dynamic-community-fba

unknown

Jul 05, 2023

CONTENTS

Welcome to the documentation of dynamic-community-fba. The documentation provides you with a comprehensive guidance on using the dynamic-community-fba Python package for modeling dynamic flux balance analysis (FBA) in microbial consortia. By utilizing the Genome Scale Metabolic Models (GSMM's) of your organisms of interest, our package enables you to analyze and explore the intricate interactions among two or more organisms.

The package covers three different ways for modelling microbial consortia's:

- Parallel FBA [refference to paper]
- Joint dynamic FBA [refference to paper]
- endPointFBA

The documentation is designed to assist users in understanding the concepts, usage, and implementation of these dynamic FBA techniques. One of the key features of dynamic-community-fba is the ability to construct and export a community matrix, which represents the joint stoichiometry matrix of the provided GSMM's models. The documentation provides in-depth explanations on how to build and utilize this matrix to study the dynamics of multi-organism interactions and metabolic networks.

To ensure a smooth start, the documentation includes a section outlining the prerequisites and installation guide. Users will find step-by-step instructions on installing the necessary dependencies and setting up the dynamic-community-fba package in their Python environment. Additionally, the documentation highlights the compatibility requirements and recommends best practices for a successful installation. By following the documentation, users will gain a comprehensive understanding of the dynamic-community-fba package and its capabilities. They will learn how to leverage its functionalities to perform Parallel FBA, joint FBA, and endPointFBA analyses, enabling them to study the dynamic behavior of metabolic networks in diverse biological systems.

The dynamic-community-fba documentation serves as a valuable resource for researchers, scientists, and students working in the field of systems biology, metabolic engineering, and microbial ecology. It empowers users to effectively model and analyze dynamic flux balance analyses, facilitating a deeper understanding of the complex interactions between organisms in various biological systems.

CHAPTER

ONE

1. INSTALLATION AND REQUIREMENTS

1.1 Prerequisites

The dynamic-community-fba package relies on the *cbmpy* library[1] for handling constraint-based metabolic models. Like cobrapy *cbmpy*` is a Python package that simplifies the creation, loading, and manipulation of constraint-based models, allowing interaction with LP-solvers like 'cplex' or 'GLPK'.

To install *cbmpy*, you can use the following command:

```
pip install cbmpy
```

For more information and detailed documentation on using *cbmpy*, please refer to the cbmpy GitHub repository and the cbmpy documentation.

1.2 Dynamic Community FBA

After the installation of *cbmpy* you can install dynamic community FBA using the following command

```
pip install NAME
```

More text if needed about the installation

1.3 Escher

Maybe we can write some easy converter functions for known maps. To display the models have to think about this

[1] PySCeS Constraint Based Modelling (http://cbmpy.sourceforge.net) Copyright (C) 2010-2023 Brett G. Olivier, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

CHAPTER

TWO

2. GETTING STARTED

With the package a the models for *E. coli core metabolism*, *Streptococcus thermophilus* and *Lactobacillus delbrueckii*, here we will show you the basic operations for loading and modifying a cbmpy model. If you've worked with *cobrapy* previously you will see *cbmpy* is not that different. If this is your first time working with GSMM's and FBA we recommend you to start with the extensive cbmpy tutorial (if we have completed this)

2.1 2.1. Loading a model using cbmpy

To load a model and perform a simple FBA analysis on it type:

```
import cbmpy
from cbmpy.CBModel import Model

model: Model = cbmpy.loadModel("data/bigg_models/e_coli_core.xml")
cbmpy.doFBA(model)
FBAsol = model.getSolutionVector(names=True)
FBAsol = dict(zip(FBAsol[1], FBAsol[0]))
```

Here the model refers to a `cbmpy.CBModel` which represents the GSMM of the loaded organism .

2.2 2.2. SBML and cobra models

The `cbmpy.load_model()` function is designed to efficiently handle a wide range of models. It seamlessly supports the import of models encoded in the standardized Systems Biology Markup Language (SBML) format, as well as models exported by *cobrapy*. This means that you can easily work with different versions of SBML and *cobrapy* models without having to specify them explicitly. This flexibility simplifies the model loading process.

Note: Sometimes the conversion of exchanges, sinks or other boundary conditions are not properly set when exporting or importing a *cobra* model into *cbmpy* therefore always check if these reactions are set correctly in the loaded model.*

2.3 2.3. Saving a model

There are two ways to save a cbmpy model. The easiest way is to save your altered model to the latest version of SBML:

```
reaction = model.getReaction("R_EX_glc__D_e") #Get a reaction from the model
reaction.setLowerBound(0) #Alter the reaction in the model

cbmpy.saveModel(model, "adjusted_model.xml") #Save the new model to a XML file
```

If you don't want to save the model to to the latest version of SBML or you wan't to save it to a *cobrapy* model you can call one of the below functions:

```
cbmpy.writeCOBRASBML(...)
cbmpy.writeFVAtoCSV(...)
cbmpy.writeModelToExcel97(...)
cbmpy.writeSBML3FBCV2(...)
```

2.4 2.4. Reactions, Reagents and Species

In *cbmpy*, the `cbmpy.CBModel` object forms the basis of the model. When working with the model, most modifications will involve manipulating this object. In the previous section, we demonstrated how to load the *E. coli core metabolism*. Now, let's explore some basic alterations that can be made to the model. For a more comprehensive understanding of the functionalities available in the `cbmpy.CBModel` object and other features of cbmpy, we recommend referring to the extensive documentation.

2.4.1 Reactions

To list all the reactions in the model, or list the reaction containing a certain string you can call the following functions:

```
modelRxns = model.getReactionIds() #All the reactions, as a list[str]
print(modelRxns)

model.getReactionIds('PG') #Outputs only reactions with "PG" in their ID
```

Once you have identified your reaction of interest, you can easily access its key details, including the reagents, upper and lower bounds, and equation, as follows:

```
from cbmpy.CBModel import Reaction, Reagent, Species

reaction: Reaction = model.getReaction("R_PGK")

reagents: list[Reagent] = reaction.getReagentObjIds()  # Get all reagent ids of the␣
↪reaction
print(reagents)

bounds = [reaction.getLowerBound(), reaction.getUpperBound()] # Get the lower and upper␣
↪bound
print(bounds)

equation = reaction.getEquation() # Get the reactions equation
print(equation)
```

Furthermore you can check if a reaction is reversible and if it is an exchange reaction:

```
print(reaction.is_exchange) #True if the reaction is an exchange reaction

print(reaction.reversible) #True if the reaction is reversible
```

You can easily add your own defined reactions to the model using `model.createReaction()`, if we for example want to add the reaction: `ATP + H2O -> ADP + Pi` we can do this with the following code:

```
model.createReaction('ATPsink', reversible = False) # Create a new empty irreversible␣
↪reaction

# Add the reagents to the reaction, All metabolites already existed in the model so we␣
↪did not
# Need to create them
model.createReactionReagent('ATPsink', metabolite = "M_atp_c" , coefficient = -1)
model.createReactionReagent('ATPsink', metabolite = "M_adp_c", coefficient =1)
model.createReactionReagent('ATPsink', metabolite =  "M_h2o_c", coefficient = -1)
model.createReactionReagent('ATPsink', metabolite = "M_pi_c" , coefficient = 1)
```

2.4.2 Reagents

Tohe `Reagent` class represents a reagent within a reaction, providing essential information about its properties and characteristics. Within the class, users can access and manipulate the reagents associated with a specific reaction within the model. The reagent itself is linked to a `Species` which we will cover shortly. You can access a reagent by retrieving it from an instance of the `Reaction` class, given the *R_PGK* reaction from the previous example we can access information about a reagent as follows:

```
reagent: Reagent = reaction.getReagent("R_PGK_M_3pg_c")

reagent.getCoefficient() # Get the reagent's stoichiometric coefficient

reagent.getCompartmentId() #Get the compartment

reagent.getSpecies() # Get the species id corresponding to this reagent
```

If a reagent has a negative coefficient it is consumed by the reaction, if the reagent has a positive coefficient it is created by the reaction.

2.4.3 Species

Species represent the metabolites in the system using the `Species` object you can easily retrieve details such as the species' molecular formula, charge, and compartment information. Furthermore you can list the reactions in which a species is consumed or created

```
species: Species = model.getSpecies("M_pi_c")

species.getChemFormula()
species.getCharge()
species.getCompartmentId() # Gives the id of the compartment in which the species lives
species.isReagentOf() # Returns a list of reaction ids in which the species is present
```

2.4.4 Objective function

To perform FBA on the model you need to set an objective function. This is the reaction for which the maximal or minimal flux will be calculated. To check what the active objective function of the model is you can write:

```
objective_ids = model.getActiveObjectiveReactionIds()
#['R_BIOMASS_Ecoli_core_w_GAM']

objective = model.getActiveObjective()
objective.getOperation()
#Maximize
```

If you would call the function `cbmpy.doFBA(model)` FBA will calculate the fluxes such that the flux through the reaction with id *R_BIOMASS_Ecoli_core_w_GAM* is maximal.

2.5 2.5. Transitioning to cbmpy from cobrapy

As previously mentioned, any model build using *cobrapy* or any other toolbox for that matter can easily be opened in *cbmpy*. By just exporting your model of interest to either SBML format or to a cobra model you can import it as an cbmpy model.

Next we will explore how cbmpy models can be used to model the behaviors of microbial communities.

CHAPTER

THREE

3. PARALLEL DYNAMIC FBA

CHAPTER

FOUR

4. THE COMMUNITY MODEL

4.1 4.1. Definition

Here we will introduce the concept of the community model, which plays a vital role in upcoming modeling techniques. In simple terms, the community model represents the combined stoichiometry matrices of N Genome-Scale Metabolic Models (GSMMs) for performing Flux Balance Analysis (FBA).

To streamline the handling of multiple models, we have developed the `CommunityModel` class. The class offers a structured representation of combined metabolic networks, integrating stoichiometric information from individual GSMMs. This facilitates in-depth analysis of complex microbial communities, their dynamics, and metabolic potentials.

The documentation delves into the underlying principles and rules governing the community model's construction, ensuring effective utilization and preventing erroneous model creation. Additionally, we provide practical usage examples to further illustrate the versatility of the `CommunityModel` class.

4.2 4.2. Creating and modifying the community model

To initialize a `CommunityModel` you have to give a list of N GSMMs as well as there biomass reaction ids:

```
import cbmpy
from endPointFBA import CommunityModel

model1 = cbmpy.loadModel("data/bigg_models/e_coli_core.xml")
model2 = cbmpy.loadModel("data/bigg_models/strep_therm.xml")

biomass_reaction_model_1 = "R_BIOMASS_Ecoli_core_w_GAM"
biomass_reaction_model_2 = "R_biomass_STR"
community_model: CommunityModel = CommunityModel(
    [model1, model2],
    [
        biomass_reaction_model_1,
        biomass_reaction_model_2,
    ],
)
```

If you are not familiar with the biomass reaction ID, there are a couple of ways to identify it. First, you can look it up in the SBML model. Alternatively, you can check if the active objective function of the model is set to the biomass reaction by using the following code: `model.getActiveObjectiveReactionIds()`. This will display a list of objective IDs, and you can verify if the biomass reaction is included.

In this example, we have utilized two models included in the package: *E. coli core metabolism* and *Streptococcus thermophilus*. If desired, you can provide an optional list of alternative IDs to refer to the single models. This can be done by supplying a list of strings as the third argument to the function. If no alternative IDs are provided, the community matrix will default to using the model identifiers. Lastly, you have the option to assign an ID of your own choosing to the new combined model. If not provided the default ID will be used.

It is important to note that the `CommunityModel` class is derived from the base `cbmpy.CBModel` class, meaning that the `CommunityModel` is still an instance of `cbmpy.CBModel`. With the newly initialized object we can obtain even more information. Check the API for all functionality.

Warning: If you want to create a community model of N identical models, it is mandatory to specify the alternative ids. Without providing alternative IDs, both models would have the same ID during the community model creation process. As a result, the code would be unable to distinguish between the two models, leading to undesired behavior.

4.3 4.3. Rules

Maybe move this to an advanced section?

During the initialization of the community model a new `cbmpy.CBModel` object is created. In the new object all compartments, reactions, reagents and species are copied from the provided model. The following considerations were made for each:

4.3.1 Compartments:

The new model comprises a total of $\Sigma_{i=1}^{n}(c_i - 1) + 1$ compartments, where c represents the number of compartments in each model i.

In the new model, all compartments from the individual models are copied, except for the *external* or *e* compartment. The external compartment is reserved to be added at the end. All other compartment ids get a prefix of the id of the model they belonged to such that we know which compartment corresponds to which organism.

This design ensures that there is only one external compartment in which all species and reactions from all models coexist.

By following this approach, the new model achieves compartmental organization while consolidating all species and reactions within a unified external compartment.

4.3.2 Reactions:

Just like the compartments, all reactions are duplicated, and each reaction ID is augmented with its corresponding model ID. This holds up for all reactions except for the exchange reactions. If two models share an exchange reaction only one is saved in the new combined model such that we have more control over all.

4.3.3 Reagents and species

Before the reactions can be copied to the new model there is a check for which species occur in more than one model. For these species a new species in the original model is created and all reactions associated to this species have there reagents changed. By doing so the initial models already have everything set correctly into place to have the reactions copied.

In contrast with the compartment IDs, and the reaction IDs the species IDs are not changed by default. But only if the species id occurs in two or more models. This is done since we can already quickly lookup to which original model the species belonged by checking the compartment which it is in.

Note: It is crucial to verify that the identical reactions and species within different models have consistent IDs before creating the community model. This is particularly significant for exchange reactions and species localized in the extracellular space. If these IDs are not uniform, despite referring to the same reactions or species, the CommunityModel class cannot determine their equivalence accurately.

Please ensure that the corresponding IDs for these reactions and species are harmonized to guarantee the proper functioning of the CommunityModel.

CHAPTER

FIVE

5. DYNAMIC JOINT FBA

The multiple metabolic models of different or the same organism that were combined in the CommunityModel as described in the previous chapter can now be used in dynamic joint FBA. The community model incorporates the metabolic reactions and interactions between the organisms, allowing for the study of their collective behavior and the emergent properties of the community as a whole.

The joint FBA approach enables the investigation of metabolic exchanges, such as the exchange of nutrients or byproducts, between organisms within the community. By simulating the community-level metabolic interactions, researchers can gain insights into the dependencies, cooperation, competition, and overall dynamics of the organisms in the community.

5.1 Joint FBA

After defining the `CommunityModel` it is easy to refine it in such a way that we can perform a Joint FBA The only thing left to do is append the biomass reaction of each individual model to create the so called *Community biomass*. So now we can define the reaction `X_c -> ` where `X_c` is the community biomass.

To perform joint FBA run the following:

```
import cbmpy
from cbmpy.CBModel import Model
from endPointFBA.CommunityModel import CommunityModel
from endPointFBA.DynamicJointFBA import DynamicJointFBA

model1: Model = cbmpy.loadModel("data/bigg_models/e_coli_core.xml")
model_2 = model1.clone()

combined_model = CommunityModel(
    [model1, model_2],
    ["R_BIOMASS_Ecoli_core_w_GAM", "R_BIOMASS_Ecoli_core_w_GAM"],
    ["ecoli_1", "ecoli_2"],
)  # Create a CommunityModel of two  E. coli strains competing for resources


# Create the joint FBA object with initial biomasses and the initial concentration of␣
↪glucose
dynamic_fba = DynamicJointFBA(
    combined_model,
    [0.1, 0.1],
    {"M_glc__D_e": 10},
)
```

(continues on next page)

(continued from previous page)

```
# Perform FBA on the new joint FBA model object
solution = cbmpy.doFBA(dynamic_fba.get_joint_model())
print(solution)
```

5.2 Making it dynamic

Dynamic FBA is an extension of the traditional FBA approach that incorporates the element of time. In dynamic FBA, a specific time step *dt* is selected, and the concentrations of external metabolites and biomass concentrations are calculated at each time point. This enables the modeling and analysis of dynamic processes such as metabolic fluxes, nutrient uptake, and product secretion over time.

The same technique can be applied for using the previously described Joint FBA which we call *Dynamic Joint FBA*

To perform the joint FBA over time using the `DynamicJointFBA` model:

```
dynamic_fba.simulate(0.1)
```

You can now easily plot the solution:

```
import matplotlib.pyplot as plt

ts, metabolites, biomasses = dynamic_fba.simulate(0.1)


ax = plt.subplot(1616)
ax.plot(ts, biomasses["ecoli_1"])
ax2 = plt.twinx(ax)
ax2.plot(ts, metabolites["X_c"], color="r")

ax.set_ylabel("Biomass ecoli 1", color="b")
ax2.set_ylabel("X_comm", color="r")

ax3 = plt.twinx(ax)
ax3.plot(ts, biomasses["ecoli_2"], color="y")
ax3.set_ylabel("Biomass ecoli 2", color="y")

plt.show()
```

5.3 Add reaction kinetics

In construction

5.3.1 Write your own kinetics!

In construction

CHAPTER

SIX

6. ENDPOINTFBA

CHAPTER

SEVEN

7. FAQ

CHAPTER

EIGHT

8. API

8.1 CommunityModel

`class endPointFBA.CommunityModel.`**`CommunityModel`**`(`*models: list[cbmpy.CBModel.Model]*, *biomass_reaction_ids: list[str]*, *ids: list[str] = []*, *combined_model_id: str = 'combined_model'*`)`

Bases: `Model`

`add_model_to_community``(`*model: Model*, *biomass_reaction: str*, *new_id: str | None = None*`)`

Adds a model to the CommunityModel

Args:

model (Model): The model that needs to be added biomass_reaction (str): The reaction id of the biomass reaction of the new model new_id (str, optional): The user set identifier. Defaults to None. If set to None the model.id will be used

`get_model_biomass_ids``()` → dict[str, str]

`get_model_specific_reactions``(`*mid: str*`)` → list[cbmpy.CBModel.Reaction]

Returns a list of reaction ids

Args:

mid (str): _description_

Raises:

NotInCombinedModel: _description_

Returns:

list[Reaction]: _description_

`get_model_specific_species``(`*mid: str*`)` → list[cbmpy.CBModel.Species]

`get_reaction_bigg_ids``(`*mid=''*`)` → list[str]

Get the reaction bigg ids of all reactions

Args:

mid (str, optional): If a model id is provided only reactions from the specific model are returned Defaults to “”.

Raises:

NotInCombinedModel: the id provided was not in the combined model

Returns:

list[str]: list containing the bigg ids

`get_species_bigg_ids`(*mid=''*) → list[str]

Get the species bigg ids of all species

Args:
mid (str, optional): When provided only the ids of a specific model are returned. Defaults to "".

Raises:
NotInCombinedModel: the id provided was not in the combined model

Returns:
list[str]: list containing the bigg ids

`identify_biomass_of_model_from_reaction_id`(*rid*) → str

`identify_biomass_reaction_for_model`(*mid: str*) → list[str]

Given a model id return the biomass reaction

Args:
mid (str): _description_

Returns:
str: _description_

`identify_model_from_reaction`(*rid: str*) → str

Given a reaction id get the single model this reaction belonged to

Args:
rid (str): reaction id of the kinetic model

Returns:
str: id of the old model

`m_identifiers: list[str]`

`m_single_model_biomass_reaction_ids: list[str] = []`

`m_single_model_identifiers: list[str]`

8.2 KineticModel

class `endPointFBA.KineticModel.KineticStruct`(*kinetics: dict[str, tuple[float, float]]*)

Bases: `object`

`get_model`() → Model

`get_model_kinetics`() → dict[str, tuple[float, float]]

`get_reaction_kinetics`(*rid*) → tuple[float, float]

`get_reaction_km`(*rid*) → float

`get_reaction_vmax`(*rid*) → float

`m_kinetics: dict[str, tuple[float, float]]`

`set_kinetics`(*kinetics: dict[str, tuple[float, float]]*)

`set_model`(*model: Model*) → None

`set_reaction_kinetics`(*rid: str*, *kinetics: tuple[float, float]*)

8.3 DynamicJointFba

class endPointFBA.DynamicJointFBA.**DynamicJointFBA**(*model: CommunityModel*, *biomasses: list[float]*, *initial_concentrations: dict[str, float] = {}*, *kinetics={}*)

Bases: `DynamicFBABase`

get_joint_model() → *CommunityModel*

m_exporters: dict[str, list[str]]

m_importers: dict[str, list[str]]

m_initial_bounds: dict[str, tuple[float, float]] = {}

m_kinetics: dict[str, tuple[float, float]]

m_model: ***CommunityModel***

set_community_biomass_reaction(*model: CommunityModel*)

simulate(*dt: float*, *epsilon=0.001*, *user_func=None*)

update_bounds(*user_func*) → None

update_concentrations(*FBAsol*, *dt*)

update_importer_bounds(*reaction: Reaction*, *X: float*)

8.4 Module contents

PYTHON MODULE INDEX

INDEX