
PhysioFit

Release 3.3.1

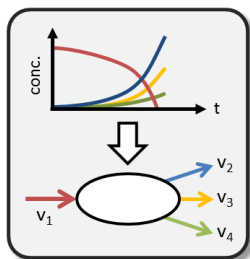
Loïc Le Grégam, Pierre Millard

Dec 11, 2023

USER DOCUMENTATION

1	Installation	2
1.1	Installation	2
1.2	Alternatives & updates	2
2	Quick start	3
2.1	Graphical user interface	3
2.2	Command line interface	4
2.3	Library	5
3	Method	6
3.1	Overview	6
3.2	Models	6
3.3	Flux calculation	6
3.4	Goodness-of-fit evaluation	7
3.5	Sensitivity analysis	7
4	Tutorial	8
4.1	Required input data file	8
4.2	Configuration file (yaml)	9
4.3	Flux calculation parameters	9
4.4	Output files	11
5	Models	12
5.1	Models shipped with PhysioFit	12
5.2	User-made models	13
6	How to cite	23
7	Library documentation	24
7.1	API reference	24
8	Frequently asked questions (FAQ)	28
8.1	How are fluxes calculated?	28
8.2	How many measurements should I use to calculate fluxes?	28
8.3	Can I calculate fluxes in case of missing values?	28
8.4	What units should be used for input data?	28
8.5	What are the flux units?	29
8.6	An error has been raised. What should I do?	29
8.7	What parameters values should I use?	29
8.8	How can I check if my data have been fitted correctly?	29
8.9	What is a χ^2 test?	29

8.10	My data hasn't been correctly fitted. Why?	30
8.11	I cannot start PhysioFit graphical user interface, can you help me?	30
8.12	I have develop a new model, can you include it in PhysioFit distribution?	31
8.13	I would like a new feature.	31
9	License	32
	Python Module Index	33
	Index	34



PhysioFit is a scientific tool designed to quantify cell growth parameters and uptake & production fluxes

Fluxes are estimated using mathematical models by fitting time-course measurements of the concentration of cells and extracellular substrates and products. PhysioFit is shipped with some common growth models, and additional tailor-made models can be implemented by users.

PhysioFit includes the following features:

- **calculation of growth rate and extracellular (uptake and production) fluxes,**
- **a set of steady-state and dynamic models** are shipped with PhysioFit,
- **tailor-made models** can be constructed by users,
- Monte-Carlo sensitivity analysis to **estimate the precision of the calculated fluxes,**
- **evaluation of the goodness of fit and visual inspection of the fitted curves,**
- shipped as a **library** with both a **graphical** and a **command line** interface,
- **open-source, free and easy to install** everywhere where Python 3 and pip run,
- **biologist-friendly.**

It is one of the routine tools that we use at the [MetaSys team](#) and [MetaToul platform](#) to calculate fluxes.

The code is open-source, and available on [GitHub](#) under a [GPLv3 license](#).

This documentation is available on Read the Docs (<https://physiofit.readthedocs.io>) and can be downloaded as a [PDF file](#).

INSTALLATION

1.1 Installation

PhysioFit requires Python 3.9 or higher and run on all platforms supporting Python3 (Windows, MacOS and Linux). If you do not have a Python environment configured on your computer, we recommend that you follow the instructions from [Anaconda](#).

To install PhysioFit using Python's built-in installer, you can just run the following command in a terminal:

```
pip install physiofit
```

Tip: We recommend the creation of isolated environments for each python tool you install in your system using the python built-in [venv](#) package or [Anaconda](#).

If this method does not work, you should ask your local system administrator or the IT department “how to install a Python 3 package from PyPi” on your computer.

PhysioFit is freely available and is distributed under open-source license at <http://github.com/MetaSys-LISBP/>.

1.2 Alternatives & updates

If you know that you do not have permission to install software system-wide, you can install PhysioFit into your user directory using the `--user` flag:

```
pip install --user physiofit
```

This does not require any special privileges.

Once the package is installed, you can update it using the following command:

```
pip install -U physiofit
```

Alternatively, you can also download all sources in a tarball from [GitHub](#), but it will be more difficult to update PhysioFit later on.

QUICK START

In this section we will explain how to launch your first job once PhysioFit has been installed onto your system.

See also:

- If you have already used PhysioFit and are looking for a more in-depth tutorial, check out the [Tutorial](#) section.

2.1 Graphical user interface

To open the Graphical User Interface, type in a terminal (e.g. Anaconda Prompt if installed on Windows):

```
physiofit
```

If you have installed the package in a specific environment, make sure to activate this environment before starting PhysioFit.

PhysioFit interface will open in a new browser window:

Welcome to PhysioFit

Select a task to execute

Calculate extracellular fluxes

Load a data file or a json configuration file



Drag and drop file here

Limit 200MB per file

Browse files



KEIO_ROBOT6_1.tsv 0.7KB



Model



--

Select an input file (which can be a `tsv` file containing the data or a `yaml` configuration file containing the run parameters and a path towards the data, see [Tutorial](#) for more details), select a model, modify the calculation parameters according to your data, and click on **Run flux calculation**. PhysioFit proceeds automatically to the flux calculation and display its progress and possibly important messages such as errors. The output of the calculations (i.e. fluxes and associated statistics) will be written in a text file as will the statistical test results, while plots will be generated individually for each metabolite (`svg` files) and combined in a multi-page `pdf` file. If multiple experiments were included in the input data, a summary (`csv` file) will also be generated. See [Output files](#) for more details.

2.2 Command line interface

To process your data, type in a terminal:

```
physiofit [command line options]
```

Here after the available options with their full names are enumerated and detailed.

```
usage: physiofit [-h] [-d DATA] [-c CONFIG] [-m MODEL] [-g] [--list] [-v]
                [-op OUTPUT_PDF] [-of OUTPUT_FLUXES] [-os OUTPUT_STATS]
                [-oc OUTPUT_CONFIG] [-or OUTPUT_RECAP] [-oz OUTPUT_ZIP]
```

2.2.1 Named Arguments

-d, --data	Path to data file in tabulated format (txt or tsv)
-c, --config	Path to config file in yaml format
-m, --model	Which model should be chosen. Useful only if generating related config file
-g, --galaxy	Is the CLI being used on the galaxy platform Default: False
--list	Return the list of models in model folder Default: False
-v, --debug_mode	Activate the debug logs Default: False
-op, --output_pdf	Path to output the pdf file containing plots
-of, --output_fluxes	Path to output the flux results
-os, --output_stats	Path to output the khi ² test
-oc, --output_config	Path to output the yaml config file
-or, --output_recap	Path to output the summary
-oz, --output_zip	Path to export zip file

2.3 Library

PhysioFit is also available as a library (a Python module) that you can import directly in your Python scripts:

```
import physiofit
```

See also:

Have a look at our [API](#) if you are interested in this feature.

3.1 Overview

Fluxes (exchange fluxes of a metabolite M_i , qM_i ; growth rate, μ), initial concentrations of species (biomass, X ; metabolites, M_i) and possibly other growth parameters (e.g. lag time) are estimated by fitting time-course measurements of metabolite and biomass concentrations, as detailed below.

Flux values provided by PhysioFit correspond the best fit. A global sensitivity analysis (Monte-Carlo approach) is available to evaluate the precision of the estimated fluxes (mean, median, standard deviation, 95% confidence intervals), plots are generated for visual inspection of the fitting quality, and a χ^2 test is performed to assess the statistical goodness of fit.

3.2 Models

Models are at the heart of the flux calculation approach implemented in PhysioFit. A flux model contains i) equations that describe the dynamics of biomass and metabolite concentrations as function of different parameters (used to simulate time-course metabolite concentrations) and ii) the list of all parameters (including fluxes) with their (default) initial values and bounds (used for flux calculation).

Different models are shipped with PhysioFit, and tailor-made models can be provided by users, as detailed in the [Models](#) section.

3.3 Flux calculation

First, PhysioFit construct a model that used to simulate the dynamics of the concentration of biomass and metabolites (substrates and products) provided in the input data. Model parameters (such as fluxes, growth rate, and initial concentrations of biomass and metabolites) are then estimated by fitting experimental metabolite and biomass dynamics. PhysioFit minimizes the following cost function:

$$residuum = \sum_i \left(\frac{sim_i - meas_i}{sd_i} \right)^2$$

where *sim* is the simulated data, *meas* denotes measurements, and *sd* is the standard deviation on measurements.

For this optimization step, PhysioFit uses the Scipy's Differential evolution method to approximate the solution, and the best solution is polished using the L-BFGS-B method (see [scipy.optimize](#) for more information on the optimization process).

3.4 Goodness-of-fit evaluation

PhysioFit performs a χ^2 test to assess the goodness of fit. Have a look at the *Frequently asked questions (FAQ)* section for more details on the interpretation of the khi2 test results.

3.5 Sensitivity analysis

To determine the precision on the fit and on the estimated parameters (including fluxes), PhysioFit performs a Monte Carlo analysis. Briefly, PhysioFit generates several datasets by adding noise to the dynamics simulated from the best fit, and calculated fluxes and other growth parameters for each of these synthetic datasets. This enables PhysioFit to compute statistics (mean, median, standard deviation and 95% confidence interval) for each parameter (including fluxes). We recommend always running a sensitivity analysis when using PhysioFit.

4.1 Required input data file

The input data must be a tabulated file (.tsv extension) with the following structure:

experiments	time	X	Glucose
Condition1	1	0.4	13
Condition1	2	0.6	12
Condition2	1	0.3	15
...	...	...	...

Columns **time** and **X** (biomass concentration) are mandatory, as is at least one metabolite column (**Glucose** in this example). If the biomass and metabolite concentrations were sampled at different moments, you can still group them together in the same table (with an empty value when no data is available). Column “experiments” contains the name of the condition/experiment, which must be provided even if only one condition/experiment is being analyzed.

Note: Flux units depend on the units of time and concentrations (of biomass and metabolites) provided in the input data file. For instance, if biomass units are in grams of cell dry weight by liter ($\text{g}_{\text{CDW}}/\text{L}$), metabolite concentrations are in millimolar (mM) and time is in hours (h), the estimated fluxes will be in $\text{mmol}/\text{g}_{\text{CDW}}/\text{h}$. Units should thus be carefully selected, and calculated fluxes must be interpreted consistently with the concentration units.

Warning: The “experiments” column must only contain letters (no numbers) and is case-sensitive!

Warning: To limit any numerical instabilities, we recommend providing values in a range not too far from unity (e.g. if a metabolite concentration is 2 mM, provide the value directly in mM and not as 0.002 M). The concentration of different metabolites can be provided using different units, but a single unit must be used for a given metabolite.

4.2 Configuration file (yaml)

The yaml configuration file contains all parameters required to calculate fluxes for a given experiment, and is thus key to ensure **reproducibility** of the flux calculation process. A configuration file is generated automatically by PhysioFit during flux calculation; it can also be created or edited manually.

Here is an example of a configuration file:

```
iterations: 100
mc: true
model:
  bounds:
    Ace_M0: (1e-06, 50)
    Ace_q: (-50, 50)
    Glc_M0: (1e-06, 50)
    Glc_q: (-50, 50)
    X_0: (0.001, 10)
    growth_rate: (0.001, 3)
  model_name: Steady-state batch model
  parameters_to_estimate:
    Ace_M0: 1
    Ace_q: 1
    Glc_M0: 1
    Glc_q: 1
    X_0: 1
    growth_rate: 1
path_to_data: ../data/test-data_single_exp.tsv
sds:
  Ace: 0.2
  Glc: 0.2
  X: 0.2
```

For a description of all calculation parameters, check the section below.

4.3 Flux calculation parameters

The first parameter is related to the **sensitivity analysis (Monte-Carlo)**, for which a check box indicates whether PhysioFit should estimate (or not) the precision on calculated fluxes. If checked, PhysioFit will let you select the number of Monte Carlo iterations. A higher number of iterations means more accurate confidence intervals on the estimated parameters, but will slow down calculations. The default number of iterations (100) is sufficient in most situations.

Next, the list of run parameters that you can tweak depends on the model you have selected. There are two types of parameters: i) **parameters to estimate**, for which you can change the solution space bounds and the initial values and ii) **fixed parameters** for which you can change the value. Here is an example when the steady-state without lag and with metabolite degradation model is selected:

Parameters 

Parameters to estimate

Parameter Name	Parameter Value	Lower Bound	Upper Bound
X _O	1	0.001	10
mu	1	0.001	3
Glc _q	1	-50	50
Glc _{M0}	1	1e-06	50
Ace _q	1	-50	50
Ace _{M0}	1	1e-06	50

Fixed parameters: Degradation

Parameter Name	Parameter Value
Glc	0
Ace	0

Standard deviation on measurements. As detailed in the methods section, reducing the standard deviation will increase the cost of the corresponding data during the optimization, thereby forcing an improvement of the fit accuracy for this measurements, but degrading the goodness-of-fit for the other measurements.

Finally, **Verbose logs**: Should debug information be written in log file. Useful in case of trouble (please join it to the issue on github). Default: False

Other default initial values are given by the model.

Note: Initial values and bounds should be carefully chosen. Ideally, initial values should be in the range of values used in the experiment. Well-defined bounds will enhance robustness and speed of the flux calculation process. The default bounds are sufficient in most cases, but may still be defined by the user when needed (e.g. the higher bound on

initial metabolite concentrations should be increased if the initial concentration of substrate is higher than 50, since it is the maximal value allowed by default.).

4.4 Output files

The following files are generated by PhysioFit in the output directory:

- `config_file.yaml` configuration file containing all parameters used for the last run.
- `flux_results.tsv` flux calculation results, i.e. fluxes and initial metabolite concentrations for the best fit, with associated precision.
- `stat_results.tsv` results from the χ^2 statistical test.
- `log.txt` run log file containing information on how the run went.
- `plots.pdf` plots of simulated and measured data.
- A number of `.svg` files: individual plots of simulated and measured data.
- `summary.csv`: summary of flux results (useful when multiple experiments are analyzed)

Note:

- When the data file (and not a `yaml` configuration file) is directly used as input in the Graphical User Interface, PhysioFit cannot get the path directly from the file metadata. For this reason, `path_to_data` is set to `None` in the generated `config_file.yaml`.
 - PhysioFit silently overwrites (results and log) files if they already exist. So take care to copy your results elsewhere if you want to protect them from overwriting.
-

The quality of the fit must be checked before interpreting the estimated fluxes. You can check the `test_results.tsv` file, which contains the detailed χ^2 statistical test results and a clear status on the quality of the fit (based on a 95% confidence interval). The generated plots also help to visualize how accurately the simulated data fits the experimental measurements. Finally, the confidence intervals estimated using the Monte-Carlo approach provides quantitative information on the precision of the estimated fluxes (mean, median, standard deviation and 95% confidence interval). Have a look to the [Frequently asked questions \(FAQ\)](#) section for help on interpreting the statistical results.

MODELS

Some common flux calculation models are provided with PhysioFit, and users can also build their own models.

Note: We would be happy to broaden the types of models shipped with PhysioFit. If you have developed a new model, it might be usefull and valuable to the fluxomics community! Please, keep in touch with us to discuss and see if we can include your model in the built-in models shipped with PhysioFit! :)

5.1 Models shipped with PhysioFit

5.1.1 Steady-state models

As detailed in Peiro et al. (2019), PhysioFit includes a steady-state model that accounts for i) non enzymatic degradation of some metabolites and ii) growth lag. This model is described by the following system of ordinary differential equations:

$$\frac{dX}{dt} = \begin{cases} 0 & \text{if } t < t_{lag} \\ \mu \cdot X & \text{ow.} \end{cases} \quad (\text{eq. 1})$$

where μ is growth rate, X is the biomass concentration, and t_{lag} is the lag time.

$$\frac{dM_i}{dt} = \begin{cases} -k_i \cdot M_i & \text{if } t < t_{lag} \\ -k_i \cdot M_i + X \cdot qM_i & \text{ow.} \end{cases} \quad (\text{eq. 2})$$

where k_i is the first-order degradation constant of the metabolite M_i and qM_i is its exchange (uptake or production) flux.

The flux qM_i is positive (negative) when M_i is produced (consumed). The sign of qM_i can thus be used to automatically identify products and substrates.

Integrating equations 1-2 provides the following analytical functions:

$$X(t) = \begin{cases} X_0 & \text{if } t < t_{lag} \\ X_0 \cdot e^{\mu \cdot (t - t_{lag})} & \text{ow.} \end{cases} \quad (\text{eq. 3})$$

$$M_i(t) = \begin{cases} M_i^0 \cdot e^{-k_i \cdot t} & \text{if } t < t_{lag} \\ qM_i \cdot \frac{X_0}{\mu + k_i} \cdot (e^{\mu \cdot (t - t_{lag})} - e^{-k_i \cdot (t - t_{lag})}) + M_i^0 \cdot e^{-k_i \cdot t} & \text{ow.} \end{cases} \quad (\text{eq. 4})$$

Three additional models are derived from this general model (without degradation, without lag phase, and without degradation nor lag phase).

Indeed, without a lag phase (i.e. $t_{lag} = 0$), equations 3-4 simplifies to:

$$X(t) = X_0 \cdot e^{\mu \cdot t} \quad (\text{eq. 5})$$

$$M_i(t) = qM_i \cdot \frac{X_0}{\mu + k_i} \cdot (e^{\mu \cdot t} - e^{-k_i \cdot t}) + M_i^0 \cdot e^{-k_i \cdot t} \quad (\text{eq. 6})$$

In the absence of degradation (i.e. $k = 0$), equation 4 simplifies to:

$$M_i(t) = \begin{cases} M_i^0 & \text{if } t < t_{lag} \\ qM_i \cdot \frac{X_0}{\mu} \cdot (e^{\mu \cdot (t-t_{lag})} - 1) + M_i^0 & \text{ow.} \end{cases} \quad (\text{eq. 7})$$

In the absence of both degradation and lag (i.e. $t_{lag} = 0$ and $k = 0$), equations 3-4 simplifies to:

$$X(t) = X_0 \cdot e^{\mu \cdot t} \quad (\text{eq. 8})$$

$$M_i(t) = qM_i \cdot \frac{X_0}{\mu} \cdot (e^{\mu \cdot (t-t_{lag})} - 1) + M_i^0 \quad (\text{eq. 9})$$

5.1.2 Dynamic model

We also implemented a dynamic model where fluxes and growth are represented by Monod kinetics, for one substrate and one product.

Time course concentrations of biomass (X), substrate (S) and product (P) are described by the following system of ordinary differential equations (ODEs):

$$\frac{dS}{dt} = -X \cdot q_S \quad (\text{eq. 10})$$

$$\frac{dX}{dt} = q_S \cdot \text{yield}_{biomass} \quad (\text{eq. 11})$$

$$\frac{dP}{dt} = q_S \cdot \text{yield}_{product} \quad (\text{eq. 12})$$

where q_S is the substrate uptake flux, $\text{yield}_{biomass}$ is the biomass yield and $\text{yield}_{product}$ is the product yield.

The dependence of the substrate uptake flux on the substrate concentration is expressed in this model by the Monod rate law:

$$q_S = q_S^{max} \cdot \frac{S}{K_M + S} \quad (\text{eq. 13})$$

where q_S^{max} is the maximal substrate uptake rate and K_M is the “half-velocity constant” (the value of S at which $\frac{\mu}{\mu_{max}} = 0.5$).

5.2 User-made models

5.2.1 Overview

Since PhysioFit 3.0.0, users can create and implement their own models to calculate fluxes and other growth parameters for any biological system. This section explains how to write your first model, how to test the model and how to implement it on your PhysioFit instance.

5.2.2 Build a template

To implement user-made models, PhysioFit leverages Python's object model to create classes that inherit from an Abstract Base Class and that handles all the heavy-lifting for implementation. A simple set of rules enables users to use their model in PhysioFit.

The model must be a class located in a dedicated module. Start by opening a text file using your IDE (Integrated Development Environment), and enter the following structure in the file:

```
from physiofit.models.base_model import Model

class ChildModel(Model):

    def __init__(self, data):
        pass

    def get_params(self):
        pass

    @staticmethod
    def simulate():
        pass

if __name__ == "__main__":
    pass
```

This is the base template to build your model. Methods `get_params` (to initialize and return model parameters) and `simulate` (to simulate metabolite dynamics for a given set of parameters) are mandatory. Additional methods are allowed if needed (e.g. to carry out intermediary steps for the simulation).

5.2.3 Populate the template

The first attribute to add in your model's `__init__` method is the model name. We strongly advise to choose a name that helps the user understand what the model is destined to simulate. You must also add two other attributes: the free parameters that PhysioFit will estimate & the fixed parameters provided by users. Finally, you must also call the `super().init(data)` method to inherit the logic from the base class:

```
from physiofit.models.base_model import Model

class ChildModel(Model):

    def __init__(self, data):
        super().__init__(data)
        self.model_name = "Tutorial model"
        self.parameters_to_estimate = None
        self.fixed_parameters = None

    def get_params(self):
        pass

    @staticmethod
    def simulate():
        pass
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    pass
```

Note: If your model does not contain fixed parameters, you must still initialize the attribute as `None`. This is considered good practice.

We can now check that the model can be initialized properly. Use the block at the end of the file for testing purposes. Here is an example of how you can test the model:

```
if __name__ == "__main__":

    test_data = pd.DataFrame(
        {
            "time": [0, 1, 2, 3],
            "experiments": ["A", "A", "A", "A"],
            "X": [0.5, 0.8, 1.2, 1.8],
            "Glucose": [12, 11.6, 11, 10.2]
        }
    )

    model = ChildModel(data=test_data)
    print(model)
```

If you now run the file, you should have a standard output in your console that looks like:

```
Selected model: Tutorial model
Model data:
   time experiments    X  Glucose
0     0             A  0.5    12.0
1     1             A  0.8    11.6
2     2             A  1.2    11.0
3     3             A  1.8    10.2
Experimental matrix:
[['A' 0.5 12.0]
 ['A' 0.8 11.6]
 ['A' 1.2 11.0]
 ['A' 1.8 10.2]]
Time vector: [0 1 2 3]
Name vector: ['X', 'Glucose']
Biomass & Metabolites: ['Glucose']
Parameters to estimate: None
Fixed parameters: None
Bounds: None
```

The next step is to define the parameters (used for simulations and optimization). PhysioFit supports two types of parameters (**parameters to estimate** and **fixed parameters**) which are detailed below.

Free parameters

The free parameters are the parameters that will be estimated by PhysioFit, and thus that require defining bounds and initial values to be initialized. The list of parameters and their initial (default) values must be returned by the `get_params` method:

```
from physiofit.models.base_model import Model

class ChildModel(Model):

    def __init__(self, data):
        super().__init__(data)
        self.model_name = "Tutorial model"
        self.parameters_to_estimate = None
        self.fixed_parameters = None

    def get_params(self):

        # Parameters are given in a dictionary, where the key is
        # the parameter name and the value is a number that will
        # be the initial value for the optimization process

        self.parameters_to_estimate = {
            "BM_0": 1,
            "growth_rate": 1
        }

        # Do the same for all metabolite parameters to estimate
        # using a for loop:

        for metabolite in self.metabolites:
            self.parameters_to_estimate.update(
                {
                    f"{metabolite}_flux" : 1,
                    f"{metabolite}_init_value" : 1
                }
            )

    @staticmethod
    def simulate():
        pass
```

Note: For a given model, the number of metabolites may vary depending on the experiment, hence the metabolite-dependent parameters can be automatically defined in this function (as illustrated here using a for loop).

The next step is to define the default bounds used for the optimization process (these bounds can be changed in the GUI). The bounds are a class of objects that handle the logic and checks. They are derived from the python `dict` base class, and as such implement the same methods (e.g. `update`). Here is an example of how to implement the bounds:

```
from physiofit.models.base_model import Model

class ChildModel(Model):
```

(continues on next page)

(continued from previous page)

```

def __init__(self, data):
    super().__init__(data)
    self.model_name = "Tutorial model"
    self.parameters_to_estimate = None
    self.fixed_parameters = None

def get_params(self):

    # Parameters are given in a dictionary, where the key is
    # the parameter name and the value is a number that will
    # be the initial value for the optimization process

    self.parameters_to_estimate = {
        "BM_0": 1,
        "growth_rate": 1
    }

    # Instantiate the bounds object

    self.bounds = Bounds(
        {
            "BM_0": (1e-3, 10),
            "growth_rate": (1e-3, 3)
        }
    )

    # Do the same for all metabolite parameters to estimate
    # using a for loop:

    for metabolite in self.metabolites:
        self.parameters_to_estimate.update(
            {
                f"{metabolite}_flux" : 1,
                f"{metabolite}_init_value" : 1
            }
        )

    # Append the default bounds to the bounds attribute
    self.bounds.update(
        {
            f"{metabolite}_flux": (-50, 50),
            f"{metabolite}_init_value": (1e-6, 50)
        }
    )

    @staticmethod
    def simulate():
        pass
    
```

Warning: The keys in the bounds and in the parameters to estimate dictionary must be the same!

Fixed parameters

The fixed parameters are parameters that are known and are not estimated by PhysioFit. For example, in the case of steady-state models that account for non enzymatic degradation (see [Steady-state models](#)), we need to provide the degradation constant of all unstable metabolites (these constants must be measured in an independent experiment, e.g. see Peiro et al., 2019):

```
self.fixed_parameters = {"Degradation": {
    metabolite: 2 for metabolite in self.metabolites
}}
```

The fixed parameters must be provided as a dictionary of dictionaries, where the first level is the name of the parameter itself (here degradation) and the second level contains the mapping of metabolite-value pairs that will be the default values initialized (here we give a default value of 2 for every metabolite for example). Each key of the first level is used to initialize a widget in the GUI, thus allowing users to change the corresponding values for the metabolites given in the second level.

Simulation function

Once the `get_params` method has been implemented, the next step is to implement the simulation function that will be called at each iteration of the optimization process to simulate the metabolite dynamics that correspond to a given set of parameters (see [Flux calculation](#) for more details). To do this, first write out the function definition:

```
@staticmethod
def simulate(
    params_opti: list,
    data_matrix: np.ndarray,
    time_vector: np.ndarray,
    params_non_opti: dict
):
    pass
```

As shown above, this function takes four arguments:

- `params_opti`: list containing the values of each parameter to estimate **in the same order as defined in the `:samp:`parameters_to_estimate`` dictionary** (see [Free parameters](#))
- `data_matrix`: numpy array containing the experimental data (or data with the same shape)
- `time_vector`: numpy array containing the time points
- `params_non_opti`: dictionary containing the fixed parameters (see [Fixed parameters](#))

Now you can start writing the body of the function. For sake of clarity, we recommend unpacking parameters values from the list of parameters to estimate into internal variables. The function `simulate` must return a matrix containing the simulation results, with the same shape as the matrix containing the experimental data. To initialize the simulated matrix, you can use the `empty_like` function from the numpy library:

```
@staticmethod
def simulate(
```

(continues on next page)

(continued from previous page)

```

        params_opti: list,
        data_matrix: np.ndarray,
        time_vector: np.ndarray,
        params_non_opti: dict
    ):
        # Get end shape
        simulated_matrix = np.empty_like(data_matrix)

        # Get initial params
        x_0 = params_opti[0]
        mu = params_opti[1]

        # Get X_0 values
        exp_mu_t = np.exp(mu * time_vector)
        simulated_matrix[:, 0] = x_0 * exp_mu_t
        fixed_params = [value for value in params_non_opti["Degradation"].values()]

        # Get parameter names and run the calculations column by column
        for i in range(1, int(len(params_opti) / 2)):
            q = params_opti[i * 2]
            m_0 = params_opti[i * 2 + 1]
            k = fixed_params[i - 1]
            exp_k_t = np.exp(-k * time_vector)
            simulated_matrix[:, i] = q * (x_0 / (mu + k)) \
                * (exp_mu_t - exp_k_t) \
                + m_0 * exp_k_t

        return simulated_matrix
    
```

The math corresponding to the simulation function provided above as example can be found [here](#) (equations 5 and 6). See [here](#) for information on how to test the completed model.

This example showcases the use of analytical functions to simulate the flux dynamics. It is also possible to use numerical derivation to solve a system of ordinary differential equations (ODEs), which can be useful when algebraic derivation is not straightforward. This requires the implementation of additional functions into the simulate function. The system of ODEs can be provided directly within the body of the simulate function:

```

from scipy.integrate import solve_ivp

@staticmethod
def simulate(
    params_opti: list,
    data_matrix: np.ndarray,
    time_vector: np.ndarray,
    params_non_opti: dict
):
    # Get parameters
    x_0 = params_opti[0]
    y_BM = params_opti[1]
    km = params_opti[2]
    qsmax = params_opti[3]
    s_0 = params_opti[4]
    
```

(continues on next page)

(continued from previous page)

```

y_P = params_opti[5]
p_0 = params_opti[6]
params = (y_BM, y_P, km, qsmax)

# initialize variables at t=0
state = [x_0, s_0, p_0]

def calculate_derivative(t, state, y_BM, y_P, km, qsmax):

    # get substrate and biomass concentrations
    s_t = state[0]
    x_t = state[1]

    # calculate fluxes
    qs_t = qsmax * (s_t / (km + s_t))
    mu_t = y_BM * qs_t
    qp_t = y_P * qs_t

    # calculate derivatives
    dx = mu_t * x_t
    ds = -qs_t * x_t
    dp = qp_t * x_t

    return dx, ds, dp

# simulate time-course concentrations
sol = solve_ivp(
    fun=calculate_derivative,
    t_span=(np.min(time_vector), np.max(time_vector)),
    y0 = state,
    args=params,
    method="LSODA",
    t_eval = list(time_vector)
)

return sol.y.T
    
```

As we can see, the function `calculate_derivative` returns the derivatives of each metabolite concentration and is used by an ODEs solver that performs the simulations. This function is thus created within the body of the `simulate` function, before being called by the solver. More information on the mathematics behind this implementation can be found [here](#).

Note: The simulation function will be called a high number of times by the optimizer for parameter estimation, so optimize this function as much as possible. When possible, implement the model using analytical solution as calculations will be faster than solving numerically the corresponding ODEs.

5.2.4 Test the model

Once you have completely populated your model file, you can now launch a round of simulations and optimizations in a programmatic way:

```
if __name__ == "__main__":
    from physiofit.base.io import IoHandler
    from physiofit.models.base_model import StandardDevs

    test_data = pd.DataFrame(
        {
            "time": [0, 1, 2, 3],
            "experiments": ["A", "A", "A", "A"],
            "X": [0.5, 0.8, 1.2, 1.8],
            "Glucose": [12, 11.6, 11, 10.2]
        }
    )

    io = IoHandler()
    model = ChildModel(data=test_data)
    model.get_params()
    fitter = io.initialize_fitter(
        model.data,
        model=model,
        mc=True,
        iterations=100,
        sd=StandardDevs({"X": 1, "Glucose": 1}),
        debug_mode=True
    )
    fitter.optimize()
    fitter.monte_carlo_analysis()
    fitter.khi2_test()
    print(fitter.parameter_stats)
```

This will return the calculated flux values and associated statistics.

Note: The test data and calculation parameters (e.g. standard deviations) defined in the test function must correspond to those expected for the new model.

To test the integration of the model into the GUI, copy the .py file in the folder `models` of PhysioFit directory. You can get the path towards this folder by opening a python kernel in your dedicated environment and initializing an IoHandler

```
from physiofit.base.io import IoHandler
io_handler = IoHandler()
print(io_handler.get_local_model_folder())
```

Note: The model file name must follow the naming convention `model_[model number].py`. If the last model in the list is the `model_5.py`, the next one should be named `model_6.py`.

You can now launch PhysioFit's GUI, load a data file corresponding to the new model, select the model, and run flux calculation. In case of errors, have a look to the error message and correct the code.

Note: We would be happy to broaden the types of models shipped with PhysioFit. If you have developed a new model, it might be usefull and valuable to the fluxomics community! Please, keep in touch with us to discuss on the model and see if we can include your model in the built-in models shipped with PhysioFit! :)

HOW TO CITE

Thank you for using PhysioFit and citing us in your work! It means a lot to us and encourage us to continue its development.

PhysioFit: a software to quantify cell growth parameters and extracellular fluxes.

Le Grégam L., Guitton Y., Bellvert F., Jourdan F., Portais J.C., Millard P.

bioRxiv preprint, doi: <https://doi.org/10.1101/2023.10.12.561695>

LIBRARY DOCUMENTATION

7.1 API reference

This module serves as reference for the different classes and associated methods of the PhysioFit package.

`fitter.py` PhysioFit software main module

class `physiofit.base.fitter.PhysioFitter`(*data, model, mc=True, iterations=100, sd=None, debug_mode=False*)

Bases: `object`

This class is responsible for most of Physiofit's heavy lifting. Features included are:

- loading of data from **csv** or **tsv** file
- **equation system initialization** using the following analytical functions (in absence of lag and degradation:
$$X(t) = X0 * \exp(\mu * t) \quad Mi(t) = qMi * (X0 / \mu) * (\exp(\mu * t) - 1) + Mi0$$
- **simulation of data points** from given initial parameters
- **cost calculation** using the equation:
$$\text{residuum} = \text{sum}((\text{sim} - \text{meas}) / \text{sd})^2$$
- **optimization of the initial parameters** using `scipy.optimize.minimize` ('Differential evolution', with polish with 'L-BFGS-B' method)
- **sensitivity analysis, khi2 tests and plotting**

Parameters

- **data** (*class: pandas.DataFrame*) – DataFrame containing data and passed by IOstream object
- **model** (`physiofit.models.base_model.Model`) – Model to initialize parameters and optimize
- **mc** (*Boolean*) – Should Monte-Carlo sensitivity analysis be performed (default=True)
- **iterations** (*int*) – number of iterations for Monte-Carlo simulation (default=50)
- **sd** (*int, float, list, dict or ndarray*) – sd matrix used for residuum calculations. Can be:
 - a matrix with the same dimensions as the measurements matrix (but without the time column)
 - a named vector containing sds for all the metabolites provided in the input file

- 0 in which case the matrix is automatically constructed from default values
- a dictionary with the data column headers as keys and the associated value as a scalar or list

initialize_sd_matrix()

Initialize the sd matrix from different types of inputs: single value, vector or matrix.

Returns

None

khi2_test()

monte_carlo_analysis()

Run a monte carlo analysis to calculate optimization standard deviations on parameters and simulated data points

optimize()

Run optimization and build the simulated matrix from the optimized parameters

verify_attrs()

Check that attributes are valid

`io.py` Module to handle inputs and outputs for PhysioFit

class `physiofit.base.io.ConfigParser`(*selected_model, sds, mc, iterations, path_to_data=None*)

Bases: `object`

allowed_keys = ['model', 'sds', 'mc', 'iterations']

export_config(*export_path*)

classmethod from_file(*yaml_file*)

classmethod from_galaxy(*galaxy_yaml*)

get_kwargs()

update_model(*model*)

class `physiofit.base.io.IoHandler`

Bases: `object`

Input/Output class that handles the former and initializes the PhysioFitter component object. It is the preferred interface for interacting with the PhysioFit package.

static add_model(*model_file*)

allowed_keys = {'debug_mode', 'iterations', 'mc', 'model', 'sd'}

get_local_model_folder() → str

Return the path towards the actual environment's used models folder

static get_model_list()

get_models(*data=None*)

Read modules containing the different models and add them to models attribute

Returns

list containing the different model objects

initialize_fitter(*data*: *pd.DataFrame*, ***kwargs*) → *PhysioFitter*

Initialize a PhysioFitter object

Parameters

- **data** – input data
- **kwargs** – Keyword arguments for fitter initialization

Returns

None

output_pdf(*fitter*: *PhysioFitter*, *export_path*: *str* | *Path* = *None*)

Handle the creation and output of a pdf file containing fit results as a plot

Parameters

export_path – Path to exported pdf. In local mode, it is sent to the `_res` directory

Returns

None

output_plots(*fitter*, *export_path*)

Handle the creation and export of the different plots in svg format :return: None

output_recap(*export_path*: *str*, *galaxy*=*False*)

output_report(*fitter*, *export_path*: *str* | *list* = *None*)

Handle creation and export of the report containing stats from monte carlo analysis of optimization parameters

Parameters

export_paths – list of paths to export the stats and fluxes. [0] is for stats and [1] for fluxes.

plot_data(*fitter*, *display*: *bool* = *False*)

Plot the data

Parameters

display – should plots be displayed

static read_data(*data*: *str*) → *DataFrame*

Read initial data file (csv or tsv)

Parameters

data – str containing the relative or absolute path to the data

Returns

pandas DataFrame containing the data

read_model(*model_file*)

Import and return the model class from .py file containing the model.

Parameters

model_file – path to the model.py file to import

static read_yaml(*yaml_file*: *str* | *bytes*) → *ConfigParser*

Import raml configuration file and parse keyword arguments

Parameters

yaml_file – path to the yaml file or json file

Return config_parser

Dictionary containing arguments parsed from yaml file

select_model(*model_name*, *data=None*)

Select a model from the list of models in the model folder of the package src directory

gui.py

class physiofit.ui.gui.**App**

Bases: object

Physiofit Graphical User Interface

check_uptodate()

Compare installed and most recent Physiofit versions.

silent_sim()

start_app()

Launch the application

base_model.py

class physiofit.models.base_model.**Bounds**(*mapping=None*, ***kwargs*)

Bases: dict

class physiofit.models.base_model.**Model**(*data: DataFrame*)

Bases: ABC

abstract get_params()

Return params_to_estimate

List of parameters to estimate

Return fixed_parameters

dict of constant parameters

Return bounds

dict of upper and lower bounds

Return default_init_values

dict containing default initial values for params

abstract static simulate(*params_opti: list*, *data_matrix: ndarray*, *time_vector: ndarray*,
params_non_opti: dict | list)

exception physiofit.models.base_model.**ModelError**

Bases: Exception

class physiofit.models.base_model.**StandardDevs**(*mapping=None*, ***kwargs*)

Bases: dict

property vector

FREQUENTLY ASKED QUESTIONS (FAQ)

8.1 How are fluxes calculated?

We provide details on the flux calculation approach implemented in PhysioFit in the [Method](#) section.

8.2 How many measurements should I use to calculate fluxes?

As in any model-based fitting procedure, more data usually means more accurate and precise flux estimates. The minimal number of measurements depend on the model used for flux calculation. For instance, for steady-state built-in models provided with PhysioFit, we recommend using at least 6 to 8 time points, which should provide reliable and meaningful estimates in most situations.

Still, the exact answer to this question strongly depends on the uptake/production/growth rates of your (micro)organism in the conditions you are investigating, on the sampling time interval, on the questions you are addressing, on the model used for flux calculation, and on many other parameters! You can make some tests by calculating fluxes from (published or theoretical) datasets similar to those you have in mind.

8.3 Can I calculate fluxes in case of missing values?

Yes, fluxes can still be calculated if some measurement(s) are missing. In this case, let empty the corresponding field of the input data file.

8.4 What units should be used for input data?

Input data (biomass concentration, metabolites concentrations, and time) can be provided to PhysioFit using any unit. Still, we recommend to use units for which values are as close to unity as possible to ensure numerical stability (e.g. 3 mM instead of $3 \cdot 10^{-3}$ M). Importantly, units of the estimated fluxes depend on units of time and metabolites and biomass concentrations. The concentration of different metabolites can be provided using different units, but a single unit must be used for all measurements of a given metabolite.

See also:

[What are the flux units?](#)

8.5 What are the flux units?

Flux units depend on the units of time and concentrations (of biomass and metabolites) provided in the input data file. For instance, if biomass units are in grams of cell dry weight by liter ($\text{g}_{\text{CDW}}/\text{L}$), metabolite concentrations are in millimolar (mM) and time is in hours (h), the estimated fluxes will be in $\text{mmol}/\text{g}_{\text{CDW}}/\text{h}$. Units should thus be carefully selected, and calculated fluxes must be interpreted consistently with the concentration units.

See also:

What units should be used for input data?

8.6 An error has been raised. What should I do?

The first thing to do is to read the error message which might contain information on how to resolve it. If not, check the FAQ section (yes, this one) to see if the error has been explained in more depth. If the error persists or if you do not understand the error, please post it in the “issues” section on [GitHub](#). We will try to respond as quickly as possible to solve your problem.

8.7 What parameters values should I use?

Details on PhysioFit parameters can be found in the [Tutorial](#) section.

8.8 How can I check if my data have been fitted correctly?

The quality of the fit can be evaluated based on:

- the plots of experimental vs simulated data for the best fit, which should be as close as possible,
- the χ^2 statistical test results given in the log file (see below for help on interpreting the results).

See also:

What is a χ^2 test? and My data hasn't been correctly fitted. Why?

8.9 What is a χ^2 test?

A χ^2 test describes how well a model fits a set of observations. Measures of goodness of fit typically summarize the discrepancy between observed values and the values expected under the model used in PhysioFit (see the [Method](#) section). It is calculated as the sum of differences between measured and simulated values, each squared and divided by the simulated value. A good fit corresponds to small differences between measured and simulated values, thereby the χ^2 value is low. In contrast, a bad fit corresponds to large differences between simulations and measurements, and the χ^2 value is high.

The resulting χ^2 value can then be compared with a χ^2 distribution to determine the goodness of fit. The p-value of one-tail χ^2 test is calculated by PhysioFit from the best fit and is given in the log file (have a look to the [Tutorial](#) section). A p-value close to 0 means poor fitting, and a p-value close to 1 means good fitting (keeping in mind that a p-value very close to 1 can be an evidence that standard deviations might be overestimated). A p-value between 0.95 and 1 means the model fits the data good enough with respect to the standard deviations provided (at a 95% confidence level). PhysioFit provides an explicit message stating whether the flux data are satisfactorily fitted or not (at a 95% confidence interval).

8.10 My data hasn't been correctly fitted. Why?

A possible reason to explain a bad fit is that standard deviations on measurements (concentration biomass and metabolites) is under-estimated, thereby making the χ^2 test too stringent. In this case, plots of measured and fitted data should be in agreement. Reliable estimated of standard deviation on measurements must be provided to PhysioFit (have a look to the [Tutorial](#) section to see how to check and adjust this parameter).

Another possible reason to explain a bad fit is that a key assumption of the flux calculation method is not respected. For instance, if you use a steady-state model shipped with PhysioFit, cells might not be strictly in metabolic steady-state, i.e. with constant fluxes during the whole experiment. If this key assumption does not occur (e.g. cells are continuously adapting to their environment and fluxes change over time), PhysioFit will not be able to fit the data satisfactorily. In this case, evaluate whether the deviation is significant or not (e.g. based on the detailed χ^2 statistics or on the plot of fitted vs measured data), and evaluate the potential biases that would be introduced by interpreting (or not) these flux values.

In rare situations, it may also be because some parameters have to be tweaked to help PhysioFit fitting the measurements, which results in obviously aberrant fits (e.g. with flat time-course profiles for all metabolites). This might happen for instance if some measurements are provided in units far from unity (e.g. $1 \cdot 10^{-5}$ M instead of 10 μ M). If this situation happens, we suggest modifying the initial values of fluxes, or changing the units of input data, and re-run the flux calculation. For more info on the run parameters and how they may affect the fitting process, please refer to section [Flux calculation parameters](#).

If you believe the problem is in PhysioFit, we would greatly appreciate if you could open a new issue on our [issue tracker](#).

8.11 I cannot start PhysioFit graphical user interface, can you help me?

If you installed PhysioFit following our standard procedure and that you are unable to start PhysioFit by opening a terminal and typing `physiofit`, then there is indeed something wrong. Do not panic, we are here to help! Please follow this simple procedure:

1. The first step of the debugging process will be to get a *traceback*, i.e. a message telling us what is actually going wrong. You should see this message in the terminal you opened.
2. Read the traceback and try to understand what is going wrong:
 - If it is related to your system or your Python installation, you will need to ask some help from your local system administrator or your IT department so they could guide you toward a clean installation. Tell them that you wanted “to use the graphical user interface of PhysioFit, a Python 3.6 software” and what you did so far (installation), give them the traceback and a link toward the documentation. They should know what to do.
 - If you believe the problem is in PhysioFit or that your local system administrator told you so, then you probably have found a bug! We would greatly appreciate if you could open a new issue on our [issue tracker](#).

8.12 I have develop a new model, can you include it in PhysioFit distribution?

If you have developed a new flux model, we would be happy to include it in PhysioFit! Open a new issue on our [issue tracker](#), and let's discuss about your model and how we could include it! :)

8.13 I would like a new feature.

We would be glad to improve PhysioFit. Please get in touch with us [here](#) so we could discuss your problem.

LICENSE

PhysioFit is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

PhysioFit is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with PhysioFit. If not, see <https://www.gnu.org/licenses/>.

- search

PYTHON MODULE INDEX

p

`physiofit.base.fitter`, [24](#)
`physiofit.base.io`, [25](#)
`physiofit.models.base_model`, [27](#)
`physiofit.ui.gui`, [27](#)

A

`add_model()` (*physiofit.base.io.IoHandler static method*), 25
`allowed_keys` (*physiofit.base.io.ConfigParser attribute*), 25
`allowed_keys` (*physiofit.base.io.IoHandler attribute*), 25
`App` (*class in physiofit.ui.gui*), 27

B

`Bounds` (*class in physiofit.models.base_model*), 27

C

`check_uptodate()` (*physiofit.ui.gui.App method*), 27
`ConfigParser` (*class in physiofit.base.io*), 25

E

`export_config()` (*physiofit.base.io.ConfigParser method*), 25

F

`from_file()` (*physiofit.base.io.ConfigParser class method*), 25
`from_galaxy()` (*physiofit.base.io.ConfigParser class method*), 25

G

`get_kwargs()` (*physiofit.base.io.ConfigParser method*), 25
`get_local_model_folder()` (*physiofit.base.io.IoHandler method*), 25
`get_model_list()` (*physiofit.base.io.IoHandler static method*), 25
`get_models()` (*physiofit.base.io.IoHandler method*), 25
`get_params()` (*physiofit.models.base_model.Model method*), 27

I

`initialize_fitter()` (*physiofit.base.io.IoHandler method*), 25

`initialize_sd_matrix()` (*physiofit.base.fitter.PhysioFitter method*), 25
`IoHandler` (*class in physiofit.base.io*), 25

K

`khi2_test()` (*physiofit.base.fitter.PhysioFitter method*), 25

M

`Model` (*class in physiofit.models.base_model*), 27
`ModelError`, 27
`module`
 physiofit.base.fitter, 24
 physiofit.base.io, 25
 physiofit.models.base_model, 27
 physiofit.ui.gui, 27
`monte_carlo_analysis()` (*physiofit.base.fitter.PhysioFitter method*), 25

O

`optimize()` (*physiofit.base.fitter.PhysioFitter method*), 25
`output_pdf()` (*physiofit.base.io.IoHandler method*), 26
`output_plots()` (*physiofit.base.io.IoHandler method*), 26
`output_recap()` (*physiofit.base.io.IoHandler method*), 26
`output_report()` (*physiofit.base.io.IoHandler method*), 26

P

physiofit.base.fitter
 module, 24
physiofit.base.io
 module, 25
physiofit.models.base_model
 module, 27
physiofit.ui.gui
 module, 27
`PhysioFitter` (*class in physiofit.base.fitter*), 24
`plot_data()` (*physiofit.base.io.IoHandler method*), 26

R

`read_data()` (*physiofit.base.io.IoHandler static method*), [26](#)
`read_model()` (*physiofit.base.io.IoHandler method*), [26](#)
`read_yaml()` (*physiofit.base.io.IoHandler static method*), [26](#)

S

`select_model()` (*physiofit.base.io.IoHandler method*), [26](#)
`silent_sim()` (*physiofit.ui.gui.App method*), [27](#)
`simulate()` (*physiofit.models.base_model.Model static method*), [27](#)
`StandardDevs` (*class in physiofit.models.base_model*), [27](#)
`start_app()` (*physiofit.ui.gui.App method*), [27](#)

U

`update_model()` (*physiofit.base.io.ConfigParser method*), [25](#)

V

`vector` (*physiofit.models.base_model.StandardDevs property*), [27](#)
`verify_attrs()` (*physiofit.base.fitter.PhysioFitter method*), [25](#)