SUPPLEMENTARY DATA
for
Population-based Parameter Identification for Dynamical Models of Biological Networks with an Application to *Saccharomyces cerevisiae*

Ewelina Weglarz-Tomczak[1,*], Jakub M. Tomczak[2,*], Agoston E. Eiben[2] and Stanley Brul[1]

[1]Swammerdam Institute for Life Sciences, Faculty of Science, University of Amsterdam, the Netherlands
[2]Department of Computer Science, Faculty of Science, Vrije Universiteit Amsterdam, the Netherlands
Contact: ewelina.weglarz.tomczak@gmail.com, jmk.tomczak@gmail.com

POPI4SB: Implementation Details

1 Requirements

The code requires Python 3.X, and uses the following packages:

- PySCeS (http://pysces.sourceforge.net/)
- Numpy (https://numpy.org/)
- SciPy (https://www.scipy.org/)
- Scikit-Learn (https://scikit-learn.org/)

The key component is PySCeS that is used for running simulations of dynamic models.

2 Code structure

The code consists of the following components:

- algorithms: The directory contains two files:
  - general_method.py,
  - population_optimization_algorithms.py.
- simulators: The directory contains two files:
  - ode_simulator.py,
  - run_simulation.py
- utils: The directory contains three files:
  - config.py,
  - general.py,
  - pysces_utils.py.

`algorithms` is the core component of the code and will be discussed in more detail in the next subsection. In the directory `simulators` there are two files that are crucial for running simulations.

In `ode_simulator.py` the fitness (or objective) function is defined. Here, we use the ODE-based simulator that takes values of parameters and returns values of quantities in a model. In the second file, `run_simulation.py`, a code for running simulations with specified optimizers is contained. A user could use one of the algorithms available in `population_optimization_algorithms.py`. However, all information are included in separate files (see Configurations).

In the `utils` directory, auxiliary functions are included that define config files, and help to internally use `PySCes`.

The file `ppopi4sb.py` contains the code for running methods specified in configuration files for a given model and data.

3 Optimization methods

All population-based optimization methods are implemented in `population_optimization_algorithms.py`. All algorithms must inherit from `GeneralMethod` in `general_method.py`, a class that is initialized with two configurations (one for the method and one for the `PySCeS` model), and possesses two functions:

- `proposal`: This function must implement a method for generating new candidates using the old population.
- `step`: This function must implement the evaluation stage (i.e., calling the fitness function from `ode_simulator.py`) and the selection mechanism, i.e., selecting a new population for the next generation.

It is important to notice that since `run_simulation.py` automatically reads all optimizers implemented in `population_optimization_algorithms.py`, a new method could be easily added to the existing set of optimizers in `population_optimization_algorithms.py` without a need of modifying other files.

An example of adding a new method Here we present an example of an algorithm that perturbs old population by adding a Gaussian noise and selects these candidates as the new population.

```
class NewAlgorithm(GeneralMethod):
    def __init__(self, config_method, config_model):
        super().__init__(config_method, config_model)

    def proposal(self, theta, E=None):
        theta_new = theta + np.random.randn(theta.shape[0], theta.shape[1])

        return theta_new

    def step(self, theta, E_old, x_obs, mod, params):
        # (1. Generate)
        theta_new = self.proposal(theta, E=None)

        # (2. Evaluate)
        E_new = calculate_fitness(x_obs, theta_new, mod, params,\\
                    self.dist, self.config_model, self.config_method)

        # (3. Select)
        # Pick all new candidates as the new population.
        return theta_new, E_new
```

After adding this optimizer to `population_optimization_algorithms.py`, it is possible to run the algorithm by specifying the configuration for it. There is no need to change any other piece of code of the program.

4 Configurations

Configuration files play a crucial role in the program. First, they contain all necessary information about an optimization method, a `PySCeS` model, and the `PySCeS` solver. Second, they are `json` files that are easy to modify/create.

Optimizer configuration A configuration file for an optimizer must contain general information, such as:

- `method_name`: a name of a method (e.g., DE);
- `generations`: the number of generations;
- `pop_size`: the population size;
- `patience`: the number of generations such that if an optimizer does not find a new, better solution, the program is terminated;
- `clip_min` and `clip_max`: the minimal and maximum values that parameters are clipped at;
- `dist_name`: the distance name that determines the fitness function (current options: `norm` for the logarithm of the normal distribution, `abs` for the absolute value);
- `scale`: the value that determines the variance in the normal distribution for the fitness function;
- `num_exps`: the number of experiment repetitions.

Any other fields are specific for a method. For instance, for Differential Evolution we must specify:

- `gamma`: the value F in Eq. (4) in the main text;
- `CR`: the value of the uniform cross-entropy probability;
- `best`: whether only the best candidate solution should be perturbed.

REMARK: Please note that adding any new optimizer requires providing values of its hyperparameters only in the configuration file. There is no need to change the code elsewhere.

Model configuration A configuration file for a `PySCeS` model must contain general information, such as:

- `model_name`: a name of a model (e.g., wolf1);
- `sim_start`: the starting moment of experiments (typically 0);
- `sim_end`: the end time of experiments;
- `sim_points`: the number of points between the start and the end of experiments;
- `noise`: if no real data is used, how much noise should be added to synthetically generated data (e.g., 0.01);
- `indices`: which quantities (timeseries) should be compared with the observed data;
- `compartment`: whether compartment should be removed from parameters of a model;
- `low` and `high`: the lower and the upper bounds of possible values for parameters of a model.

REMARK: If you use real data, please be cautious with setting `sim_start`, `sim_end` and `sim_points` so that they match the real measurements.

Solver configuration In the program, we use `LSODA` solver for ODEs. The configuration file for the solver contains the following fields:

- `mode_sim_max_iter`: default 0;
- `lsoda_atol`: default 1.0e-012;
- `lsoda_rtol`: default 1.0e-007;
- `lsoda_mxordn`: default 12;
- `lsoda_mxords`: default 5;
- `lsoda_mxstep`: default 0.

For details, please read `PySCeS` docs. In `POPI4SB`, we provide the solver configuration (`solver.json`).

5 An example

We provide an example how to use `POPI4SB`. In the code repository, we provide the following files:

- `de.json`: the configuration file for Differential Evolution;
- `eda.json`: the configuration file for Estimation of Distribution Algorithm;
- `wolf.json`: the configuration file for a `PySCeS` model (`wolf1.psc`);
- `wolf1.psc`: the `PySCeS` model file (the glycolysis in baker's yeast described in this document);
- `wolf_x.npy`: the Numpy file containing an example of possible observed data.

After running `popi4sb.py`, please provide the directory where the example files are (e.g., `/example/`).