NEMF

Jun 23, 2020

Contents:

1 An introduction to NEMF 3

2 Examples 9

3 A novel three step, network-based, general ecosystem modeling framework 17

4 Installation 19

5 Manual 21

6 API reference 31

7 Interaction Functions 37

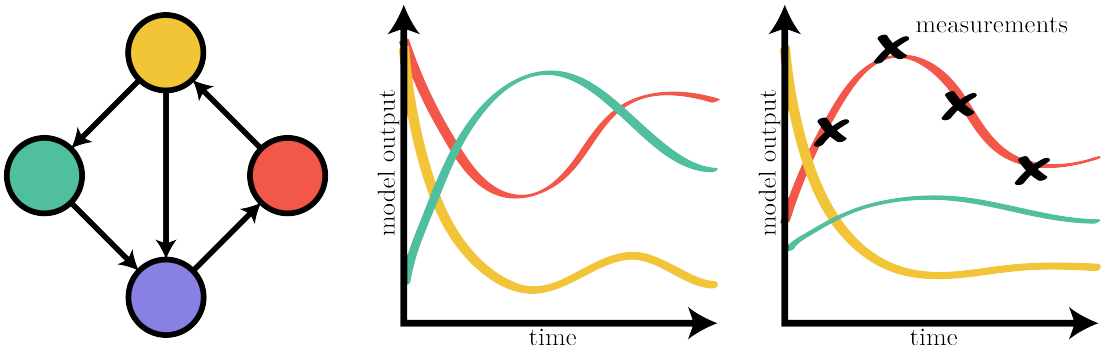
8 Releases 43

Bibliography 47

Python Module Index 49

Index 51

NEMF is a ecosystem modelling framework written in python. It is designed to offer an easy to use method for modelling ecosystems with low- to intermediate complexity. It consists of three conceptual parts:



1. *Network-based model description*
2. *Forward modelling*
3. *Inverse modelling*

   (For details see *Introductory notes*.)

The framework offers the functionality to handle *non-equilibrium*, *non-linear interactions*. For the typical use-cases, user do need to write any code but only provide a model configuration.

The range of use-cases can easily be extended with simple user-written functions without needing to change any of the framework code. NEMF offers an easy to use method to *fit any model parameter* such that the model mimics the studied system.

For simplicity, the current version of the framework is limited to *non-spatially resolved models* (box-models).

For a brief introduction to the ideas behind this library, you can read the *introductory notes*. A more detailed description can be found in the *paper*. Visit the *installation page* to see how you can download the package. You can browse the *examples* to see what you can do with NEMF. A more detailed description of the parts the library can be found in the *manual*, while the code is documented in the *API reference* section.

To see the code, please visit the github repository. To report a bug, please raise a 'Issue' in githubs issue tracker. Please feel free to contact laurin.steidle@uni-hamburg.de with general issues (i.e. if the installation fails).

CHAPTER 1

An introduction to NEMF

The network-based ecosystem modelling framework (NEMF) is a python software tool to model ecosystems. From a modelling perspective, it consists of three conceptual parts:

1. **Network-based model description:** Each model is described by a network. A network consists of two components. Nodes, which describe ecosystem compartments, and edges, which link two compartments together through some interaction.
2. **Forward modelling:** A model is defined by a set of differential equations implicitly defined by the network structure. Theses differential equations can be solved numerically for a certain initial state. Hence, providing forecasts for how the model behave over time.
3. **Inverse modelling:** Fitting of model forecast to to observational or other reference data, by varying model parameters.

It aims to keep the configuration complexity minimal for the user such that it can be quickly learned and applied for i.e. rapid prototyping of model ideas. To keep configuration and computational complexity low it can only applied to non-spatially-resolved, also known as box-models.

1.1 Network and Forward

We will now go over the the first two parts of the model with the help of a simple NPZD type model. NPZD stands for the **N**utrient- **P**hytoplankton- **Z**ooplanktion-**D**etritus model, which is a simple well studied marine ecosystem model. The details of it are not important here, as it is solely used as placeholder for any sort of ecosystem model.

To calculate the forecast of a model we run the following few lines of code:

```
import nemf
model = nemf.load_model('exemplary_npzd_model.yml')
nemf.interaction_graph(model)
results = nemf.forward_model(model)
nemf.output_summary(results)
```

Let's go through the lines one by one to see what happened:

1. First we imported the nemf python library.

```
import nemf
```

   This tells python that we want to us this library and because not stated otherwise that we will address it as *nemf*

2. We tell the nemf library which model we want to use.

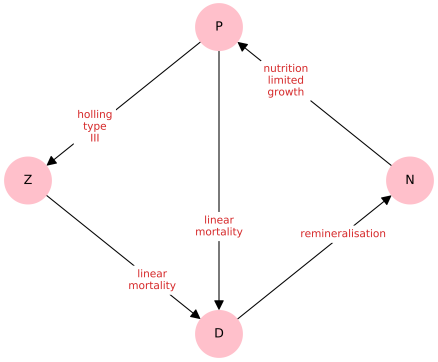
```
model = nemf.load_model('exemplary_npzd_model.yml')
```

   Models are typically defined in an extra file. This file contains the description of the model in a humon-readable standard called YAML. Hence, the file extension *.yml* More on the yml standard and how it is used to define models can be found here.

3. We visualize the network defined in the model configuration by

```
nemf.interaction_graph(model)
```

   which returns the following plot:



   NEMF offers the option to draw the network defined in the model configuration. This is helps to catch errors that might have happened during the configuration and gives a nice overview over the model. Each node represents a compartment in the model, i.e. a population or a chemical quantity. The arrows between them show what flows from one compartment to another while the label on the arrow describes how it does that.

4. We solve the differential equations underlying the model numerically with:

```
results = nemf.forward_model(model)
```

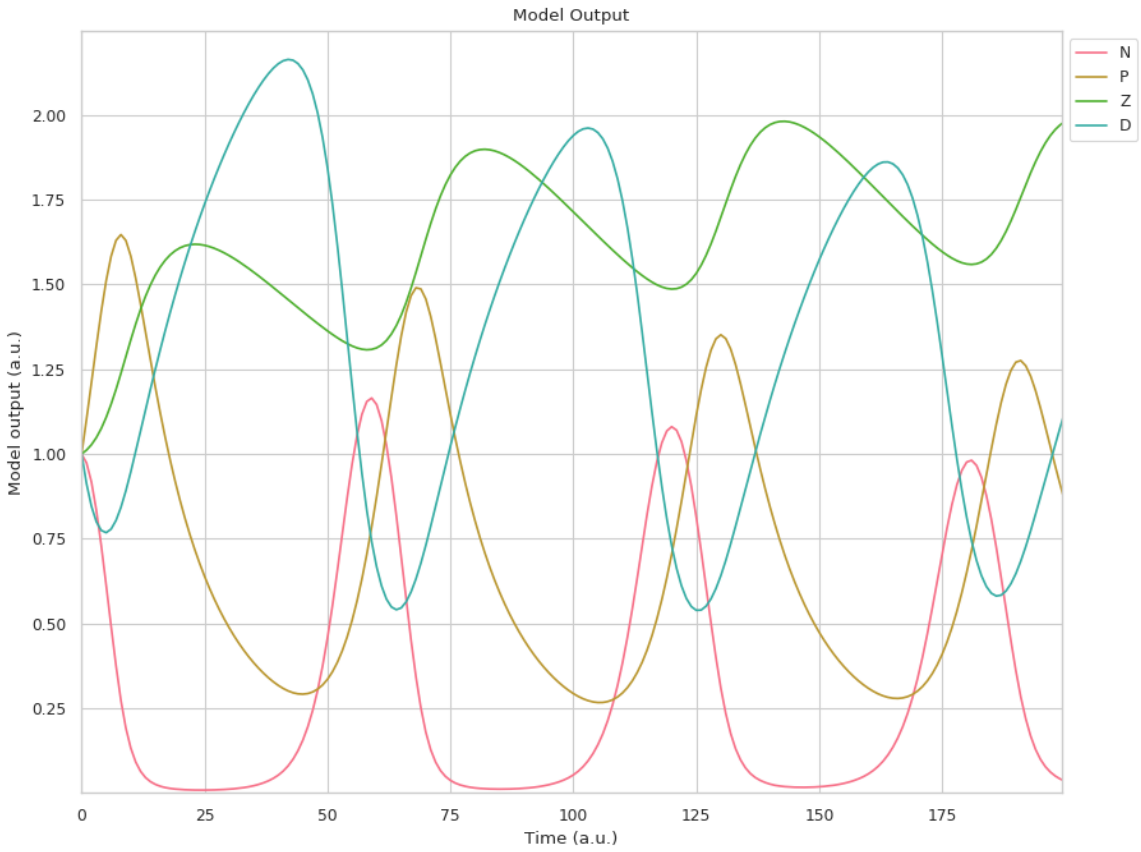
   The network implicitly defines a set of differential equation that couples the compartments to each other, through the interactions between them. The framework solves these differential equations to give a forecast how the model is expected to evolve over time. This is often also called 'time evolution'.

5. The result of the time evolution are visualized by calling

```
nemf.output_summary(results)
```

   which generates the following plot:

   Each line represents one compartment and how its associated quantity (i.e. a population size) changes over time.


Model Output
N
P
Z
D
Model output (a.u.)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
25
50
75
100
125
150
175
Time (a.u.)

1.2 Model description via YAML configuration

In the example above, we assumed that a model (*'exemplary_npzd_model'*) has already been defined. If we want to construct a new model, we need to write our own configuration file.

There are three major parts of the configuration file:

1. **Compartments** contain a list of all model compartments, like species population pools or nutrition pools.
2. **Interactions** contains a list of all interactions between compartments, like what eats what and what happens when it dies
3. **Configuration** contains a list of technical details that decide the framework behavior during the forecast and fitting.

The configuration file is written in the YAML standard.

It consists of what is called key-value pairs. Each key is associated with a value. These values can also be lists which are indicated by leading "-", like bullet points. You can find details about the YAML standard on http://yaml.org/. Note that the YAML website is itself perfectly valid yaml.

A simple example for the compartment section looks like this:

```
compartment:      # header of the compartment section
  A:              # name of compartment
    value: 1.0    # initial value of the compartment
  B:
    value: 2.0
  ..
  ..
```

Interactions are defined similarly:

```
A:B:              # flow from B to A (predator:prey)
- fkt: grazing    # type of interaction
- parameters:     # parameters used for the interaction
  - 1             # i.e. hunting rate
  - 2             #      food processing time
B:A
- fkt: natural mortality
- parameters:
  - 0.01          # natural mortality rate
```

A description of how this works in detail can be found in the YAML section of the manual.

1.3 Inverse modelling

So far, we covered the first two aspect; the network-based approach and the forward modelling. We can also fit unknown, or imprecisely known parameters such that the forecast resembles a provided data set as closely as possible.

We can achieve this with the *inverse_model* method.

```
import nemf
model = nemf.load_model('exemplary_npzd_model.yml')
results = nemf.inverse_model(model)
nemf.output_summary(results)
```

Most of this code is the same as previously shown. The only new line is:

```
results = nemf.inverse_model(model)
```

Instead of calculating the forecast once as previously, the *inverse_model* we now calculates it for different sets of parameters in such a way that we find the best solution quickly.

However, for this to work we implicitly provided some additional information in the yaml configuration file. There are two things we need to provide:

1. Reference data (i.e observational data)
2. Optimized parameters

The reference data is expected do be in a separate file. Details about its format and how it can be imported can be found in the reference data section of the manual.

The parameters that shall be optimized are selected in the YAML configuration file by adding the 'optimise' key and providing its upper and lower bounds in which the method tries to find the best solution.

```
compartment:       # header of the compartment section
  A:               # name of compartment
    value: 1.0     # initial value of the compartment
    optimise:
      lower: 0     # lower and
      upper: 2     # upper bound during the fitting process
```

Detail on the configuration of the YAML file can be found in the yaml section of the manual.

The results are then visualized with:

```
nemf.output_summary(results)
```

Which creates the following figure:

1.3.1 Next steps

You have a few options for where to go next. You might first want to learn how to *install namf on your machine*. Once that's done, you can browse the *examples* to get a broader sense for what kind problems nemf is designed for. You can read through the manual for a deeper discussion of the different parts of the library and how they are designed. If you want to know specifics of the nemf functions implementations, you could check out the *API reference*, which documents each function and its parameters.

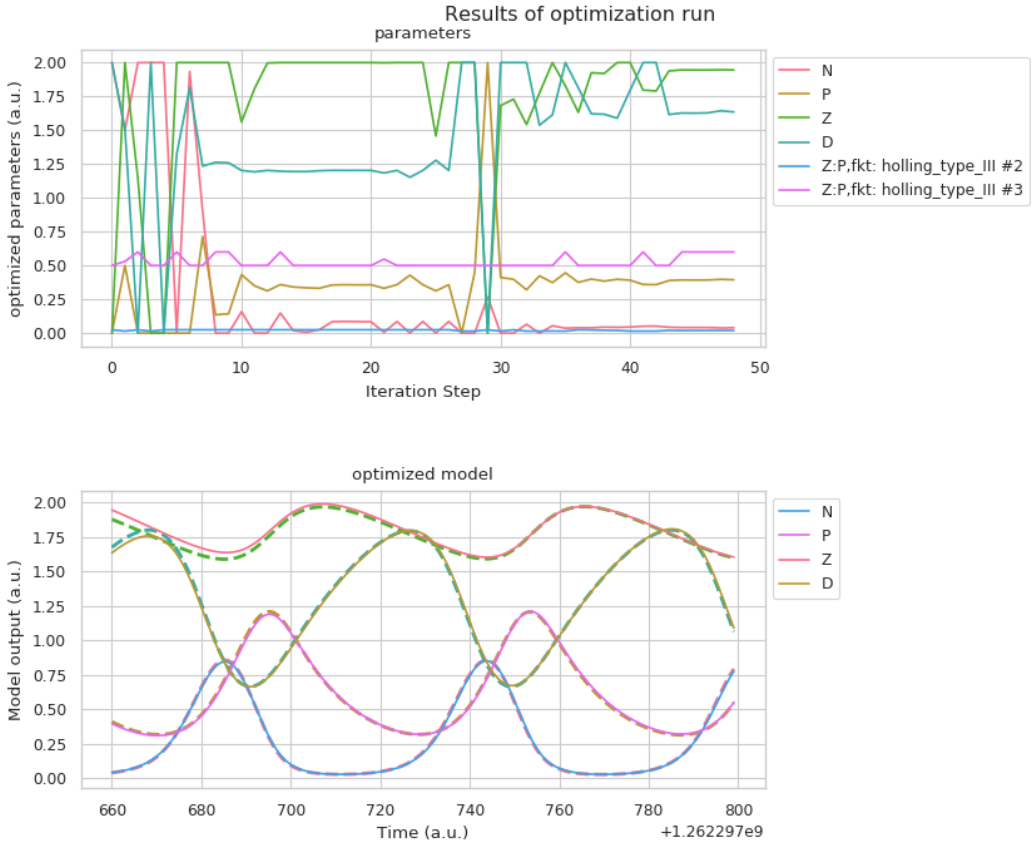


Fig. 1: Visualization of the results of the model fit. The upper figure shows the tested parameter during the fitting process, while the lower figure shows the "optimally" fitted model.

CHAPTER 2

Examples

We will present some small examples in full to show what the model can be used for and how.

2.1 Predator-Prey

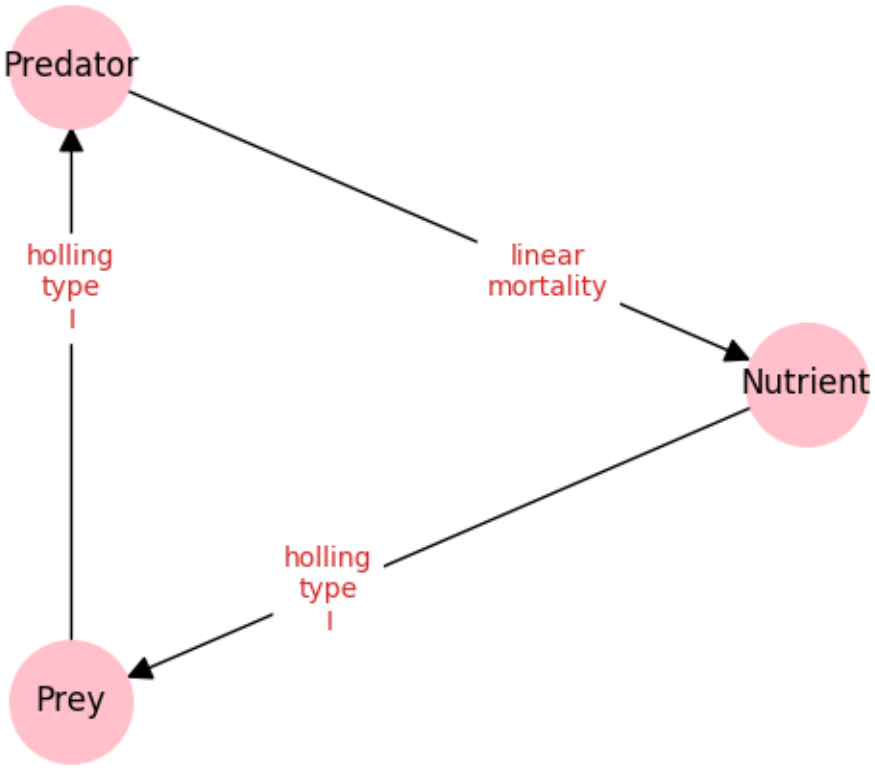
The probably most famous ecosystem model is the so called "Lotka-Volterra" model. It has been developed in the 1920s and was designed to described the population dynamics of hares and lynxes. It is described by the following set of equations,

eq:Lotka_Volterra

where x and y represent the predator and prey populations, while the remaining variables are constants.

While famous, this minimalistic predator-prey model is not "closed". Hence, it implicitly depends on external compartments, most importantly nutrients for the prey. We present a closed extension of this model, where the predator dies, is transformed in some form of nutrient and consumed by the prey.

The interaction graph of this model is presented below.



graph:closed_LV

This graph is generated by the NEMF framework base on the following *YAML-configuration-file*:

```yaml
# List of all present compartments in the model.
compartment:
  Nutrient:
    value: 0.1
    optimise:

  Predator:                          # name of the compartment
    value:  0.1                      # initial value of the compartment
    optimise:                        # if the above value shall not be optimized,
                                     #this remains empty
  Prey:
    value: 0.1
    optimise:

# List of all interactions/flows between compartments
interactions:
  Prey:Nutrient:                     # names of the interacting compartments
  - fkt: holling_type_I              # name of the type of interaction
    parameters:                      # list of parameters required for interaction
    - 'Nutrient'                     # Parameters can also be other compartments
    - 2
    optimise:                        # whether or not some of these parameters
                                     # shall be optimized
  Predator:Prey:
```

(continues on next page)

(continued from previous page)

```
  - fkt: holling_type_I
    parameters:
    - 'Prey'
    - 100

  Nutrient:Predator:
  - fkt: linear_mortality
    parameters:
    - Predator
    - 1

# list of details for the forward and inverse modelling modules
configuration:
  # time evolution / forward modelling
  time_evo_max: 100
  dt_time_evo: 0.1
```

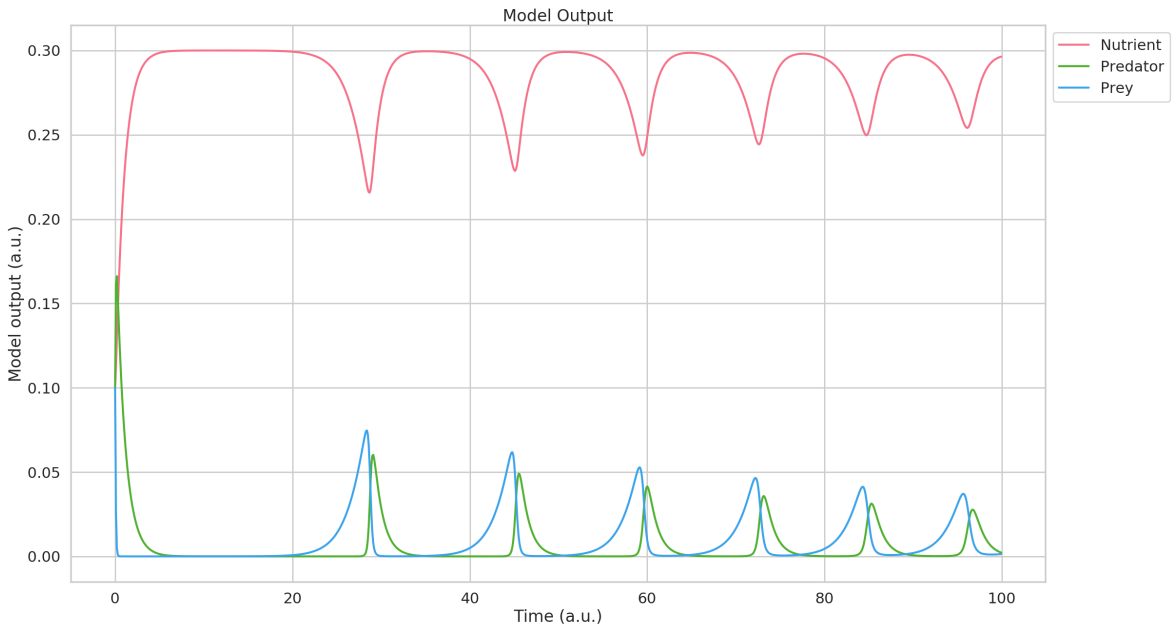
You can read more about the *YAML configuration file* here.

To create the graph and also calculate a forecast/time-evolution of the model we execute the following lines of code:

```
import nemf

# provide the path of model/system configuration
model_path = 'path/to/the/yaml/file/presented/above/example.yml'

# load the model configuration
model_config = nemf.model_class(model_path)

# visualise the model configuration to check for errors
nemf.interaction_graph(model_config)

# for a time evolution of the model call:
output_dict = nemf.forward_model(model_config)

# the results of the time evolution can be visualized with:
nemf.output_summary(output_dict)
```

To read more about the functions present in the NEMF framework, take a look at the API references.

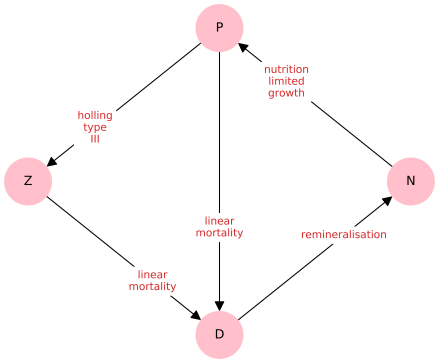
This generates both the graph as shown above, and also the following time-evolution of the model:



plot:timeevo_LV

2.2 NPZD-Model (Nutrient-Phytoplankton-Zooplankton-Detritus)

The simple Predator Prey example presented the fundamentals of the Framework. In the NPZD example used in the *introduction* we also presented the inverse- or fitting-capabilities of the framework which we will present now.



The model interaction graph looks like this: graph:NPZD

This model is a little more complicated. Hence, the configuration file will also be a little longer.

Note especially how we declare which parameters shall be fitted and the range in which the fitted value shall remain.

```
# List of all present compartments in the model.
compartment:
  N:              # name of the compartment
    value:  1     # initial value of the compartment
    optimise:     # if the above value shall not be optimized this remains empty
      lower: 1.0e-9    # if not, lower and
      upper: 2         # upper constraints must be defined
  P:
    value: 1 #.20835024e+00
```

(continues on next page)

(continued from previous page)

```
    optimise:
      lower: 1.0e-9
      upper: 2
  Z:
    value: 1 #8.84333950e-01
    optimise:
      lower: 1.0e-9
      upper: 2
  D:
    value: 1 # 8.57333742e-01
    optimise:
      lower: 1.0e-9
      upper: 2

# List of all interactions/flows between compartments
interactions:

  P:N:                              # names of the interacting compartments
  - fkt: nutrient_limited_growth    # name of the type of interaction
    parameters:                     # list of parameters required for interaction
    - 'N'                           # Parameters can also be other compartments
    - 0.27
    - 0.7
    optimise:                       # whether or not some of these parameters
                                    # shall be optimized
  Z:P:
  - fkt: holling_type_III
    parameters:
    - 'P'
    - 0.02
    - 0.575
    optimise:
      - parameter_no: 2
        lower: 0.015
        upper: 0.025
      - parameter_no: 3
        lower: 0.5
        upper: 0.6
  D:P:
  - fkt: linear_mortality
    parameters:
    - 'P'
    - 0.04
    optimise:
  D:Z:
  - fkt: linear_mortality
    parameters:
    - 'Z'
    - 0.01
    optimise:
  N:D:
  - fkt: remineralisation
    parameters:
    - 'D'
    - 0.148
    optimise:
```

(continues on next page)

(continued from previous page)

```
# list of details for the forward and inverse modelling modules
configuration:
  # time evolution / forward modelling
  time_evo_max: 1000
  dt_time_evo: 1
```

For the framework to fit we also need a date set to be used. It will then try to find a model configuration (in the allowed constraints) that lies closest to the reference data points.

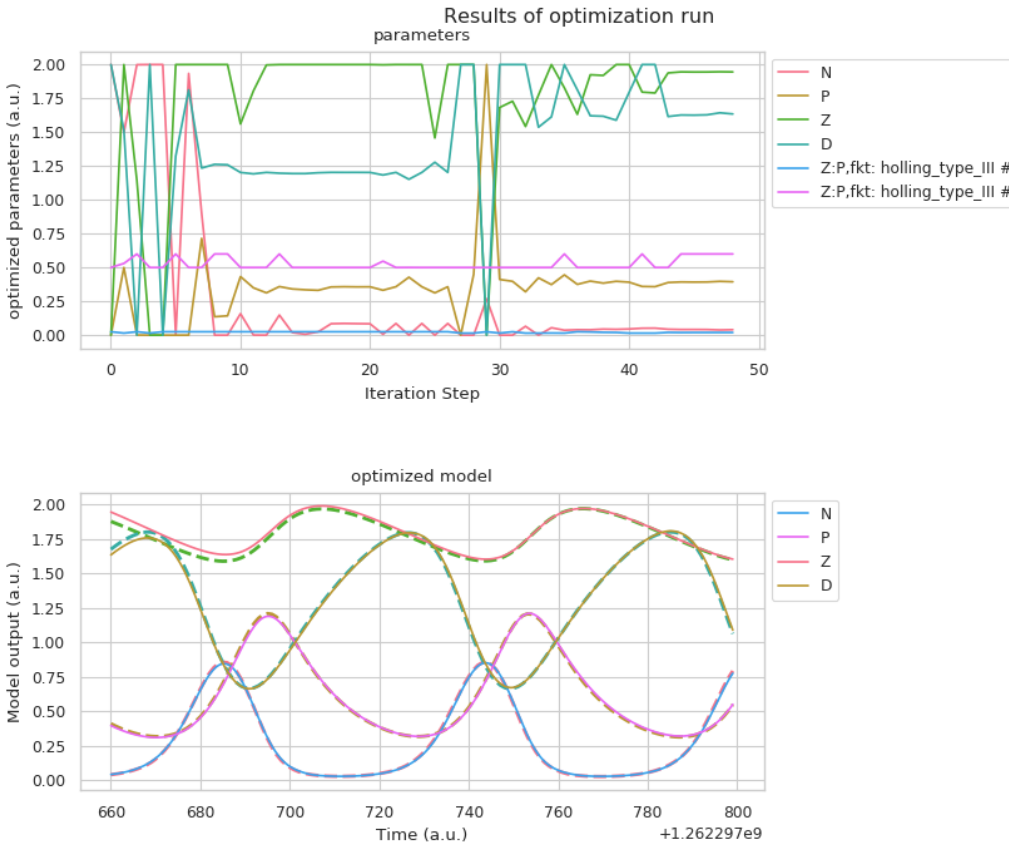
For our example we use the following data set:

```
'Datetime','N','P','Z','D'
1.262297670e+09,1.51e-01,3.15e-01,1.74e+00,1.79e+00
1.262297681e+09,7.22e-01,4.97e-01,1.60e+00,1.17e+00
1.262297692e+09,5.23e-01,1.13e+00,1.66e+00,6.69e-01
1.262297703e+09,4.98e-02,9.03e-01,1.94e+00,1.09e+00
1.262297714e+09,2.82e-02,4.97e-01,1.93e+00,1.54e+00
1.262297725e+09,8.95e-02,3.29e-01,1.79e+00,1.79e+00
1.262297736e+09,5.07e-01,3.92e-01,1.64e+00,1.45e+00
1.262297747e+09,7.57e-01,9.44e-01,1.60e+00,6.88e-01
1.262297758e+09,9.99e-02,1.07e+00,1.89e+00,9.35e-01
1.262297769e+09,2.52e-02,5.88e-01,1.95e+00,1.42e+00
1.262297780e+09,5.41e-02,3.58e-01,1.83e+00,1.74e+00
1.262297791e+09,3.04e-01,3.30e-01,1.68e+00,1.67e+00
```

Read more about the reference data format here.

To tell the framework to perfore a fitting-run we execute the following small script. It will also generate the interaction graph above and will draw a plot presenting the result.

```
import nemf

# provide the path of model/system configuration
model_path = 'path/to/the/yaml/file/presented/above/example.yml'
reference_data_path = 'path/to/the/data/file/representing/the/model_ref.csv'

# load the model configuration
model_config = nemf.model_class(model_path)

# visualise the model configuration to check for errors
nemf.interaction_graph(model_config)

# for a simple time evolution of the model call:
output_dict = nemf.inverse_model(model_config)

# the results of the time evolution can be visualized with:
nemf.output_summary(output_dict)
```



which will create the following plot:

In the top you can see the different set tested during the fitting process. The bottom half shows the "fitted" model time evolution, which represents the frameworks best guess for the parameters. The reference data points are shown as dashed lines.

2.3 Enzymatic Reaction

[placesholder]

CHAPTER 3

A novel three step, network-based, general ecosystem modeling framework

Below, you can find the pre-published paper that provides a bigger picture of the framework and its key features.

A novel three step network-based general ecosystem modelling framework

CHAPTER 4

Installation

4.1 On Linux

Python and its package manger (pip) should be preinstalled. Hence, it can simply be installed through:

```
pip install nemf
```

4.2 On Windows:

4.2.1 Using pip

If you haven't installed python already, get the latest python version from here. This also installs pythons integrated package manager (pip).Then, the following install command can be used in windows *powershell*:

```
python -m pip install nemf
```

4.2.2 Using Anaconda

Alternatively, you can use *Anaconda* to manage both the python installation and your python packages.

To do so, download and install Anaconda from their homepage

After Anaconda is installed you can use *Anaconda Prompt* to install the package by executing

```
conda install nemf
```

4.3 On Mac

If you haven't installed python already, get the latest python version from here. Scroll down to get the *"macOS 64-bit installer"*

To installs pythons integrated package manager (pip).

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python get-pip.py
```

Then, the following install command can be used in the *terminal*:

```
python -m pip install nemf
```

4.3.1 Using Anaconda

Alternatively, you can use *Anaconda* to manage both the python installation and your python packages.

To do so, download and install Anaconda from their homepage

After Anaconda is installed you can use *Anaconda Prompt* to install the package by executing

```
conda install nemf
```

CHAPTER 5

Manual

Manual

5.1 Conceptual overview of library

5.1.1 ODE's and their time evolution

To keep it as general as possible we do not assume anything about the dependency of the model on the optimized input parameters. Therefore, we are required to study their effect on the model by examining the results of the integrated ODE's.

We assume a first-order coupled ODE system of the following form:

Where alpha is the set of real valued coefficients. A simple example would be something of the form:

Here, alpha takes the form:

5.1.2 Fitting

Stability and Suitability

We require that the desired state of the ODE system is a steady state of the system. From an optimization point of view,we usually also desire a certain steady state of the system. This gives us the following two constrains of stability and suitability:

Where u_{0} is the vector containing the quanteties observed in the ODE in their initial state (t=0). alpha is a matrix containg the coupling coefficients of them. G(u,alpha) is the map of time evolution of the system defined by u, alpha. The subscripts (i and j) represent the iteration step of the time evolution and both epsilon are small (compared to F(x)) positive real valued objects. The desired state of the system is represented by the vector y. The subscripts (i and j) represent the iteration step of the time evolution and both epsilon are two small (compared to G) positive real valued objects.

In general, the parameters we might desire to be optimised are elements of u and alpha. In the following we will refer to them as free parameters (x). As stated above they are

All elements of u_{0} and alpha are constant in the optimization process. Therefore, we will in the following only discuss the behaviour of the time evolution dependant on the free parameters F(x). F(x) is a different representation of the same map. Hence,

Cost/Loss function

As implied by the suitability condition, we can define a measure for how close the system represents the desired system. Keep in mind that we only consider steady state outputs of the integrated ODE F(x). This measure is usually referred to as 'objective function', but often also called cost- or loss function. Hence, for every set of free parameter (x) we can calculate a cost function J(x). The standard objective function is of the form

The effect of the different free parameters can be examined through a perturbation approach. The free parameters are perturbed slightly.

Theoretically, we could continue to sample the parameter space of the free parameters in this way or by testing a homogeneous distribution of free parameters over the sample space. However, this is computationally costly because it requires the evaluation of a large set of free parameters. Especially for a high dimensional search space, meaning a large set of observed quantities modelled by the ODE and/or a large set of free parameters, this becomes unfeasible.

Therefore we require a better sampling approach to reduce the computational cost of finding an optimal set of free parameters.

Gradient Descent

The approach chosen in this library is called 'Gradient Descent'. Its core idea is that we calculate a local gradient of the unknown cost function field and always move along the direction of the steepest descent. This way we find in principle the closest local minima without sampling the whole region but only one path. This drastically cuts the computational cost.

In its simplest form such a gradient descent is given by

where x^{n+1} is the next tested free parameter set and lambda is the step size, sometimes also called *learning rate*, of the gradient of J. The gradient always points into direction of the steepest ascent. Hence, by going into the opposite direction we are always walking downhill on the steepest path.

This approach is also used in many machine learning problems and therefore well studied. Its great advantage is that we only require the field of the cost function J(x,y) to be continuous and differentiable. This enables this approach to be applicable in many different problems. However, the generality of this approach introduces certain problems as well. Firstly, it is generally unknown if an ideal solution exists. Secondly, assuming such an ideal solution exists, it is not guaranteed that the algorithm finds it. Hence, if the best output of the algorithm is still not satisfactory, one does not know if the studied system does not have a better solution or if some parameter of the algorithm was badly chosen.

5.2 Overview of the YAML configuration file

The idea of this configuration file is to provide a simple, well structured, easily human-readable configuration of the full framework. A run of the framework is fully determined by this configuration file. It in This contains the configuration of the interaction network, the resulting ODEs, their time integration, the fitting methods and its configuration including the reference to the data that shall be used to fit.

If you first want to take a look at a rough overview of the library, see README.md. If you would like to get a better understanding how the library works internally, see README_concept.md.

Even a the small NPZD-type model that is presented in the library overview has numerous interactions and many more parameters and constants. In addition to the configuration of the model, we also need to define what parameters we want to fit and how. Therefore, the configuration of them can becomes become confusing and we we which need a clear structure to define them. To avoid loosing track of all these individual things, we store all in one well structured file. YAML provides a ideal format for this, since it is easily readable by humans as well as machines.

5.2.1 What is defined in the file?

The file contains three sections: Compartments, Interactions, Configuration. Compartments and Interaction are used to construct the network that is used to construct the set of ODEs. Configuration defines the methods for time integration and fitting.

Compartments

Compartments are the observed and modelled quantity of the ODE system. In the NPZD model, they are: (N)utritients, (P)hytoplankton, (Z)ooplankton, (D)etritus. All are counted in a shared currency, i.e. carbon mass.

The relevant information that needs to be defined is:

- name
- initial value (in the time evolution)
- upper and lower bound, if this value shall be fitted

In the YAML format this looks like the following:

```
compartment:            # header of the compartment section
  N:                 # name of compartment
    value: 1.0       # initial value of the compartment
    optimise:
      lower: 0       # lower and
      upper: 2       # upper bound during the fitting process
  P:
    value: 1.0
    optimise: null # no fitting required for this compartment
```

A more detailed description of all the possible options is provided further down.

Interaction

Interactions represent any flow from one compartment to an other. A interaction is described either by a function (keyword: 'fkt') defined in the framework library (see models.py/#interaction_models),or by a user defined function. The only condition to the function name provided in the yaml file is that has to match the function known to the python interpreter exactly.

In addition to the name of the compartments and their interaction function, we require the set of parameters used in the function to be defined. These can either be constants, or compartments. Furthermore, the direction of the flow needs to be defined ('sign').

We can define individual parameters to be fitted. This is done by providing the index of the parameter (i.e.: second parameter) and its range. Naturally, this does not apply if the selected parameter is a compartment because they are calculated during the time evolution.

A small example of the interaction of between N and P might look like the following:

```
interactions:
  # the functions are automatically multiplied by the value
  # of the second compartment
  N:P:                              # name of source:destination of the flow
  - fkt: nutrient_limited_growth    # function defining the type of flow
    parameters:
    - 'N'                           # the first parameter is the current value
    - 0.27                          # of the 'N' compartment
    - 0.7
    sign: '-1'                      # direction of flow (from P to N)
    optimise:
      - parameter_no: 2             # second parameter (0.27) is optimized
        lower: 0.1                  # and values in the range of
        upper: 0.3                  # [0.1,0.3] are allowed to be used
```

A crucial implementation detail is, that by default, an interaction is always linear with respect to the second named compartment. Meaning that if I want to model the following flow,

```
A:B:
  - fkt: f_AB
  - parameters:
    - 1
    - 2
  - sign: '-1'
  - optimise:
```

Configuration

This section of the file contains all technical implementation details of the ODE solver and inverse problem solver. Find a annotated example below.

```
# ode solver related details:
integration_scheme: euler_forward
time_evo_max: 100
dt_time_evo: 0.1
ode_coeff_model: interaction_model_generator

# to check if the ode time evolution is stable and converges:
stability_rel_tolerance: 1e-3
tail_length_stability_check: 10
start_stability_check: 50

# fitting related details
sources: null
sinks: null
fit_model: direct_fit_model
fit_target: [1,1,0,0]
```

ode solver related configuration details:

- **integration scheme**: name of the function that calculates the next time integration step. It can either be chosen from models.py/#integration_schemes, or defined by the user.
- **time_exo_max**: Ideally the ode system reached a stable state and stops automatically. If this is not the case it stops the latest at the ode-model time defined here.

- **dt_time_evo** defines the step size in ode-model time
- **ode_coeff_model**: Every step in the time time integration can be represented by a matrix multiplication. ode_coeff_model defines how this matrix is constructed. Usually, you do not want to change this from the default. However, it is implemented in such a way that this function can be defined by the user if needed.

convergence check configurations:

- **start_stability_check**: time step in model time after which the convergence of the ode-model output is checked.
- **stability_check_tail_length** defines the number last ode-model outputs (hence, tail) to use to check the stability
- **stability_rel_tolerance**: to test the convergence, we define a range in which the model is allowed to fluctuate. This range is defined relatively to its value. Hence, a value of 1e-3 allows the ode-model to fluctuate by one permille. If this range is exceeded the model is (not yet) stable.

fitting related details:

- **fit_model**: The ODE-model output can be of arbitrary shape. Depending on its application, there are many possible derived quantities you might want to fit your model to. One of the simplest examples could be that the model reaches a certain steady-state. However, more complex derived complex quantities might also be of interest. I.e. for a ecosystem model it might be desirable that your model does have a net-zero energy flow. All this can be done with a suitable choice of the the fit_model function. A fit model can either be chosen from the models.py/#fit_models functions or be by the user.
- **fit_target**: defines that output value or the set of values that your model is fitted to. In the case that a zero net flux is desired would be set to zero.
- **sources/sinks**: marks certain compartments as sources or sinks, needed for a net-flux fit model.

5.3 Interaction Functions

A short description what interaction functions are, how they are used and how they are implemented.

5.3.1 General

Compartments in the model are linked through flows. These flows, are characterized through, what we call, interaction functions. Each interaction function precisely describe the change per time step between two compartments. [ref paper for details]

The user has to deicide what type of interaction suits best to describe an interaction between to compartments. A common examples of such types are *holling type* functional responses. A more simple example might be simple linear mortality, where a percentage of compartment dies and is transformed into some sort of detritus.

5.3.2 Implementation

The framework offers a wide range of typical functional responses to be chosen as an interaction function.

Conceptually

Conceptually a interaction function *f* is of the form: f(a,b,c, ...) -> y where *a,b,c* are the interaction parameters, such a natural mortality rate or the amount of predators. *y* is a scalar representing the change per time step in the "flow-destination" compartment.

The user has to decide on which function *f* they want to use to represent a certain interaction, as well as to provide the function dependent parameter *a,b,c* ... This could be for example a linear mortality of the formf(alpha,A) = alpha * A where alpha represents a mortality rate, expressed in units of (unit of A)/time.

Technically

For technical reasons this is implemented slightly different. Each interaction function also takes the a list of all compartments, including the name or index of the origin and destination compartment. Hence, the previous function signature ends up looking like this:

f(X,[i_origin,j_destination],a,b,c, ...) -> y,

where X represents the array containing all compartments, and i and j represent the corresponding indices of the origin and destination compartments.

5.3.3 Units

In Ecology, or in Biology in general there are often many different units used to describe similar things. Therefore, it can sometimes be difficult to gather all the necessary information in a consistent form to use them in interaction functions.

Particular attention is therefore required when creating the interaction functions, as any wrongly interpreted or transformed unit may render the model invalid.

In general, the unit of an interaction function, as for a compartment, are arbitrary. It is up for the user to decide what kind of unit they may like to use to represent their model. However, this design choice has to be consistent in the entire model.

As the interaction functions represent changes in model compartment over time, they need to be expressed as a rate of the same unit as the corresponding compartment that they are summed up with.

I.e.: Assume we wish to represent the flow between two compartments *A* and *B*. Compartment B is described in units of gram. If we wish to describe a flow from compartment A to compartment B, this flow needs also to be expressed in units of gram as it is effectively summed up with the quantity in compartment B.

We strongly recommend, that when designing an interaction function, a sanity check is performed to verify that all units used add up to the necessary one. I.e.: In a simple example the flow from A to B might be represented by a linear function *f_AB* of the form:f_AB = alpha * A

Because f_AB is added to B for every time step dt, f_AB needs to take the form of a rate. If unit_B is the unit of the compartment B and the time steps are expressed in seconds than f_AB needs to add up to the unit unit_B/s. Hence, alpha needs to be in the unit 1/s. Otherwise the interaction is invalid, and with it the resulting model.

In general compartments might have different units. However, to keep the models simple and avoid further confusion in the choice of unit we designed the framework ins such a way that all compartments need to share a common unit. An example of such a shared unit might be carbon mass in kg, in contrast to a wet weight of a certain species.

5.3.4 Renaming or providing user-defined interaction functions

Even a simple model can contain many different interactions. However, often many of those interaction share the same underlying behavior. I.e. a natural mortality might scale linearly with the total population just as a exudation

process might scale linearly as well. For model simplicity we recommend using the same function to represent these interactions. Nevertheless, it might be helpful to distinguish the processes with different names. This is especially helpful when drawing a larger model as a network, as it helps to identify the actual processes one wants to describe.

For this reason, the model allows to rename existing interaction functions easily by the user. This can be achieved by two different ways:

- The first option is to define a list of the alternative names in the yaml configuration file. An example of this might look like:

```
configuration:
  [...]
  alternative_interaction_names:
    'alternative_name_one': 'existing_function'
    'alternative_name_two': 'existing_function'
```

- The second option is to define them in the python script that is executed to run the model. In that case the newly written or renamed functions need to be past to the function *import_interaction_functions([func1,func2,...])*

  To give an example:

  - In the case of renaming a existing function

```
import nemf
alternative_name_one = existing_function
alternative_name_two = existing_function

nemf.model.import_interaction_functions([alternative_name_one,alternative_
↪name_two])
```

  - When a new interaction function is written, it has to use the same signature as the existing ones. See interaction functions code documentation

    Currently something like this would be expected:

```
def new_function(X, ii, jj, args):
    [...] # calculates the changes per time step
    return df

import nemf
nemf.model.import_interaction_functions([new_function])
```

5.4 Reference Data

5.4.1 Why?

A major part of the NEMF framework is its capability to inverse model or fit a designed model output to data. These data sets might for example be measurements take in the field. They could also be some sort of expected behavior by the modeler. I.e. something like: "if my assumptions are correct, I expect this to converge to twice the size of the other compartment" However they were created, the user needs to provide them somehow to the framework. The framework expects the data in a standardized form. This ensures that the data is interpreted correctly while avoiding tedious data import configurations by the user,

5.4.2 Format

The format required is sometimes referred to as the "tidy-data" format. We expect rows to be 'observations' of the system. The columns represent the compartments in the model.

5.4.3 Example

Assume that A & B are names of two compartments in the model. Further, assume that t_i is the time at which the i'th observation took place. A(t_i) represents the value of the compartment A at the specific moment in time.

A data set containing several observations may then look like the following table:

time	A	B
t_1	A(t_1)	B(t_1)
t_2	A(t_2)	B(t_2)
t_3	A(t_3)	B(t_3)

An other potential outcome is that the desired state of the model is a steady-state. Meaning, that the described compartment is expected to converge to a certain value. In that case the we expect that the compartment will reach the steady-state value at t=infinity. Such situation is expected to be described by

time	A	B
inf	A(inf)	B(inf)

The framework does not specifically enforce a convergence as this behaviors is governed by the model. Enforcing to only consider steady-states in the optimization process is currently no possible in the framework. This means that the model might reach the desired values A(inf) & B(inf) at the end of the time evolution while the model has not (yet) converged. To avoid such an output it is possible to specify the the steady-state solution repeatably as in the first table. However, the values for A and B remain the same for all time steps. This suppress non-steady-states solutions in the model optimization.

Note: It is not possible to mix the 'inf' timestamp with date-time values in the reference data sets. If both is present the optimizer will default to the non-inf type fitting if the 'inf' timestamp is not set in the very row of the data set.

5.4.4 Time format

[placeholder]

- posix
  - integer counting seconds since 1.1.1970 UTC
  - also negative possible, seconds before 1970
- Datetime
  - which datetime formats
  - how in excel/csv

5.4.5 File format

Their are two data types that are supported:

- **plain text files**, (typical file extensions are '.txt' or '.csv')
- **excel files** (.xlsx)

The plain text files are required to be in a specific standard to be correctly interpreted by the framework. Besides the general structure introduced in the previous section we require the headers to be named exactly the same as the compartments in the model. This is necessary to match them without any potential misinterpretation. Any columns which do not have a exact counterpart in the model will be ignored.

Plain text files

Every data sets needs to contain a header defining the names of the columns which are compared to the model compartments. The names of the compartment are required to be separated by the chosen delimiter (default `','`)

An example of such a file can be found in the GitHub Repository

The headers has some flexibility how it is defined. The line may start with a comment symbol (default: "#')

```
# Datetime, A, B, [...]
```

Leading and trailing white spaces in a name are ignored. Hence,

```
Datetime,A,B = Datetime, A, B
```

It is also possible to write the name inside of quotation marks. This is typically used to mark character strings. Hence,

```
"Datetime","A","B" = 'Datetime','A','B' = Datetime, A, B
```

Note: If quotation marks are used, the names are parsed literally. As a result of that, leading and trailing whitespaces are no longer ignored.

Generally, it is assumed that the names of the column are defined in the in the very first row of the document.

```
1 Datetime, A, B
2 t0,A(t0),B(t0)
```

If this is not the case, i.e. the file is formatted something like this,

```
Some additional information about the file and its origin
before the data column headers are parsed
[Data]
```

the framework might not know how to interpret it. There are two option how to deal with this. We recommend using the following format to avoid this problem.

- Use the comment mark (#) for the non-header lines of the data while *NO* comment mark is used for the column headers. I.e:

  This will work,

```
# Some additional information about the file and its origin
# before the data column headers are parsed
Datetime, A, B
[...]
```

this will *NOT* work

```
# Some additional information about the file and its origin
# before the data column headers are parsed
# Datetime, A, B
[...]
```

while THIS will also work.

```
# Datetime, A, B
[...]
```

- Alternatively, the *load_ref_data()* method has the option to *"skip_header= "*. However, this requires a manual re-import of the data and is not recommended.

```
model = nemf.load_model('path/to/model.yml')
model.load_ref_data('path/to/reference/data.csv',skip_header=5)
```

Excel files

CHAPTER 6

API reference

6.1 Importing Model

nemf.**load_model**(*model_path*, *ref_data_path=None*)

Loads a model by reading its model configuration from from file

Parameters

- **model_path** (*string*) – path to yaml file containing the model
- **ref_data_path** (*string (optional)*) – path to file, plain text or xls(x), containing reference data to the model. See: https://nemf.readthedocs.io/en/latest/README_reference_data.html

Returns **model** – class objects that contains the model

Return type nemf_model class

6.2 Model Functions

class nemf.models.**model_class**(*model_path*, *ref_data_path=None*)

load_reference_data(*ref_data_path=None*, ***kwargs*)

Loads reference data used in model optimization from file

Either, the path to the reference data is provided in the yaml configuration file or passed to this function. Latter overwrites the path in the configuration file.

Parameters **ref_data_path** (*string (optional)*) – path to the file containing the reference data

nemf.models.**import_interaction_functions**(*func*)

Adds the functions from 'func' to the globals in models

func [list] List containing the functions that will be added to globals

6.3 Visualizing Model

nemf.**interaction_graph**(*model*)
shows a graph/network of all compartments and their interactions

6.4 Forcasting Model

nemf.**forward_model**(*model*, *method='RK45'*, *verbose=False*, *t_eval=None*)
Runs the time integration for a provided model configuration.

model [model_class object] class object containing the model configuration and its related methods. See load_configuration

method [string, optional]

Type of solver used for the initial-value problem aka forecasting. Should be on of:

- 'RK45' (default): Explicit Runge-Kutta method of order 5(4) [1]_.

The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula (local extrapolation is done). A quartic interpolation polynomial is used for the dense output [2]_. Can be applied in the complex domain.

- 'RK23': Explicit Runge-Kutta method of order 3(2) [3]_. The error is controlled assuming accuracy of the second-order method, but steps are taken using the third-order accurate formula (local extrapolation is done). A cubic Hermite polynomial is used for the dense output. Can be applied in the complex domain.
- 'DOP853': Explicit Runge-Kutta method of order 8 [13]_. Python implementation of the "DOP853" algorithm originally written in Fortran [14]_. A 7-th order interpolation polynomial accurate to 7-th order is used for the dense output. Can be applied in the complex domain.
- 'Radau': Implicit Runge-Kutta method of the Radau IIA family of order 5 [4]_. The error is controlled with a third-order accurate embedded formula. A cubic polynomial which satisfies the collocation conditions is used for the dense output.
- 'BDF': Implicit multi-step variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation [5]_. The implementation follows the one described in [6]_. A quasi-constant step scheme is used and accuracy is enhanced using the NDF modification. Can be applied in the complex domain.
- 'LSODA': Adams/BDF method with automatic stiffness detection and switching [7]_, [8]_. This is a wrapper of the Fortran solver from ODEPACK.

Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used

for non-stiff problems and implicit methods ('Radau', 'BDF') for stiff problems [9]_. Among Runge-Kutta methods, 'DOP853' is recommended for solving with high precision (low values of *rtol* and *atol*). If not sure, first try to run 'RK45'. If it makes unusually many iterations, diverges, or fails, your problem is likely to be stiff and you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

verbose [bool, optional] Flag for extra verbosity during runtime

t_eval [1d-array, optional] contains time stamps in posix time for which a solution shall be found and returned.

model [model_class object] class object containing the model configuration, model run results, and its related methods

References

6.5 Fitting Model

nemf.**inverse_model**(*model*, *nlp_method='SLSQP'*, *ivp_method='Radau'*, *sample_sets=3*, *maxiter=1000*, *seed=137*, *verbose=False*, *debug=False*)

Fits the model to data.

Optimizes a set of randomly generated free parameters and returns their optimized values and the corresponding fit-model and cost- function output

model [model_class object] class object containing the model configuration and its related methods. See load_configuration()

nlp_method [string, optional] Type of solver for the non-linear-programming problem aka fitting. Should be one of:

- 'trust-constr'
- 'SLSQP'
- 'L-BFGS-B'
- 'TNC
- 'Powell'

For problems with constraints use one of:

- 'trust-constr'
- 'SLSQP'

ivp_method [string, optional]

Type of solver used for the initial-value problem aka forecasting. Should be on of:

- **'Radau' (default):**

  Implicit Runge-Kutta method of the Radau IIA family

  – 'RK45': Explicit Runge-Kutta method of order 5(4) [1]_.

- 'RK23': Explicit Runge-Kutta method of order 3(2) [3]_.
- 'DOP853': Explicit Runge-Kutta method of order 8 [13]_.
- 'BDF': Implicit multi-step variable-order (1 to 5) method
- 'LSODA': Adams/BDF method with automatic stiffness detection

  Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used

for non-stiff problems and implicit methods ('Radau', 'BDF') for stiff problems [9]_. Among Runge-Kutta methods, 'DOP853' is recommended for solving with high precision (low values of *rtol* and *atol*). If not sure, first try to run 'RK45'. If it makes unusually many iterations, diverges, or fails,

your problem is likely to be stiff and you should use ‘Radau’ or ‘BDF’. ‘LSODA’ can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

sample_sets [positive integer, optional] Amount of randomly generated sample sets used as initial free parameters

maxiter [positive integer, optional] Maximal amount of iterations allowed in the gradient descent algorithm.

seed [positive integer, optional] Initializes the random number generator. Used to recreate the same set of pseudo-random numbers. Helpfull when debugging.

verbose [boo, optional] Flag for extra verbosity during runtime

model [model_class object] class object containing the model configuration, model run results (parameters, model, prediction, cost), and its related methods

Bound-Constrained minimization

Method *L-BFGS-B* uses the L-BFGS-B algorithm [B6], [B7] for bound constrained minimization.

Method *Powell* is a modification of Powell's method [B3], [B4] which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken. If bounds are not provided, then an unbounded line search will be used. If bounds are provided and the initial guess is within the bounds, then every function evaluation throughout the minimization procedure will be within the bounds. If bounds are provided, the initial guess is outside the bounds, and *direc* is full rank (default has full rank), then some function evaluations during the first iteration may be outside the bounds, but every function evaluation after the first iteration will be within the bounds. If *direc* is not full rank, then some parameters may not be optimized and the solution is not guaranteed to be within the bounds.

Method *TNC* uses a truncated Newton

algorithm [B5], [B8] to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *Newton-CG* method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

Constrained Minimization

Method SLSQP uses Sequential Least SQuares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [B12]. Note that the wrapper handles infinite values in bounds by converting them into large floating values. Method trust-constr is a trust-region algorithm for constrained optimization. It swiches between two implementations depending on the problem definition. It is the most versatile constrained minimization algorithm implemented in SciPy and the most appropriate for large-scale problems. For equality constrained problems it is an implementation of Byrd-Omojokun Trust-Region SQP method described in [B17] and in [B5], p. 549. When inequality constraints are imposed as well, it swiches to the trust-region interior point method described in [B16]. This interior point algorithm, in turn, solves inequality constraints by introducing slack variables and solving a sequence of equality-constrained barrier problems for progressively smaller values of the barrier parameter. The previously described equality constrained SQP method is used to solve the subproblems with increasing levels of accuracy as the iterate gets closer to a solution.

The available options are:

- **‘RK45’ (default): Explicit Runge-Kutta method of order 5(4) [1]_.** The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula (local extrapolation is done). A quartic interpolation polynomial is used for the dense output [2]_. Can be applied in the complex domain.

- **'RK23': Explicit Runge-Kutta method of order 3(2) [3]_. The error** is controlled assuming accuracy of the second-order method, but steps are taken using the third-order accurate formula (local extrapolation is done). A cubic Hermite polynomial is used for the dense output. Can be applied in the complex domain.
- **'DOP853': Explicit Runge-Kutta method of order 8 [13]_.** Python implementation of the "DOP853" algorithm originally written in Fortran [14]_. A 7-th order interpolation polynomial accurate to 7-th order is used for the dense output. Can be applied in the complex domain.
- **'Radau': Implicit Runge-Kutta method of the Radau IIA family of** order 5 [4]_. The error is controlled with a third-order accurate embedded formula. A cubic polynomial which satisfies the collocation conditions is used for the dense output.
- **'BDF': Implicit multi-step variable-order (1 to 5) method based** on a backward differentiation formula for the derivative approximation [5]_. The implementation follows the one described in [6]_. A quasi-constant step scheme is used and accuracy is enhanced using the NDF modification. Can be applied in the complex domain.
- **'LSODA': Adams/BDF method with automatic stiffness detection and** switching [7]_, [8]_. This is a wrapper of the Fortran solver from ODEPACK.

Explicit Runge-Kutta methods ('RK23', 'RK45', 'DOP853') should be used for non-stiff problems and implicit methods ('Radau', 'BDF') for stiff problems [9]_. Among Runge-Kutta methods, 'DOP853' is recommended for solving with high precision (low values of *rtol* and *atol*). If not sure, first try to run 'RK45'. If it makes unusually many iterations, diverges, or fails, your problem is likely to be stiff and you should use 'Radau' or 'BDF'. 'LSODA' can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

References

6.6 Visualizing Results

nemf.**output_summary**(*model*)

reads the data saved in the model class and depending on this data chooses a visualization method with the help of draw_output_summary to present the results

CHAPTER 7

Interaction Functions

Here a list of all currently implemented interaction functions is presented. These can be used in the YAML model configuration to describe interactions between to compartments.

Note: The compartments are referenced via indices in the implementation of the interaction functions. This is simply an implementation detail. The user can (and should) use the compartment names when referencing them in the YAML configuration file. The framework handles the mapping of the names to the corresponding indeces internally.

nemf.interaction_functions.**excretion**(*X*, *idx_A*, *idx_B*, *coefficient*)
linear response with respect to *origin/prey* compartment

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns df – change in the origin and destitnation compartment. Calculated by coefficient*origin_compartment

Return type float

nemf.interaction_functions.**exudation**(*X*, *idx_A*, *idx_B*, *coefficient*)
linear response with respect to *origin/prey* compartment

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment

- **coefficient** (*float*) – governs the slope of the linear response

Returns df – change in the origin and destitnation compartment. Calculated by coefficient*origin_compartment

Return type float

nemf.interaction_functions.**holling_type_0**(*X*, *idx_A*, *coefficient*)

linear response with respect to *destination/predator* compartment

For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns df – change in the origin and destitnation compartment. Calculated by coefficient*destination_compartment

Return type float

nemf.interaction_functions.**holling_type_I**(*X*, *idx_A*, *idx_B*, *coefficient*)

linear response with respect to both *origin/pray* and *destination/predator* compartment.

For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns df – change in the origin and destitnation compartment. Calculated by coefficient * origin_compartment * destination_compartment

Return type float

nemf.interaction_functions.**holling_type_II**(*X*, *idx_A*, *idx_B*, *food_processing_time*, *hunting_rate*)

non-linear response with respect to *origin/pray* with linear response with respect to *destination/predator* compartment

The response with respect to the origin compartment 'B' is approximately linear for small 'B' and converges towards an upper limit governed by the 'food_processing_time' for large 'B'. For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns **df** – change in the origin and destitnation compartment. Calculated by consumption_rate = ((hunting_rate * origin_compartment) / (1 + hunting_rate * food_processing_time * origin_compartment)) * destination_compartment

Return type float

nemf.interaction_functions.**holling_type_III**(*X*, *idx_A*, *idx_B*, *saturation_rate*, *consumption_rate_limit*)

non-linear response with respect to *origin/pray* with linear response with respect to *destination/predator* compartment

The response with respect to the origin compartment 'B' is approximately quadratic for small 'B' and converges towards an upper limit governed by the 'food_processing_time' for large 'B'. For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **saturation_rate** (*float*) – first parameter of the interaction. governs the slope of the non lineare response for small pray populations.
- **consumption_rate_limit** (*float*) – second parameter of the interaction. governs the upper limit of the response.

Returns **df** – change in the origin and destitnation compartment. Calculated by consumption_rate = ((consumption_rate_limit * saturation_rate * B**2)/ (consumption_rate_limit + (saturation_rate*B**2)))*A

Return type float

nemf.interaction_functions.**inverse_type_0**(*X*, *idx_A*, *idx_B*, *coefficient*)

linear response with respect to *origin/prey* compartment

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns **df** – change in the origin and destitnation compartment. Calculated by coefficient*origin_compartment

Return type float

nemf.interaction_functions.**linear_mortality**(*X*, *idx_A*, *idx_B*, *coefficient*)

linear response with respect to *origin/prey* compartment

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment

- **coefficient** (*float*) – governs the slope of the linear response

Returns **df** – change in the origin and destitnation compartment. Calculated by coefficient*origin_compartment

Return type float

nemf.interaction_functions.**nutrient_limited_growth**(*X*, *idx_A*, *idx_B*, *growth_rate*, *half_saturation*)

non-linear response with respect to *destination/predator* compartment

Similar to holling_type_II and is a reparameterization of holling II. The response with respect to the origin compartment 'B' is approximately linear for small 'B' and converges towards an upper limit governed by the 'growth_rate' for large 'B'. For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **growth_rate** (*float*) – first parameter of the interaction. governs the upper limit of the response.
- **half_saturation** (*float*) – second parameter of the interaction. governs the slope of the response.

Returns **df** – change in the origin and destitnation compartment. Calculated by consumption_rate = ((hunting_rate * origin_compartment) / (1 + hunting_rate * food_processing_time * origin_compartment)) * destination_compartment

Return type float

nemf.interaction_functions.**remineralisation**(*X*, *idx_A*, *idx_B*, *coefficient*)

linear response with respect to both *origin/pray* and *destination/predator* compartment.

For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns **df** – change in the origin and destitnation compartment. Calculated by coefficient * origin_compartment * destination_compartment

Return type float

nemf.interaction_functions.**sloppy_feeding**(*holling_type*, *coeff*, **args*)

calls holling_type functions with an extra "efficiency" coefficient. the inverse of the efficiency is then supposed to flow into a different compartment

nemf.interaction_functions.**stress_dependant_exudation**(*X*, *idx_A*, *idx_B*, *coefficient*)

linear response with respect to both *origin/pray* and *destination/predator* compartment.

For examples see: Examples

Parameters

- **X** (*np.array*) – containing the current state of the contained quantity of each compartment
- **idx_A** (*integer*) – index of the element representing the destination/predator compartment
- **idx_B** (*integer*) – index of the element representing the origin/pray compartment
- **coefficient** (*float*) – governs the slope of the linear response

Returns **df** – change in the origin and destitnation compartment. Calculated by coefficient * origin_compartment * destination_compartment

Return type float

CHAPTER 8

Releases

8.1 v0.3.3 (June 2020)

This is a minor release introducing new options for both non-linear-programming aka fitting solvers and initial-value-problem aka forecasting solvers. It also brings some bugfixes and consistency changes as well as additional documentation.

8.2 New Non-Linear-Programming (NLP) solvers

Previously, only the two solvers that worked in all cases (bounded, constraint & bounded) were properly implemented.

However, both 'SLSQP' and 'trust-constr' showed to be sometimes outperformed by other solvers. 'trust-constr' showed to be much slower, and 'SLSQP' while fast sometimes exited without actually optimizing the result. For that reason we added additional solves which are available in the case where no constraints are present.

The full list of solvers is now:* 'SLSQP' * 'trust-constr' * **'L-BFGS-B'** * **'TNC'** * **'Powell'** where the bold ones are newly implemented. To distinguish the NLP and IVP solvers, the key used to pass them to the inverse_model has been changed from 'method' to 'nlp_method'. For more detail, see the API references

8.3 New Initial-Value-Problem (IVP) solvers

Previously, the IVP solver was not free for the user to change and always defaulted to de 'RK45' solver. However, the 'RK45' showed to have some issues. In some cases it got stuck, mostly because the numerical errors resulted in negative compartment values even if the ODE would not allow it. Hence, the error increased over time drastically and the output, if calculated in a reasonable time were unusable.

Other solvers showed to performed better in these circumstances. The default solver is now 'Radau'. While it is generally slower then the 'RK45' it showed to perform better in the above described circumstances.

The full list of IVP solvers is now: * ‘Radau’ (default) * ‘RK45’ * ‘RK23’ * ‘DOP853’ * ‘BDF’ * ‘LSODA’ To distinguish the NLP and IVP solvers, the key used to pass them is called ‘ivp_method’. For more detail, see the API references

8.4 Bug fixed and consistency changes

8.4.1 Changes

- Different verbosity options can now be used for ‘trust-constr’, providing more information during runtime. See API references for more details.
- Changed the output of the *load_reference_data()* function. Previously it returned the output directly. Hence, the user had to pass it back to the model. Because there is (currently) no application to use it without passing it back to the model it does this now automatically.

8.4.2 Bugs

- *load_model()* did not allow to not pass a reference data path as presented in the examples. This is now possible.
- *load_reference_data()* did not correctly accept keyword arguments but only non-keyword ones. This as an error as all optional ones are kwargs.

8.4.3 Minor internal changes

- *internal:* In some cases the reference data sets have been referred to as ‘fit data’ because it has been at some point during development used exclusively for fitting. Because is does not need to be used to fit anymore but can also be used to plot, we now call it reference data everywhere.
- *internal*: The data type used to describe the bounds of parameters during the optimization process was inconsistent. The compartments used list while the interactions used sets. This was not also inconsistent but also caused issues as some solvers did not parse them correctly. Now, all bounds are passed as lists.

8.5 v0.3.2 (June 2020)

This is a minor release introducing the new name “nemf” of the framework as well as a improved reference data import and some changes in the plotting of the framework output.

8.5.1 New name

The framework has been renamed **G**eneral **E**cosystem **M**odelling **F**ramework (GEMF) to **N**etwork-based **E**cosystem **M**odelling **F**ramework (NEMF).

While it still remains a *general* modelling framework, the name change shall emphasis the core distinguishing concept that distinguishes it from other other modelling frameworks. Additionally, the network concept is key concept that users need to work with and keep in mind while using the framework, while the *general*-aspekt fades into the background once the framework has been chosen.

8.5.2 Better reference data input

- Previously reference data had to be provided for all modelled compartments. It is now possible to provide reference data for only a subset of the compartment.
  - Datasets must now contain a header that contains the names of the columns which will be compared to the compartment names.
  - Data that is present in the reference data set which does not correspond to modelled compartment will now be simply ignored and does not need be removed manually. However, this also requires the header names to match the compartment names exactly.
- New feature of "Datetime" timestamps for reference data in excel files. "Datetimes" (i.e. '18.09.1783 12:34') are automatically transformed into the default POSIX timestamp (seconds before/after 1.1.1970).

8.5.3 Plotting changes

- *Major*: Output summary plots now automatically use scatter plots instead of line plots if the amount of data points is low. This has two main reasons:
  - Plots with only one point are now also proprly drawn. (Lines require two points, and are drawn empty if only one is present)
  - Avoids implying a linear behavior between two distant points, which can confuse the user while interpret ing the plot.
- *Minor*: Implemented a new color selection scheme when plotting. This avoids that colors are reused when many lines are present. Additionally, it makes sure that the reference data and model output use the same colors, even if only a subset of the model compartments is present in the refer data.

8.5.4 Other changes

- The interaction function *nutrition_limited_growth* has been renamed to *nutrient_limited_growth*.
- *load_model* now also allows the import of reference data.
- Many new descriptions and comments have been added to the documentation

8.6 v0.3.1 (June 2020)

This is a minor release introducing a user-friendly option to rename interaction functions as well as constraints for optimized parameters. An initial documentation has been drafted and is hosted on readthedocs.io.

8.6.1 Importing and renaming interaction functions

- *init_alternative_interaction_names( )*This is called when ever a new model i initialized. It checks if alternative names for an existing interaction were declared in the yaml configuration file. For details check the docs.
- *import_interaction_functions( )*This can be called in the execution file by the user to import user-defined or renamed interaction functions. For details check the docs.

The previous version of renaming interaction functions was to declare the alternative names in the execution file. However, this did not work consistently in all execution environments. Therefore, we now introduce to dedicated functions to rename and/or import interaction functions.

8.6.2 Importing constraints

- Added *import_constraints( )*Reads constraints from an python file and adds them to the model. For details check the docs.

In some circumstances, the parameter fitted in an optimization run are not independent of each other. I.e. one might want to enforce that sum of two parameters is always equal to one. Such a behavior is enforced through 'constraints'.

8.6.3 Documentation

- Hosted the current drafts of the documentation on readthedocs.io.
- Documentation created via sphinx
- Added automated API reference creation based on docstrings via autodoc

8.7 v0.3.0 (May 2020)

This is a major release introducing scipy's integration and optimization routines into the framework.

8.7.1 Replaced reference data import

- Removed the old reference data input which only allowed for simple steady-state fitting
- New version requires the data to be stored in a separate file and imported either by passing the link in the yaml file or when calling the inverse_method routine. The latter overwrites the path given in the yaml file.

8.7.2 New forward modelling

- Replaced the self written *time_integration* routine with *scipy.integrate.solve_ivp*.
- Adjusted *forward_model* and plotting routines accordingly and introduced supporting functions.

8.7.3 New inverse modelling

- Replaced the self written *gradient_descent* method with *scipy.optimize.minimize*.
- Adjusted *inverse_model* and plotting routines accordingly and introduced supporting functions.

8.7.4 Visualization

- Added *initial_guess* plotting function to visualize the model output before any optimization has been applied.

Bibliography

[B3] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal 7: 155-162.

[B4] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.

[B5] Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.

[B6] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208.

[B7] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550-560.

[B8] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770-778.

[B12] Kraft, D. A software package for sequential quadratic programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center – Institute for Flight Mechanics, Koln, Germany.

[B16] Byrd, Richard H., Mary E. Hribar, and Jorge Nocedal. 1999. An interior point algorithm for large-scale nonlinear programming. SIAM Journal on Optimization 9.4: 877-900.

[B17] Lalee, Marucha, Jorge Nocedal, and Todd Plantega. 1998. On the implementation of an algorithm for large-scale equality constrained optimization. SIAM Journal on Optimization 8.3: 682-706.

Python Module Index

n

nemf, 31
nemf.interaction_functions, 37
nemf.models, 31

Index

E

excretion() (*in module nemf.interaction_functions*), 37
exudation() (*in module nemf.interaction_functions*), 37

H

holling_type_0() (*in module nemf.interaction_functions*), 38
holling_type_I() (*in module nemf.interaction_functions*), 38
holling_type_II() (*in module nemf.interaction_functions*), 38
holling_type_III() (*in module nemf.interaction_functions*), 39

I

import_interaction_functions() (*in module nemf.models*), 31
inverse_type_0() (*in module nemf.interaction_functions*), 39

L

linear_mortality() (*in module nemf.interaction_functions*), 39
load_model() (*in module nemf*), 31
load_reference_data() (*nemf.models.model_class method*), 31

M

model_class (*class in nemf.models*), 31

N

nemf (*module*), 31
nemf.interaction_functions (*module*), 37
nemf.models (*module*), 31
nutrient_limited_growth() (*in module nemf.interaction_functions*), 40

R

remineralisation() (*in module nemf.interaction_functions*), 40

S

sloppy_feeding() (*in module nemf.interaction_functions*), 40
stress_dependant_exudation() (*in module nemf.interaction_functions*), 40