: int = 1*, *amici_object_builder: pypesto.objective.amici.AmiciObjectBuilder = None*, *calculator: pypesto.objective.amici_calculator.AmiciCalculator = None*)

Bases: pypesto.objective.base.ObjectiveBase

This class allows to create an objective directly from an amici model.

__init__(*amici_model: Union[amici.Model, amici.ModelPtr]*, *amici_solver: Union[amici.Solver, amici.SolverPtr]*, *edatas: Union[Sequence[amici.ExpData], amici.ExpData]*, *max_sensi_order: int = None*, *x_ids: Sequence[str] = None*, *x_names: Sequence[str] = None*, *parameter_mapping: ParameterMapping = None*, *guess_steadystate: bool = True*, *n_threads: int = 1*, *amici_object_builder: pypesto.objective.amici.AmiciObjectBuilder = None*, *calculator: pypesto.objective.amici_calculator.AmiciCalculator = None*)

Constructor.

Parameters

- **amici_model** – The amici model.
- **amici_solver** – The solver to use for the numeric integration of the model.
- **edatas** – The experimental data. If a list is passed, its entries correspond to multiple experimental conditions.
- **max_sensi_order** – Maximum sensitivity order supported by the model. Defaults to 2 if the model was compiled with o2mode, otherwise 1.
- **x_ids** – Ids of optimization parameters. In the simplest case, this will be the AMICI model parameters (default).
- **x_names** – Names of optimization parameters.
- **parameter_mapping** – Mapping of optimization parameters to model parameters. Format as created by *amici.petab_objective.create_parameter_mapping*. The default is just to assume that optimization and simulation parameters coincide.
- **guess_steadystate** – Whether to guess steadystates based on previous steadystates and respective derivatives. This option may lead to unexpected results for models with conservation laws and should accordingly be deactivated for those models.
- **n_threads** – Number of threads that are used for parallelization over experimental conditions. If amici was not installed with openMP support this option will have no effect.
- **amici_object_builder** – AMICI object builder. Allows recreating the objective for pickling, required in some parallelization schemes.
- **calculator** – Performs the actual calculation of the function values and derivatives.

apply_steadystate_guess(*condition_ix: int*, *x_dct: Dict*)

Use the stored steadystate as well as the respective sensitivity (if available) and parameter value to approximate the steadystate at the current parameters using a zeroth or first order taylor approximation: x_ss(x') = x_ss(x) [+ dx_ss/dx(x)*(x'-x)]

call_unprocessed(*x*, *sensi_orders*, *mode*)
Call objective function without pre- or post-processing and formatting.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns A dict containing the results.

Return type result

check_mode(*mode*)
Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

check_sensi_orders(*sensi_orders*, *mode*) → bool
Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

initialize()
Initialize the objective function. This function is used at the beginning of an analysis, e.g. optimization, and can e.g. reset the objective memory. By default does nothing.

par_arr_to_dct(*x: Sequence[float]*) → Dict[str, float]
Create dict from parameter vector.

reset_steadystate_guesses()
Resets all steadystate guess data

store_steadystate_guess(*condition_ix: int*, *x_dct: Dict*, *rdata: amici.ReturnData*)
Store condition parameter, steadystate and steadystate sensitivity in steadystate_guesses if steadystate guesses are enabled for this condition

class pypesto.objective.**CsvHistory**(*file: str*, *x_names: Sequence[str] = None*, *options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*, *load_from_file: bool = False*)
Bases: pypesto.objective.history.History

Stores a representation of the history in a CSV file.

Parameters

- **file** – CSV file name.
- **x_names** – Parameter names.
- **options** – History options.
- **load_from_file** – If True, history will be initialized from data in the specified file

__init__(*file: str, x_names: Sequence[str] = None, options: Union[pypesto.objective.history.HistoryOptions, Dict] = None, load_from_file: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

finalize()
Finalize history. Called after a run.

get_chi2_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Chi2 values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_fval_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Function values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_grad_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Gradients.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_hess_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Hessians.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_res_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Residuals.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_schi2_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Chi2 sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_sres_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Residual sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_time_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Cumulative execution times.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_x_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Parameters.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None
Update history after a function evaluation.

Parameters

- **x** – The parameter vector.
- **sensi_orders** – The sensitivity orders computed.
- **mode** – The objective function mode computed (function value or residuals).
- **result** – The objective function values for parameters *x*, sensitivities *sensi_orders* and mode *mode*.

class pypesto.objective.**Hdf5History**(*id: str*, *file: str*, *options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)
Bases: pypesto.objective.history.History

Stores a representation of the history in an HDF5 file.

Parameters

- **id** – Id of the history
- **file** – HDF5 file name.
- **options** – History options.

__init__(*id: str*, *file: str*, *options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)
Initialize self. See help(type(self)) for accurate signature.

finalize()
Finalize history. Called after a run.

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None
Update history after a function evaluation.

Parameters

- **x** – The parameter vector.
- **sensi_orders** – The sensitivity orders computed.
- **mode** – The objective function mode computed (function value or residuals).
- **result** – The objective function values for parameters *x*, sensitivities *sensi_orders* and mode *mode*.

class pypesto.objective.**History**(*options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)
Bases: pypesto.objective.history.HistoryBase

Tracks numbers of function evaluations only, no trace.

Parameters **options** – History options.

__init__(*options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)
Initialize self. See help(type(self)) for accurate signature.

finalize()
Finalize history. Called after a run.

property n_fval
Number of function evaluations.

property n_grad
Number of gradient evaluations.

property n_hess
Number of Hessian evaluations.

property n_res
Number of residual evaluations.

property n_sres
Number or residual sensitivity evaluations.

property start_time
Start time.

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None
Update history after a function evaluation.

Parameters

- **x** – The parameter vector.
- **sensi_orders** – The sensitivity orders computed.
- **mode** – The objective function mode computed (function value or residuals).
- **result** – The objective function values for parameters *x*, sensitivities *sensi_orders* and mode *mode*.

class pypesto.objective.**HistoryBase**
Bases: abc.ABC

Abstract base class for history objects.

Can be used as a dummy history, but does not implement any history functionality.

finalize()
Finalize history. Called after a run.

get_chi2_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[float], float]
Chi2 values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_fval_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[float], float]
Function values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_grad_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[Union[numpy.ndarray, np.nan]], numpy.ndarray, np.nan]
Gradients.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_hess_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[Union[numpy.ndarray, np.nan]], numpy.ndarray, np.nan]
Hessians.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_res_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[Union[numpy.ndarray, np.nan]], numpy.ndarray, np.nan]
Residuals.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_schi2_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[Union[numpy.ndarray, np.nan]], numpy.ndarray, np.nan]
Chi2 sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_sres_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[Union[numpy.ndarray, np.nan]], numpy.ndarray, np.nan]
Residual sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_time_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[float], float]
Cumulative execution times.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_x_trace(*ix: Optional[Union[int, Sequence[int]]] = None*) → Union[Sequence[numpy.ndarray], numpy.ndarray]
Parameters.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

property n_fval
Number of function evaluations.

property n_grad
Number of gradient evaluations.

property n_hess
Number of Hessian evaluations.

property n_res
Number of residual evaluations.

property n_sres
Number or residual sensitivity evaluations.

property start_time
Start time.

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None

Update history after a function evaluation.

Parameters

- **x** – The parameter vector.
- **sensi_orders** – The sensitivity orders computed.
- **mode** – The objective function mode computed (function value or residuals).
- **result** – The objective function values for parameters *x*, sensitivities *sensi_orders* and mode *mode*.

class pypesto.objective.**HistoryOptions**(*trace_record: bool = False*, *trace_record_grad: bool = True*, *trace_record_hess: bool = True*, *trace_record_res: bool = True*, *trace_record_sres: bool = True*, *trace_record_chi2: bool = True*, *trace_record_schi2: bool = True*, *trace_save_iter: int = 10*, *storage_file: str = None*)

Bases: dict

Options for the objective that are used in optimization, profiles and sampling.

In addition implements a factory pattern to generate history objects.

Parameters

- **trace_record** – Flag indicating whether to record the trace of function calls. The trace_record_* flags only become effective if trace_record is True. Default: False.
- **trace_record_grad** – Flag indicating whether to record the gradient in the trace. Default: True.
- **trace_record_hess** – Flag indicating whether to record the Hessian in the trace. Default: False.
- **trace_record_res** – Flag indicating whether to record the residual in the trace. Default: False.
- **trace_record_sres** – Flag indicating whether to record the residual sensitivities in the trace. Default: False.
- **trace_record_chi2** – Flag indicating whether to record the chi2 in the trace. Default: True.
- **trace_record_schi2** – Flag indicating whether to record the chi2 sensitivities in the trace. Default: True.
- **trace_save_iter** – After how many iterations to store the trace.
- **storage_file** – File to save the history to. Can be any of None, a "{filename}.csv", or a "{filename}.hdf5" file. Depending on the values, the *create_history* method creates the appropriate object. Occurrences of "{id}" in the file name are replaced by the *id* upon creation of a history, if applicable.

__init__(*trace_record: bool = False*, *trace_record_grad: bool = True*, *trace_record_hess: bool = True*, *trace_record_res: bool = True*, *trace_record_sres: bool = True*, *trace_record_chi2: bool = True*, *trace_record_schi2: bool = True*, *trace_save_iter: int = 10*, *storage_file: str = None*)

Initialize self. See help(type(self)) for accurate signature.

static assert_instance(*maybe_options: Union[HistoryOptions, Dict]*) → pypesto.objective.history.HistoryOptions

Returns a valid options object.

Parameters **maybe_options** (*HistoryOptions or dict*) –

create_history(*id: str*, *x_names: Sequence[str]*) → pypesto.objective.history.History

Factory method creating a *History* object.

Parameters

- **id** – Identifier for the history.
- **x_names** – Parameter names.

class pypesto.objective.**MemoryHistory**(*options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)

Bases: pypesto.objective.history.History

Tracks numbers of function evaluations and keeps an in-memory trace of function evaluations.

Parameters **options** – History options.

__init__(*options: Union[pypesto.objective.history.HistoryOptions, Dict] = None*)

Initialize self. See help(type(self)) for accurate signature.

get_chi2_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Chi2 values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_fval_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Function values.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_grad_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Gradients.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_hess_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Hessians.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_res_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Residuals.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_schi2_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]

Chi2 sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_sres_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Residual sensitivities.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_time_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Cumulative execution times.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

get_x_trace(*ix: Optional[Union[Sequence[int], int]] = None*) → Union[Sequence[Union[float, numpy.ndarray, np.nan]], float, numpy.ndarray, np.nan]
Parameters.

Takes as parameter an index or indices and returns corresponding trace values. If only a single value is requested, the list is flattened.

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None
Update history after a function evaluation.

Parameters

- **x** – The parameter vector.
- **sensi_orders** – The sensitivity orders computed.
- **mode** – The objective function mode computed (function value or residuals).
- **result** – The objective function values for parameters *x*, sensitivities *sensi_orders* and mode *mode*.

class pypesto.objective.**NegLogParameterPriors**(*prior_list: List[Dict]*, *x_names: Sequence[str] = None*)
Bases: pypesto.objective.base.ObjectiveBase

This class implements Negative Log Priors on Parameters.

Contains a list of prior dictionaries for the individual parameters of the format

{'index': [int], 'density_fun': [Callable], 'density_dx': [Callable], 'density_ddx': [Callable]}

A prior instance can be added to e.g. an objective, that gives the likelihood, by an AggregatedObjective.

Notes

All callables should correspond to log-densities. That is, they return log-densities and their corresponding derivatives. Internally, values are multiplied by -1, since pyPESTO expects the Objective function to be of a negative log-density type.

__init__(*prior_list: List[Dict]*, *x_names: Sequence[str] = None*)
Constructor

Parameters

- **prior_list** – List of dicts containing the individual parameter priors. Format see above.

- **x_names** – Sequence of parameter names (optional).

call_unprocessed(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*) → Dict[str, Union[float, numpy.ndarray, Dict]]

Call objective function without pre- or post-processing and formatting.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns A dict containing the results.

Return type result

check_mode(*mode*) → bool

Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

check_sensi_orders(*sensi_orders: Tuple[int, ...]*, *mode: str*) → bool

Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

gradient_neg_log_density(*x*)

Computes the gradient of the negative log-density for a parameter vector x.

hessian_neg_log_density(*x*)

Computes the hessian of the negative log-density for a parameter vector x.

hessian_vp_neg_log_density(*x*, *p*)

Computes the hessian vector product of the hessian of the negative log-density for a parameter vector x with a vector p.

neg_log_density(*x*)

Computes the negative log-density for a parameter vector x.

class pypesto.objective.**NegLogPriors**(*objectives: Sequence[pypesto.objective.base.ObjectiveBase]*, *x_names: Sequence[str] = None*)

Bases: pypesto.objective.aggregated.AggregatedObjective

Aggregates different forms of negative log-prior distributions.

Allows to distinguish priors from the likelihood by testing the type of an objective.

Consists basically of a list of individual negative log-priors, given in self.objectives.

class pypesto.objective.**Objective**(*fun: Callable = None*, *grad: Union[Callable, bool] = None*, *hess: Callable = None*, *hessp: Callable = None*, *res: Callable = None*, *sres: Union[Callable, bool] = None*, *x_names: Sequence[str] = None*)

Bases: pypesto.objective.base.ObjectiveBase

The objective class allows the user explicitely specify functions that compute the function value and/or residuals as well as respective derivatives.

Parameters

- **fun** – The objective function to be minimized. If it only computes the objective function value, it should be of the form

  ```
  fun(x) -> float
  ```

  where x is an 1-D array with shape (n,), and n is the parameter space dimension.

- **grad** – Method for computing the gradient vector. If it is a callable, it should be of the form

  ```
  grad(x) -> array_like, shape (n,).
  ```

  If its value is True, then fun should return the gradient as a second output.

- **hess** – Method for computing the Hessian matrix. If it is a callable, it should be of the form

  ```
  hess(x) -> array, shape (n,n).
  ```

  If its value is True, then fun should return the gradient as a second, and the Hessian as a third output, and grad should be True as well.

- **hessp** – Method for computing the Hessian vector product, i.e.

  ```
  hessp(x, v) -> array_like, shape (n,)
  ```

  computes the product H*v of the Hessian of fun at x with v.

- **res** – Method for computing residuals, i.e.

  ```
  res(x) -> array_like, shape(m,).
  ```

- **sres** – Method for computing residual sensitivities. If its is a callable, it should be of the form

  ```
  sres(x) -> array, shape (m,n).
  ```

  If its value is True, then res should return the residual sensitivities as a second output.

- **x_names** – Parameter names. None if no names provided, otherwise a list of str, length dim_full (as in the Problem class). Can be read by the problem.

__init__(*fun: Callable = None*, *grad: Union[Callable, bool] = None*, *hess: Callable = None*, *hessp: Callable = None*, *res: Callable = None*, *sres: Union[Callable, bool] = None*, *x_names: Sequence[str] = None*)

Initialize self. See help(type(self)) for accurate signature.

call_unprocessed(*x*, *sensi_orders*, *mode*)

Call objective function without pre- or post-processing and formatting.

Returns A dict containing the results.

Return type result

check_mode(*mode*)

Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

check_sensi_orders(*sensi_orders*, *mode*)
Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

property has_fun

property has_grad

property has_hess

property has_hessp

property has_res

property has_sres

class pypesto.objective.**ObjectiveBase**(*x_names: Sequence[str] = None*)
Bases: abc.ABC

The objective class is a simple wrapper around the objective function, giving a standardized way of calling. Apart from that, it manages several things including fixing of parameters and history.

The objective function is assumed to be in the format of a cost function, log-likelihood function, or log-posterior function. These functions are subject to minimization. For profiling and sampling, the sign is internally flipped, all returned and stored values are however given as returned by this objective function. If maximization is to be performed, the sign should be flipped before creating the objective function.

history
For storing the call history. Initialized by the methods, e.g. the optimizer, in *initialize_history()*.

pre_post_processor
Preprocess input values to and postprocess output values from __call__. Configured in *update_from_problem()*.

__call__(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...] = 0*, *mode: str = 'mode_fun'*, *return_dict: bool = False*) → Union[float, numpy.ndarray, Tuple, Dict[str, Union[float, numpy.ndarray, Dict]]]
Method to obtain arbitrary sensitivities. This is the central method which is always called, also by the get_* methods.

There are different ways in which an optimizer calls the objective function, and in how the objective function provides information (e.g. derivatives via separate functions or along with the function values). The different calling modes increase efficiency in space and time and make the objective flexible.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

- **return_dict** – If False (default), the result is a Tuple of the requested values in the requested order. Tuples of length one are flattened. If True, instead a dict is returned which can carry further information.

Returns By default, this is a tuple of the requested function values and derivatives in the requested order (if only 1 value, the tuple is flattened). If *return_dict*, then instead a dict is returned with function values and derivatives indicated by ids.

Return type result

__init__(*x_names: Sequence[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

abstract call_unprocessed(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*) → Dict[str, Union[float, numpy.ndarray, Dict]]
Call objective function without pre- or post-processing and formatting.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns A dict containing the results.

Return type result

check_grad(*x: numpy.ndarray*, *x_indices: Sequence[int] = None*, *eps: float = 1e-05*, *verbosity: int = 1*, *mode: str = 'mode_fun'*) → pandas.core.frame.DataFrame
Compare gradient evaluation: Firstly approximate via finite differences, and secondly use the objective gradient.

Parameters

- **x** – The parameters for which to evaluate the gradient.
- **x_indices** – Indices for which to compute gradients. Default: all.
- **eps** – Finite differences step size. Default: 1e-5.
- **verbosity** – Level of verbosity for function output. * 0: no output, * 1: summary for all parameters, * 2: summary for individual parameters. Default: 1.
- **mode** – Residual (MODE_RES) or objective function value (MODE_FUN, default) computation mode.

Returns gradient, finite difference approximations and error estimates.

Return type result

abstract check_mode(*mode*) → bool
Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

abstract check_sensi_orders(*sensi_orders*, *mode*) → bool
Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.

- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

get_fval(*x: numpy.ndarray*) → float
Get the function value at x.

get_grad(*x: numpy.ndarray*) → numpy.ndarray
Get the gradient at x.

get_hess(*x: numpy.ndarray*) → numpy.ndarray
Get the Hessian at x.

get_res(*x: numpy.ndarray*) → numpy.ndarray
Get the residuals at x.

get_sres(*x: numpy.ndarray*) → numpy.ndarray
Get the residual sensitivities at x.

property has_fun

property has_grad

property has_hess

property has_hessp

property has_res

property has_sres

initialize()
Initialize the objective function. This function is used at the beginning of an analysis, e.g. optimization, and can e.g. reset the objective memory. By default does nothing.

static output_to_tuple(*sensi_orders: Tuple[int, ...]*, *mode: str*, ***kwargs: Union[float, numpy.ndarray]*) → Tuple
Return values as requested by the caller, since usually only a subset is demanded. One output is returned as-is, more than one output are returned as a tuple in order (fval, grad, hess).

update_from_problem(*dim_full: int*, *x_free_indices: Sequence[int]*, *x_fixed_indices: Sequence[int]*, *x_fixed_vals: Sequence[float]*)
Handle fixed parameters. Later, the objective will be given parameter vectors x of dimension dim, which have to be filled up with fixed parameter values to form a vector of dimension dim_full >= dim. This vector is then used to compute function value and derivatives. The derivatives must later be reduced again to dimension dim.

This is so as to make the fixing of parameters transparent to the caller.

The methods preprocess, postprocess are overwritten for the above functionality, respectively.

Parameters

- **dim_full** – Dimension of the full vector including fixed parameters.
- **x_free_indices** – Vector containing the indices (zero-based) of free parameters (complimentary to x_fixed_indices).
- **x_fixed_indices** – Vector containing the indices (zero-based) of parameter components that are not to be optimized.
- **x_fixed_vals** – Vector of the same length as x_fixed_indices, containing the values of the fixed parameters.

class pypesto.objective.**OptimizerHistory**(*history: pypesto.objective.history.History*, *x0: numpy.ndarray*, *generate_from_history: bool = False*)

Bases: object

Objective call history. Container around a History object, which keeps track of optimal values.

fval0, fval_min
Initial and best function value found.

chi20, chi2_min
Initial and best chi2 value found.

x0, x_min
Initial and best parameters found.

grad_min
gradient for best parameters

hess_min
hessian (approximation) for best parameters

res_min
residuals for best parameters

sres_min
residual sensitivities for best parameters

Parameters

- **history** – History object to attach to this container. This history object implements the storage of the actual history.
- **x0** – Initial values for optimization
- **generate_from_history** – If set to true, this function will try to fill attributes of this function based on the provided history

__init__(*history: pypesto.objective.history.History*, *x0: numpy.ndarray*, *generate_from_history: bool = False*) → None
Initialize self. See help(type(self)) for accurate signature.

extract_from_history(*var*, *ix*)

finalize()

update(*x: numpy.ndarray*, *sensi_orders: Tuple[int]*, *mode: str*, *result: Dict[str, Union[float, numpy.ndarray]]*) → None
Update history and best found value.

pypesto.objective.**res_to_chi2**(*res: numpy.ndarray*)
We assume that the residuals res are related to an objective function value chi2 via:

```
chi2 = sum(res**2)
```

which is consistent with the AMICI definition but NOT the 'Linear' formulation in scipy.

pypesto.objective.**sres_to_schi2**(*res: numpy.ndarray*, *sres: numpy.ndarray*)
In line with the assumptions in res_to_chi2.

CHAPTER

EIGHT

PROBLEM

A problem contains the objective as well as all information like prior describing the problem to be solved.

pypesto.problem.**Iterable**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Iterable

pypesto.problem.**List**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of List

class pypesto.problem.**NegLogPriors**(*objectives: Sequence[pypesto.objective.base.ObjectiveBase]*, *x_names: Sequence[str] = None*)
Bases: pypesto.objective.aggregated.AggregatedObjective

Aggregates different forms of negative log-prior distributions.

Allows to distinguish priors from the likelihood by testing the type of an objective.

Consists basically of a list of individual negative log-priors, given in self.objectives.

class pypesto.problem.**ObjectiveBase**(*x_names: Sequence[str] = None*)
Bases: abc.ABC

The objective class is a simple wrapper around the objective function, giving a standardized way of calling. Apart from that, it manages several things including fixing of parameters and history.

The objective function is assumed to be in the format of a cost function, log-likelihood function, or log-posterior function. These functions are subject to minimization. For profiling and sampling, the sign is internally flipped, all returned and stored values are however given as returned by this objective function. If maximization is to be performed, the sign should be flipped before creating the objective function.

history
For storing the call history. Initialized by the methods, e.g. the optimizer, in *initialize_history()*.

pre_post_processor
Preprocess input values to and postprocess output values from __call__. Configured in *update_from_problem()*.

__call__(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...] = 0*, *mode: str = 'mode_fun'*, *return_dict: bool = False*) → Union[float, numpy.ndarray, Tuple, Dict[str, Union[float, numpy.ndarray, Dict]]]

Method to obtain arbitrary sensitivities. This is the central method which is always called, also by the get_* methods.

There are different ways in which an optimizer calls the objective function, and in how the objective function provides information (e.g. derivatives via separate functions or along with the function values). The different calling modes increase efficiency in space and time and make the objective flexible.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.
- **return_dict** – If False (default), the result is a Tuple of the requested values in the requested order. Tuples of length one are flattened. If True, instead a dict is returned which can carry further information.

Returns By default, this is a tuple of the requested function values and derivatives in the requested order (if only 1 value, the tuple is flattened). If *return_dict*, then instead a dict is returned with function values and derivatives indicated by ids.

Return type result

__init__(*x_names: Sequence[str] = None*)

Initialize self. See help(type(self)) for accurate signature.

abstract call_unprocessed(*x: numpy.ndarray*, *sensi_orders: Tuple[int, ...]*, *mode: str*) → Dict[str, Union[float, numpy.ndarray, Dict]]

Call objective function without pre- or post-processing and formatting.

Parameters

- **x** – The parameters for which to evaluate the objective function.
- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns A dict containing the results.

Return type result

check_grad(*x: numpy.ndarray*, *x_indices: Sequence[int] = None*, *eps: float = 1e-05*, *verbosity: int = 1*, *mode: str = 'mode_fun'*) → pandas.core.frame.DataFrame

Compare gradient evaluation: Firstly approximate via finite differences, and secondly use the objective gradient.

Parameters

- **x** – The parameters for which to evaluate the gradient.
- **x_indices** – Indices for which to compute gradients. Default: all.
- **eps** – Finite differences step size. Default: 1e-5.
- **verbosity** – Level of verbosity for function output. * 0: no output, * 1: summary for all parameters, * 2: summary for individual parameters. Default: 1.
- **mode** – Residual (MODE_RES) or objective function value (MODE_FUN, default) computation mode.

Returns gradient, finite difference approximations and error estimates.

Return type result

abstract check_mode(*mode*) → bool
Check if the objective is able to compute in the requested mode.

Parameters **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether mode is supported

Return type flag

abstract check_sensi_orders(*sensi_orders*, *mode*) → bool
Check if the objective is able to compute the requested sensitivities.

Parameters

- **sensi_orders** – Specifies which sensitivities to compute, e.g. (0,1) -> fval, grad.
- **mode** – Whether to compute function values or residuals.

Returns Boolean indicating whether combination of sensi_orders and mode is supported

Return type flag

get_fval(*x: numpy.ndarray*) → float
Get the function value at x.

get_grad(*x: numpy.ndarray*) → numpy.ndarray
Get the gradient at x.

get_hess(*x: numpy.ndarray*) → numpy.ndarray
Get the Hessian at x.

get_res(*x: numpy.ndarray*) → numpy.ndarray
Get the residuals at x.

get_sres(*x: numpy.ndarray*) → numpy.ndarray
Get the residual sensitivities at x.

property has_fun

property has_grad

property has_hess

property has_hessp

property has_res

property has_sres

initialize()
Initialize the objective function. This function is used at the beginning of an analysis, e.g. optimization, and can e.g. reset the objective memory. By default does nothing.

static output_to_tuple(*sensi_orders: Tuple[int, ...]*, *mode: str*, ***kwargs: Union[float, numpy.ndarray]*) → Tuple
Return values as requested by the caller, since usually only a subset is demanded. One output is returned as-is, more than one output are returned as a tuple in order (fval, grad, hess).

update_from_problem(*dim_full: int*, *x_free_indices: Sequence[int]*, *x_fixed_indices: Sequence[int]*, *x_fixed_vals: Sequence[float]*)
Handle fixed parameters. Later, the objective will be given parameter vectors x of dimension dim, which have to be filled up with fixed parameter values to form a vector of dimension dim_full >= dim. This

vector is then used to compute function value and derivatives. The derivatives must later be reduced again to dimension dim.

This is so as to make the fixing of parameters transparent to the caller.

The methods preprocess, postprocess are overwritten for the above functionality, respectively.

Parameters

- **dim_full** – Dimension of the full vector including fixed parameters.
- **x_free_indices** – Vector containing the indices (zero-based) of free parameters (complimentary to x_fixed_indices).
- **x_fixed_indices** – Vector containing the indices (zero-based) of parameter components that are not to be optimized.
- **x_fixed_vals** – Vector of the same length as x_fixed_indices, containing the values of the fixed parameters.

class pypesto.problem.**Problem**(*objective: pypesto.objective.base.ObjectiveBase*, *lb: Union[numpy.ndarray, List[float]]*, *ub: Union[numpy.ndarray, List[float]]*, *dim_full: Optional[int] = None*, *x_fixed_indices: Optional[Iterable[int]] = None*, *x_fixed_vals: Optional[Iterable[float]] = None*, *x_guesses: Optional[Iterable[float]] = None*, *x_names: Optional[Iterable[str]] = None*, *x_scales: Optional[Iterable[str]] = None*, *x_priors_defs: Optional[pypesto.objective.priors.NegLogPriors] = None*)

Bases: object

The problem formulation. A problem specifies the objective function, boundaries and constraints, parameter guesses as well as the parameters which are to be optimized.

Parameters

- **objective** – The objective function for minimization. Note that a shallow copy is created.
- **ub** (*lb,*) – The lower and upper bounds. For unbounded directions set to inf.
- **dim_full** – The full dimension of the problem, including fixed parameters.
- **x_fixed_indices** – Vector containing the indices (zero-based) of parameter components that are not to be optimized.
- **x_fixed_vals** – Vector of the same length as x_fixed_indices, containing the values of the fixed parameters.
- **x_guesses** – Guesses for the parameter values, shape (g, dim), where g denotes the number of guesses. These are used as start points in the optimization.
- **x_names** – Parameter names that can be optionally used e.g. in visualizations. If objective.get_x_names() is not None, those values are used, else the values specified here are used if not None, otherwise the variable names are set to ['x0', ... 'x{dim_full}']. The list must always be of length dim_full.
- **x_scales** – Parameter scales can be optionally given and are used e.g. in visualisation and prior generation. Currently the scales 'lin', 'log`and 'log10' are supported.
- **x_priors_defs** – Definitions of priors for parameters. Types of priors, and their required and optional parameters, are described in the *Prior* class.
- **dim** – The number of non-fixed parameters. Computed from the other values.
- **x_free_indices** (*array_like of int*) – Vector containing the indices (zero-based) of free parameters (complimentary to x_fixed_indices).

Notes

On the fixing of parameter values:

The number of parameters dim_full the objective takes as input must be known, so it must be either lb a vector of that size, or dim_full specified as a parameter.

All vectors are mapped to the reduced space of dimension dim in __init__, regardless of whether they were in dimension dim or dim_full before. If the full representation is needed, the methods get_full_vector() and get_full_matrix() can be used.

__init__(*objective: pypesto.objective.base.ObjectiveBase*, *lb: Union[numpy.ndarray, List[float]]*, *ub: Union[numpy.ndarray, List[float]]*, *dim_full: Optional[int] = None*, *x_fixed_indices: Optional[Iterable[int]] = None*, *x_fixed_vals: Optional[Iterable[float]] = None*, *x_guesses: Optional[Iterable[float]] = None*, *x_names: Optional[Iterable[str]] = None*, *x_scales: Optional[Iterable[str]] = None*, *x_priors_defs: Optional[pypesto.objective.priors.NegLogPriors] = None*)
Initialize self. See help(type(self)) for accurate signature.

property dim

fix_parameters(*parameter_indices: Union[Iterable[int], int]*, *parameter_vals: Union[Iterable[float], float]*) → None
Fix specified parameters to specified values

full_index_to_free_index(*full_index: int*)
Calculate index in reduced vector from index in full vector.

Parameters **full_index** (*The index in the full vector.*) –

Returns **free_index**

Return type The index in the free vector.

get_full_matrix(*x: Optional[numpy.ndarray]*) → Optional[numpy.ndarray]
Map matrix from dim to dim_full. Usually used for hessian.

Parameters **x** (*array_like, shape=(dim, dim)*) – The matrix in dimension dim.

get_full_vector(*x: Optional[numpy.ndarray]*, *x_fixed_vals: Iterable[float] = None*) → Optional[numpy.ndarray]
Map vector from dim to dim_full. Usually used for x, grad.

Parameters

- **x** (*array_like, shape=(dim,)*) – The vector in dimension dim.
- **x_fixed_vals** (*array_like, ndim=1, optional*) – The values to be used for the fixed indices. If None, then nans are inserted. Usually, None will be used for grad and problem.x_fixed_vals for x.

get_reduced_matrix(*x_full: Optional[numpy.ndarray]*) → Optional[numpy.ndarray]
Map matrix from dim_full to dim, i.e. delete fixed indices.

Parameters **x_full** (*array_like, ndim=2*) – The matrix in dimension dim_full.

get_reduced_vector(*x_full: Optional[numpy.ndarray]*) → Optional[numpy.ndarray]
Map vector from dim_full to dim, i.e. delete fixed indices.

Parameters **x_full** (*array_like, ndim=1*) – The vector in dimension dim_full.

property lb

normalize() → None
Reduce all vectors to dimension dim and have the objective accept vectors of dimension dim.

print_parameter_summary() → None
Prints a summary of what parameters are being optimized and parameter boundaries.

property ub

unfix_parameters(*parameter_indices: Union[Iterable[int], int]*) → None
Free specified parameters

property x_free_indices

property x_guesses

CHAPTER

NINE

PETAB

pyPESTO support for the PEtab data format.

class pypesto.petab.**PetabImporter**(*petab_problem: petab.Problem*, *output_folder: str = None*, *model_name: str = None*)

Bases: pypesto.objective.amici.AmiciObjectBuilder

MODEL_BASE_DIR = 'amici_models'

__init__(*petab_problem: petab.Problem*, *output_folder: str = None*, *model_name: str = None*)

petab_problem: Managing access to the model and data.

output_folder: Folder to contain the amici model. Defaults to './amici_models/{model_name}'.

model_name: Name of the model, which will in particular be the name of the compiled model python module.

compile_model(***kwargs*)

Compile the model. If the output folder exists already, it is first deleted.

Parameters **kwargs** (Extra arguments passed to *amici.SbmlImporter.sbml2amici*.) –

create_edatas(*model: amici.Model = None*, *simulation_conditions=None*) → List[amici.ExpData]

Create list of amici.ExpData objects.

create_model(*force_compile: bool = False*, ***kwargs*) → amici.Model

Import amici model. If necessary or force_compile is True, compile first.

Parameters

- **force_compile** – If False, the model is compiled only if the output folder does not exist yet. If True, the output folder is deleted and the model (re-)compiled in either case.

  Warning: If *force_compile*, then an existing folder of that name will be deleted.

- **kwargs** (*Extra arguments passed to amici.SbmlImporter.sbml2amici*) –

create_objective(*model: amici.Model = None*, *solver: amici.Solver = None*, *edatas: Sequence[amici.ExpData] = None*, *force_compile: bool = False*, ***kwargs*) → pypesto.objective.amici.AmiciObjective

Create a pypesto.AmiciObjective.

Parameters

- **model** – The AMICI model.
- **solver** – The AMICI solver.

- **edatas** – The experimental data in AMICI format.
- **force_compile** – Whether to force-compile the model if not passed.
- ****kwargs** – Additional arguments passed on to the objective.

Returns A `pypesto.AmiciObjective` for the model and the data.

Return type objective

create_prior() → pypesto.objective.priors.NegLogParameterPriors
Creates a prior from the parameter table. Returns None, if no priors are defined.

create_problem(*objective: pypesto.objective.amici.AmiciObjective = None*, ***kwargs*) → *pypesto.problem.Problem*
Create a `pypesto.Problem`.

Parameters

- **objective** – Objective as created by *create_objective*.
- ****kwargs** – Additional key word arguments passed on to the objective, if not provided.

Returns A `pypesto.Problem` for the objective.

Return type problem

create_solver(*model: amici.Model = None*) → amici.Solver
Return model solver.

static from_yaml(*yaml_config: Union[dict, str]*, *output_folder: str = None*, *model_name: str = None*) → pypesto.petab.importer.PetabImporter
Simplified constructor using a petab yaml file.

rdatas_to_measurement_df(*rdatas: Sequence[amici.ReturnData]*, *model: amici.Model = None*) → pandas.core.frame.DataFrame
Create a measurement dataframe in the petab format from the passed *rdatas* and own information.

Parameters

- **rdatas** – A list of rdatas as produced by pypesto.AmiciObjective.__call__(x, return_dict=True)['rdatas'].
- **model** – The amici model.

Returns A dataframe built from the rdatas in the format as in self.petab_problem.measurement_df.

Return type measurement_df

rdatas_to_simulation_df(*rdatas: Sequence[amici.ReturnData]*, *model: amici.Model = None*) → pandas.core.frame.DataFrame
Same as *rdatas_to_measurement_df*, execpt a petab simulation dataframe is created, i.e. the measurement column label is adjusted.

CHAPTER

TEN

OPTIMIZE

Multistart optimization with support for various optimizers.

class pypesto.optimize.**DlibOptimizer**(*method: str*, *options: Dict = None*)
Bases: pypesto.optimize.optimizer.Optimizer

Use the Dlib toolbox for optimization.

__init__(*method: str*, *options: Dict = None*)
Default constructor.

get_default_options()
Create default options specific for the optimizer.

is_least_squares()

minimize(*problem*, *x0*, *id*, *allow_failed_starts*, *history_options=None*)

class pypesto.optimize.**IpoptOptimizer**(*options: Dict = None*)
Bases: pypesto.optimize.optimizer.Optimizer

Use IpOpt (https://pypi.org/project/ipopt/) for optimization.

__init__(*options: Dict = None*)

Parameters **options** – Options are directly passed on to *ipopt.minimize_ipopt*.

is_least_squares()

minimize(*problem*, *x0*, *id*, *allow_failed_starts*, *history_options=None*)

class pypesto.optimize.**OptimizeOptions**(*startpoint_resample: bool = False*, *allow_failed_starts: bool = True*)
Bases: dict

Options for the multistart optimization.

Parameters

- **startpoint_resample** – Flag indicating whether initial points are supposed to be resampled if function evaluation fails at the initial point
- **allow_failed_starts** (*bool, optional*) – Flag indicating whether we tolerate that exceptions are thrown during the minimization process.

__init__(*startpoint_resample: bool = False*, *allow_failed_starts: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

static assert_instance(*maybe_options: Union[OptimizeOptions, Dict]*) → pypesto.optimize.options.OptimizeOptions
Returns a valid options object.

Parameters **maybe_options** (OptimizeOptions *or dict*) –

class pypesto.optimize.**Optimizer**

Bases: abc.ABC

This is the optimizer base class, not functional on its own.

An optimizer takes a problem, and possibly a start point, and then performs an optimization. It returns an OptimizerResult.

__init__()

Default constructor.

get_default_options()

Create default options specific for the optimizer.

abstract is_least_squares()

abstract minimize(*problem*, *x0*, *id*, *allow_failed_starts*, *history_options=None*)

class pypesto.optimize.**OptimizerResult**(*id: str = None*, *x: numpy.ndarray = None*, *fval: float = None*, *grad: numpy.ndarray = None*, *hess: numpy.ndarray = None*, *res: numpy.ndarray = None*, *sres: numpy.ndarray = None*, *n_fval: int = None*, *n_grad: int = None*, *n_hess: int = None*, *n_res: int = None*, *n_sres: int = None*, *x0: numpy.ndarray = None*, *fval0: float = None*, *history: pypesto.objective.history.History = None*, *exitflag: int = None*, *time: float = None*, *message: str = None*)

Bases: dict

The result of an optimizer run. Used as a standardized return value to map from the individual result objects returned by the employed optimizers to the format understood by pypesto.

Can be used like a dict.

id

Id of the optimizer run. Usually the start index.

x

The best found parameters.

fval

The best found function value, *fun(x)*.

grad

The gradient at *x*.

hess

The Hessian at *x*.

res

The residuals at *x*.

sres

The residual sensitivities at *x*.

n_fval

Number of function evaluations.

n_grad

Number of gradient evaluations.

n_hess
Number of Hessian evaluations.

n_res
Number of residuals evaluations.

n_sres
Number of residual sensitivity evaluations.

x0
The starting parameters.

fval0
The starting function value, *fun(x0)*.

history
Objective history.

exitflag
The exitflag of the optimizer.

time
Execution time.

message
Textual comment on the optimization result.

Type str

Notes

Any field not supported by the optimizer is filled with None.

__init__(*id: str = None*, *x: numpy.ndarray = None*, *fval: float = None*, *grad: numpy.ndarray = None*, *hess: numpy.ndarray = None*, *res: numpy.ndarray = None*, *sres: numpy.ndarray = None*, *n_fval: int = None*, *n_grad: int = None*, *n_hess: int = None*, *n_res: int = None*, *n_sres: int = None*, *x0: numpy.ndarray = None*, *fval0: float = None*, *history: pypesto.objective.history.History = None*, *exitflag: int = None*, *time: float = None*, *message: str = None*)
Initialize self. See help(type(self)) for accurate signature.

update_to_full(*problem:* pypesto.problem.Problem) → None
Updates values to full vectors/matrices

Parameters **problem** – problem which contains info about how to convert to full vectors or matrices

class pypesto.optimize.**PyswarmOptimizer**(*options: Dict = None*)
Bases: pypesto.optimize.optimizer.Optimizer

Global optimization using pyswarm.

__init__(*options: Dict = None*)
Default constructor.

is_least_squares()

minimize(*problem*, *x0*, *id*, *allow_failed_starts*, *history_options=None*)

class pypesto.optimize.**ScipyOptimizer**(*method: str = 'L-BFGS-B'*, *tol: float = 1e-09*, *options: Dict = None*)
Bases: pypesto.optimize.optimizer.Optimizer

Use the SciPy optimizers.

__init__(*method: str = 'L-BFGS-B'*, *tol: float = 1e-09*, *options: Dict = None*)
Default constructor.

get_default_options()
Create default options specific for the optimizer.

is_least_squares()

minimize(*problem*, *x0*, *id*, *allow_failed_starts*, *history_options=None*)

pypesto.optimize.**minimize**(*problem: pypesto.problem.Problem*, *optimizer: pypesto.optimize.optimizer.Optimizer = None*, *n_starts: int = 100*, *ids: Iterable[str] = None*, *startpoint_method: Union[Callable, bool] = None*, *result: pypesto.result.Result = None*, *engine: pypesto.engine.base.Engine = None*, *options: pypesto.optimize.options.OptimizeOptions = None*, *history_options: pypesto.objective.history.HistoryOptions = None*) → *pypesto.result.Result*

This is the main function to call to do multistart optimization.

Parameters

- **problem** – The problem to be solved.
- **optimizer** – The optimizer to be used n_starts times.
- **n_starts** – Number of starts of the optimizer.
- **ids** – Ids assigned to the startpoints.
- **startpoint_method** – Method for how to choose start points. False means the optimizer does not require start points, e.g. 'pso' method in 'GlobalOptimizer'
- **result** – A result object to append the optimization results to. For example, one might append more runs to a previous optimization. If None, a new object is created.
- **engine** – Parallelization engine. Defaults to sequential execution on a SingleCoreEngine.
- **options** – Various options applied to the multistart optimization.
- **history_options** – Optimizer history options.

Returns Result object containing the results of all multistarts in *result.optimize_result*.

Return type result

CHAPTER

ELEVEN

PROFILE

class pypesto.profile.**ProfileOptions**(*default_step_size: float = 0.01*, *min_step_size: float = 0.001*, *max_step_size: float = 1.0*, *step_size_factor: float = 1.25*, *delta_ratio_max: float = 0.1*, *ratio_min: float = 0.145*, *reg_points: int = 10*, *reg_order: int = 4*, *magic_factor_obj_value: float = 0.5*)

Bases: dict

Options for optimization based profiling.

Parameters

- **default_step_size** – Default step size of the profiling routine along the profile path (adaptive step lengths algorithms will only use this as a first guess and then refine the update).
- **min_step_size** – Lower bound for the step size in adaptive methods.
- **max_step_size** – Upper bound for the step size in adaptive methods.
- **step_size_factor** – Adaptive methods recompute the likelihood at the predicted point and try to find a good step length by a sort of line search algorithm. This factor controls step handling in this line search.
- **delta_ratio_max** – Maximum allowed drop of the posterior ratio between two profile steps.
- **ratio_min** – Lower bound for likelihood ratio of the profile, based on inverse chi2-distribution. The default 0.145 is slightly lower than the 95% quantile 0.1465 of a chi2 distribution with one degree of freedom.
- **reg_points** – Number of profile points used for regression in regression based adaptive profile points proposal.
- **reg_order** – Maximum degree of regression polynomial used in regression based adaptive profile points proposal.
- **magic_factor_obj_value** – There is this magic factor in the old profiling code which slows down profiling at small ratios (must be >= 0 and < 1).

__init__(*default_step_size: float = 0.01*, *min_step_size: float = 0.001*, *max_step_size: float = 1.0*, *step_size_factor: float = 1.25*, *delta_ratio_max: float = 0.1*, *ratio_min: float = 0.145*, *reg_points: int = 10*, *reg_order: int = 4*, *magic_factor_obj_value: float = 0.5*)

Initialize self. See help(type(self)) for accurate signature.

static create_instance(*maybe_options: Union[ProfileOptions, Dict]*) → pypesto.profile.options.ProfileOptions

Returns a valid options object.

Parameters **maybe_options** (*ProfileOptions or dict*) –

class pypesto.profile.**ProfilerResult**(*x_path: numpy.ndarray*, *fval_path: numpy.ndarray*, *ratio_path: numpy.ndarray*, *gradnorm_path: numpy.ndarray = None*, *exitflag_path: numpy.ndarray = None*, *time_path: numpy.ndarray = None*, *time_total: float = 0.0*, *n_fval: int = 0*, *n_grad: int = 0*, *n_hess: int = 0*, *message: str = None*)

Bases: dict

The result of a profiler run. The standardized return return value from pypesto.profile, which can either be initialized from an OptimizerResult or from an existing ProfilerResult (in order to extend the computation).

Can be used like a dict.

x_path
The path of the best found parameters along the profile (Dimension: n_par x n_profile_points)

fval_path
The function values, fun(x), along the profile.

ratio_path
The ratio of the posterior function along the profile.

gradnorm_path
The gradient norm along the profile.

exitflag_path
The exitflags of the optimizer along the profile.

time_path
The computation time of the optimizer runs along the profile.

time_total
The total computation time for the profile.

n_fval
Number of function evaluations.

n_grad
Number of gradient evaluations.

n_hess
Number of Hessian evaluations.

message
Textual comment on the profile result.

Notes

Any field not supported by the profiler or the profiling optimizer is filled with None. Some fields are filled by pypesto itself.

__init__(*x_path: numpy.ndarray*, *fval_path: numpy.ndarray*, *ratio_path: numpy.ndarray*, *gradnorm_path: numpy.ndarray = None*, *exitflag_path: numpy.ndarray = None*, *time_path: numpy.ndarray = None*, *time_total: float = 0.0*, *n_fval: int = 0*, *n_grad: int = 0*, *n_hess: int = 0*, *message: str = None*)
Initialize self. See help(type(self)) for accurate signature.

append_profile_point(*x: numpy.ndarray*, *fval: float*, *ratio: float*, *gradnorm: float = nan*, *time: float = nan*, *exitflag: float = nan*, *n_fval: int = 0*, *n_grad: int = 0*, *n_hess: int = 0*) → None
This function appends a new point to the profile path.

Parameters

- **x** – The parameter values.
- **fval** – The function value at *x*.
- **ratio** – The ratio of the function value at *x* by the optimal function value.
- **gradnorm** – The gradient norm at *x*.
- **time** – The computation time to find *x*.
- **exitflag** – The exitflag of the optimizer (useful if an optimization was performed to find *x*).
- **n_fval** – Number of function evaluations performed to find *x*.
- **n_grad** – Number of gradient evaluations performed to find *x*.
- **n_hess** – Number of Hessian evaluations performed to find *x*.

flip_profile() → None

This function flips the profiling direction (left-right) Profiling direction needs to be changed once (if the profile is new), or twice if we append to an existing profile.

All profiling paths are flipped in-place.

pypesto.profile.**approximate_parameter_profile**(*problem: pypesto.problem.Problem*, *result: pypesto.result.Result*, *profile_index: Iterable[int] = None*, *profile_list: int = None*, *result_index: int = 0*, *n_steps: int = 100*) → *pypesto.result.Result*

Calculate profiles based on an approximation via a normal likelihood centered at the chosen optimal parameter value, with the covariance matrix being the Hessian or FIM.

Parameters

- **problem** – The problem to be solved.
- **result** – A result object to initialize profiling and to append the profiling results to. For example, one might append more profiling runs to a previous profile, in order to merge these. The existence of an optimization result is obligatory.
- **profile_index** – Array with parameter indices, whether a profile should be computed (1) or not (0). Default is all profiles should be computed.
- **profile_list** – Integer which specifies whether a call to the profiler should create a new list of profiles (default) or should be added to a specific profile list.
- **result_index** – Index from which optimization result profiling should be started (default: global optimum, i.e., index = 0).
- **n_steps** – Number of profile steps in each dimension.

Returns The profile results are filled into *result.profile_result*.

Return type result

pypesto.profile.**calculate_approximate_ci**(*xs: numpy.ndarray*, *ratios: numpy.ndarray*, *confidence_ratio: float*) → Tuple[float, float]

Calculate approximate confidence interval based on profile. Interval bounds are linerly interpolated.

Parameters

- **xs** – The ordered parameter values along the profile for the coordinate of interest.
- **ratios** – The likelihood ratios corresponding to the parameter values.

- **confidence_ratio** – Minimum confidence ratio to base the confidence interval upon, as obtained via *pypesto.profile.chi2_quantile_to_ratio*.

Returns Bounds of the approximate confidence interval.

Return type lb, ub

pypesto.profile.**chi2_quantile_to_ratio**(*alpha: float = 0.95*, *df: int = 1*)

Transform lower tail probability *alpha* for a chi2 distribution with *df* degrees of freedom to a profile likelihood ratio threshold.

Parameters

- **alpha** – Lower tail probability, defaults to 95% interval.
- **df** – Degrees of freedom. Defaults to 1.

Returns Corresponds to a likelihood ratio.

Return type ratio

pypesto.profile.**parameter_profile**(*problem: pypesto.problem.Problem*, *result: pypesto.result.Result*, *optimizer: pypesto.optimize.optimizer.Optimizer*, *profile_index: numpy.ndarray = None*, *profile_list: int = None*, *result_index: int = 0*, *next_guess_method: Union[Callable, str] = 'adaptive_step_regression'*, *profile_options: pypesto.profile.options.ProfileOptions = None*) → *pypesto.result.Result*

This is the main function to call to do parameter profiling.

Parameters

- **problem** – The problem to be solved.
- **result** – A result object to initialize profiling and to append the profiling results to. For example, one might append more profiling runs to a previous profile, in order to merge these. The existence of an optimization result is obligatory.
- **optimizer** – The optimizer to be used along each profile.
- **profile_index** – Array with parameter indices, whether a profile should be computed (1) or not (0). Default is all profiles should be computed.
- **profile_list** – Integer which specifies whether a call to the profiler should create a new list of profiles (default) or should be added to a specific profile list.
- **result_index** – Index from which optimization result profiling should be started (default: global optimum, i.e., index = 0).
- **next_guess_method** – Function handle to a method that creates the next starting point for optimization in profiling.
- **profile_options** – Various options applied to the profile optimization.

Returns The profile results are filled into *result.profile_result*.

Return type result

CHAPTER

TWELVE

SAMPLING

Draw samples from the distribution, with support for various samplers.

class pypesto.sampling.**AdaptiveMetropolisSampler**(*options: Dict = None*)
Bases: pypesto.sampling.metropolis.MetropolisSampler

Metropolis-Hastings sampler with adaptive proposal covariance.

__init__(*options: Dict = None*)
Initialize self. See help(type(self)) for accurate signature.

classmethod default_options()
Convenience method to set/get default options.

Returns Default sampler options.

Return type default_options

initialize(*problem:* pypesto.problem.Problem, *x0: numpy.ndarray*)
Initialize the sampler.

Parameters

- **problem** – The problem for which to sample.
- **x0** – Should, but is not required to, be used as initial parameter.

class pypesto.sampling.**AdaptiveParallelTemperingSampler**(*internal_sampler: pypesto.sampling.sampler.InternalSampler*, *betas: Sequence[float] = None*, *n_chains: int = None*, *options: Dict = None*)
Bases: pypesto.sampling.parallel_tempering.ParallelTemperingSampler

Parallel tempering sampler with adaptive temperature adaptation.

adjust_betas(*i_sample: int*, *swapped: Sequence[bool]*)
Update temperatures as in Vousden2016.

classmethod default_options() → Dict
Convenience method to set/get default options.

Returns Default sampler options.

Return type default_options

class pypesto.sampling.**InternalSampler**(*options: Dict = None*)
Bases: pypesto.sampling.sampler.Sampler

Sampler to be used inside a parallel tempering sampler.

The last sample can be obtained via *get_last_sample* and set via *set_last_sample*.

abstract get_last_sample() → pypesto.sampling.sampler.InternalSample
Get the last sample in the chain.

Returns The last sample in the chain in the exchange format.

Return type internal_sample

make_internal(*temper_lpost: bool*)
This function can be called by parallel tempering samplers during initialization to allow the inner samplers to adjust to them being used as inner samplers. Default: Do nothing.

Parameters **temper_lpost** – Whether to temperate the posterior or only the likelihood.

abstract set_last_sample(*sample: pypesto.sampling.sampler.InternalSample*)
Set the last sample in the chain to the passed value.

Parameters **sample** – The sample that will replace the last sample in the chain.

class pypesto.sampling.**McmcPtResult**(*trace_x: numpy.ndarray*, *trace_neglogpost: numpy.ndarray*, *trace_neglogprior: numpy.ndarray*, *betas: Iterable[float]*, *burn_in: int = None*, *time: float = 0.0*, *message: str = None*)
Bases: dict

The result of a sampler run using Markov-chain Monte Carlo, and optionally parallel tempering.

Can be used like a dict.

Parameters

- **trace_x** (*[n_chain, n_iter, n_par]*) – Parameters.
- **trace_neglogpost** (*[n_chain, n_iter]*) – Negative log posterior values.
- **trace_neglogprior** (*[n_chain, n_iter]*) – Negative log prior values.
- **betas** (*[n_chain]*) – The associated inverse temperatures.
- **burn_in** (*[n_chain]*) – The burn in index.
- **message** (*str*) – Textual comment on the profile result.
- **n_chain denotes the number of chains, n_iter the number of** (*Here,*) –
- **(i.e., the chain length), and n_par the number of parameters.** (*iterations*) –

__init__(*trace_x: numpy.ndarray*, *trace_neglogpost: numpy.ndarray*, *trace_neglogprior: numpy.ndarray*, *betas: Iterable[float]*, *burn_in: int = None*, *time: float = 0.0*, *message: str = None*)
Initialize self. See help(type(self)) for accurate signature.

class pypesto.sampling.**MetropolisSampler**(*options: Dict = None*)
Bases: pypesto.sampling.sampler.InternalSampler

Simple Metropolis-Hastings sampler with fixed proposal variance.

__init__(*options: Dict = None*)
Initialize self. See help(type(self)) for accurate signature.

classmethod default_options()
Convenience method to set/get default options.

Returns Default sampler options.

Return type default_options

get_last_sample() → pypesto.sampling.sampler.InternalSample
Get the last sample in the chain.

Returns The last sample in the chain in the exchange format.

Return type internal_sample

get_samples() → pypesto.sampling.result.McmcPtResult
Get the generated samples.

initialize(*problem:* pypesto.problem.Problem, *x0: numpy.ndarray*)
Initialize the sampler.

Parameters

- **problem** – The problem for which to sample.
- **x0** – Should, but is not required to, be used as initial parameter.

make_internal(*temper_lpost: bool*)
This function can be called by parallel tempering samplers during initialization to allow the inner samplers to adjust to them being used as inner samplers. Default: Do nothing.

Parameters **temper_lpost** – Whether to temperate the posterior or only the likelihood.

sample(*n_samples: int*, *beta: float = 1.0*)
Perform sampling.

Parameters

- **n_samples** – Number of samples to generate.
- **beta** – Inverse of the temperature to which the system is elevated.

set_last_sample(*sample: pypesto.sampling.sampler.InternalSample*)
Set the last sample in the chain to the passed value.

Parameters **sample** – The sample that will replace the last sample in the chain.

class pypesto.sampling.**ParallelTemperingSampler**(*internal_sampler: pypesto.sampling.sampler.InternalSampler*, *betas: Sequence[float] = None*, *n_chains: int = None*, *options: Dict = None*)
Bases: pypesto.sampling.sampler.Sampler

Simple parallel tempering sampler.

__init__(*internal_sampler: pypesto.sampling.sampler.InternalSampler*, *betas: Sequence[float] = None*, *n_chains: int = None*, *options: Dict = None*)
Initialize self. See help(type(self)) for accurate signature.

adjust_betas(*i_sample: int*, *swapped: Sequence[bool]*)
Adjust temperature values. Default: Do nothing.

classmethod default_options() → Dict
Convenience method to set/get default options.

Returns Default sampler options.

Return type default_options

get_samples() → pypesto.sampling.result.McmcPtResult
Concatenate all chains.

initialize(*problem:* pypesto.problem.Problem, *x0: Union[numpy.ndarray, List[numpy.ndarray]]*)
Initialize the sampler.

Parameters

- **problem** – The problem for which to sample.
- **x0** – Should, but is not required to, be used as initial parameter.

sample(*n_samples: int*, *beta: float = 1.0*)
Perform sampling.

Parameters

- **n_samples** – Number of samples to generate.
- **beta** – Inverse of the temperature to which the system is elevated.

swap_samples() → Sequence[bool]
Swap samples as in Vousden2016.

class pypesto.sampling.**Pymc3Sampler**(*step_function=None*, ***kwargs*)
Bases: pypesto.sampling.sampler.Sampler

Wrapper around Pymc3 samplers.

Parameters

- **step_function** – A pymc3 step function, e.g. NUTS, Slice. If not specified, pymc3 determines one automatically (preferable).
- ****kwargs** – Options are directly passed on to *pymc3.sample*.

__init__(*step_function=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_samples() → pypesto.sampling.result.McmcPtResult
Get the generated samples.

initialize(*problem:* pypesto.problem.Problem, *x0: numpy.ndarray*)
Initialize the sampler.

Parameters

- **problem** – The problem for which to sample.
- **x0** – Should, but is not required to, be used as initial parameter.

sample(*n_samples: int*, *beta: float = 1.0*)
Perform sampling.

Parameters

- **n_samples** – Number of samples to generate.
- **beta** – Inverse of the temperature to which the system is elevated.

classmethod translate_options(*options*)
Convenience method to translate options and fill in defaults.

Parameters **options** – Options configuring the sampler.

class pypesto.sampling.**Sampler**(*options: Dict = None*)
Bases: abc.ABC

Sampler base class, not functional on its own.

The sampler maintains an internal chain, which is initialized in *initialize*, and updated in *sample*.

__init__(*options: Dict = None*)
Initialize self. See help(type(self)) for accurate signature.

classmethod default_options() → Dict
Convenience method to set/get default options.

Returns Default sampler options.

Return type default_options

abstract get_samples() → pypesto.sampling.result.McmcPtResult
Get the generated samples.

abstract initialize(*problem:* pypesto.problem.Problem, *x0: Union[numpy.ndarray, List[numpy.ndarray]]*)
Initialize the sampler.

Parameters

- **problem** – The problem for which to sample.
- **x0** – Should, but is not required to, be used as initial parameter.

abstract sample(*n_samples: int*, *beta: float = 1.0*)
Perform sampling.

Parameters

- **n_samples** – Number of samples to generate.
- **beta** – Inverse of the temperature to which the system is elevated.

classmethod translate_options(*options*)
Convenience method to translate options and fill in defaults.

Parameters **options** – Options configuring the sampler.

pypesto.sampling.**geweke_test**(*result:* pypesto.result.Result, *zscore: float = 2.0*) → int
Calculates the burn-in of MCMC chains.

Parameters

- **result** – The pyPESTO result object with filled sample result.
- **zscore** – The Geweke test threshold. Default 2.

Returns Iteration where the first and the last fraction of the chain do not differ significantly regarding Geweke test -> Burn-In

Return type burn_in

pypesto.sampling.**sample**(*problem:* pypesto.problem.Problem, *n_samples: int*, *sampler: pypesto.sampling.sampler.Sampler = None*, *x0: Union[numpy.ndarray, List[numpy.ndarray]] = None*, *result:* pypesto.result.Result *= None*) → *pypesto.result.Result*
This is the main function to call to do parameter sampling.

Parameters

- **problem** – The problem to be solved. If None is provided, a `pypesto.AdaptiveMetropolisSampler` is used.
- **n_samples** – Number of samples to generate.
- **sampler** – The sampler to perform the actual sampling.

- **x0** – Initial parameter for the Markov chain. If None, the best parameter found in optimization is used. Note that some samplers require an initial parameter, some may ignore it. x0 can also be a list, to have separate starting points for parallel tempering chains.
- **result** – A result to write to. If None provided, one is created from the problem.

Returns A result with filled in sample_options part.

Return type result

CHAPTER

THIRTEEN

VISUALIZE

pypesto comes with various visualization routines. To use these, import pypesto.visualize.

class pypesto.visualize.**ReferencePoint**(*reference=None*, *x=None*, *fval=None*, *color=None*, *legend=None*)

Bases: dict

Reference point for plotting. Should contain a parameter value and an objective function value, may alos contain a color and a legend.

Can be used like a dict.

x
Reference parameters.

Type ndarray

fval
Function value, fun(x), for reference parameters.

Type float

color
Color which should be used for reference point.

Type RGBA, optional

auto_color
flag indicating whether color for this reference point should be assigned automatically or whether it was assigned by user

Type boolean

legend
legend text for reference point

Type str

__init__(*reference=None*, *x=None*, *fval=None*, *color=None*, *legend=None*)
Initialize self. See help(type(self)) for accurate signature.

pypesto.visualize.**assign_clustered_colors**(*vals*, *balance_alpha=True*, *highlight_global=True*)

Cluster and assign colors.

Parameters

- **vals** (*numeric list or array*) – List to be clustered and assigned colors.
- **balance_alpha** (*bool (optional)*) – Flag indicating whether alpha for large clusters should be reduced to avoid overplotting (default: True)

- **highlight_global** (*bool (optional)*) – flag indicating whether global optimum should be highlighted

Returns **colors** – One for each element in 'vals'.

Return type list of RGBA

pypesto.visualize.**assign_clusters**(*vals*)

Find clustering.

Parameters **vals** (*numeric list or array*) – List to be clustered.

Returns

- **clust** (*numeric list*) – Indicating the corresponding cluster of each element from 'vals'.
- **clustsize** (*numeric list*) – Size of clusters, length equals number of clusters.

pypesto.visualize.**assign_colors**(*vals*, *colors=None*, *balance_alpha=True*, *highlight_global=True*)

Assign colors or format user specified colors.

Parameters

- **vals** (*numeric list or array*) – List to be clustered and assigned colors.
- **colors** (*list, or RGBA, optional*) – list of colors, or single color
- **balance_alpha** (*bool (optional)*) – Flag indicating whether alpha for large clusters should be reduced to avoid overplotting (default: True)
- **highlight_global** (*bool (optional)*) – flag indicating whether global optimum should be highlighted

Returns **colors** – One for each element in 'vals'.

Return type list of RGBA

pypesto.visualize.**create_references**(*references=None*, *x=None*, *fval=None*, *color=None*, *legend=None*) → List[pypesto.visualize.reference_points.ReferencePoint]

This function creates a list of reference point objects from user inputs

Parameters

- **references** (*ReferencePoint or dict or list, optional*) – Will be converted into a list of RefPoints
- **x** (*ndarray, optional*) – Parameter vector which should be used for reference point
- **fval** (*float, optional*) – Objective function value which should be used for reference point
- **color** (*RGBA, optional*) – Color which should be used for reference point.
- **legend** (*str*) – legend text for reference point

Returns **colors** – One for each element in 'vals'.

Return type list of RGBA

pypesto.visualize.**delete_nan_inf**(*fvals: numpy.ndarray*, *x: numpy.ndarray = None*) → Tuple[numpy.ndarray, numpy.ndarray]

Delete nan and inf values in fvals. If parameters 'x' are passend, also the corresponding entries are deleted.

Parameters

- **x** – array of parameters

- **fvals** – array of fval

Returns

- **x** (*np.array*) – array of parameters without nan or inf
- **fvals** (*np.array*) – array of fval without nan or inf

pypesto.visualize.**optimizer_history**(*results*, *ax=None*, *size=18.5, 10.5*, *trace_x='steps'*, *trace_y='fval'*, *scale_y='log10'*, *offset_y=None*, *colors=None*, *y_limits=None*, *start_indices=None*, *reference=None*, *legends=None*)

Plot history of optimizer. Can plot either the history of the cost function or of the gradient norm, over either the optimizer steps or the computation time.

Parameters

- **results** (*pypesto.Result or list*) – Optimization result obtained by 'optimize.py' or list of those
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified
- **trace_x** (*str, optional*) – What should be plotted on the x-axis? Possibilities: 'time', 'steps' Default: 'steps'
- **trace_y** (*str, optional*) – What should be plotted on the y-axis? Possibilities: 'fval', 'gradnorm', 'stepsize' Default: 'fval'
- **scale_y** (*str, optional*) – May be logarithmic or linear ('log10' or 'lin')
- **offset_y** (*float, optional*) – Offset for the y-axis-values, as these are plotted on a log10-scale Will be computed automatically if necessary
- **colors** (*list, or RGBA, optional*) – list of colors, or single color color or list of colors for plotting. If not set, clustering is done and colors are assigned automatically
- **y_limits** (*float or ndarray, optional*) – maximum value to be plotted on the y-axis, or y-limits
- **start_indices** (*list or int*) – list of integers specifying the multistart to be plotted or int specifying up to which start index should be plotted
- **reference** (*list, optional*) – List of reference points for optimization results, containing et least a function value fval
- **legends** (*list or str*) – Labels for line plots, one label per result object

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**optimizer_history_lowlevel**(*vals*, *scale_y='log10'*, *colors=None*, *ax=None*, *size=18.5, 10.5*, *x_label='Optimizer steps'*, *y_label='Objective value'*, *legend_text=None*)

Plot optimizer history using list of numpy arrays.

Parameters

- **vals** (*list of numpy arrays*) – list of 2xn-arrays (x_values and y_values of the trace)

- **scale_y** (*str, optional*) – May be logarithmic or linear ('log10' or 'lin')
- **colors** (*list, or RGBA, optional*) – list of colors, or single color color or list of colors for plotting. If not set, clustering is done and colors are assigned automatically
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – see waterfall
- **x_label** (*str*) – label for x-axis
- **y_label** (*str*) – label for y-axis
- **legend_text** (*str*) – Label for line plots

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**parameters**(*results*, *ax=None*, *free_indices_only=True*, *lb=None*, *ub=None*, *size=None*, *reference=None*, *colors=None*, *legends=None*, *balance_alpha=True*, *start_indices=None*)

Plot parameter values.

Parameters

- **results** (*pypesto.Result or list*) – Optimization result obtained by 'optimize.py' or list of those
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **free_indices_only** (*bool, optional*) – If True, only free parameters are shown. If False, also the fixed parameters are shown.
- **ub** (*lb,*) – If not None, override result.problem.lb, problem.problem.ub. Dimension either result.problem.dim or result.problem.dim_full.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified
- **reference** (*list, optional*) – List of reference points for optimization results, containing et least a function value fval
- **colors** (*list, or RGBA, optional*) – list of colors, or single color color or list of colors for plotting. If not set, clustering is done and colors are assigned automatically
- **legends** (*list or str*) – Labels for line plots, one label per result object
- **balance_alpha** (*bool (optional)*) – Flag indicating whether alpha for large clusters should be reduced to avoid overplotting (default: True)
- **start_indices** (*list or int*) – list of integers specifying the multistarts to be plotted or int specifying up to which start index should be plotted

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**parameters_lowlevel**(*xs*, *fvals*, *lb=None*, *ub=None*, *x_labels=None*, *ax=None*, *size=None*, *colors=None*, *linestyle='-'*, *legend_text=None*, *balance_alpha=True*)

Plot parameters plot using list of parameters.

Parameters

- **xs** (*nested list or array*) – Including optimized parameters for each startpoint. Shape: (n_starts, dim).

- **fvals** (*numeric list or array*) – Function values. Needed to assign cluster colors.
- **ub** (*lb,*) – The lower and upper bounds.
- **x_labels** (*array_like of str, optional*) – Labels to be used for the parameters.
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – see parameters
- **colors** (*list of RGBA*) – One for each element in 'fvals'.
- **linestyle** (*str, optional*) – linestyle argument for parameter plot
- **legend_text** (*str*) – Label for line plots
- **balance_alpha** (*bool (optional)*) – Flag indicating whether alpha for large clusters should be reduced to avoid overplotting (default: True)

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**process_offset_y**(*offset_y: Optional[float]*, *scale_y: str*, *min_val: float*) → float

compute offset for y-axis, depend on user settings

Parameters

- **offset_y** – value for offsetting the later plotted values, in order to ensure positivity if a semilog-plot is used
- **scale_y** – Can be 'lin' or 'log10', specifying whether values should be plotted on linear or on log10-scale
- **min_val** – Smallest value to be plotted

Returns **offset_y** – value for offsetting the later plotted values, in order to ensure positivity if a semilog-plot is used

Return type float

pypesto.visualize.**process_result_list**(*results*, *colors=None*, *legends=None*)

assigns colors and legends to a list of results, check user provided lists

Parameters

- **results** (*list or pypesto.Result*) – list of pypesto.Result objects or a single pypesto.Result
- **colors** (*list, optional*) – list of RGBA colors
- **legends** (*str or list*) – labels for line plots

Returns

- **results** (*list of pypesto.Result*) – list of pypesto.Result objects
- **colors** (*list of RGBA*) – One for each element in 'results'.
- **legends** (*list of str*) – labels for line plots

pypesto.visualize.**process_y_limits**(*ax*, *y_limits*)

apply user specified limits of y-axis

Parameters

- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **y_limits** (*ndarray*) – y_limits, minimum and maximum, for current axes object
- **min_val** (*float*) – Smallest value to be plotted

Returns **ax** – Axes object to use.

Return type matplotlib.Axes, optional

pypesto.visualize.**profile_cis**(*result: pypesto.result.Result*, *confidence_level: float = 0.95*, *profile_indices: Sequence[int] = None*, *profile_list: int = 0*, *color: Union[str, tuple] = 'C0'*, *show_bounds: bool = False*, *ax: matplotlib.axes._axes.Axes = None*) → matplotlib.axes._axes.Axes

Plot approximate confidence intervals based on profiles.

Parameters

- **result** – The result object after profiling.
- **confidence_level** – The confidence level in (0,1), which is translated to an approximate threshold assuming a chi2 distribution, using *pypesto.profile.chi2_quantile_to_ratio*.
- **profile_indices** – List of integer values specifying which profiles should be plotted. Defaults to the indices for which profiles were generated in profile list *profile_list*.
- **profile_list** – Index of the profile list to be used.
- **color** – Main plot color.
- **show_bounds** – Whether to show, and extend the plot to, the lower and upper bounds.
- **ax** – Axes object to use. Default: Create a new one.

pypesto.visualize.**profile_lowlevel**(*fvals*, *ax=None*, *size: Tuple[float, float] = 18.5, 6.5*, *color=None*, *legend_text: str = None*, *show_bounds: bool = False*, *lb: float = None*, *ub: float = None*)

Lowlevel routine for plotting one profile, working with a numpy array only

Parameters

- **fvals** (*numeric list or array*) – Values to plot.
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified.
- **color** (*RGBA, optional*) – Color for profiles in plot.
- **legend_text** (*str*) – Label for line plots.
- **show_bounds** – Whether to show, and extend the plot to, the lower and upper bounds.
- **lb** – Lower bound.
- **ub** – Upper bound.

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**profiles**(*results: Union[pypesto.result.Result, Sequence[pypesto.result.Result]], ax=None, profile_indices: Sequence[int] = None, size: Sequence[float] = 18.5, 6.5, reference: Union[pypesto.visualize.reference_points.ReferencePoint, Sequence[pypesto.visualize.reference_points.ReferencePoint]] = None, colors=None, legends: Sequence[str] = None, x_labels: Sequence[str] = None, profile_list_ids: Union[int, Sequence[int]] = 0, ratio_min: float = 0.0, show_bounds: bool = False*)

Plot classical 1D profile plot (using the posterior, e.g. Gaussian like profile)

Parameters

- **results** (*list or pypesto.Result*) – List of or single *pypesto.Result* after profiling.
- **ax** (*list of matplotlib.Axes, optional*) – List of axes objects to use.
- **profile_indices** (*list of integer values*) – List of integer values specifying which profiles should be plotted.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified.
- **reference** (*list, optional*) – List of reference points for optimization results, containing at least a function value fval.
- **colors** (*list, or RGBA, optional*) – List of colors, or single color.
- **legends** (*list or str, optional*) – Labels for line plots, one label per result object.
- **x_labels** (*list of str*) – Labels for parameter value axes (e.g. parameter names).
- **profile_list_ids** (*int or list of ints, optional*) – Index or list of indices of the profile lists to be used for profiling.
- **ratio_min** – Minimum ratio below which to cut off.
- **show_bounds** – Whether to show, and extend the plot to, the lower and upper bounds.

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**profiles_lowlevel**(*fvals, ax=None, size: Tuple[float, float] = 18.5, 6.5, color=None, legend_text: str = None, x_labels=None, show_bounds: bool = False, lb_full=None, ub_full=None*)

Lowlevel routine for profile plotting, working with a list of arrays only, opening different axes objects in case

Parameters

- **fvals** (*numeric list or array*) – Values to plot.
- **ax** (*list of matplotlib.Axes, optional*) – List of axes object to use.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified.
- **size** – Figure size (width, height) in inches. Is only applied when no ax object is specified.
- **color** (*RGBA, optional*) – Color for profiles in plot.
- **legend_text** (*List[str]*) – Label for line plots.
- **legend_text** – Label for line plots.

- **show_bounds** – Whether to show, and extend the plot to, the lower and upper bounds.
- **lb_full** – Lower bound.
- **ub_full** – Upper bound.

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**sampling_1d_marginals**(*result:* pypesto.result.Result, *i_chain: int = 0*, *stepsize: int = 1*, *plot_type: str = 'both'*, *bw: str = 'scott'*, *suptitle: str = None*, *size: Tuple[float, float] = None*)

Plot marginals.

Parameters

- **result** – The pyPESTO result object with filled sample result.
- **i_chain** – Which chain to plot. Default: First chain.
- **stepsize** – Only one in *stepsize* values is plotted.
- **plot_type** (*{'hist'|'kde'|'both'}*) – Specify whether to plot a histogram ('hist'), a kernel density estimate ('kde'), or both ('both').
- **bw** (*{'scott', 'silverman' | scalar | pair of scalars}*) – Kernel bandwidth method.
- **suptitle** – Figure super title.
- **size** – Figure size in inches.

Returns **ax**

Return type matplotlib-axes

pypesto.visualize.**sampling_fval_trace**(*result:* pypesto.result.Result, *i_chain: int = 0*, *full_trace: bool = False*, *stepsize: int = 1*, *title: str = None*, *size: Tuple[float, float] = None*, *ax: matplotlib.axes._axes.Axes = None*)

Plot log-posterior (=function value) over iterations.

Parameters

- **result** – The pyPESTO result object with filled sample result.
- **i_chain** – Which chain to plot. Default: First chain.
- **full_trace** – Plot the full trace including warm up. Default: False.
- **stepsize** – Only one in *stepsize* values is plotted.
- **title** – Axes title.
- **size** (*ndarray*) – Figure size in inches.
- **ax** – Axes object to use.

Returns The plot axes.

Return type ax

pypesto.visualize.**sampling_parameters_trace**(*result: pypesto.result.Result, i_chain: int = 0, full_trace: bool = False, stepsize: int = 1, use_problem_bounds: bool = True, suptitle: str = None, size: Tuple[float, float] = None, ax: matplotlib.axes._axes.Axes = None*)

Plot parameter values over iterations.

Parameters

- **result** – The pyPESTO result object with filled sample result.
- **i_chain** – Which chain to plot. Default: First chain.
- **full_trace** – Plot the full trace including warm up. Default: False.
- **stepsize** – Only one in *stepsize* values is plotted.
- **use_problem_bounds** – Defines if the y-limits shall be the lower and upper bounds of parameter estimation problem.
- **suptitle** – Figure suptitle.
- **size** – Figure size in inches.
- **ax** – Axes object to use.

Returns The plot axes.

Return type ax

pypesto.visualize.**sampling_scatter**(*result: pypesto.result.Result, i_chain: int = 0, stepsize: int = 1, suptitle: str = None, size: Tuple[float, float] = None*)

Parameter scatter plot.

Parameters

- **result** – The pyPESTO result object with filled sample result.
- **i_chain** – Which chain to plot. Default: First chain.
- **stepsize** – Only one in *stepsize* values is plotted.
- **suptitle** – Figure super title.
- **size** – Figure size in inches.

Returns The plot axes.

Return type ax

pypesto.visualize.**waterfall**(*results, ax=None, size=18.5, 10.5, y_limits=None, scale_y='log10', offset_y=None, start_indices=None, reference=None, colors=None, legends=None*)

Plot waterfall plot.

Parameters

- **results** (*pypesto.Result or list*) – Optimization result obtained by 'optimize.py' or list of those
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – Figure size (width, height) in inches. Is only applied when no ax object is specified
- **y_limits** (*float or ndarray, optional*) – maximum value to be plotted on the y-axis, or y-limits

- **scale_y** (*str, optional*) – May be logarithmic or linear ('log10' or 'lin')
- **offset_y** – offset for the y-axis, if it is supposed to be in log10-scale
- **start_indices** (*list or int*) – list of integers specifying the multistart to be plotted or int specifying up to which start index should be plotted
- **reference** (*list, optional*) – List of reference points for optimization results, containing et least a function value fval
- **colors** (*list, or RGBA, optional*) – list of colors, or single color color or list of colors for plotting. If not set, clustering is done and colors are assigned automatically
- **legends** (*list or str*) – Labels for line plots, one label per result object

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**waterfall_lowlevel**(*fvals*, *scale_y='log10'*, *offset_y=0.0*, *ax=None*, *size=18.5, 10.5*, *colors=None*, *legend_text=None*)

Plot waterfall plot using list of function values.

Parameters

- **fvals** (*numeric list or array*) – Including values need to be plotted.
- **scale_y** (*str, optional*) – May be logarithmic or linear ('log10' or 'lin')
- **offset_y** – offset for the y-axis, if it is supposed to be in log10-scale
- **ax** (*matplotlib.Axes, optional*) – Axes object to use.
- **size** (*tuple, optional*) – see waterfall
- **colors** (*list, or RGBA, optional*) – list of colors, or single color color or list of colors for plotting. If not set, clustering is done and colors are assigned automatically
- **legend_text** (*str*) – Label for line plots

Returns **ax** – The plot axes.

Return type matplotlib.Axes

CHAPTER

FOURTEEN

RESULT

The pypesto.Result object contains all results generated by the pypesto components. It contains sub-results for optimization, profiles, sampling.

class pypesto.result.**OptimizeResult**
Bases: object

Result of the minimize() function.

__init__()
Initialize self. See help(type(self)) for accurate signature.

append(*optimizer_result: pypesto.optimize.result.OptimizerResult*)
Append an optimizer result to the result object.

Parameters **optimizer_result** – The result of one (local) optimizer run.

as_dataframe(*keys=None*) → pandas.core.frame.DataFrame
Get as pandas DataFrame. If keys is a list, return only the specified values.

as_list(*keys=None*) → Sequence
Get as list. If keys is a list, return only the specified values.

Parameters **keys** (*list(str), optional*) – Labels of the field to extract.

get_for_key(*key*) → list
Extract the list of values for the specified key as a list.

sort()
Sort the optimizer results by function value fval (ascending).

class pypesto.result.**ProfileResult**
Bases: object

Result of the profile() function.

It holds a list of profile lists. Each profile list consists of a list of *ProfilerResult* objects, one for each parameter.

__init__()
Initialize self. See help(type(self)) for accurate signature.

append_empty_profile_list() → int
Append an empty profile list to the list of profile lists.

Returns The index of the created profile list.

Return type index

append_profiler_result(*profiler_result: Optional[pypesto.profile.result.ProfilerResult] = None*, *profile_list: int = None*) → None
Append the profiler result to the profile list.

Parameters

- **profiler_result** – The result of one profiler run for a parameter, or None if to be left empty.
- **profile_list** – Index specifying the profile list to which we want to append. Defaults to the last list.

get_profiler_result(*i_par: int*, *profile_list: int = None*) → pypesto.profile.result.ProfilerResult
Get theprofiler result at parameter index *i_par* of profile list *profile_list*.

Parameters

- **i_par** – Integer specifying the profile index.
- **profile_list** – Index specifying the profile list. Defaults to the last list.

set_profiler_result(*profiler_result: pypesto.profile.result.ProfilerResult*, *i_par: int*, *profile_list: int = None*) → None
Write a profiler result to the result object at *i_par* of profile list *profile_list*.

Parameters

- **profiler_result** – The result of one (local) profiler run.
- **i_par** – Integer specifying the parameter index.
- **profile_list** – Index specifying the profile list. Defaults to the last list.

class pypesto.result.**Result**(*problem=None*)
Bases: object

Universal result object for pypesto. The algorithms like optimize, profile, sample fill different parts of it.

problem
The problem underlying the results.

Type pypesto.Problem

optimize_result
The results of the optimizer runs.

profile_result
The results of the profiler run.

sample_result
The results of the sampler run.

__init__(*problem=None*)
Initialize self. See help(type(self)) for accurate signature.

class pypesto.result.**SampleResult**
Bases: object

Result of the sample() function.

__init__()
Initialize self. See help(type(self)) for accurate signature.

pypesto.result.**Sequence**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Sequence

CHAPTER

FIFTEEN

ENGINES

The execution of the multistarts can be parallelized in different ways, e.g. multi-threaded or cluster-based. Note that it is not checked whether a single task itself is internally parallelized.

class pypesto.engine.**Engine**

Bases: abc.ABC

Abstract engine base class.

__init__()

Initialize self. See help(type(self)) for accurate signature.

abstract execute(*tasks: List[pypesto.engine.task.Task]*)

Execute tasks.

Parameters **tasks** – List of tasks to execute.

class pypesto.engine.**MultiProcessEngine**(*n_procs: int = None*)

Bases: pypesto.engine.base.Engine

Parallelize the task execution using multiprocessing.

Parameters **n_procs** – The maximum number of processes to use in parallel. Defaults to the number of CPUs available on the system according to *os.cpu_count()*. The effectively used number of processes will be the minimum of *n_procs* and the number of tasks submitted.

__init__(*n_procs: int = None*)

Initialize self. See help(type(self)) for accurate signature.

execute(*tasks: List[pypesto.engine.task.Task]*)

Pickle tasks and distribute work over parallel processes.

class pypesto.engine.**MultiThreadEngine**(*n_threads: int = None*)

Bases: pypesto.engine.base.Engine

Parallelize the task execution using multithreading.

Parameters **n_threads** – The maximum number of threads to use in parallel. Defaults to the number of CPUs available on the system according to *os.cpu_count()*. The effectively used number of threads will be the minimum of *n_threads* and the number of tasks submitted.

__init__(*n_threads: int = None*)

Initialize self. See help(type(self)) for accurate signature.

execute(*tasks: List[pypesto.engine.task.Task]*)

Deepcopy tasks and distribute work over parallel threads.

class pypesto.engine.**OptimizerTask**(*optimizer: pypesto.optimize.optimizer.Optimizer*, *problem: pypesto.problem.Problem*, *x0: numpy.ndarray*, *id: str*, *options: pypesto.optimize.options.OptimizeOptions*, *history_options: pypesto.objective.history.HistoryOptions*)

Bases: pypesto.engine.task.Task

A multistart optimization task, performed in *pypesto.minimize*.

__init__(*optimizer: pypesto.optimize.optimizer.Optimizer*, *problem: pypesto.problem.Problem*, *x0: numpy.ndarray*, *id: str*, *options: pypesto.optimize.options.OptimizeOptions*, *history_options: pypesto.objective.history.HistoryOptions*)

Create the task object.

Parameters

- **optimizer** – The optimizer to use.
- **problem** – The problem to solve.
- **x0** – The point from which to start.
- **id** – The multistart id.
- **options** – Options object applying to optimization.
- **history_options** – Optimizer history options.

execute() → pypesto.optimize.result.OptimizerResult

Execute the task and return its results.

class pypesto.engine.**SingleCoreEngine**

Bases: pypesto.engine.base.Engine

Dummy engine for sequential execution on one core. Note that the objective itself may be multithreaded.

__init__()

Initialize self. See help(type(self)) for accurate signature.

execute(*tasks: List[pypesto.engine.task.Task]*)

Execute all tasks in a simple for loop sequentially.

CHAPTER

SIXTEEN

STARTPOINT

Methods for selecting points that can be used as start points for multistart optimization. All methods have the form

```
method(**kwargs) -> startpoints
```

where the kwargs can/should include the following parameters, which are passed by pypesto:

n_starts: int Number of points to generate.

lb, ub: ndarray Lower and upper bound, may for most methods not contain nan or inf values.

x_guesses: ndarray, shape=(g, dim), optional Parameter guesses by the user, where g denotes the number of guesses. Note that these are only possibly taken as reference points to generate new start points (e.g. to maximize some distance) depending on the method, but regardless of g, there are always n_starts points generated and returned.

objective: pypesto.Objective, optional The objective can be used to evaluate the goodness of start points.

max_n_fval: int, optional The maximum number of evaluations of the objective function allowed.

pypesto.startpoint.**assign_startpoints**(*n_starts: int*, *startpoint_method: Callable*, *problem:* pypesto.problem.Problem, *options: pypesto.optimize.options.OptimizeOptions*) → numpy.ndarray

Assign startpoints.

pypesto.startpoint.**latin_hypercube**(***kwargs*) → numpy.ndarray

Generate latin hypercube points.

pypesto.startpoint.**uniform**(***kwargs*) → numpy.ndarray

Generate uniform points.

CHAPTER

SEVENTEEN

STORAGE

Saving and loading traces and results objects.

class pypesto.storage.**OptimizationResultHDF5Reader**(*storage_filename: str*)
Bases: object

Reader of the HDF5 result files written by class OptimizationResultHDF5Writer.

storage_filename
HDF5 result file name

__init__(*storage_filename: str*)

Parameters **storage_filename** (*str*) – HDF5 result file name

read() → *pypesto.result.Result*
Read HDF5 result file and return pyPESTO result object.

class pypesto.storage.**OptimizationResultHDF5Writer**(*storage_filename: str*)
Bases: object

Writer of the HDF5 result files.

storage_filename
HDF5 result file name

__init__(*storage_filename: str*)

Parameters **storage_filename** (*str*) – HDF5 result file name

write(*result:* pypesto.result.Result, *overwrite=False*)
Write HDF5 result file from pyPESTO result object.

class pypesto.storage.**ProblemHDF5Reader**(*storage_filename: str*)
Bases: object

Reader of the HDF5 problem files written by class ProblemHDF5Writer.

storage_filename
HDF5 problem file name

__init__(*storage_filename: str*)

Parameters **storage_filename** (*str*) – HDF5 problem file name

read(*objective: pypesto.objective.base.ObjectiveBase = None*) → *pypesto.problem.Problem*
Read HDF5 problem file and return pyPESTO problem object.

Parameters **objective** – Objective function which is currently not save to storage.

Returns A problem instance with all attributes read in.

Return type problem

class pypesto.storage.**ProblemHDF5Writer**(*storage_filename: str*)

Bases: object

Writer of the HDF5 problem files.

storage_filename
HDF5 result file name

__init__(*storage_filename: str*)

Parameters **storage_filename** (*str*) – HDF5 problem file name

write(*problem*, *overwrite: bool = False*)
Write HDF5 problem file from pyPESTO problem object.

CHAPTER

EIGHTEEN

LOGGING

Logging convenience functions.

pypesto.logging.**log**(*name: str = 'pypesto'*, *level: int = 10*, *console: bool = False*, *filename: str = ''*)
Log messages from a specified name with a specified level to any combination of console and file.

Parameters

- **name** – The name of the logger.
- **level** – The output level to use.
- **console** – If True, messages are logged to console.
- **filename** – If specified, messages are logged to a file with this name.

pypesto.logging.**log_to_console**(*level: int = 10*)
Log to console.

Parameters the log method. (*See*) –

pypesto.logging.**log_to_file**(*level: int = 10*, *filename: str = '.pypesto_logging.log'*)
Log to file.

Parameters the log method. (*See*) –

CHAPTER

NINETEEN

RELEASE NOTES

19.1 0.1 series

19.1.1 0.1.0 (2020-06-17)

Objective

- Write solver settings to stream to enable serialization for distributed systems (#308)
- Refactor objective function (#347) * Removes necessity for all of the nasty binding/undbinding in AmiciObjective * Substantially reduces the complexity of the AggregatedObjective class * Aggregation of functions with inconsistent sensi_order/mode support * Introduce ObjectiveBase as an abstract Objective class * Introduce FunctionObjective for objectives from functions
- Implement priors with gradients, integrate with PEtab (#357)
- Fix minus sign in AmiciObjective.get_error_output (#361)
- Implement a prior class, derivatives for standard models, interface with PEtab (#357)
- Use *amici.import_model_module* to resolve module loading failure (#384)

Problem

- Tidy up problem vectors using properties (#393)

Optimization

- Interface IpOpt optimizer (#373)

Profiles

- Tidy up profiles (#356)
- Refactor profiles; add locally approximated profiles (#369)
- Fix profiling and visualization with fixed parameters (#393)

Sampling

- Geweke test for sampling convergence (#339)
- Implement basic Pymc3 sampler (#351)
- Make theano for pymc3 an optional dependency (allows using pypesto without pymc3) (#356)
- Progress bar for MCMC sampling (#366)
- Fix Geweke test crash for small sample sizes (#376)
- In parallel tempering, allow to only temperate the likelihood, not the prior (#396)

History and storage

- Allow storing results in a pre-filled hdf5 file (#290)
- Various fixes of the history (reduced vs. full parameters, read-in from file, chi2 values) (#315)
- Fix proper dimensions in result for failed start (#317)
- Create required directories before creating hdf5 file (#326)
- Improve storage and docs documentation (#328)
- Fix storing x_free_indices in hdf5 result (#334)
- Fix problem hdf5 return format (#336)
- Implement partial trace extraction, simplify History API (#337)
- Save really all attributes of a Problem to hdf5 (#342)

Visualization

- Customizable xLabels and tight layout for profile plots (#331)
- Fix non-positive bottom ylim on a log-scale axis in waterfall plots (#348)
- Fix "palette list has the wrong number of colors" in sampling plots (#372)
- Allow to plot multiple profiles from one result (#399)

Logging

- Allow easier specification of only logging for submodules (#398)

Tests

- Speed up travis build (#329)
- Update travis test system to latest ubuntu and python 3.8 (#330)
- Additional code quality checks, minor simplifications (#395)

19.2 0.0 series

19.2.1 0.0.13 (2020-05-03)

- Tidy up and speed up tests (#265 and others).
- Basic self-implemented Adaptive Metropolis and Adaptive Parallel Tempering sampling routines (#268).
- Fix namespace sample -> sampling (#275).
- Fix covariance matrix regularization (#275).
- Fix circular dependency *PetabImporter* - *PetabAmiciObjective* via *AmiciObjectBuilder*, *PetabAmiciObjective* becomes obsolete (#274).
- Define *AmiciCalculator* to separate the AMICI call logic (required for hierarchical optimization) (#277).
- Define initialize function for resetting steady states in *AmiciObjective* (#281).
- Fix scipy least squares options (#283).
- Allow failed starts by default (#280).
- Always copy parameter vector in objective to avoid side effects (#291).

- Add Dockerfile (#288).
- Fix header names in CSV history (#299).

Documentation:

- Use imported members in autodoc (#270).
- Enable python syntax highlighting in notebooks (#271).

19.2.2 0.0.12 (2020-04-06)

- Add typehints to global functions and classes.
- Add *PetabImporter.rdatas_to_simulation_df* function (all #235).
- Adapt y scale in waterfall plot if convergence was too good (#236).
- Clarify that *Objective* is of type negative log-posterior, for minimization (#243).
- Tidy up *AmiciObjective.parameter_mapping* as implemented in AMICI now (#247).
- Add *MultiThreadEngine* implementing multi-threading aside the *MultiProcessEngine* implementing multi-processing (#254).
- Fix copying and pickling of *AmiciObjective* (#252, #257).
- Remove circular dependence history-objective (#254).
- Fix problem of visualizing results with failed starts (#249).
- Rework history: make thread-safe, use factory methods, make context-specific (#256).
- Improve PEtab usage example (#258).
- Define history base contract, enabling different backends (#260).
- Store optimization results to HDF5 (#261).
- Simplify tests (#263).

Breaking changes:

- *HistoryOptions* passed to *pypesto.minimize* instead of *Objective* (#256).
- *GlobalOptimizer* renamed to *PyswarmOptimizer* (#235).

19.2.3 0.0.11 (2020-03-17)

- Rewrite AmiciObjective and PetabAmiciObjective simulation routine to directly use amici.petab_objective routines (#209, #219, #225).
- Implement petab test suite checks (#228).
- Various error fixes, in particular regarding PEtab and visualization.
- Improve trace structure.
- Fix conversion between fval and chi2, fix FIM (all #223).

19.2.4 0.0.10 (2019-12-04)

- Only compute FIM when sensitivities are available (#194).
- Fix documentation build (#197).
- Add support for pyswarm optimizer (#198).
- Run travis tests for documentation and notebooks only on pull requests (#199).

19.2.5 0.0.9 (2019-10-11)

- Update to AMICI 0.10.13, fix API changes (#185).
- Start using PEtab import from AMICI to be able to import constant species (#184, #185)
- Require PEtab>=0.0.0a16 (#183)

19.2.6 0.0.8 (2019-09-01)

- Add logo (#178).
- Fix petab API changes (#179).
- Some minor bugfixes (#168).

19.2.7 0.0.7 (2019-03-21)

- Support noise models in Petab and Amici.
- Minor Petab update bug fixes.

19.2.8 0.0.6 (2019-03-13)

- Several minor error fixes, in particular on tests and steady state.

19.2.9 0.0.5 (2019-03-11)

- Introduce AggregatedObjective to use multiple objectives at once.
- Estimate steady state in AmiciObjective.
- Check amici model build version in PetabImporter.
- Use Amici multithreading in AmiciObjective.
- Allow to sort multistarts by initial value.
- Show usage of visualization routines in notebooks.
- Various fixes, in particular to visualization.

19.2.10 0.0.4 (2019-02-25)

- Implement multi process parallelization engine for optimization.
- Introduce PrePostProcessor to more reliably handle pre- and post-processing.
- Fix problems with simulating for multiple conditions.
- Add more visualization routines and options for those (colors, reference points, plotting of lists of result obejcts)

19.2.11 0.0.3 (2019-01-30)

- Import amici models and the petab data format automatically using pypesto.PetabImporter.
- Basic profiling routines.

19.2.12 0.0.2 (2018-10-18)

- Fix parameter values
- Record trace of function values
- Amici objective to directly handle amici models

19.2.13 0.0.1 (2018-07-25)

- Basic framework and implementation of the optimization

CHAPTER

TWENTY

AUTHORS

This package was mainly developed by:

- Jan Hasenauer
- Yannik Schälte
- Fabian Fröhlich
- Daniel Weindl
- Paul Stapor
- Leonard Schmiester
- Dantong Wang
- Leonard Schmiester
- Caro Loos

CHAPTER

TWENTYONE

CONTACT

Discovered an error? Need help? Not sure if something works as intended? Please contact us!

- Yannik Schälte: yannik.schaelte@gmail.com

CHAPTER

TWENTYTWO

LICENSE

```
Copyright (c) 2018, Jan Hasenauer
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are␣
↪permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list␣
↪of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this␣
↪list of conditions and the following disclaimer in the documentation and/or other␣
↪materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be␣
↪used to endorse or promote products derived from this software without specific␣
↪prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY␣
↪EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED␣
↪WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.␣
↪IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,␣
↪INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT␣
↪NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR␣
↪PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,␣
↪WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)␣
↪ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE␣
↪POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER

TWENTYTHREE

LOGO

pyPESTO's logo can be found in multiple variants in the doc/logo directory on github, in svg and png format. It is made available under a creative commons CC0 license. You are encouraged to use it e.g. in presentations and posters.

We thank Patrick Beart for his contribution to the logo.

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

X