pyPESTO Documentation

Release 0.0.7

The pyPESTO developers

Mar 21, 2019

User's guide

Version: 0.0.7
Source code: https://github.com/icb-dcm/pypesto

CHAPTER 1

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 2

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
# get the optimizer trace
objective_options = pypesto.ObjectiveOptions(trace_record=True, trace_save_iter=1)
objective1 = pypesto.Objective(fun=sp.optimize.rosen,
                               grad=sp.optimize.rosen_der,
                               hess=sp.optimize.rosen_hess,
                               options=objective_options)

# 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)
```

(continues on next page)

(continued from previous page)

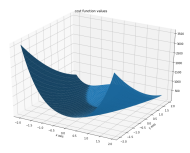
```
dim_full = 10
lb = -5 * np.ones((dim_full, 1))
ub = 5 * np.ones((dim_full, 1))

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

2.1.2 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')
```



2.1.3 Run optimization

[5]:
```
# 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

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

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

(continues on next page)

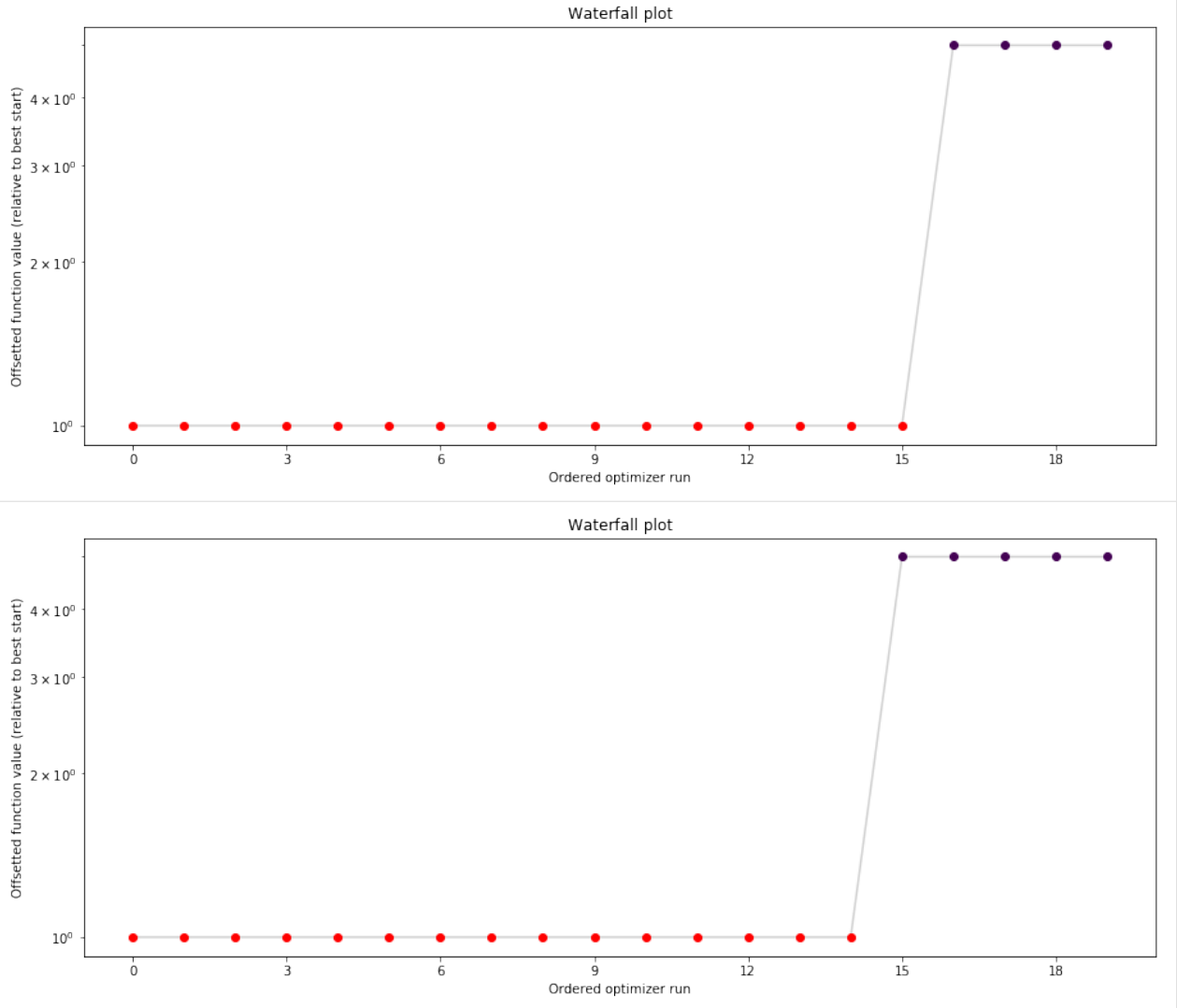
(continued from previous page)

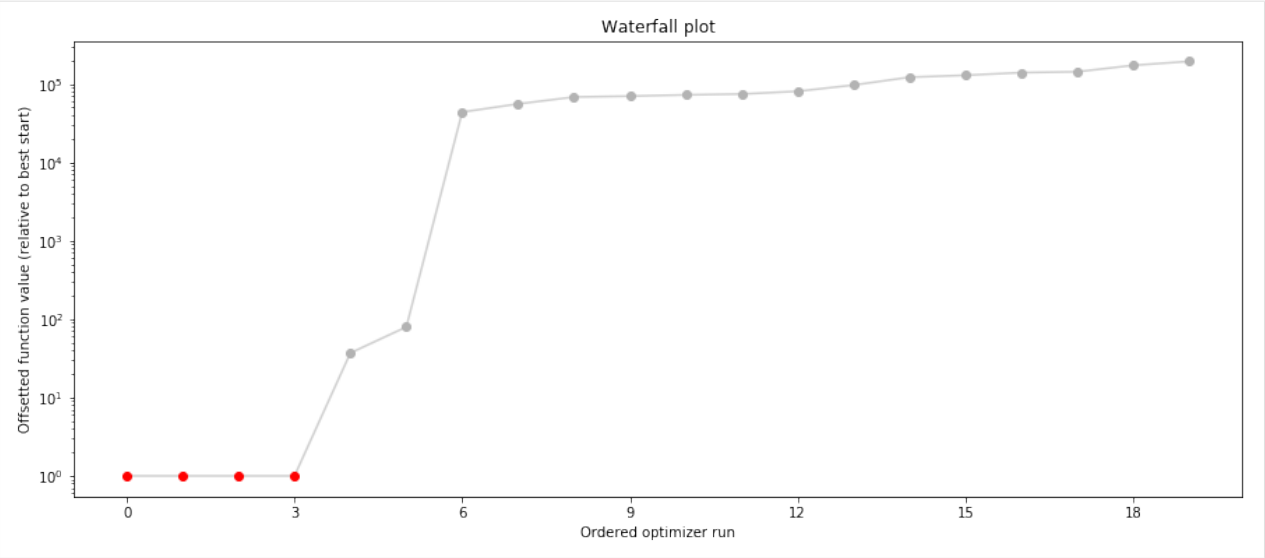
```
/home/yannik/yenv/lib/python3.6/site-packages/scipy/optimize/_minimize.py:518:␣
↪RuntimeWarning: Method dogleg cannot handle constraints nor bounds.
  RuntimeWarning)
```

2.1.4 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 0x7fac6554b550>
```

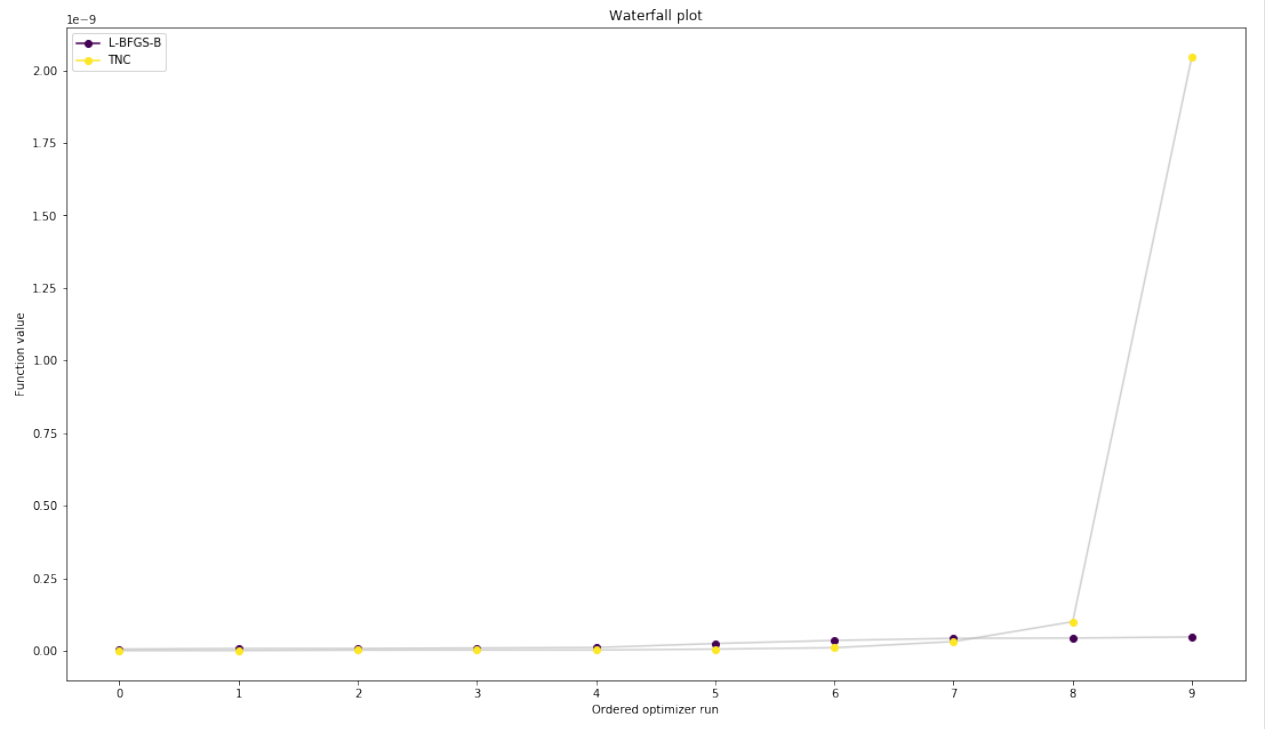




We can now have a closer look, which method perfomred better: Let's first compare bfgs and TNC, since both methods gave good results. How does they fine convergence look like?

[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 0x7fac65342438>
```



[8]:
```
# retrieve second optimum
all_x = result1_bfgs.optimize_result.get_for_key('x')
```

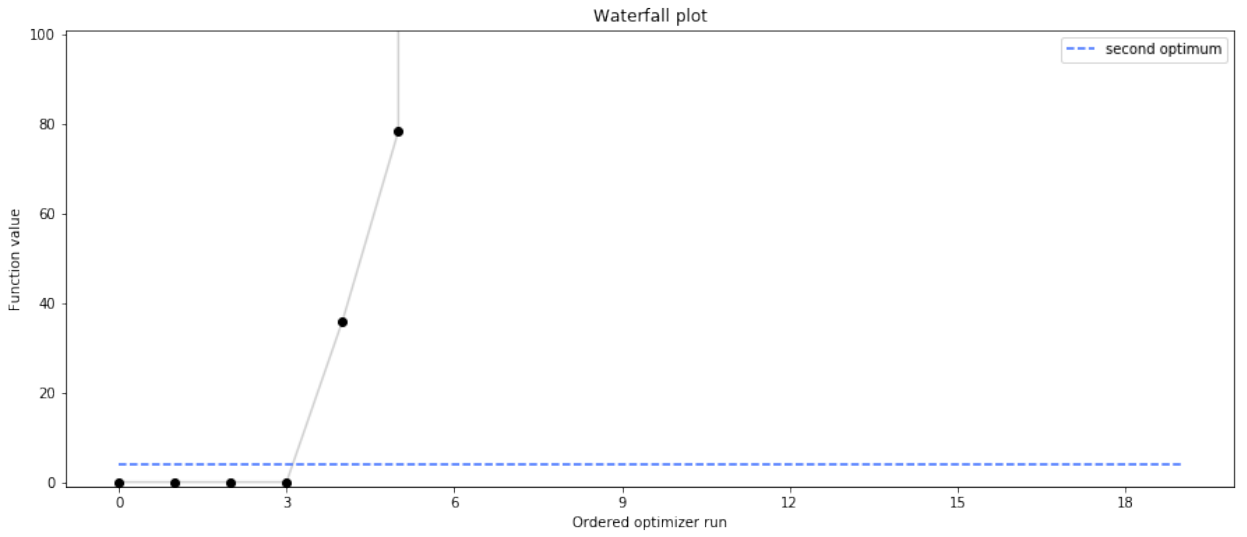
(continues on next page)

(continued from previous page)

```
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: 3.9865791128196464
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fac652265f8>
```

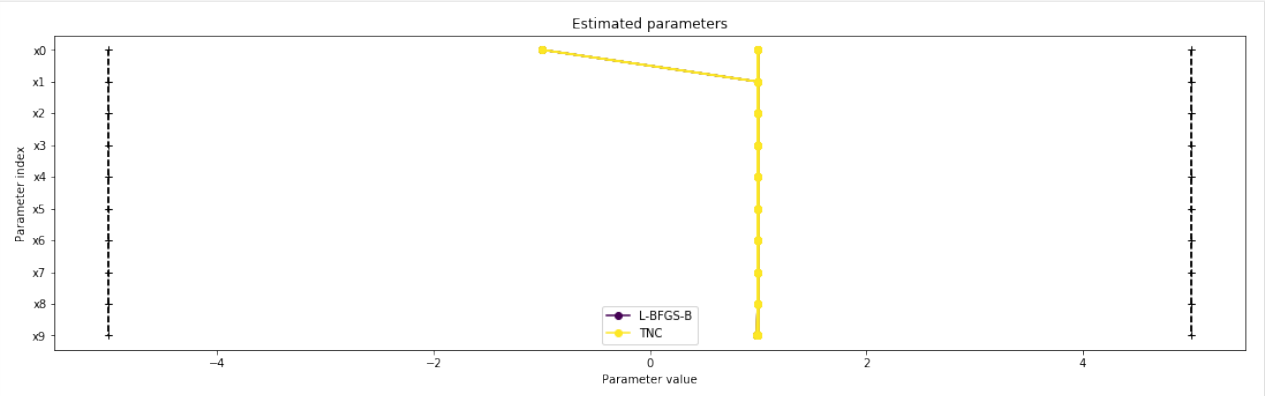


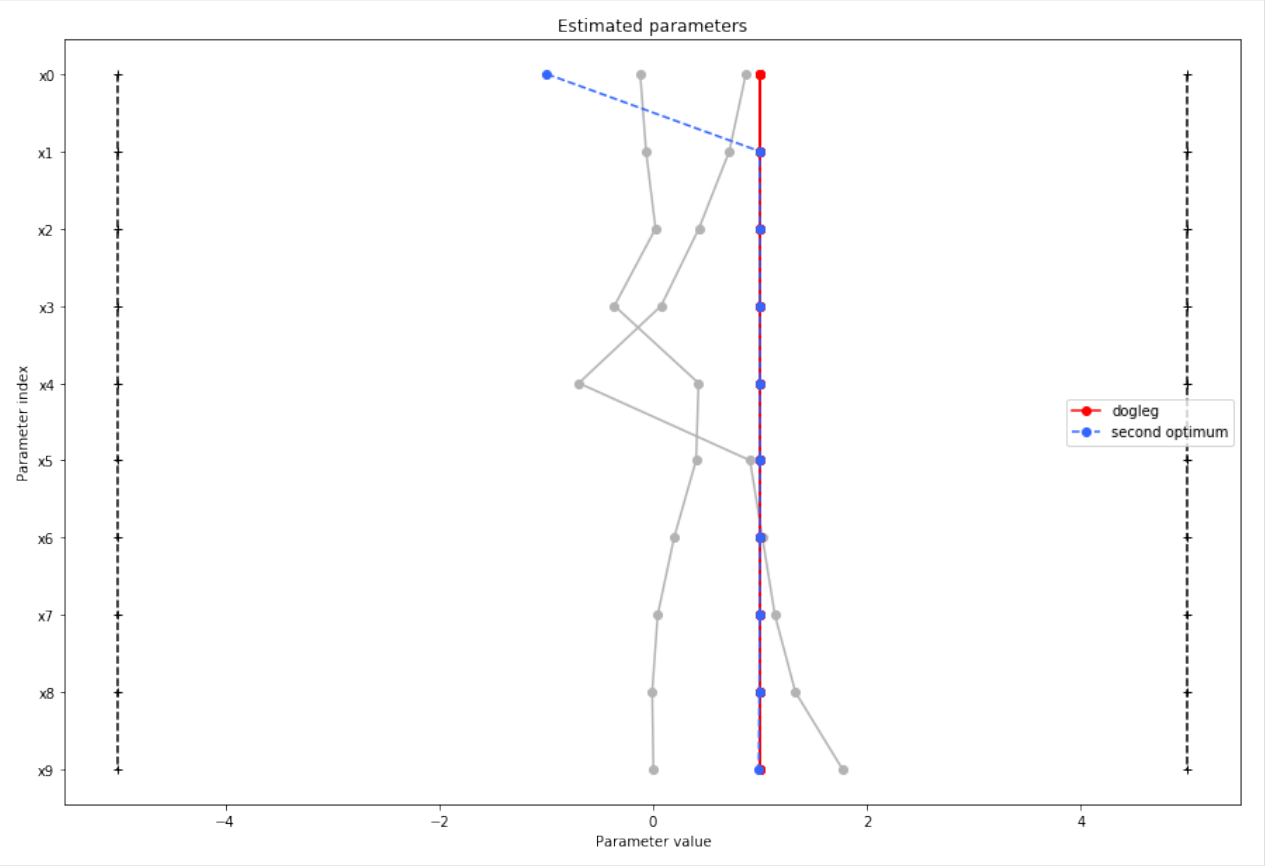
2.1.5 Visualize parameters

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 0x7fac654570b8>
```



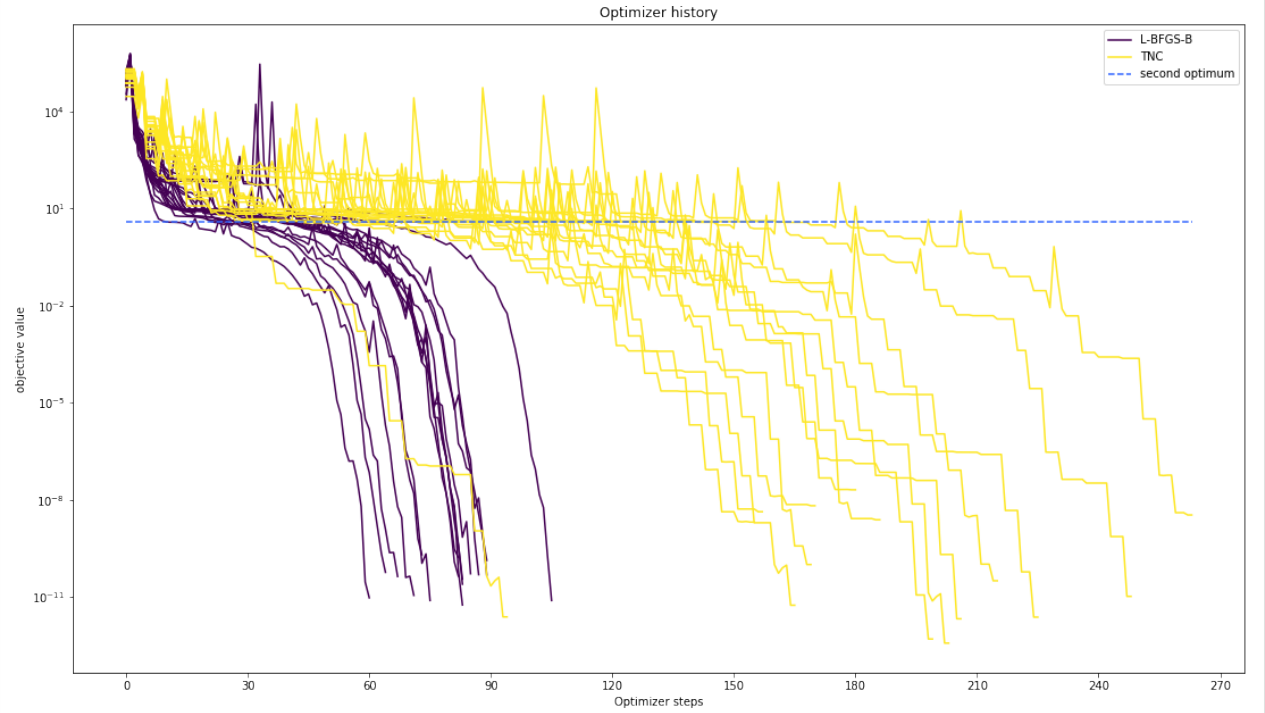


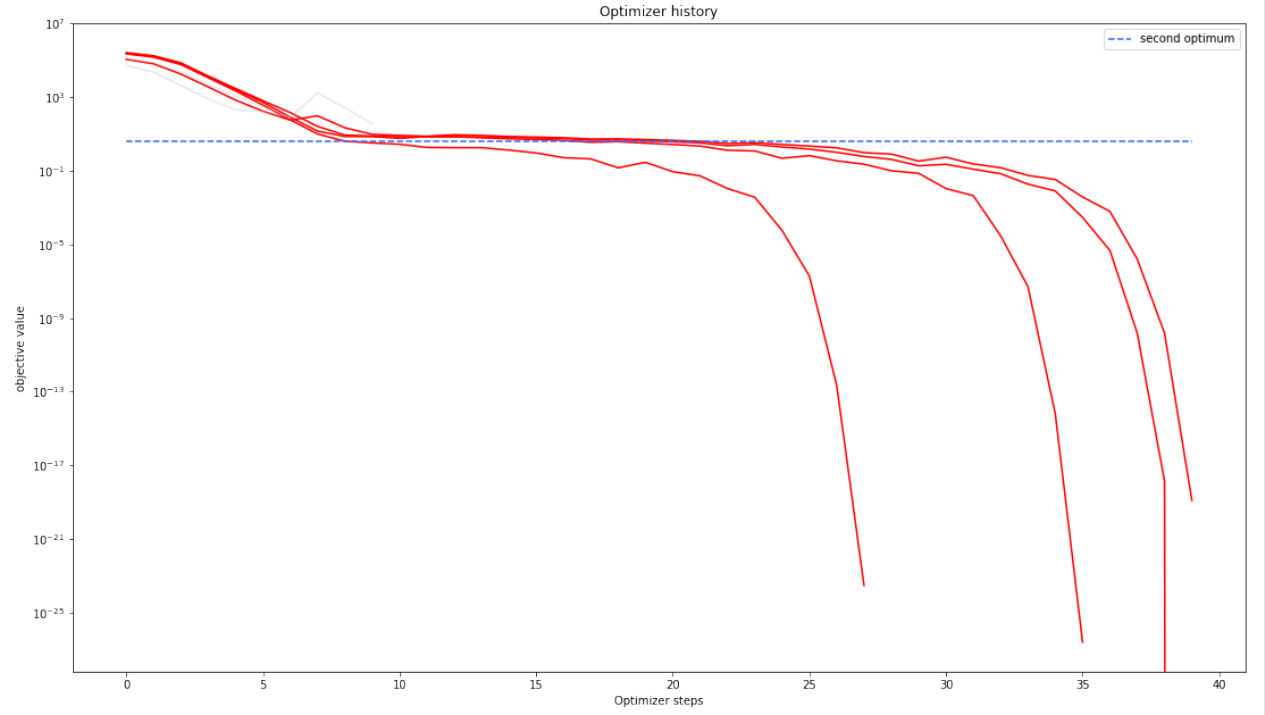
2.1.6 Optimizer history

Let's compare optimzer progress over time.

[10]:
```
# 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)
```

[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fac6549ef60>
```





We can also visualize this usign other scalings or offsets…

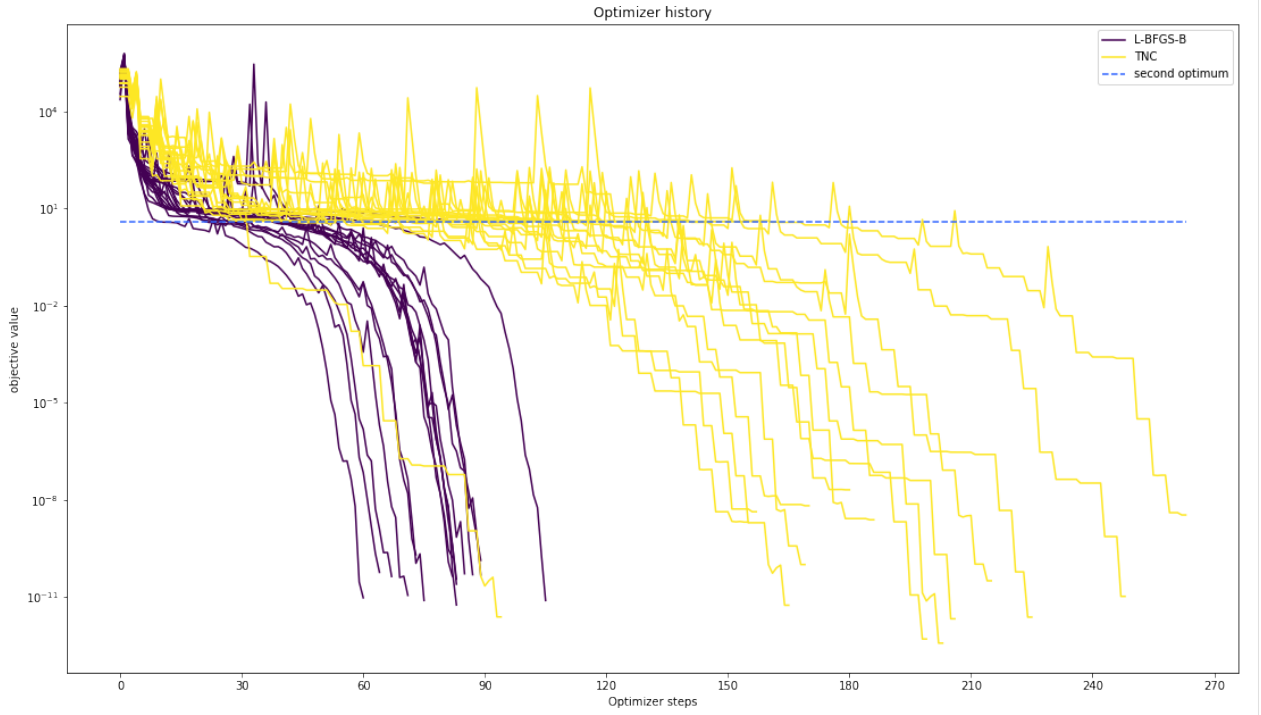
[11]:
```
# plot one list of waterfalls
pypesto.visualize.optimizer_history([result1_bfgs, result1_tnc],
                                    legends=['L-BFGS-B', 'TNC'],
```

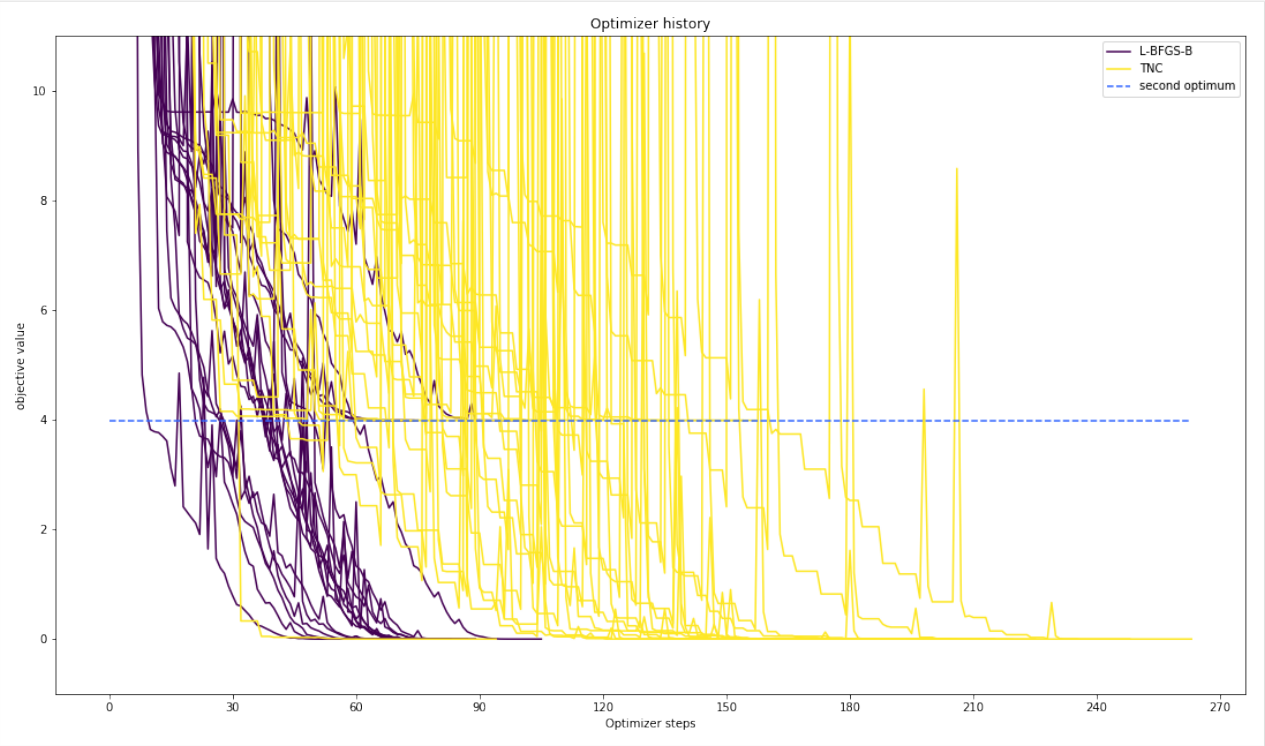
(continues on next page)

(continued from previous page)

```
                                    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.])
```

[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fac6501beb8>





2.1.7 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.

[12]:
```
# compute profiles
profile_options = pypesto.ProfileOptions(min_step_size=0.0005,
    delta_ratio_max=0.05,
    default_step_size=0.005,
    ratio_min=0.03)

result1_tnc = pypesto.parameter_profile(
    problem=problem1,
    result=result1_tnc,
    optimizer=optimizer_tnc,
    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_tnc = pypesto.parameter_profile(
    problem=problem1,
    result=result1_tnc,
    optimizer=optimizer_tnc,
    profile_index=np.array([1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0]),
```

(continues on next page)

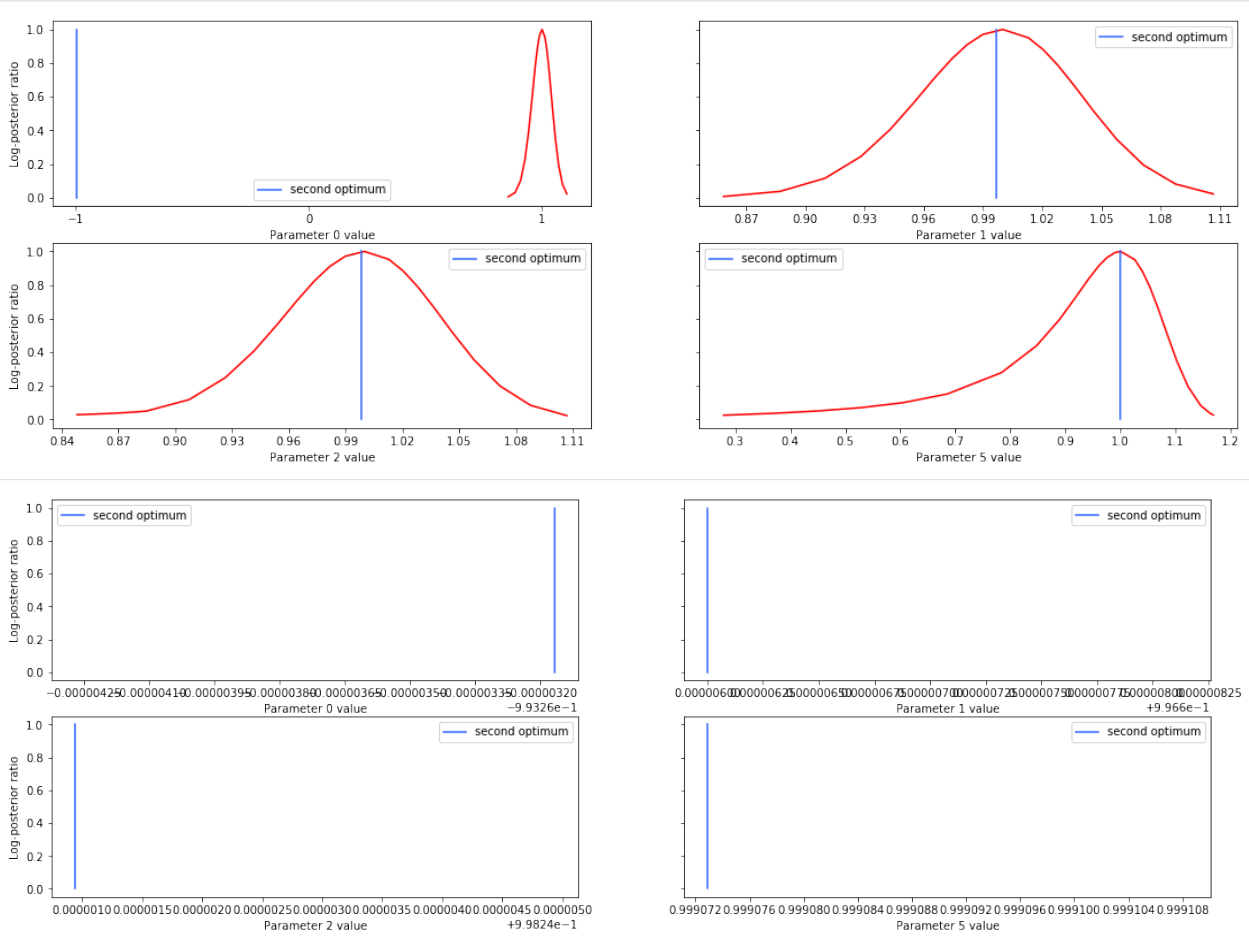
(continued from previous page)

```
    result_index=19,
    profile_options=profile_options)
```

2.1.8 Visualize and analyze results

pypesto offers easy-to-use visualization routines:

[13]:
```
# specify the parameters, for which profiles should be computed
ax = pypesto.visualize.profiles(result1_tnc, profile_indices = [0,1,2,5],
                                reference=ref, profile_list_id=0)
# plot profiles again, now from second optimum
ax = pypesto.visualize.profiles(result1_tnc, profile_indices = [0,1,2,5],
                                reference=ref, profile_list_id=1)
```



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

[14]:
```
result1_tnc.optimize_result.as_dataframe(['fval', 'n_fval', 'n_grad',
                                          'n_hess', 'n_res', 'n_sres', 'time'])
```

[14]:

	fval	n_fval	n_grad	n_hess	n_res	n_sres	time
0	3.727426e-13	204	204	0	0	0	2.277306
1	4.994350e-13	200	200	0	0	0	2.517795
2	2.106192e-12	207	207	0	0	0	2.310546

(continues on next page)

(continued from previous page)

```
3   2.368548e-12     226     226       0       0       0  2.805628
4   2.407578e-12      95      95       0       0       0  1.029478
5   5.525261e-12     166     166       0       0       0  1.876049
6   1.038073e-11     249     249       0       0       0  2.957421
7   3.148873e-11     216     216       0       0       0  2.358464
8   9.991588e-11     170     170       0       0       0  2.292490
9   2.045469e-09     156     156       0       0       0  2.203442
10  2.414312e-09     187     187       0       0       0  2.385922
11  3.400104e-09     264     264       0       0       0  3.420013
12  4.277078e-09     158     158       0       0       0  2.009642
13  6.561378e-09     171     171       0       0       0  1.820322
14  2.038944e-08     181     181       0       0       0  2.000105
15  3.986579e+00     158     158       0       0       0  1.917194
16  3.986579e+00     106     106       0       0       0  1.417423
17  3.986579e+00      90      90       0       0       0  1.017045
18  3.986579e+00     138     138       0       0       0  1.473538
19  3.986579e+00     155     155       0       0       0  2.147978
```

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)
```

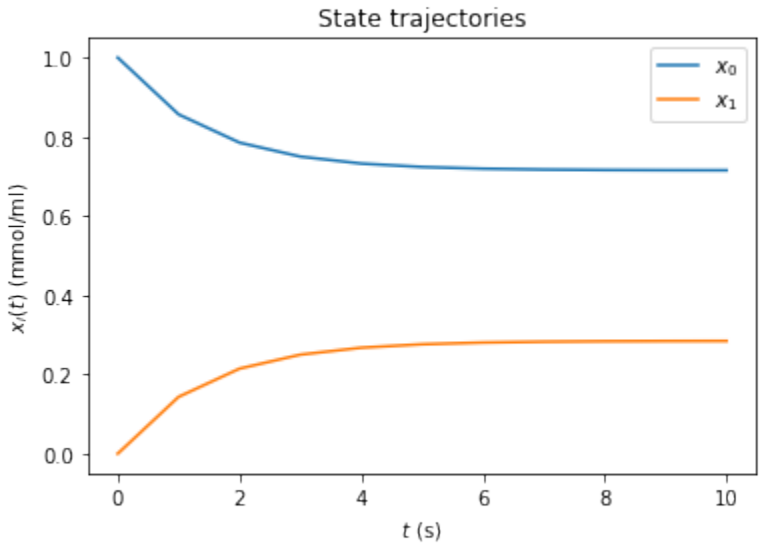
2.2.2 Load AMICI model

[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()
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

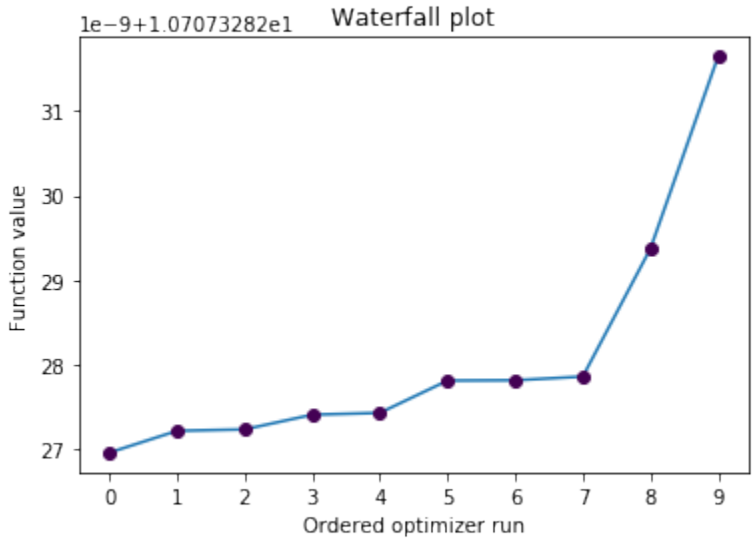
(continued from previous page)

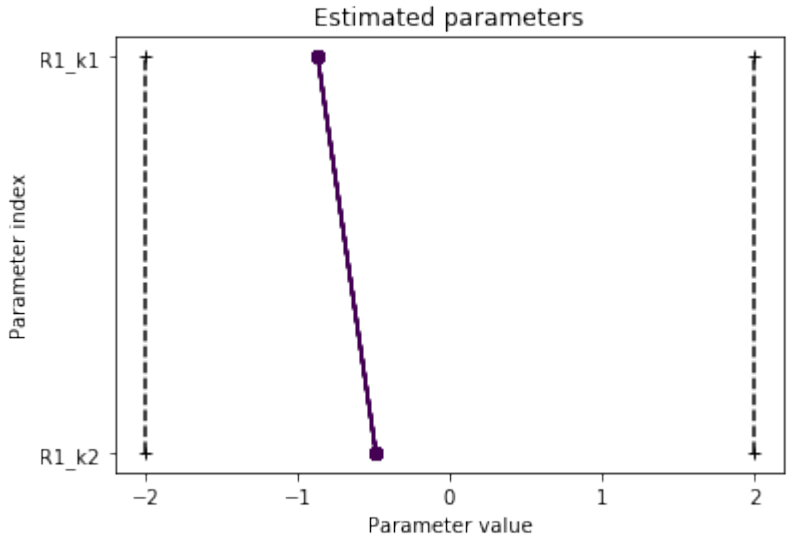
```
problem = pypesto.Problem(objective=objective,
                          lb=[-2,-2], ub=[2,2])

# 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 0x7f3dd2b98be0>
```





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)
```

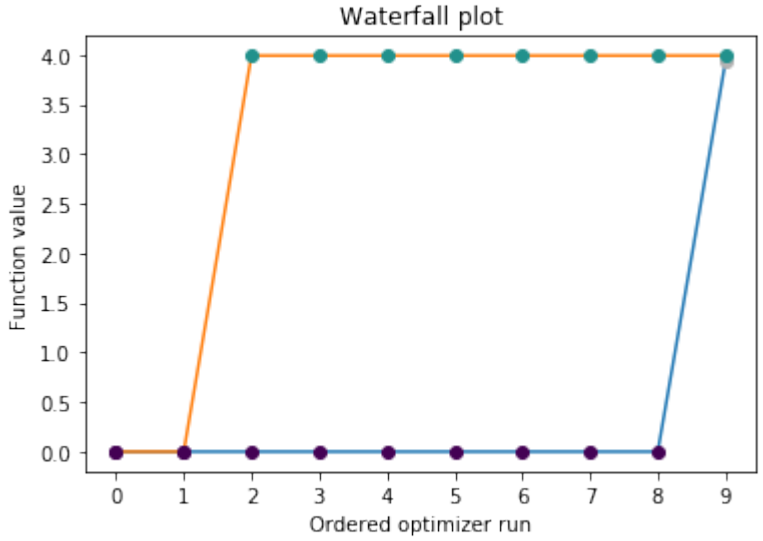
2.3.2 Optimize

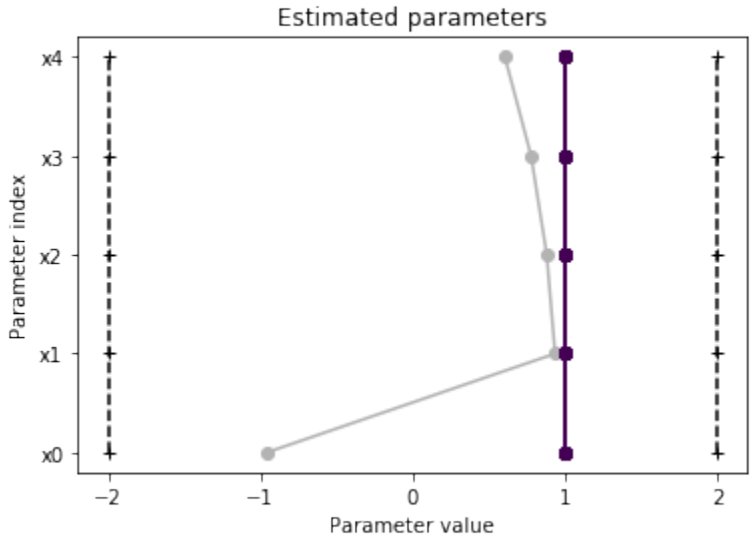
[3]:
```
optimizer = pypesto.ScipyOptimizer()
n_starts = 10

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

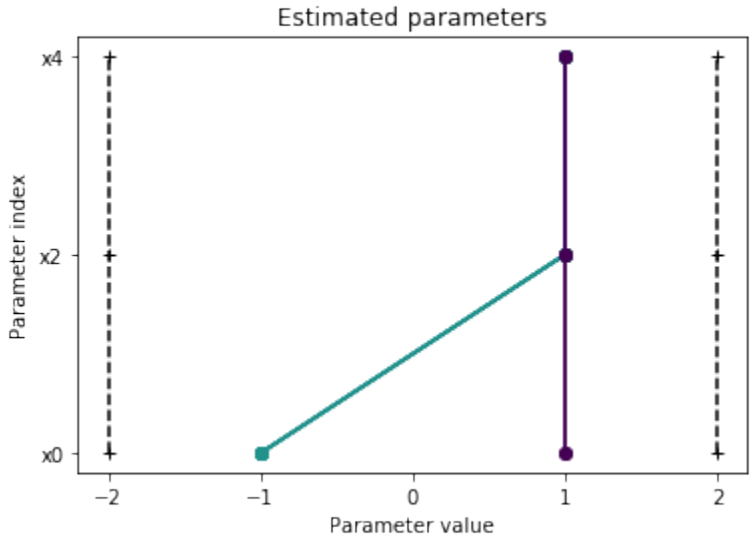
2.3.3 Visualize

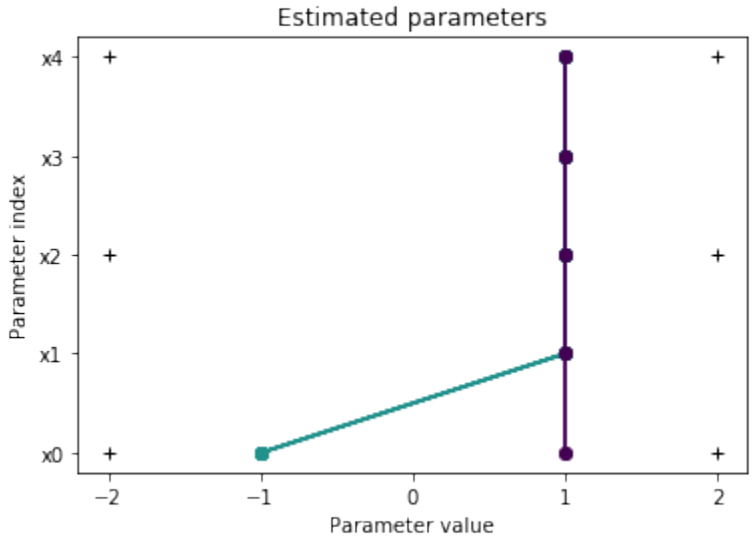
[4]:
```
fig, ax = plt.subplots()
pypesto.visualize.waterfall(result1, ax)
pypesto.visualize.waterfall(result2, ax)
pypesto.visualize.parameters(result1)
pypesto.visualize.parameters(result2)
pypesto.visualize.parameters(result2, free_indices_only=False)
```

[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f54c174de48>









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

[5]:
```
           fval                                                grad  \
0  6.647442e-15  [-9.471566723124671e-07, 2.1021349555995795e-0...
1  3.457526e-13  [-5.2800846471136764e-06, -3.3572245747142896e...
2  3.977885e-13  [2.849416601133684e-06, -1.631494200631376e-05...
3  5.131069e-13  [-1.7633333620389112e-05, 2.2702306040974444e-...
4  1.136030e-12  [-5.235885777815437e-06, 2.0492304488450417e-0...
5  1.408337e-12  [1.907331444582324e-05, 2.095463179570122e-05,...
6  2.901510e-11  [5.880961607340145e-05, -0.0001001725246473881...
7  1.178049e-10  [0.00034230906294693863, -0.000134688339183694...
8  3.892358e-10  [-0.0003125535760549204, -0.000540050381524208...
9  3.930839e+00  [1.0349712058044247e-05, 3.916386398739036e-06...

                                                   x
0  [0.9999999965730911, 0.9999999954969394, 0.999...
1  [0.9999999337976915, 0.9999998804645882, 0.999...
2  [1.0000000203239299, 1.0000000336259385, 1.000...
```

(continues on next page)

(continued from previous page)

```
3  [0.9999999656666152, 0.9999999752449003, 0.999...
4  [0.9999999555620058, 0.9999999239915387, 0.999...
5  [0.9999999800522175, 0.9999999123214094, 0.999...
6  [0.9999996346096833, 0.9999991203684541, 0.999...
7  [1.0000006892941309, 1.0000005262631377, 1.000...
8  [0.999999304112937, 0.9999993861314045, 1.0000...
9  [-0.9620510054878277, 0.9357393936941063, 0.88...
```

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

[6]:
```
           fval                                                grad  \
0  1.848017e-13  [1.161320144703573e-05, nan, 1.138211418469940...
1  1.950306e-13  [1.5797936376845116e-05, nan, 8.69990872045727...
2  3.989975e+00  [3.708520157630346e-08, nan, 8.708029498088168...
3  3.989975e+00  [-2.0752946916502424e-08, nan, 1.7918193731319...
4  3.989975e+00  [-7.013172895753428e-07, nan, 6.60039187652859...
5  3.989975e+00  [4.8148038946926874e-06, nan, 5.98716656044703...
6  3.989975e+00  [-3.6444866702289858e-06, nan, -1.258053682496...
7  3.989975e+00  [-1.9515809019488017e-05, nan, 2.2359531416176...
8  3.989975e+00  [-2.509148223639457e-05, nan, 1.94825262340907...
9  3.989975e+00  [3.7141079881841677e-05, nan, -2.5976819306825...

                                                   x
0  [1.0000000144803007, 1.0, 1.0000000113593952, ...
1  [1.0000000196981744, 1.0, 1.0000000086825436, ...
2  [-0.9949747467836382, 1.0, 1.0000000000869065,...
3  [-0.9949747468568538, 1.0, 1.0000000001788243,...
4  [-0.9949747477183607, 1.0, 1.0000000006587217,...
5  [-0.994974740735661, 1.0, 1.000000005975216, 1...
6  [-0.9949747514440345, 1.0, 0.9999999874445739,...
7  [-0.9949747715350857, 1.0, 1.000000022314901, ...
8  [-0.9949747785931701, 1.0, 1.0000000194436385,...
9  [-0.9949746998147513, 1.0, 0.9999999740750298,...
```

[ ]:
```
```

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
```

(continues on next page)

(continued from previous page)

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

```
/home/paul/Documents/pesto/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()
```

(continues on next page)

(continued from previous page)

```
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()

# 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:
47.97654457818761
```

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
```

(continues on next page)

(continued from previous page)

```
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)
```

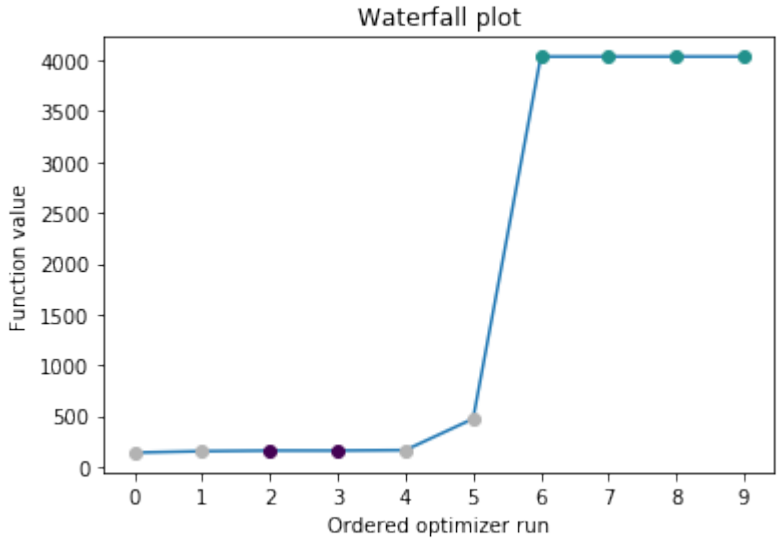
[17]:
```
# do the optimization
result = pypesto.minimize(problem=problem,
                          optimizer=optimizer,
                          n_starts=10) # 200
```

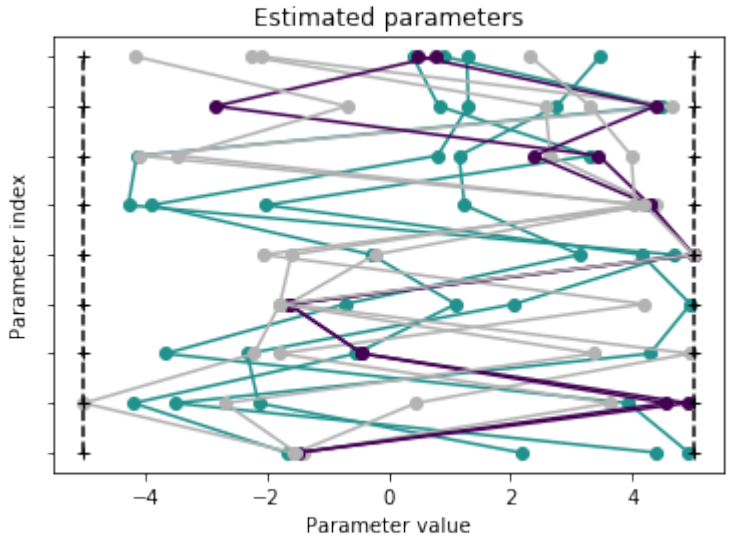
2.4.5 Visualization

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 0x7f697f679710>
```





[ ]:

2.5 Model import using the Petab format

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

import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Download benchmark models - Note: 200MB :(
!git clone --depth 1 https://github.com/LoosC/Benchmark-Models.git tmp/benchmark-
↪models || (cd tmp/benchmark-models && git pull)
```

(continues on next page)

(continued from previous page)

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

2.5.1 Manage PETAB model

[2]:
```
folder_base = "tmp/benchmark-models/hackathon_contributions_new_data_format/"
model_name = "Zheng_PNAS2012"
model_name = "Boehm_JProteomeRes2014"

petab_problem = petab.Problem.from_folder(folder_base + model_name)

# print(petab.lint.check_measurement_df(manager.measurement_df))
petab_problem.get_optimization_to_simulation_parameter_mapping()
```

[2]:
```
[['Epo_degradation_BaF3',
  'k_exp_hetero',
  'k_exp_homo',
  'k_imp_hetero',
  'k_imp_homo',
  'k_phos',
  'ratio',
  'sd_pSTAT5A_rel',
  'sd_pSTAT5B_rel',
  'sd_rSTAT5A_rel',
  'specC17']]
```

2.5.2 Import model to AMICI

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

model = importer.create_model()
print(model.getParameterScale())
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')
```

```
<amici.amici.ParameterScalingVector; proxy of <Swig Object of type 'std::vector< enum␣
↪amici::ParameterScaling,std::allocator< enum amici::ParameterScaling > > *' at␣
↪0x10cea20c0> >
Model parameters: ['Epo_degradation_BaF3', 'k_exp_hetero', 'k_exp_homo',␣
↪'k_imp_hetero', 'k_imp_homo', 'k_phos', 'ratio', 'noiseParameter1_pSTAT5A_rel',␣
↪'noiseParameter1_pSTAT5B_rel', 'noiseParameter1_rSTAT5A_rel', 'specC17']

Model const parameters: []

Model outputs:    ['observable_pSTAT5A_rel', 'observable_pSTAT5B_rel',␣
↪'observable_rSTAT5A_rel']
```

(continues on next page)

(continued from previous page)

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

2.5.3 Create objective function

[4]:
```
obj = importer.create_objective()

#print(amici.getDataObservablesAsDataFrame(obj.amici_model, edatas))
#print(edatas[0].fixedParametersPreequilibration)
#print(obj.dim, obj.x_names, len(obj.x_ids), obj.opt_to_sim_par_mapping)
#print(edatas[0].fixedParametersPreequilibration, edatas[0].fixedParameters)

print("Nominal parameter values:\n", petab_problem.x_nominal)

obj(petab_problem.x_nominal)
```

```
Nominal parameter values:
 [-1.5689175884, -4.9997048936, -2.2096987817, -1.7860065475, 4.9901140088, 4.
↪1977354885, 0.693, 0.5857552706, 0.8189828192, 0.49868440399999997, 0.107]
```

[4]:
```
138.22199805132536
```

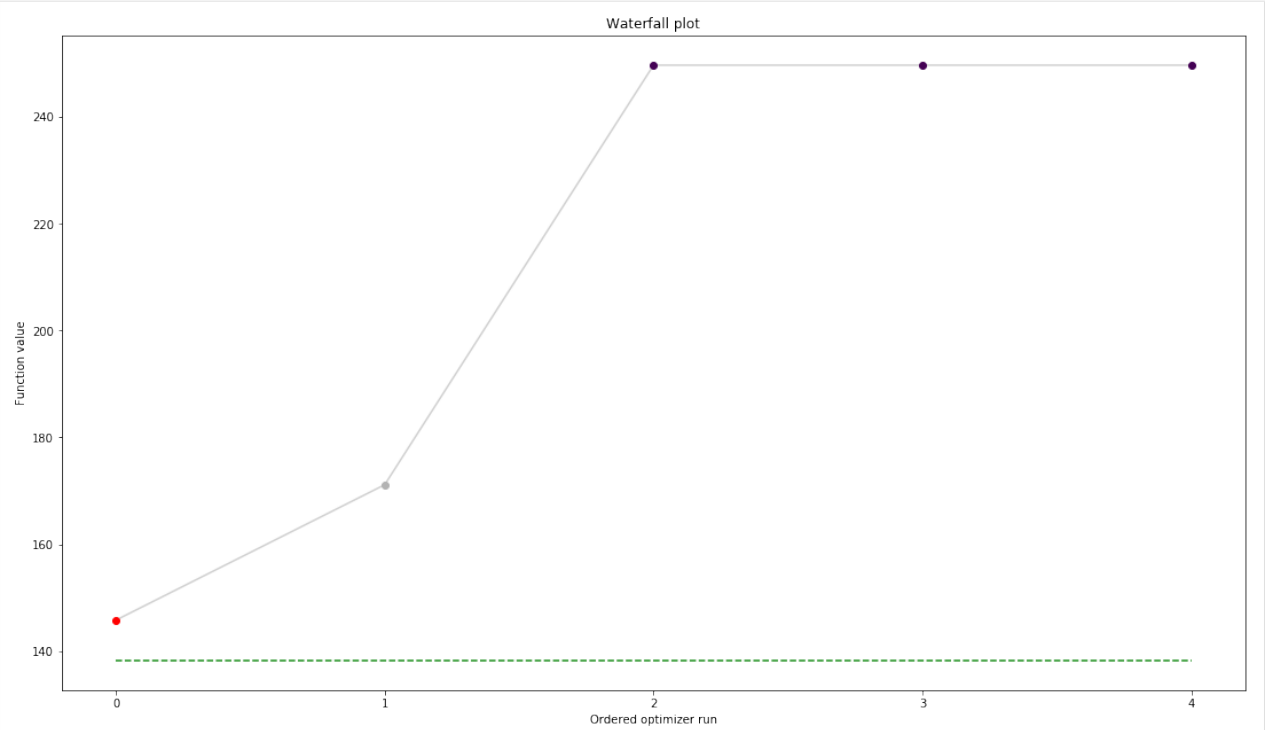
2.5.4 Run optimization

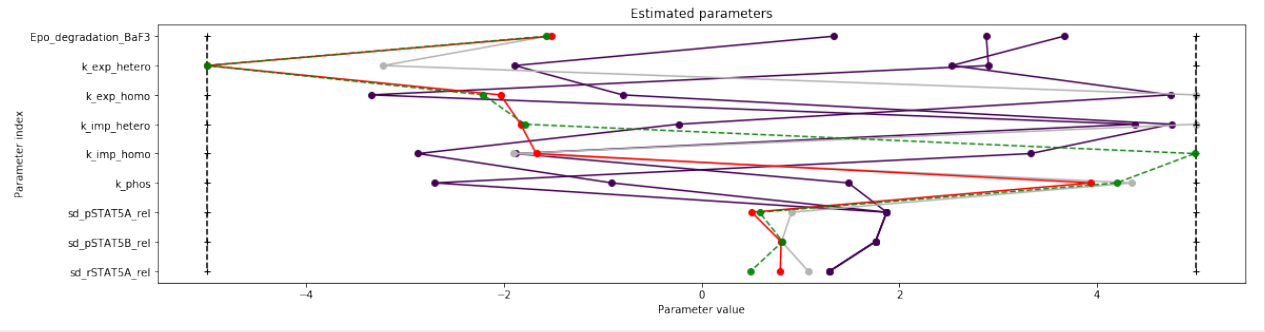
[5]:
```
optimizer = pypesto.ScipyOptimizer()

problem = importer.create_problem(obj)

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

# do the optimization
result = pypesto.minimize(problem=problem, optimizer=optimizer,
                          n_starts=5, engine=engine)
```

2.5.5 Visualize

[6]:
```
import pypesto.visualize

ref = pypesto.visualize.create_references(x=petab_problem.x_nominal, fval=obj(petab_
↪problem.x_nominal))

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

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

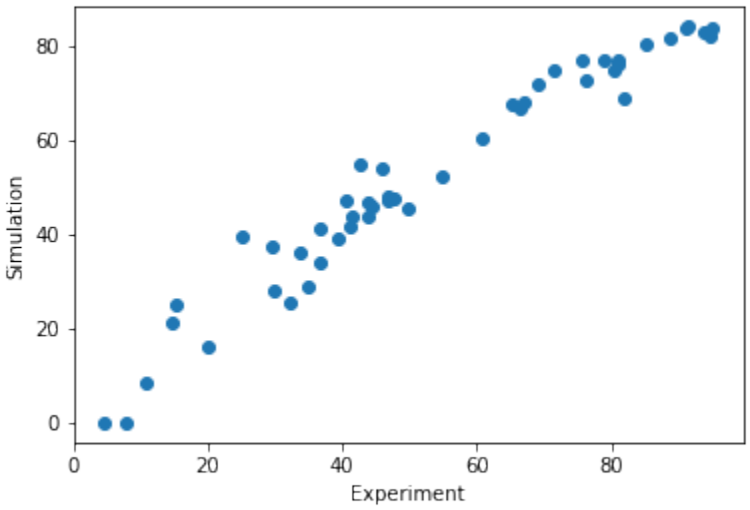
```
[145.75941368562795, 171.1340772947248, 249.7459755797753, 249.74599741458377, 249.
↪74599743902127]
```





[7]:
```
rdatas = obj(result.optimize_result.get_for_key('x')[0], return_dict=True)['rdatas']
df = importer.rdatas_to_measurement_df(rdatas)

plt.xlabel("Experiment")
plt.ylabel("Simulation")

# note: here we still assume that both files have the same order
plt.scatter(petab_problem.measurement_df['measurement'], df['measurement'])

# print(df)
```

[7]:
```
<matplotlib.collections.PathCollection at 0x1241a22b0>
```



[ ]:

2.6 Download the examples as notebooks

- `Rosenbrock`
- `Conversion reaction`
- `Fixed parameters`
- `Boehm model`
- `Petab import`

Note: Some of the notebooks have extra dependencies.

CHAPTER 3

Contribute

3.1 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
```

3.2 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.

3.2.1 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.

3.3 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 4

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.

4.1 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.

4.2 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.

4.2.1 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.

4.2.2 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 5

Objective

class pypesto.objective.**Objective**(*fun=None*, *grad=None*, *hess=None*, *hessp=None*, *res=None*, *sres=None*, *fun_accept_sensi_orders=False*, *res_accept_sensi_orders=False*, *x_names=None*, *options=None*)

Bases: `object`

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.

Parameters

- **fun** (*callable, optional*) – 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** (*callable, bool, optional*) – 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** (*callable, optional*) – 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** (*callable, optional*) –

  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** (*{callable, bool}, optional*) –

  Method for computing residuals, i.e. `res(x) -> array_like, shape(m,).`

- **sres** (*callable, optional*) – 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.

- **fun_accept_sensi_orders** (*bool, optional*) – Flag indicating whether fun takes sensi_orders as an argument. Default: False.

- **res_accept_sensi_orders** (*bool, optional*) – Flag indicating whether res takes sensi_orders as an argument. Default: False

- **x_names** (*list of str*) – 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.

- **options** (*pypesto.ObjectiveOptions, optional*) – Options as specified in pypesto.ObjectiveOptions.

history
For storing the call history. Initialized by the optimizer in reset_history().

Type pypesto.ObjectiveHistory

preprocess
Preprocess input values to __call__.

Type callable

postprocess
Postprocess output values from __call__.

Type callable

sensitivity_orders
Temporary variable to store requested sensitivity orders

Type tuple

Notes

preprocess, postprocess are configured in update_from_problem() and can be reset using the reset() method.

If fun_accept_sensi_orders resp. res_accept_sensi_orders is True, fun resp. res can also return dictionaries instead of tuples. In that case, they are expected to follow the naming conventions in `constants.py`. This is of interest, because when __call__ is called with return_dict = True, the full dictionary is returned, which can contain e.g. also simulation data or debugging information.

__call__ (*x*, *sensi_orders: tuple = (0,)*, *mode='mode_fun'*, *return_dict=False*)
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** (*array_like*) – The parameters for which to evaluate the objective function.

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

__init__ (*fun=None*, *grad=None*, *hess=None*, *hessp=None*, *res=None*, *sres=None*, *fun_accept_sensi_orders=False*, *res_accept_sensi_orders=False*, *x_names=None*, *options=None*)
Initialize self. See help(type(self)) for accurate signature.

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

Parameters

- **x** (*array_like*) – The parameters for which to evaluate the gradient.
- **x_indices** (*array_like, optional*) – List of index values for which to compute gradients. Default: all.
- **eps** (*float, optional*) – Finite differences step size. Default: 1e-5.
- **verbosity** (*int*) –

  Level of verbosity for function output 0: no output 1: summary for all parameters 2: summary for individual parameters

  Default: 1.
- **mode** (*str*) – Residual (MODE_RES) or objective function value (MODE_FUN, default) computation mode.

Returns **result** – gradient, finite difference approximations and error estimates.

Return type pd.DataFrame

check_sensi_orders (*sensi_orders*, *mode*)
Check if the objective is able to compute the requested sensitivities. If not, throw an exception.

finalize_history ()
Finalize the history object.

get_fval (*x*)
Get the function value at x.

get_grad (*x*)
Get the gradient at x.

get_hess (*x*)
Get the Hessian at x.

get_res (*x*)
Get the residuals at x.

get_sres (*x*)
Get the residual sensitivities at x.

has_fun

has_grad

has_hess

has_hessp

has_res

has_sres

static output_to_dict(*sensi_orders*, *mode*, *output_tuple*)
Convert output tuple to dict.

static output_to_tuple(*sensi_orders*, *mode*, ***kwargs*)
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).

reset()
Completely reset the objective, i.e. undo the modifications in update_from_problem().

reset_history(*index=None*)
Reset the objective history and specify temporary saving options.

Parameters **index** (*As in ObjectiveHistory.index.*) –

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

class pypesto.objective.**ObjectiveOptions**(*trace_record=False*, *trace_record_grad=True*, *trace_record_hess=False*, *trace_record_res=False*, *trace_record_sres=False*, *trace_record_chi2=True*, *trace_record_schi2=True*, *trace_all=True*, *trace_file=None*, *trace_save_iter=10*)

Bases: dict

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

Parameters

- **trace_record** (*bool, optional*) – 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** (*bool, optional*) – Flag indicating whether to record the gradient in the trace. Default: True.
- **trace_record_hess** (*bool, optional*) – Flag indicating whether to record the Hessian in the trace. Default: False.

- **trace_record_res** (*bool, optional*) – Flag indicating whether to record the residual in the trace. Default: False.
- **trace_record_sres** (*bool, optional.*) – Flag indicating whether to record the residual sensitivities in the trace. Default: False.
- **trace_record_chi2** (*bool, optional*) – Flag indicating whether to record the chi2 in the trace. Default: True.
- **trace_record_schi2** (*bool, optional*) – Flag indicating whether to record the chi2 sensitivities in the trace. Default: True.
- **trace_all** (*bool, optional*) – Flag indicating whether to record all (True, default) or only better (False) values.
- **trace_file** (*str or True, optional*) – Either pass a string here denoting the file name for storing the trace, or True, in which case the default file name "tmp_trace_{index}.dat" is used. A contained substring {index} is converted to the multistart index. Default: None, i.e. no file is created.
- **index, optional** (*trace_save_iter.*) – Trace is saved every tr_save_iter iterations. Default: 10.

__init__(*trace_record=False*, *trace_record_grad=True*, *trace_record_hess=False*, *trace_record_res=False*, *trace_record_sres=False*, *trace_record_chi2=True*, *trace_record_schi2=True*, *trace_all=True*, *trace_file=None*, *trace_save_iter=10*)
Initialize self. See help(type(self)) for accurate signature.

static assert_instance(*maybe_options*)
Returns a valid options object.

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

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

```
fval = 0.5 * sum(res**2)
```

which is the 'Linear' formulation in scipy.

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

class pypesto.objective.**AmiciObjective**(*amici_model*, *amici_solver*, *edatas*, *max_sensi_order=None*, *x_ids=None*, *x_names=None*, *mapping_par_opt_to_par_sim=None*, *mapping_scale_opt_to_scale_sim=None*, *guess_steadystate=True*, *n_threads=1*, *options=None*)
Bases: pypesto.objective.objective.Objective

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

__init__(*amici_model*, *amici_solver*, *edatas*, *max_sensi_order=None*, *x_ids=None*, *x_names=None*, *mapping_par_opt_to_par_sim=None*, *mapping_scale_opt_to_scale_sim=None*, *guess_steadystate=True*, *n_threads=1*, *options=None*)
Constructor.

Parameters

- **amici_model** (*amici.Model*) – The amici model.

- **amici_solver** (*amici.Solver*) – The solver to use for the numeric integration of the model.
- **edatas** (*amici.ExpData or list of amici.ExpData*) – The experimental data. If a list is passed, its entries correspond to multiple experimental conditions.
- **max_sensi_order** (*int, optional*) – Maximum sensitivity order supported by the model. Defaults to 2 if the model was compiled with o2mode, otherwise 1.
- **x_ids** (*list of str, optional*) – Ids of optimization parameters. In the simplest case, this will be the AMICI model parameters (default).
- **x_names** (*list of str, optional*) – See `Objective.x_names`.
- **mapping_par_opt_to_par_sim** (*optional*) – Mapping of optimization parameters to model parameters. List array of size n_simulation_parameters * n_conditions. The default is just to assume that optimization and simulation parameters coincide. The default is to assume equality of both.
- **mapping_scale_opt_to_scale_sim** (*optional*) – Mapping of optimization parameter scales to simulation parameter scales. The default is to just use the scales specified in the *amici_model* already.
- **guess_steadystate** (*bool, optional (default = True)*) – 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** (*int, optional (default = 1)*) – 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.
- **options** (*pypesto.ObjectiveOptions, optional*) – Further options.

apply_steadystate_guess(*condition_ix*, *x*)
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)]

get_bound_fun()
Generate a fun function that calls _call_amici with MODE_FUN. Defining a non-class function that references self as a local variable will bind the function to a copy of the current self object and will accordingly not take future changes to self into account.

get_bound_res()
Generate a res function that calls _call_amici with MODE_RES. Defining a non-class function that references self as a local variable will bind the function to a copy of the current self object and will accordingly not take future changes to self into account.

get_error_output(*rdatas*)

rebind_fun()
Replace the current fun function with one that is bound to the current instance

rebind_res()
Replace the current res function with one that is bound to the current instance

reset()
Resets the objective, including steadystate guesses

reset_steadystate_guesses()
Resets all steadystate guess data

set_par_sim_for_condition(*condition_ix*, *x*)
Set the simulation parameters from the optimization parameters for the given condition.

Parameters

- **condition_ix** (*int*) – Index of the current experimental condition.
- **x** (*array_like*) – Optimization parameters.

set_parameter_scale(*condition_ix*)

set_plist_for_condition(*condition_ix*)
Set the plist according to the optimization parameters for the given condition.

Parameters

- **condition_ix** (*int*) – Index of the current experimental condition.
- **x** (*array_like*) – Optimization parameters.

store_steadystate_guess(*condition_ix*, *x*, *rdata*)
Store condition parameter, steadystate and steadystate sensitivity in steadystate_guesses if steadystate guesses are enabled for this condition

class pypesto.objective.**AggregatedObjective**(*objectives*, *x_names=None*, *options=None*)
Bases: pypesto.objective.objective.Objective

This class allows to create an aggregateObjective from a list of Objective instances.

__init__(*objectives*, *x_names=None*, *options=None*)
Constructor.

Parameters **objectives** (*list*) – List of pypesto.objetive instances

aggregate_fun(*x*)

aggregate_fun_sensi_orders(*x*, *sensi_orders*)

aggregate_grad(*x*)

aggregate_hess(*x*)

aggregate_hessp(*x*)

aggregate_res(*x*)

aggregate_res_sensi_orders(*x*, *sensi_orders*)

aggregate_sres(*x*)

reset_steadystate_guesses()
Propagates reset_steadystate_guesses() to child objectives if available (currently only applies for amici_objective)

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

MODEL_BASE_DIR = 'amici_models'

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

petab_problem: petab.Problem Managing access to the model and data.

output_folder: str, optional Folder to contain the amici model. Defaults to './amici_models/model_name'.

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

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

create_edatas(*model=None*, *simulation_conditions=None*)
Create list of amici.ExpData objects.

create_model(*force_compile=False*)
Import amici model. If necessary or force_compile is True, compile first.

Parameters **force_compile** (*str, optional*) – 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.

create_objective(*model=None*, *solver=None*, *edatas=None*, *force_compile: bool = False*)
Create a pypesto.PetabAmiciObjective.

create_problem(*objective*)

create_solver(*model=None*)
Return model solver.

static from_folder(*folder*, *output_folder: str = None*, *model_name: str = None*)
Simplified constructor exploiting the standardized petab folder structure.

Parameters

- **folder** (*str*) – Path to the base folder of the model, as in petab.Problem.from_folder.
- **output_folder** (*See __init__.*) –
- **model_name** (*See __init__.*) –

rdatas_to_measurement_df(*rdatas*, *model=None*)
Create a measurement dataframe in the petab format from the passed *rdatas* and own information.

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

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

Return type pandas.DataFrame

CHAPTER 6

Problem

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

class pypesto.problem.**Problem**(*objective*, *lb*, *ub*, *dim_full=None*, *x_fixed_indices=None*, *x_fixed_vals=None*, *x_guesses=None*, *x_names=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** (*pypesto.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** (*int, optional*) – The full dimension of the problem, including fixed parameters.
- **x_fixed_indices** (*array_like of int, optional*) – Vector containing the indices (zero-based) of parameter components that are not to be optimized.
- **x_fixed_vals** (*array_like, optional*) – Vector of the same length as x_fixed_indices, containing the values of the fixed parameters.
- **x_guesses** (*array_like, optional*) – 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** (*array_like of str, optional*) – 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.

dim

The number of non-fixed parameters. Computed from the other values.

Type int

x_free_indices
Vector containing the indices (zero-based) of free parameters (complimentary to x_fixed_indices).

Type array_like of int

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*, *lb*, *ub*, *dim_full=None*, *x_fixed_indices=None*, *x_fixed_vals=None*, *x_guesses=None*, *x_names=None*)
Initialize self. See help(type(self)) for accurate signature.

fix_parameters(*parameter_indices*, *parameter_vals*)
Fix specified parameters to specified values

get_full_matrix(*x*)
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*, *x_fixed_vals=None*)
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*)
Map matrix from dim_full to dim, i.e. delete fixed indices.

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

get_reduced_vector(*x_full*)
Map vector from dim_full to dim, i.e. delete fixed indices.

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

normalize_input(*check_x_guesses=True*)
Reduce all vectors to dimension dim and have the objective accept vectors of dimension dim.

print_parameter_summary()
Prints a summary of what parameters are being optimized and what parameter boundaries are

unfix_parameters(*parameter_indices*)
Free specified parameters

CHAPTER 7

Optimize

pypesto.optimize.**minimize**(*problem*, *optimizer=None*, *n_starts=100*, *startpoint_method=None*, *result=None*, *engine=None*, *options=None*) → pypesto.result.Result

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

Parameters

- **problem** (*pypesto.Problem*) – The problem to be solved.
- **optimizer** (*pypesto.Optimizer*) – The optimizer to be used n_starts times.
- **n_starts** (*int*) – Number of starts of the optimizer.
- **startpoint_method** (*{callable, False}, optional*) – Method for how to choose start points. False means the optimizer does not require start points
- **result** (*pypesto.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.
- **options** (*pypesto.OptimizeOptions, optional*) – Various options applied to the multistart optimization.

class pypesto.optimize.**OptimizeOptions**(*startpoint_resample=False*, *allow_failed_starts=False*)

Bases: dict

Options for the multistart optimization.

Parameters

- **startpoint_resample** (*bool, optional*) – 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=False*, *allow_failed_starts=False*)

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

static assert_instance(*maybe_options*)
Returns a valid options object.

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

class pypesto.optimize.**OptimizerResult**(*x=None*, *fval=None*, *grad=None*, *hess=None*, *n_fval=None*, *n_grad=None*, *n_hess=None*, *n_res=None*, *n_sres=None*, *x0=None*, *fval0=None*, *trace=None*, *exitflag=None*, *time=None*, *message=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.

x
The best found parameters.

Type ndarray

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

Type float

grad, hess
The gradient and Hessian at x.

Type ndarray

n_fval
Number of function evaluations.

Type int

n_grad
Number of gradient evaluations.

Type int

n_hess
Number of Hessian evaluations.

Type int

exitflag
The exitflag of the optimizer.

Type int

message
Textual comment on the optimization result.

Type str

Notes

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

__init__(*x=None*, *fval=None*, *grad=None*, *hess=None*, *n_fval=None*, *n_grad=None*, *n_hess=None*, *n_res=None*, *n_sres=None*, *x0=None*, *fval0=None*, *trace=None*, *exitflag=None*, *time=None*, *message=None*)
Initialize self. See help(type(self)) for accurate signature.

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.

static get_default_options()
Create default options specific for the optimizer.

is_least_squares()

minimize(*problem*, *x0*, *index*)
" Perform optimization.

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

Use the SciPy optimizers.

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

static get_default_options()
Create default options specific for the optimizer.

is_least_squares()

minimize(*problem*, *x0*, *index*)

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

Use the Dlib toolbox for optimization.

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

static get_default_options()
Create default options specific for the optimizer.

is_least_squares()

minimize(*problem*, *x0*, *index*)

CHAPTER 8

Profile

pypesto.profile.**parameter_profile**(*problem*, *result*, *optimizer*, *profile_index=None*, *profile_list=None*, *result_index=0*, *next_guess_method=None*, *profile_options=None*, *optimize_options=None*) → pypesto.result.Result

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

Parameters

- **problem** (*pypesto.Problem*) – The problem to be solved.
- **result** (*pypesto.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** (*pypesto.Optimizer*) – The optimizer to be used along each profile.
- **profile_index** (*ndarray of integers, optional*) – array with parameter indices, whether a profile should be computed (1) or not (0) Default is all profiles should be computed
- **profile_list** (*integer, optional*) – 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** (*integer, optional*) – index from which optimization result profiling should be started (default: global optimum, i.e., index = 0)
- **next_guess_method** (*callable, optional*) – function handle to a method that creates the next starting point for optimization in profiling.
- **profile_options** (*pypesto.ProfileOptions, optional*) – Various options applied to the profile optimization.
- **optimize_options** (*pypesto.OptimizeOptions, optional*) – Various options applied to the optimizer.

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

Bases: dict

Options for optimization based profiling.

Parameters

- **default_step_size** (*float, optional*) – 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** (*float, optional*) – lower bound for the step size in adaptive methods
- **max_step_size** (*float, optional*) – upper bound for the step size in adaptive methods
- **step_size_factor** (*float, optional*) – 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** (*float, optional*) – maximum allowed drop of the posterior ratio between two profile steps
- **ratio_min** (*float, optional*) – lower bound for likelihood ratio of the profile, based on inverse chi2-distribution. The default corresponds to 95% confidence
- **reg_points** (*float, optional*) – number of profile points used for regression in regression based adaptive profile points proposal
- **reg_order** (*float, optional*) – maximum dregee of regriossion polynomial used in regression based adaptive profile points proposal
- **magic_factor_obj_value** (*float, optional*) – 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=0.01*, *min_step_size=0.001*, *max_step_size=1.0*, *step_size_factor=1.25*, *delta_ratio_max=0.1*, *ratio_min=0.145*, *reg_points=10*, *reg_order=4*, *magic_factor_obj_value=0.5*)

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

static create_instance(*maybe_options*)

Returns a valid options object.

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

class pypesto.profile.**ProfilerResult**(*x_path*, *fval_path*, *ratio_path*, *gradnorm_path=None*, *exitflag_path=None*, *time_path=None*, *time_total=0.0*, *n_fval=0*, *n_grad=0*, *n_hess=0*, *message=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 extent the compputation).

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)

Type ndarray

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

Type ndarray

ratio_path
The ratio of the posterior function along the profile.

Type ndarray

gradnorm_path
The gradient norm along the profile.

Type ndarray

exitflag_path
The exitflags of the optimizer along the profile.

Type ndarray

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

Type ndarray

time_total
The total computation time for the profile.

Type ndarray

n_fval
Number of function evaluations.

Type int

n_grad
Number of gradient evaluations.

Type int

n_hess
Number of Hessian evaluations.

Type int

message
Textual comment on the profile result.

Type str

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*, *fval_path*, *ratio_path*, *gradnorm_path=None*, *exitflag_path=None*, *time_path=None*, *time_total=0.0*, *n_fval=0*, *n_grad=0*, *n_hess=0*, *message=None*)
Initialize self. See help(type(self)) for accurate signature.

append_profile_point(*x*, *fval*, *ratio*, *gradnorm=nan*, *exitflag=nan*, *time=nan*, *n_fval=0*, *n_grad=0*, *n_hess=0*)
This function appends a new OptimizerResult to an existing ProfilerResults

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

CHAPTER 9

Sample

CHAPTER 10

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*)

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*) → list

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.

__init__()

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

add_profile(*profiler_result*, *i_parameter*)
Writes a profiler result to the result object at i_parameter.

Parameters

- **profiler_result** – The result of one (local) profiler run.
- **i_parameter** – integer specifying the parameter index

create_new_profile(*profiler_result=None*)
Append an profiler result to the result object.

Parameters

- **profiler_result** – The result of one (local) profiler run or None, if to be left empty
- **profile_list** (*integer*) – index specifying the list of profiles, to which we want to append

create_new_profile_list()
Append an profiler result to the result object.

get_current_profile(*i_parameter*)
Append an profiler result to the result object.

Parameters **i_parameter** – integer specifying the profile index

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.

CHAPTER 11

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 multistart itself is parallelized.

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*)
Execute all tasks in a simple for loop sequentially.

class pypesto.engine.**MultiProcessEngine**(*n_procs: int = None*)
Bases: pypesto.engine.base.Engine

Parallelize the task execution using the *multiprocessing.Pool* environment.

n_procs
The number of cores to use. Defaults to the number of cpus available on the system according to os.cpu_count(). The effectively used number of cores will be the minimum of n_procs and the number of tasks submitted (and the number of CPUs available).

Type int, optional

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

execute(*tasks*)

class pypesto.engine.**OptimizerTask**(*optimizer*, *problem*, *startpoint*, *j_start*, *options*, *handle_exception*)
Bases: pypesto.engine.task.Task

A multistart optimization task, performed in *pypesto.minimize*.

__init__(*optimizer*, *problem*, *startpoint*, *j_start*, *options*, *handle_exception*)
Create the task object.

Parameters

- **optimizer**(*the optimizer to use.*)–
- **problem**(*the problem to solve.*)–
- **startpoint**(*the point from which to start.*)–
- **j_start**(*the index of the multistart.*)–
- **options**(*options object applying to optimization.*)–
- **handle_exception** (*callable to apply when the optimization fails.*)–

execute()

Execute the task and return its results.

CHAPTER 12

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.**create_references**(*references=None*, *x=None*, *fval=None*, *color=None*, *legend=None*)

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.**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_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_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.**process_result_list**(*results*, *colors=None*, *legends=None*)
assigns colors and legends to a list of results, chekc 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_offset_y**(*offset_y*, *scale_y*, *min_val*)
compute offset for y-axis, depend on user settings

Parameters

- **offset_y** (*float*) – value for offsetting the later plotted values, in order to ensure positivity if a semilog-plot is used
- **min_val** (*float*) – 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_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.**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

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.**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.**profiles**(*results*, *ax=None*, *profile_indices=None*, *size=(18.5, 6.5)*, *reference=None*, *colors=None*, *legends=None*, *profile_list_id=0*)

Plot classical 1D profile plot (using the posterior, e.g. Gaussian like profile)

Parameters

- **results** (*list or pypesto.Result*) – list of pypesto.Result or single pypesto.Result
- **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 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, optional*) – Labels for line plots, one label per result object
- **profile_list_id** (*int, optional*) – index of the profile list to be used for profiling

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**profiles_lowlevel**(*fvals*, *ax=None*, *size=(18.5, 6.5)*, *color=None*, *legend_text=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*) – Including values need to be plotted.
- **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** (*str*) – Label for line plots

Returns **ax** – The plot axes.

Return type matplotlib.Axes

pypesto.visualize.**profile_lowlevel**(*fvals*, *ax=None*, *size=(18.5, 6.5)*, *color=None*, *legend_text=None*)

Lowlevel routine for plotting one profile, working with a numpy array only

Parameters

- **fvals** (*numeric list or array*) – Including values need to be plotted.
- **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

Returns **ax** – The plot axes.

Return type matplotlib.Axes

CHAPTER 13

Startpoint

Method 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.`**`uniform`**(***kwargs*)
Generate uniform points.

`pypesto.startpoint.`**`latin_hypercube`**(***kwargs*)
Generate latin hypercube points.

`pypesto.startpoint.`**`assign_startpoints`**(*n_starts*, *startpoint_method*, *problem*, *options*)
Assign startpoints.

CHAPTER 14

Release notes

14.1 0.0 series

14.1.1 0.0.7 (2019-03-21)

- Support noise models in Petab and Amici.
- Minor Petab update bug fixes.

14.1.2 0.0.6 (2019-03-13)

- Several minor error fixes, in particular on tests and steady state.

14.1.3 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.

14.1.4 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)

14.1.5 0.0.3 (2019-01-30)

- Import amici models and the petab data format automatically using pypesto.PetabImporter.
- Basic profiling routines.

14.1.6 0.0.2 (2018-10-18)

- Fix parameter values
- Record trace of function values
- Amici objective to directly handle amici models

14.1.7 0.0.1 (2018-07-25)

- Basic framework and implementation of the optimization

CHAPTER 15

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 16

Contact

Discovered an error? Need help? Not sure if something works as intended? Please contact us!

- Yannik Schälte: yannik.schaelte@gmail.com

CHAPTER 17

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 18

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

D

E

F

G

H

S

T

U

W

X