**MiMICS User Guide**

**Introduction to MiMICS**

Multi-scale Model of Metabolism In Cellular Systems, abbreviated MiMICS, is an extendable computational framework executed in Python and Java to simulate metabolism in 2D and 3D microbial communities. MiMICS couples a genome-scale metabolic network reconstruction (GENRE) with the established platform Hybrid Automata Library (HAL)1, which contains an agent-based model, and a continuum-scale reaction-diffusion model. Individual agents can represent single-cell bacteria or bacteria populations that exist 2D or 3D world, of which the dimensions can be defined by the user.

A key feature of MiMICS is the user’s ability to incorporate multiple -omics integrated GENREs, which can represent unique metabolic states that differ in predicted parameter values passed to the extracellular models. To generate multiple -omics integrated GENREs, we recommend algorithms such as RiPTIDe2 or GiMME3. Individual agents can decide which metabolic model state to execute based on mechanistic rules input by the user.

At each simulation time step, each MiMICS sub-model is performed to update agent properties and metabolite concentrations. For each agent, the biomass and metabolite concentration from the continuum-scale grid corresponding to the agent’s location is converted to a metabolite uptake flux used to constrain the agent’s GENRE. Constraint-based flux-balance analysis is used to optimize each agent’s GENRE to predict a biomass growth rate, as well as metabolite secretion and uptake fluxes. The biomass growth rate is passed to the ABM to update an agent’s biomass. Metabolite secretion and uptake fluxes are passed to the in the continuum-scale reaction-diffusion model to update the metabolite concentrations. In the ABM, bacteria agents can perform behaviors like cell division, motility, and cell shoving mechanics. Simulation outputs such as agent locations, agent intracellular metabolic fluxes and metabolite concentrations can be provided at each simulation time step.

**Overview of MiMICS framework structure**

Relevant agent parameter values are passed between Java and Python MiMICS models using a Py4J Gateway Server, which is first initialized by the user (Figure 1). Next, a MiMICS python file runs and calls upon methods in the MiMICS Java class to initialize MiMICS, diffuse metabolites, and run agent methods (Figure 1). Each agent’s metabolic model is optimized in Python using the Python package COBRApy and parallel computing (Figure 1). To perform a set of agent behaviors, the MiMICS Java class accesses individual agent methods defined in the Cell3D Java agent class. All Java classes (outlined in pink in Figure 1) can be compiled into a MIMICS.jar file to run MiMICS on a High Performance Computing system.

**A screenshot of a computer screen

Description automatically generated**

**Figure 1. Overview of MiMICS framework structure**. Code based in Python and Java are outlined in green and pink, respectively. A Py4J Gateway Server, initialized by the user, is required to pass agent parameter values between Python and Java models. The Java classes outlined in pink are used to compile a MIMICS.jar file that can be used to run MiMICS on a High Performance Computer.

**DEPENDENCIES**

Hybrid Automata Library (HAL) is a Java library that contains methods to execute the agent-based model and a reaction-diffusion model1. As recommended by HAL, MiMICS uses IntelliJ IDEA as the IDE to edit and run Java code, and build a compiled MIMICS.jar file. More detailed description of HAL’s applications, methods, and tutorials can be found in the HAL [publication](https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007635) and in the [HAL User Guide](https://halloworld.org/setup.html#getting_started).

Each agent’s metabolic network model was optimized using the [COBRApy](https://cobrapy.readthedocs.io/en/latest/) Python package. To improve MiMICS runtime, MiMICS uses the Multiprocessing Python package that facilitates parallel computing to asynchronously optimize the agent’s metabolic models across multiple central processing units (CPUs). The [Py4J](https://www.py4j.org/) python program sets up a Py4J Gateway Server, which is used to pass information between the HAL (Java) and metabolic model (Python) models.

**Computational Specifications**

Java v.15.0.2

Python v.3.9

Py4J v.0.10.9.1

IntelliJ IDEA v.2020.3.2 Community Edition

HAL v.1.1.0

**Overview of MiMICS simulation files**

1. *MIMICS.java*: contains the Cell3D agent class and MIMICS class. Contains methods to simulate an agent-based model and a reaction-diffusion model. Contains methods to save MiMICS agent and metabolite concentration outputs. Executed in Java.
2. *MIMICS\_gateway.java*: contains methods to start the Py4J Gateway Server, which interfaces MiMICS files written in Java and Python languages. Also contains methods to automatically start the MiMICS python file, *agents\_FBA.py,* if desired. Executed in Java.
3. *agents\_FBA.py*: optimizes each agent’s metabolic model. Contains functions to initialize the MiMICS agent and metabolite grids, pass agent information between the metabolic model (Python) and HAL (Java) using the Py4J Gateway Server, and run agent-based model and metabolite diffusion methods using the Py4J Gateway Server. Executed in python.

Forsimulation of MiMICS on a High Performance Computing system, a slurm script, was used, named *A\_MIMICS\_PS.slurm,* to submit simulation jobs.

**Installation of MiMICS**

Download the MIMICS source code here: <https://github.com/tracykuper/mimics>.

IntelliJ IDEA is a useful developer environment to edit and run Java and Python code. Download [IntelliJ IDEA](https://www.jetbrains.com/help/idea/installation-guide.html). Download HAL source code following HAL’s instructions, the download link located in the [HAL User Guide](https://halloworld.org/setup.html#getting_started), which contains the HAL method library. Set up the HAL library in a new IntelliJ IDEA project according to the instructions in the [HAL User Guide](https://halloworld.org/setup.html#getting_started). Name the new project *‘MIMICS-master’*.

One change is required to make in the HAL library code for MiMICS to run. Within the HAL library, navigate to HAL > GridsAndAgents > AgentsGrid3D. Within the AgentsGrid3D class, change the return method of the GetAgentsRadApprox method from *void* to *public void*.

Right-click on the *‘MIMICS-master’* project to create a new package named *‘MIMICS’.* Drag and drop the files *MIMICS.java*, *MIMICS\_gateway.java*, and *agents\_FBA.py* into the ‘*MIMICS’* package folder.

**Configure a Python Interpreter and add Py4J in the IntelliJ IDEA project.**

Because MiMICS is written partially in python, a python interpreter must be added to the IntelliJ IDEA project. Adding a python interpreter to an IntelliJ IDEA project is outlined [here](https://www.jetbrains.com/help/idea/creating-virtual-environment.html). Briefly, install the Python Plugin in IntelliJ IDEA. Next, navigate to *‘File’* > *‘Project Structure’* > *‘SDKs’* > Click *‘+’* > Select *‘Add Python SDK’* > Select *‘System Interpreter’* > In the *‘Interpreter’* text box, select the folder where the Python virtual environment exists > Click *‘OK’* > Click *‘Apply’*.

Check the necessary python packages are installed, which may include cobra, py4j, numpy, multiprocessing, sys, pandas, math, and time that are used in MiMICS. To add python packages, navigate to *‘File’* > *‘Project Structure’* > *‘SDKs’ >* Click on the python SDK *> ‘Packages’ >* Click *‘+’ >* Install the necessary packages.

Set the python interpreter in the *‘MIMICS-master’* project*.* Navigate to *‘File’* > *‘Project Structure’* > *‘Modules’* > Right click on you’re the project name *‘MIMICS-master’* > Click *‘+’* > *‘Python’* > Set the *‘Python Interpreter’* to the Python virtual environment in the *‘MIMICS-master’* project folder.

**Description of MiMICS Python File, *agents\_FBA.py***

The python file, *agents\_FBA.py*, optimizes each agent’s metabolic model, and executes the HAL agent-based model and reaction-diffusion model. Python runs each of these models for a defined number of simulation time steps set by the user. Python executes the agent-based model and reaction-diffusion model by calling functions defined in Java (described in the *MIMICS\_gateway.java* User Guide section) using the Py4J Gateway Server.

To call these Java functions, Python initializes accesses a Py4J Gateway Server, designated by a variable named *gateway* in the main function of *agents\_FBA.py*:

gateway = JavaGateway(); # INITIALIZE THE PY4J JAVA GATEWAY

To execute the methods outlined in *MIMICS\_gateway.java* from python, use the following code structure in agents\_FBA.py: gateway.entry\_point.*MIMICS\_gateway\_method*(), where gateway.entry\_point is used to access the PY4J Gateway Server entry point, and *MIMICS\_gateway\_method*() is the name of the desired method contained in *MIMICS\_gateway.java*.

**PY4J function to initialize MiMICS ABM and metabolite world**

In the *agents\_FBA.py* main program, the MiMICS agent-based model world and metabolite grid is initialized by calling the *run\_model0()* method defined in *MIMICS\_gateway.java*. In the code below, *num\_cell* corresponds to the initial number of agents*, initial\_oxygen* corresponds to an initial oxygen concentration, and *initial\_glucose* corresponds to an initial glucose concentration passed to HAL.

gateway.entry\_point.run\_model0(int(num\_cell),initial\_oxygen, initial\_glucose); # INITIALIZE MIMICS MODEL

**PY4J function to run MiMICS for multiple simulation time steps**

Defined in the *agents\_FBA.py* main program, the *run\_MIMICS()* python function contains a simulation time step *for loop*, which performs the optimization of agent metabolic models, agent-based model, and metabolite reaction-diffusion model for a desired number of simulation time steps. The code below shows the *run\_MIMICS()* function with inputs of the number of simulation time steps (*num\_dt*), a list of metabolic models (*models*), an XML file of the nutrient media concentrations (*media*), a list of metabolite reaction IDs (*rxns*), the filename of the reaction flux output (*file\_name\_rxn*), the simulation job number (*job\_num*), initial oxygen concentration (*initial\_oxygen*), the oxygen diffusion coefficient (*D\_O2*), initial glucose concentration (*initial\_glucose*), and the glucose diffusion coefficient (*D\_G*)

run\_MIMICS(int(num\_dt),models,media,rxns,file\_name\_rxn,int(job\_num), initial\_oxygen, D\_O2, initial\_glucose, D\_G);

**PY4J function to execute agent methods from python**

Within thetime step *for loop of* *run\_MIMICS(),* agent-based model methods are first executed using the PY4J function *run\_model().*

**PY4J function to pass agent values from HAL (Java) to metabolic models (Python)**

Next, Within thetime step *for loop of* *run\_MIMICS()*, arrays of agent parameter values, including agent biomass, index, metabolic state, and local metabolite concentrations, are passed from HAL (Java) to Python by python calling the PY4J methods defined in *MIMICS\_gateway.java*. This agent information passed from HAL is used to define and optimize the agent’s metabolic models in Python.

This code block in the *run\_GENRE()* shows Python obtaining arrays containing each agents’ biomass, index, metabolite concentrations, and metabolic state assignment numbers.

biomass\_values = list(gateway.entry\_point.getBiomassFromHal()) ## GET AGENT BIOMASS FROM ABM  
biomass\_values = [x / 1e14 for x in biomass\_values]

pos\_count = len(biomass\_values) ## COUNT NUMBER OF AGENTS  
index\_values = list(gateway.entry\_point.getIndexFromHal()) ## GET AGENT INDEX FROM ABM

patch\_values\_from\_java = np.array(gateway.entry\_point.getPatchFromHal\_All()) ## GET METABOLITE CONCENTRATIONS AT AGENT'S PATCH FROM ABM

metabolic\_states = list(gateway.entry\_point.getMetabolicStateFromHal()) ## GET AGENT METABOLIC STATE FROM ABM

**Optimization of agent metabolic models in Python**

Next, agent information is passed to the *run\_GENRE()* multiprocessing function to optimize each agent’s metabolic model. Each agent’s metabolic model state, biomass, index, and metabolite concentrations are passed as inputs into the *run\_GENRE()* multiprocessing function. Within the *run\_GENRE()* function, an agent’s metabolic model state is assigned, defined by the *model* variable:

model = models[metabolic\_state] ## ASSIGN METABOLIC MODEL STATE TO AGENT

The exchange fluxes of the metabolic model are constrained on the agent’s local metabolic concentrations. The agent’s local patch metabolite concentrations and metabolic model exchange flux are converted with the following equation:

(Equation 1)

Where is the metabolite flux to set the GENRE exchange reaction lower bounds (mmol/(g \*hr),  is the agent’s local patch metabolite concentration (mM), is the patch volume (L), is the metabolite uptake time step, and is the agent biomass (g). The agent’s metabolic model exchange reaction bounds are then constrained by the corresponding metabolite exchange flux.

The agent’s metabolic model is optimized with COBRApy’s flux-balance analysis.

The predicted biomass synthesis rate was used as a growth rate to update the agent’s biomass using an exponential growth rate:

(Equation 2)

Where is the updated agent biomass in grams, is the initial biomass in grams, is the growth rate informed from the optimized GENRE (*hr-1*) and is the growth time step (*hr*).

In addition, in the *run\_GENRE()* function, the agent’s metabolic model predicted exchange metabolite flux values are converted to a metabolite concentration (Equation 1) to update the agent’s local extracellular metabolite concentrations:

(Equation 3)

where is the updated metabolite concentration (mM), is the initial metabolite concentration, and is the metabolite concentration consumed or secreted by the agent.

Next, in the *run\_GENRE()* function, the predicted intracellular and exchange reaction fluxes from each optimized agent’s metabolic model are obtained and passed as an output from the multiprocessing *run\_GENRE()* function.

Lastly, in the *run\_GENRE()* function, the outputs from the *run\_GENRE()* multiprocessing function contain the agent’s updated biomass, agent indices, updated metabolite concentrations where agents reside, and reaction fluxes from each agent’s metabolic model.

**PY4J function to pass agent values from metabolic models (Python) to HAL (Java)**

Next, within the time step for loop in the *run\_MIMICS()* function, the results from the agents metabolic models multiprocessing optimization are passed to the HAL Java environment to update agent properties and metabolite concentrations at the agent’s locations.

The agent’s updated parameter values are formatted into an array that can be read in Java. Each array contains an agent property with the parameter values indexed corresponding to the agent’s index. The arrays are formatted as a 1D or 2D *jvm* array that can be read in Java in the following Python code:

## INITIALIZE ARRAYS OF UPDATED AGENT INFORMATION TO BE PASSED TO JAVA ABM  
new\_biomass\_total = gateway.new\_array(gateway.jvm.double,int(pos\_count))  
growth\_rates\_total = gateway.new\_array(gateway.jvm.double,int(pos\_count))  
index\_values\_for\_java = gateway.new\_array(gateway.jvm.int,int(pos\_count))  
patch\_value\_for\_java = gateway.new\_array(gateway.jvm.double,int(pos\_count),2) # NUMBER OF METABOLITES

Next, python calls upon the PY4J methods, defined in *MIMICS\_gateway.java,* to pass the agent property arrays, and update the agent property values in the HAL Java environment. The below code demonstrates python passing values to update agent properties in the HAL Java environment, including updated value arrays containing each agent’s growth rate, biomass, and metabolite concentrations where each agent is located.

## SEND UPDATED GROWTH RATE, BIOMASS, METABOLITE CONCENTRATIONS TO JAVA ABM  
gateway.entry\_point.setGrowthRateFromPython(growth\_rates\_total, index\_values\_for\_java)  
gateway.entry\_point.setBiomassFromPython(new\_biomass\_total, index\_values\_for\_java)  
gateway.entry\_point.setPatchFromPython(patch\_value\_for\_java,index\_values\_for\_java)

**Diffuse metabolites**

Next in the time step for loop in the *run\_MIMICS()* function, python calls upon the PY4J method Diffuse\_Metabolites() defined in the *MIMICS\_gateway.java* to perform metabolite diffusion.

# SAVE MODEL OUTPUTS: AGENT AND METABOLITE PATCH INFORMATION  
gateway.entry\_point.Diffuse\_Metabolites(initial\_oxygen, D\_O2, initial\_glucose, D\_G)

**Save MiMICS simulation outputs**

Lastly in the time step for loop in the *run\_MIMICS()* function, python calls upon the PY4J methods Save\_cell\_info() and Save\_met\_info(), both methods defined in the *MIMICS\_gateway.java*, to save MiMICS simulation outputs of agent properties and metabolite concentrations.

# SAVE MODEL OUTPUTS: AGENT AND METABOLITE PATCH INFORMATION  
if t == 0:  
 gateway.Save\_cell\_info(int(t), int(job\_num))  
if t >0:  
 gateway.Save\_cell\_info(int(t), int(job\_num))  
times\_save=[0,25,50,91,92,93,94,95,96,97,98,99]  
if t in times\_save:  
 gateway.Save\_met\_info(int(t), int(job\_num))

In this code, *job\_num* is the variable that contains an integer of the simulation job input by the user and *t* is the simulation time step.

Note in this example code, Save\_met\_info() was only run for desired simulation time points. If this 3D metabolite grid is very large, we recommend this because the output file from Save\_met\_info() can be very large datasets if saved for more than 10-20 time step.

The PY4J method Save\_cell\_info() saves a CSV file named ‘agent\_properties*#*.csv’, where is contains the simulation job number. This agent property CSV file contains the parameter values for each agent’s property, where rows correspond to each agent and columns correspond a unique agent’s property. The agent property CSV file contains each agent’s x,y,z coordinates, index, biomass, growth rate, metabolic state, and metabolite concentrations at the agent’s location. In addition, the simulation time step and simulation job number are included. Other desired parameter values related to MiMICS can be added by the user.

The PY4J method Save\_met\_info() saves a CSV file named ‘met\_grid*#*.csv’, where *#* is contains the simulation job number. This metabolite grid CSV file contains the metabolite concentrations at each metabolite grid location, where rows correspond to each x,y,z coordinate value and columns correspond to each metabolite (i.e. oxygen, glucose).

In addition, each agent’s metabolic model predicted intracellular and exchange reaction flux values are saved as a CSV file directly from python, named ‘rxn\_fluxes*#*.csv’, where *#* contains an integer of the simulation job number.

**Description of MiMICS Java file, *MIMICS\_gateway.java***

MiMICS implements a Py4J Gateway Server to facilitate passing parameter values for each agent between the metabolic model Python environment and HAL Java environment. Methods to pass information through the Py4J Gateway Server are written in MIMICS\_gateway class in the *MIMICS\_gateway.java* file, and are defined below:

run\_model0( int b0 ): method that calls MIMICS class methods to initialize agents and initialize metabolite concentrations. The input parameter *b0* is the initial number of agents.

run\_model(): method that calls the MIMICS class method StepCells() to execute agent-based model functions.

Diffuse\_Metabolites(double initial\_oxygen, double D\_O2, double initial\_glucose, double D\_G): method that calls MIMICS class metabolite diffusion methods, i.e. Gas\_Diffuse() and Carbon\_Diffuse(), to simulate metabolite diffusion. The inputs initial\_oxygen and initial\_glucose are initial concentrations of oxygen and glucose, respectively. D\_O2 and D\_G are oxygen and glucose diffusion coefficients, respectively.

Save\_met\_info( *“inputs”* ): method that calls MIMICS class method Save\_Met\_Info( *“inputs”* ). Recommended inputs are the simulation time step and simulation job number.

Save\_cell\_info( *“inputs”* ): method that calls MIMICS class method Save Cell\_Info( *“inputs”* ). Recommended inputs are the simulation time step, simulation job number, and parameter values used to define agent metabolic state decisions.

getBiomassFromHal(): method to pass biomass (units: grams) of each agent, contained as a double array in order of the agent’s index, from HAL (Java) to the metabolic model (Python).

getIndexFromHal(): method to pass the index of each agent, contained as an integer array in order of the agent’s index, from HAL (Java) to the metabolic model (Python).

getMetabolicStateFromHal(): method to pass the metabolic state of each agent, contained as an integer array in order of the agent’s index, from HAL (Java) to the metabolic model (Python).

getPatchFromHal\_All(): method to pass the metabolite concentrations at each agent’s grid location, contained as an double 2d array in order of the agent’s index, from HAL (Java) to the metabolic model (Python).

setPatchFromPython(double[][] met, int[] index): method to pass updated metabolite concentration values and agent index values, as a double 2d array and integer array, respectively, in order of the agent’s index, from the metabolic model (Python) to HAL (Java). This method is used to update the metabolite grid concentrations that were predicted to be consumed or secreted by an agent’s metabolic model.

setGrowthRateFromPython(double[] growth\_rate, int[] index): method to pass the agent’s predicted growth rate and index, each contained in a double and integer array, respectively, and in order of the agent’s index, from the metabolic model (Python) to HAL (Java). This method is used to update the agent’s growth rate.

setBiomassFromPython(double[] mass, int[] index): method to pass the agent’s updated biomass and index, each contained in a double and integer array, respectively, and in order of the agent’s index, from the metabolic model (Python) to HAL (Java). This method is used to update the agent’s biomass.

main() function: method used to initialize the Py4J Gateway Server and calls to run the *agents\_FBA.py* file. Once the *agents\_FBA.py* file is complete, the MIMICS\_gateway main() method automatically shuts down the Py4J Gateway Server. Agent-based model and metabolite grid dimensions are defined by the user here:

* + xdim: x-dimension, in units of micrometers scaled to the patch size
  + ydim: y-dimension, in units of micrometers scaled to the patch size
  + zdim: z-dimensions, in units of micrometers scaled to the patch size

**Description of MiMICS Java file, *MIMICS.java***

The MIMICS model class defines methods for extracellular metabolite initialization and diffusion, agent initialization and execution of agent class methods, saving simulation outputs, and interfacing with the Py4J Gateway Server.

**MiMICS extracellular reaction-diffusion model**

The reaction-diffusion metabolite model methods used in MiMICS were constructed using the HAL reaction-diffusion metabolite model library. Detailed information about these methods can be found in the HAL User Guide or in HAL tutorial files.

Extracellular metabolite grids are initialized in the MIMICS class constructor in *MIMICS.java.* Metabolite initialization and diffusion methods are written in the MIMICS class constructor in *MIMICS.java.* Metabolite concentrations were initialized by the method Initialize\_Metabolites(), which is called by the MIMICS initialization method, run\_model0() in *MIMICS\_gateway.java*. Diffusion of gaseous metabolites and carbon substrates was simulated by Gas\_Diffuse() and Carbon\_Diffuse() methods, respectively. Each metabolite diffusion method is called in Diffuse\_Metabolites() methods in *MIMICS\_gateway.java.*

**MiMICS agent initialization**

Agents were randomly initialized in space by the method Initialize\_Random( *int b0* ), where *b0* is an integer representing the initial number of cells input by the user.

**Ordering execution of agent class methods**

Agent class methods were executed in a defined order for each agent by the method StepCells( *“inputs”* ). *“inputs”* represents any input parameter values required to as inputs for agent class methods.

**Saving MiMICS simulation outputs**

The MIMICS class contains methods to save simulation outputs of agent locations and metabolic states, and extracellular metabolite concentrations.

Save\_Cell\_Info( *“inputs”* ) method saves information about each agents (x, y, z) coordinates, index, metabolic state, biomass, growth rate, and local metabolite concentration. In addition, the simulation time step, desired simulation parameter values, and the simulation job number are saved. The output filename from this method is ‘agent properties#.csv’, where the # is the simulation job number. *“inputs”* into the Save\_Cell\_Info() method are recommended to be the simulation time step, simulation job number, total number of agents, and values of any other desired parameter values.

Save\_Met\_Info( *“inputs”* ) method saves information about the extracellular metabolite concentrations in each of the metabolite grid locations. The output filename from this method is ‘met\_grid#.csv’, where the # is the simulation job number. *“inputs”* into the Save\_Met\_Info() method are recommended to be the simulation time step and simulation job number.

**MiMICS agent java class methods**

The agent class methods used in MiMICS were constructed using the HAL agent-based model library. Detailed information about the agent class methods can be found in the HAL User Guide or in HAL tutorial files.

Init( “*inputs”* ): initializes agent properties such as agent biomass, index, growth rate, metabolic state, motility state, force parameters, directional angle, size, and time scale for mRNA synthesis of a specific metabolic state.

* *“inputs”* can include input values to initialize the agent properties.

ForceCalc(double overlap, Cell3D other), CalcMove(), and MoveDiv(): calculates and enforces force calculations between agents to prevent agent overlap. Parameter values for these methods (i.e. overlap radius, force\_scaler, friction coefficient) are defined in the MIMICS class constructer. More information about the force calculations can be found in the HAL User Guide or in HAL tutorial files.

Biomass\_Divide(int pop): simulates cellular division when agent biomass is above a maximum biomass threshold. Calls the Init() method to initialize a daughter agent.

* Integer “pop” was used as assign the index for new daughter agents.

Change\_metabolic\_state( “*inputs”* ): simulates cell’s switching metabolic states based on mechanistic rules defined by the user (i.e. cells sensing the extracellular metabolite environment, stochastic rules)

* “*inputs”* can include metabolite concentration thresholds and stochastic proportion thresholds for agents to switch metabolic states
* For each rule, update the agent’s metabolic state assignment number, i.e. “ this.metabolic\_state = # ” . The metabolic state number maps to a desired metabolic model (example shown in Table 1).
* If desired, reset the time scale for mRNA synthesis of other metabolic states in this code section.

**Printing statements from MiMICS**

To print statements from MiMICS when simulating on a personal computer, you can print statements directly from Python and Java as normal.

When running on a High Performance Computer, MiMICS does not easily allow for print statements from Python. If needed, the PY4J function *Print\_Phrase()* can be used to pass statements from Python to Java to print. An example code is shown below. The *sarray* variable defines a string that can be read and printed in Java.

sarray = gateway.new\_array(gateway.jvm.java.lang.String,2) ## STRING FOR PRINTING FROM JAVA  
sarray[0]="WORKER FAILED" ## ERROR STATEMENT  
gateway.entry\_point.Print\_Phrase(sarray[0]) ## PRINT ERROR STATEMENT USING PY4J GATEWAY

**Running MiMICS on personal computer**

For populations of less than 1,000 simulated agents, MiMICS may efficiently run on a personal computer. If desired, particularly for troubleshooting code, the the uncompiled MiMICS source code can be run on a personal computer using IntelliJ IDEA. Compared to the compiled MIMICS.jar file, running the uncompiled source code MIMICS files is more advantageous to receive error or print statements from both the Python and Java model components.

To run the uncompiled MiMICS source code on a personal computer, open *agents\_FBA.py,* *MIMICS\_gateway.java*, and *MIMICS.java* in IntelliJ IDEA. Run *MIMICS\_gateway.java* file to open the Py4J Gateway Sever.

The following command in the *MIMICS\_gateway.java* main method can be used to execute the *agents\_FBA.py* file automatically from *MIMICS\_gateway.java.*

String command = "python agents\_FBA.py"; // PYTHON FILE TO OPTIMIZE EACH AGENT'S GENRE  
Process p = Runtime.*getRuntime*().exec(command); // CALL PYTHON FILE  
p.waitFor();  
  
// PRINT STATEMENTS CALLED FROM THE PYTHON FILE  
try (BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()))) {  
 String line;  
 while ((line = br.readLine()) != null) {  
 System.*out*.println(line);  
 }  
}  
gatewayServer.shutdown(); //SHUT DOWN JAVA GATEWAY SERVER WHEN MULTI-SCALE MODEL IS FINISHED

If desired, the function above can be commented out so that the *agents\_FBA.py* can be manually executed by the user. Ensure the *MIMICS\_gateway.java* is running first so that a Py4J Gateway Server is open to receive commands from python, then run *agents\_FBA.py*.

**Running MiMICS on a High Performance Computer**

The MIMICS.jar file is advantageous to easily run MiMICS on a high performance computing (HPC) system that provides large CPU resources for parallel processing. In addition, HPC systems can provide multiple computing nodes to simultaneously run multiple MiMICS simulations for replicate simulations or simulation perturbations. More information to compile a .jar file with IntelliJ IDEA can be found [here.](https://www.jetbrains.com/help/idea/compiling-applications.html#compile_module)

To compile the MIMICS.jar file in IntelliJ IDEA:

1. Select *‘File’* > select *‘Project Structure’*
2. Select *‘Artifacts’* from the side bar menu > Click *‘+’* > select *‘JAR’* > select *‘From Modules with Dependencies’*
3. In the *‘Create Jar From Modules’* window > select *‘Main Class’* > select *‘Project’* > select the *‘MIMICS\_gateway’* class > Click *‘OK’* to exit.
4. Select *‘Apply’* in *‘Project Structure’*. Click ‘*OK’* to exit.
5. In the main toolbar select ‘*Build’* > select *‘Build Artifacts’* > in the *‘Action’* menu, select *‘Build’*. The MIMICS.jar file will be compiled and saved in the artifacts folder of the project.

A slurm job script can be used to execute the MIMICS.jar file in the HPC environment. In addition, the slurm script can define the number of desired nodes, number of tasks, number of tasks per node, number of CPUS per tasks, total time for one simulation, a result output file, computing partition, and memory for one simulation. Shown here, because a Py4J Gateway Server is set up for each MIMICS simulation, only one MIMICS simulation can be performed on one node (i.e. multiple MIMICS simulations **can not** be performed on one computing node). Therefore, the ntasks-per-node slurm parameter, which defines the number of MiMICS simulations per computing node, **must be set to one** to avoid an error.

The slurm script also defines simulation parameter values (in this code example O2\_t, NO\_t), number of CPUs, and simulation replicate numbers. These values are exported and used in the *agents\_FBA.py* file.

Below is an example of a slurm job script:

#!/bin/bash

#SBATCH –ntasks=15

#SBATCH –nodes=15

#SBATCH –ntasks-per-node=1

#SBATCH –cpus-per-task=35

#SBATCH –time=0-00:45:00

#SBATCH –output=result%a

#SBATCH –partition=parallel

#SBATCH –mem=45000

module load anaconda

module load java

O2\_T\_LIST =(“0.19” “0.19” “0.19” “0.2” “0.2” “0.2” “0.21” “0.21” “0.21” “0.22” “0.22” “0.22” “0.23” “0.23” “0.23”)

export O2\_t=${O2\_T\_LIST[$SLURM\_ARRAY\_TASK\_ID]}

NO\_T\_LIST =(“1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0” “1.0”)

export NO\_t =${NO\_T\_LIST[$SLURM\_ARRAY\_TASK\_ID]}

export NUM\_PROCS=$SLURM\_CPUS\_PER\_TASK

export NUM\_ARRAY=$SLURM\_ARRAY\_TASK\_ID

java -jar MIMICS.jar

Run this command in the Slurm workload manager to execute the slurm job script:

sbatch --array=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14 A\_MIMICS\_PS

The ‘--array’ is used to define the job index array. The job index number will be used as the SLURM\_ARRAY\_TASK\_ID and used as the index to select parameter value from the O2\_t and NO\_t list. The job name is A\_MIMICS\_PS. In this example slurm command, 15 simulations were performed. For example, the fifth job will name the output file names with the last number as 5, and the o2\_t value will be 0.2 mM, the fifth number in the O2\_T\_LIST, and the no\_t value will be 1.0 M, the fifth number in the NO\_T\_LIST.

**User inputs and customization of MiMICS code**

The MiMICS source code allows for users to extend the framework to simulate metabolic processes in multi-cell communities of interest. We recommend users read the HAL manual to understand the HAL’s extendable agent and metabolic grid functions available to users. This section outlines where users need to input names of input files, and make corresponding adjustments to the MiMICS source code. All areas in the Python and Java code files where the user is required to input file names or suggested to alter MiMICS source code are marked above by ‘USER INPUT:’.

**Input the simulation job number**

In the main function in *‘agents\_FBA.py’*, the user must input the simulation job number used to name MiMICS output files. This simulation job number is helpful when running multiple simulation replicates or parameter sweeps. When running MiMICS on a personal computer, define the simulation number in directly in *job\_num* variable. In the case where MiMICS is run on a high-performance computing (HPC) system, the *job\_num* variable is defined and imported from the MiMICS HPC slurm script job file. In the source code shown below, the simulation job number was set to 0.

## USER INPUT: SET OR IMPORT THE SIMULATION JOB NUMBER  
job\_num = 0 # DEFINE HERE WHEN RUNNING MIMICS ON PERSONAL COMPUTER  
#job\_num=int(os.getenv('NUM\_ARRAY')) # COMMENT OUT WHEN RUNNING ON HPC SYSTEM. JOB NUMBER VALUE DEFINED IN MIMICS HPC SLURM JOB FILE.

**Input metabolic model files**

In the main function in *‘agents\_FBA.py’*, the user must input the names of the metabolic model files intended to be used in MiMICS. The metabolic model files should be an XML format and are imported using COBRApy. The *‘models’* list variable contains the metabolic models the cellular agents can use to simulate metabolism. In simulations where agents can use different metabolic model states, in the *‘models’* list variable, list the metabolic model variable names in the order corresponding to the agent’s metabolic state number (example of metabolic state index assignment shown in Table 1).

Table 1. Indexing mapping of metabolic model states.

|  |  |
| --- | --- |
| **Metabolic state #** | **Metabolic Model State** |
| 0 | Aerobic |
| 1 | Denitrification |
| 2 | Denitrification + NO |
| 3 | Oxidative stress |

In the case where only metabolic model is imported and available to the agents, only list the one metabolic model variable name in the *‘models’* list variable.

## USER INPUT: IMPORT METABOLIC MODELS  
model0 = cobra.io.read\_sbml\_model("insert metabolic model filename")  
model1 = cobra.io.read\_sbml\_model("insert metabolic model filename")  
models = [model0, model1] ## SET LIST OF METABOLIC MODELS THAT CAN BE USED BY THE AGENTS

To generate multiple -omics-guided metabolic states of a GENRE, we recommend using published algorithms, such as RIPTiDe2 or GiMME3.

Example code with two different metabolic model inputs:

## USER INPUT: IMPORT METABOLIC MODELS  
model0 = cobra.io.read\_sbml\_model("Aerobic\_state.xml")  
model1 = cobra.io.read\_sbml\_model("Denitrification\_state.xml")  
models = [model0, model1] ## SET LIST OF METABOLIC MODELS THAT CAN BE USED BY THE AGENTS

In the example above, the *‘models’* list contains *‘model0’* at index 0 and *‘model1’* at index 1. Correspondingly, agents assigned a value of 0 for their metabolic state # will be assigned the metabolic model *‘model0’* . Agents assigned a value of 1 for their metabolic state # will be assigned the metabolic model *‘model1’* .

**Input nutrient media concentrations**

In the main function in *‘agents\_FBA.py’*, the user must input the name of the nutrient media concentrations (in an XML file type) intended to be used to simulate the aqueous bulk nutrient media concentrations in MiMICS. These nutrient media concentrations can be used to constrain the exchange bounds of each agent’s metabolic model.

## USER INPUT: IMPORT NUTRIENT MEDIA  
media = pd.read\_excel('insert nutrient media file name”, engine='openpyxl')

The first row the nutrient media XML file should contain the column titles of ‘Name’, ‘Metabolite ID’, and ‘mM in patch (mmol/L)’. In the ‘Name’ column list the general name of the metabolite (e.g. oxygen, glucose, etc.). In the ‘Metabolite ID’ column list the metabolite ID number used in the metabolic model. In the ‘mM in patch (mmol/L)’ column, list the metabolite concertation in units of mM.

An example of the nutrient media XML file format is shown below:

|  |  |  |
| --- | --- | --- |
| Name | Metabolite ID | mM in patch (mmol/L) |
| Water | cpd00001\_e | 1000 |
| Glucose | cpd00027\_e | 3.2 |

**Input list of intracellular reactions to track in MiMICS**

At each time step, MiMICS outputs the flux (in the *rxns\_flux.csv* output file) of intracellular reactions for each agent. In the main function in *‘agents\_FBA.py’*, the user inputs the IDs of the desired reactions to track in MiMICS define in the *‘rxns’* variable.

## USER INPUT: IMPORT REACTION NAMES TO SAVE REACTION FLUXES FOR EACH AGENT  
rxns = pd.read\_csv('insert filename of reaction IDs')

The reaction IDs should correspond to reaction IDs in the metabolic model. The list of reaction IDs should be CSV file with the reaction IDS name listed in one column (example shown below).

|  |
| --- |
| RXN ID 1 |
| RXN ID 2 |
| RXN ID 3 |

We recommend the user input less than 1000 reactions and only the reactions with expected non-zero flux to avoid generation of large datasets.

**Set total simulation time and time steps**

In the main function in *‘agents\_FBA.py’*, the user can input the total simulation time in units of hours defined in the *total\_sim\_time variable*. In addition,the user can input the simulation time step in units of minutes defined in the *time\_step variable*. These inputs calculate the number of MiMICS simulation time steps defined by the *num\_dt* variable.

## USER INPUT: SET NUMBER OF SIMULATION TIME STEPS  
total\_sim\_time = 0.5 # TOTAL SIMULATION TIME, HOUR  
time\_step = 5 # SIMULATION TIME STEP, MINUTES  
num\_dt = total\_sim\_time \* 60/time\_step; # NUMBER OF SIMULATION TIME STEPS

**Initialize the number of cells in agent-based model**

When running MiMICS on a personal computer, in the main function in *‘agents\_FBA.py’*, the user can input the initial number (integer) of cellular agents defined in the *num\_cell* variable. In the case where MiMICS is run on a high-performance computing (HPC) system, the *num\_cell* variable is defined and imported from the MiMICS HPC slurm script job file. In the source code shown below, two agents were initialized.

## USER\_INPUT: SET OR IMPORT THE INITIAL NUMBER OF CELLS  
num\_cell = 2 # DEFINE HERE WHEN RUNNING MIMICS ON PERSONAL COMPUTER   
#num\_cell = int(os.getenv('NUM\_CELLS\_LIST')) # COMMENT OUT WHEN RUNNING ON HPC SYSTEM. NUM CELLS VALUE DEFINED IN MIMICS HPC SLURM JOB FILE.

**Define dimensions of agent-based model and metabolite grids**

The user should input the desired dimensions used to define the agent-based model and metabolite grids. These dimensions are defined in the MAIN METHOD in *‘MIMICS\_gateway.java’.* The dimensions correspond to the number of grids or patches that define the agent-based model and metabolite 2D or 3D grids.

The source code shown below defines the 3D grid dimensions. The x and y dimensions were set as 50 , corresponding to 50 patches of 1 length in the x and y directions. The z dimension were set as 10 , corresponding to 10 patches of 1 length in the z direction.

// USER INPUT: DEFINE 3D WORLD DIMENSIONS  
int xdim = (int) 50; // X-DIMENSION, UNITS: MICROMETERS  
int ydim = (int) 50; // Y-DIMENSION, UNITS: MICROMETERS  
int zdim = (int) 10; // Z-DIMENSION, UNITS: MICROMETERS

**Define and initialize metabolite reaction-diffusion PDE grids**

In the MiMICS class in *‘MIMICS.java’,* metabolite 3D grids were initialized in the using HAL’s built in PDE3D class. Here, the user must define each metabolite grid desired to be simulated in MiMCIS. The source code below shows metabolite 3D grids generated for oxygen and glucose.

// USER INPUT: INITIALIZE PDE GRIDS FOR METABOLITES IN BIOFILM  
PDEGrid3D oxygen = new PDEGrid3D(xDim, yDim,zDim);  
PDEGrid3D glucose = new PDEGrid3D(xDim, yDim,zDim);

Initial and aqueous metabolite concentrations for the metabolite 3D grids are defined the main function in the *‘agents\_FBA.py’* file, shown below:

## USER INPUT: DEFINE INITIAL METABOLITE CONCENTRATIONS  
initial\_oxygen = 0.25; # INITIAL OXYGEN CONCENTRATION; UNITS: mM  
initial\_glucose = 3.2; # INITIAL GLUCOSE CONCENTRATION; UNITS: mM

Metabolite diffusion coefficients, scaled according to HAL’s methods, are defined the main function in the *‘agents\_FBA.py’* file, shown below:

## USER INPUT: DEFINE METABOLITE DIFFUSION COEFFICIENTS  
D\_O2 = 0.5; # SCALED OXYGEN DIFFUSION COEFFICIENT  
D\_G = 0.72; # SCALED GLUCOSE DIFFUSION COEFFICIENT

Metabolite diffusion is simulated using HAL’s alternating direction implicit ADI) method PDE solver. Diffusion of gaseous metabolites was solved with the method Gas\_Diffuse() in *‘MiMICS.java’*. Diffusion of carbon metabolites was solved with the method Carbon\_Diffuse() in *‘MiMICS.java’*. Both methods can be adjusted by the user as desired.

Adding new metabolites will require the user to update corresponding variables and inputs in the Python functions and Java methods that simulate metabolite diffusion or pass information about metabolite concentrations. These Python functions include: run\_GENRE() gateway.entry\_point.Diffuse\_Metabolites(), and gateway.entry\_point.run\_model0(), and run\_GENRE(). In addition the python array variables, *patch\_value\_from\_java* and *patch\_value\_for\_java,* which hold metabolite concentrations for each agent passed from and to java, respectively, should be updated accordingly.

These Java methods that should be updated to simulate new metabolite grids used in the MIMICS Gateway and MIMICS class include Carbon\_Diffuse(), Gas\_Diffuse(), Diffuse\_Metabolites(), and run\_model0(), getPatchFromHal\_All(), setPatchFromPython(), Save\_Cell\_Info(), and Save\_Met\_Info().

**Define agent parameter values**

In addition to the agent attributes assigned in HAL, agents were assigned attributes including agent biomass, growth rate, directional angel, index, metabolic state, time to switch to a new metabolic state, and sum of mechanical forces. In this way, each agent can contain its unique value of each attribute. The user can define the values of x-coordinate, y-coordinate, z-coordinate, biomass, directional angle, parameters, metabolic state assignment number of initial agents in the Initialize\_Random() method in *‘MIMICS.java’*. In the Biomass\_Divide() agent class method in *‘MIMICS.java’*, the user can input the maximum biomass for an agent defined by the *max\_biomass* variable.

// FUNCTION TO RANDOMLY INITIALIZE AGENTS   
public void Initialize\_Random(int b0) {  
 max\_cell\_index = b0; // SET MAXIMUM CELL INDEX  
 for (int j = 0; j <= max\_cell\_index-1; j++) { // j DEFINES THE AGENT INDEX  
 double x = rn.Double(xDim); // X-COORDINATE  
 double y = rn.Double(yDim); // Y-COORDINATE  
 Random r = new Random();  
 double old\_mass = 1e-12 + (2e-12 - 1e-12) \* r.nextDouble(); // SET AGENT BIOMASS  
 int angle = rn.Int(360); // INITIALIZE CELL ANGLE  
 int metabolic\_state = 0;  
 NewAgentPT(x, y, 0).Init(old\_mass, angle, metabolic\_state, j, 0, 0);  
 }  
}

For cases where agent’s can simulate different metabolic model states, a time delay was added for agents to switch between states to represent the time delay for new mRNA synthesis upon detecting new environmental cues. This time delay was captured by the agent class attributes *‘t\_switch\_#*’ where the # corresponds to the metabolic model state assignment number. We recommend each metabolic model state has a defined and initialized *‘t\_switch\_#*’ agent class attribute. All methods in the MiMICS java code where *‘t\_switch\_#*’ attributes are used will need to be updated accordingly.

The Biomass\_Divide() agent class method defines processes related to cell division which can be adjusted by the user as desired. To perform HAL’s agent cell shoving mechanic behaviors, the necessary values for the agent’s radius, force scaler, and friction are defined in the MAIN METHOD of *‘MiMICS.java’*

**Define agent behaviors**

Agent behaviors can be defined using HAL’s built in agent class functions or by the user defining an agent class function. For each agent that is alive, these agent class functions are performed in an order defined the StepCells() method in *‘MiMICS.java’.* The source code contains a recommended order of these functions which can be adjusted as desired.

In cases where multiple metabolic model states are used, rules for agents to be assigned a metabolic state number (corresponding to a metabolic model state) can be defined by the user in the Change\_metabolic\_state() method in *‘MiMICS.java’.* Example code of these rules is demonstrated in the code for simulations of *P. aeruginosa* biofilms.

Once all inputs and desired code modifications are accounted for, run the simulation following the sections: **Running MiMICS on a High Performance Computer** and **Running MiMICS on a personal computer**

**Description of output files**

1. *Metabolite concentrations.* The metabolite concentrations at each x,y,z coordiante in the metabolite grid are output in ‘met\_grid#.csv’, where *#* defines the simulation job number. The row corresponds to the x,y,z position in the metabolite grid. Columns correspond to metabolite concentrations. Define the directory for this file, represented as the variable *file\_name* in the code below, in *MiMICS\_gateway.java* class method Save\_met\_info() method in *‘MiMICS.java’.*
2. // FUNCTION TO SAVE MODEL OUTPUTS (EXECUTED FROM PYTHON)  
   public static void Save\_met\_info(int time, int rep) throws IOException {  
    String file\_name = "met\_grid"+ Integer.*toString*((rep))+".csv";  
    *bis*.Save\_Met\_Info(time,rep,file\_name); // SAVE METABOLITE PATCH INFORMATION  
   }
3. *Agent values.* The attribute values for each agent are output in a file named ‘agent\_properties*#*.csv’, where *#* defines the simulation job number. The method Save\_cell\_info() in *‘MiMICS.java’* saves this output file. In the CSV file, rows correspond to each agent and columns correspond a unique agent’s property. The agent property CSV file contains each agent’s x,y,z coordinates, index, biomass, growth rate, metabolic state, and metabolite concentrations at the agent’s location. In addition, the simulation time step and simulation job number are included. Other desired values of agent attributes be added by the user in the Save\_cell\_info() method. Define the directory for this file, represented as the variable *file\_name* in the code below, in *MiMICS\_gateway.java* class method Save\_cell\_info() method.

public static void Save\_cell\_info(int time, int rep) throws IOException {  
 String file\_name ="agent\_properties"+ Integer.*toString*((rep))+".csv";  
 *bis*.Save\_Cell\_Info(time,rep,*bis*.Pop(),file\_name); // SAVE AGENT INFORMATION  
}

1. *Agent intracellular reaction fluxes.* Each agent’s metabolic model predicted intracellular and exchange reaction flux values are saved as a CSV file named ‘rxns\_fluxs*#*.csv’, where *#* contains an integer of the simulation job number. Each row corresponds to each agent and the columns contains each agent index and reaction flux value. Reaction IDs are listed in the first row. The simulation time point is also listed. This file is saved directly from python. Define the directory for this file in the main method method in *‘agents\_FBA.py,* as described above.

**REFERENCES**

1. Bravo, R. R. *et al.* Hybrid Automata Library: A flexible platform for hybrid modeling with real-time visualization. *PLoS Comput Biol* **16**, e1007635 (2020).

2. Jenior, M. L., Moutinho, T. J., Dougherty, B. V. & Papin, J. A. Transcriptome-guided parsimonious flux analysis improves predictions with metabolic networks in complex environments. *PLoS Comput Biol* **16**, e1007099 (2020).

3. Becker, S. A. & Palsson, B. O. Context-Specific Metabolic Networks Are Consistent with Experiments. *PLoS Comput Biol* **4**, e1000082 (2008).