iDynoMiCS: individual-based Dynamics of Microbial Communities Simulator

Software Setup and Tutorial

Laurent Lardon, Brian Merkey, Sónia Martins, Chinmay Kanchi, Edd Miles, Robert Clegg, Jan-Ulrich Kreft

June 2009: Initial Version
November 2009: Revised Version
May 2010: Revised Version
February 2011: Revised Version
May 2011: Updated to reflect the official release 1.0 version of the code
October 2012: Updated for release 1.1

www.idynomics.org

I Introduction

The iDynoMiCS software simulates the growth of microbial communities. iDynoMiCS is written in Java, and uses XML protocol files that allow one to specify easily many different types of simulations. iDynoMiCS writes plain-text XML files as output, and these may be processed using any number of software tools (though we provide some general post-processing routines that run in Matlab and R). In addition to XML files, iDynoMiCS also writes files for POV-Ray, which is used to render 3-D images of the simulation. A 'default' installation of iDynoMiCS will require the following software:

- A Java Virtual Machine: for running simulations
- Matlab: for post-processing of simulations results
- POV-Ray (and the QuietPOV extension, if desired): for visualizing agent locations
- Eclipse (Java development environment): for viewing or modifying the iDynoMiCS source code
- ImageMagick: for modifying output images and movies (used by some Matlab scripts)

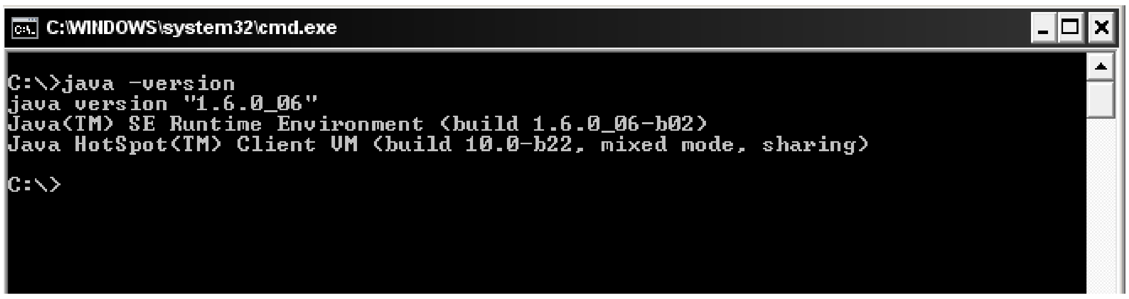
In practice, only the first is truly required to run iDynoMiCS, though interpretation of simulation results requires the Matlab/POV-Ray combination or some other post-processing tools. Those interested in the particular details of modeling, or who wish to extend iDynoMiCS to model new problems, will benefit from installing Eclipse.

This tutorial describes the installation of the required software and how to run simulations. We will describe how to specify simulations using the XML protocol files, and will provide a brief introduction into post-processing simulation results using Matlab. This tutorial is for PCs running Microsoft Windows. However, like any Java program, iDynoMiCS also runs on other operating systems including MacOS X and Linux. There are also Matlab, POV-Ray and Eclipse versions for those operating systems.

II Required Installations

II.1 Java Virtual Machine

To determine whether a Java Virtual Machine is installed in your computer, open a Command prompt window (From the Start menu choose 'Run' and type '`cmd`') and type '`java -version`'. If a JAVA run-time engine is installed you should see a screen like this:



```
C:\>java -version
java version "1.6.0_06"
Java(TM) SE Runtime Environment (build 1.6.0_06-b02)
Java HotSpot(TM) Client VM (build 10.0-b22, mixed mode, sharing)

C:\>
```

If you do not have a JAVA runtime engine or if the version is earlier than 1.5, install the newest JVM from www.java.com.

Note that the Eclipse installation already includes a Java compiler and other utilities that are part of the Java Development Kit, so you do not need to install the Java Development Kit (there is no harm doing that though) – only the Java runtime.

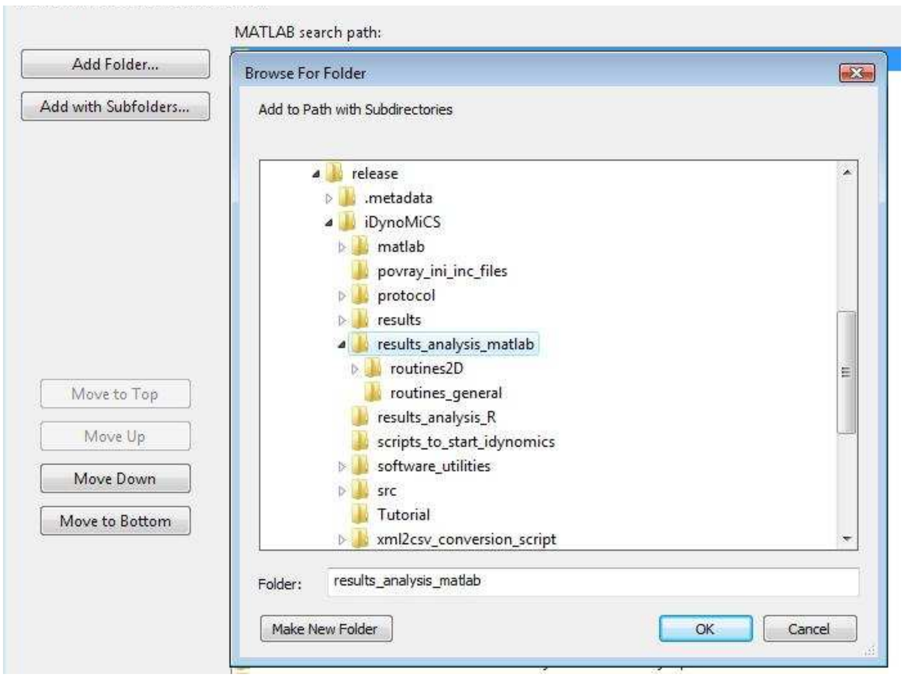
II.2 iDynoMiCS Software

The iDynoMiCS software requires no additional installation aside from putting the software folder in an appropriate location. Because simulation outputs will take up a nontrivial amount of hard drive space, it is best to put the iDynoMiCS software on a larger drive if possible. If at a later point in time additional disk space is needed, you can always move previous simulation results elsewhere to clear space.

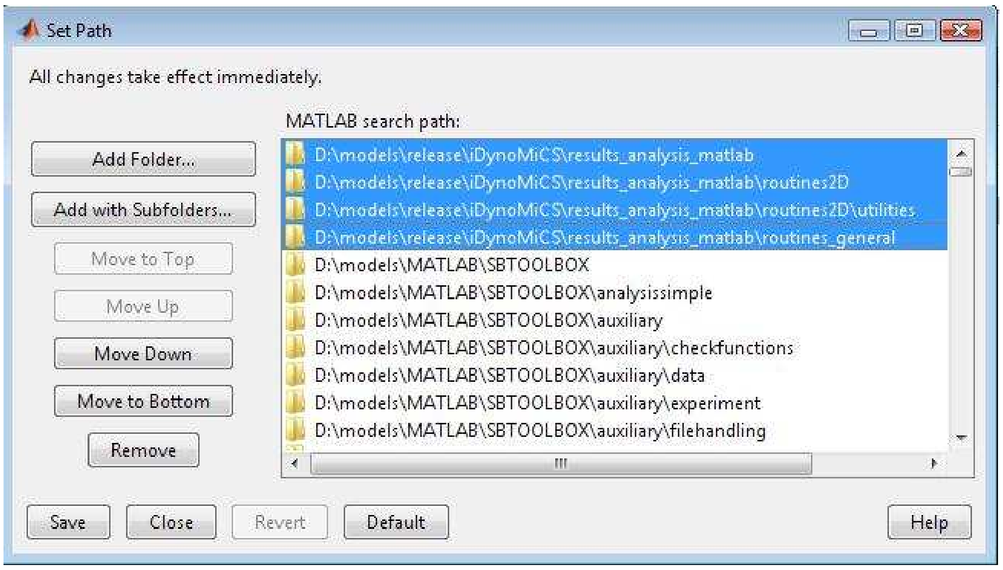
II.3 Matlab

The Matlab software requires a software license, though many universities have a site-license allowing use on your own computer; check with your IT department regarding Matlab availability. If you do not have access to Matlab, you may consider the open-source software Octave (http://www.gnu.org/software/octave/) as an alternative.

In order to use the provided Matlab routines, you need to add the iDynoMiCS Matlab routines to your Matlab path. To do this, open Matlab and choose 'File -> Set Path'. Press the 'Add with Subfolders' button, and choose the 'iDynoMiCS\results_analysis_matlab' directory.



Verify that the folders are visible (as shown above), then click 'Save' and 'Close'. Test that the path is correct by typing '`help showRunInfo`' at the Matlab command line; there will be no error if the path was set correctly.



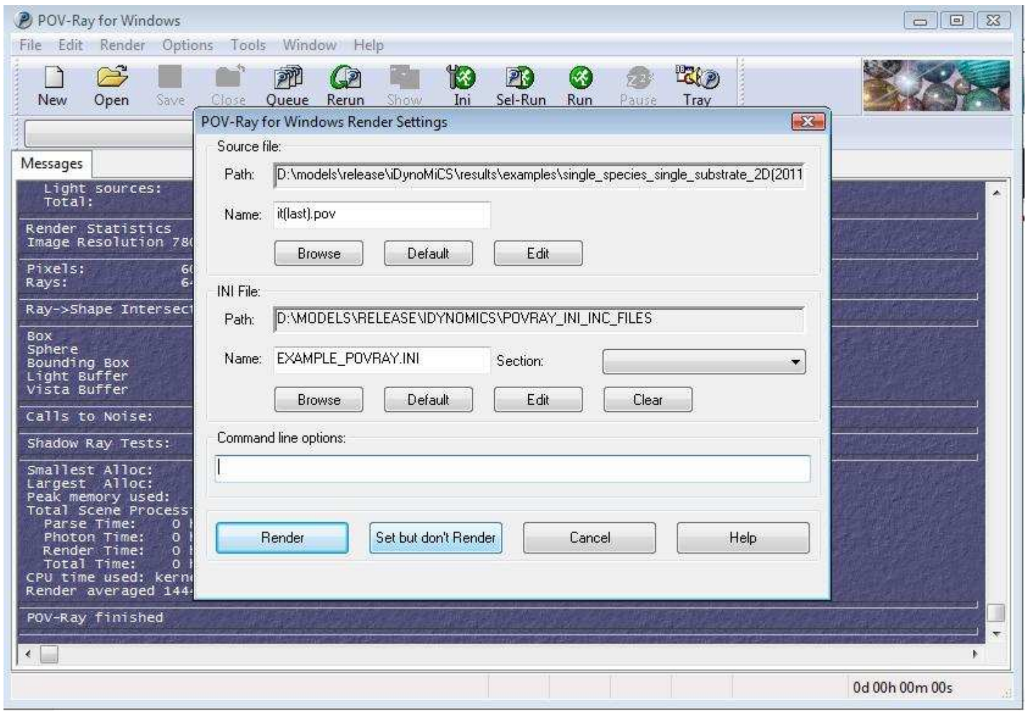
All changes take effect immediately.

MATLAB search path:

D:\models\release\iDynoMiCS\results_analysis_matlab
D:\models\release\iDynoMiCS\results_analysis_matlab\routines2D
D:\models\release\iDynoMiCS\results_analysis_matlab\routines2D\utilities
D:\models\release\iDynoMiCS\results_analysis_matlab\routines_general
D:\models\MATLAB\SBTOOLBOX
D:\models\MATLAB\SBTOOLBOX\analysissimple
D:\models\MATLAB\SBTOOLBOX\auxiliary
D:\models\MATLAB\SBTOOLBOX\auxiliary\checkfunctions
D:\models\MATLAB\SBTOOLBOX\auxiliary\data
D:\models\MATLAB\SBTOOLBOX\auxiliary\experiment
D:\models\MATLAB\SBTOOLBOX\auxiliary\filehandling

II.4 POV-Ray and QuietPOV

Install POV-Ray using the **povwin36.exe** file. When choosing the installation directory, it will make things easier later if you pick a path easy to write (*e.g.* `C:\idynomics\povray`). You can define the rendering settings using .ini files. One example is in the "povray_ini_inc_files" folder, example_povray.ini, which renders images in .png format and sets the width/height of the images to 780 pixels. You can load this file from PovRay, check under the 'Render' menu the option 'Edit settings/render' and browse to the directory containing the example_povray.ini file. Then choose 'Set but don't Render' and PovRay will now use the settings contained in the ini file to render the images.


POV-Ray for Windows
File Edit Render Options Tools Window Help
New Open Save Close Queue Rerun Show Ini Sel-Run Run Pause Tray
Messages
POV-Ray for Windows Render Settings
Browse
Default
Edit
Clear
Render
Set but don't Render
Cancel
Help

```
Light sources:
  Total:
Render Statistics
Image Resolution 78
Pixels:          6
Rays:            6
Ray->Shape Intersect
Box
Sphere
Bounding Box
Light Buffer
Vista Buffer
Calls to Noise:
Shadow Ray Tests:
Smallest Alloc:
Largest  Alloc:
Peak memory used:
Total Scene Process
  Parse Time:    0
  Photon Time:   0
  Render Time:   0
  Total Time:    0
CPU time used: kern
Render averaged 144
POV-Ray finished
```

Source file:
Path: D:\models\release\iDynoMiCS\results\examples\single_species_single_substrate_2D(2011
Name: it(last).pov

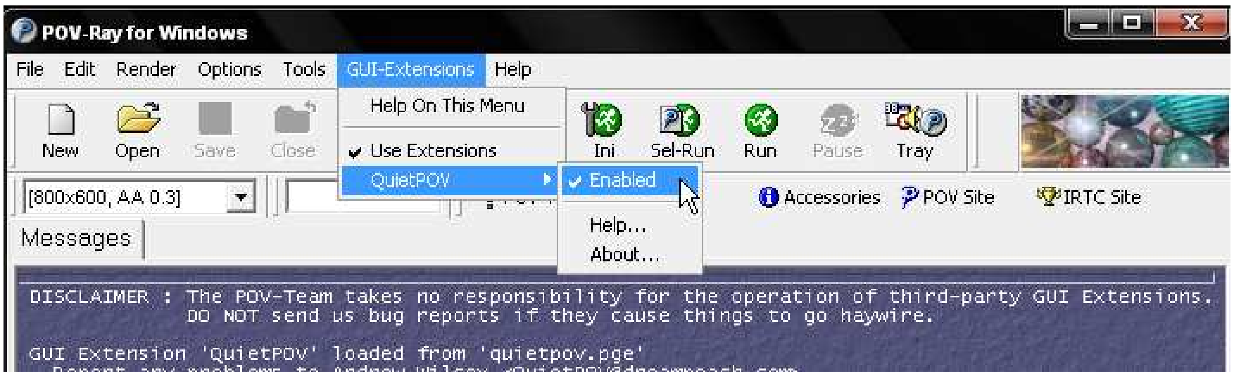
INI File:
Path: D:\MODELS\RELEASE\IDYNOMICS\POVRAY_INI_INC_FILES
Name: EXAMPLE_POVRAY.INI
Section:

Command line options:

0d 00h 00m 00s

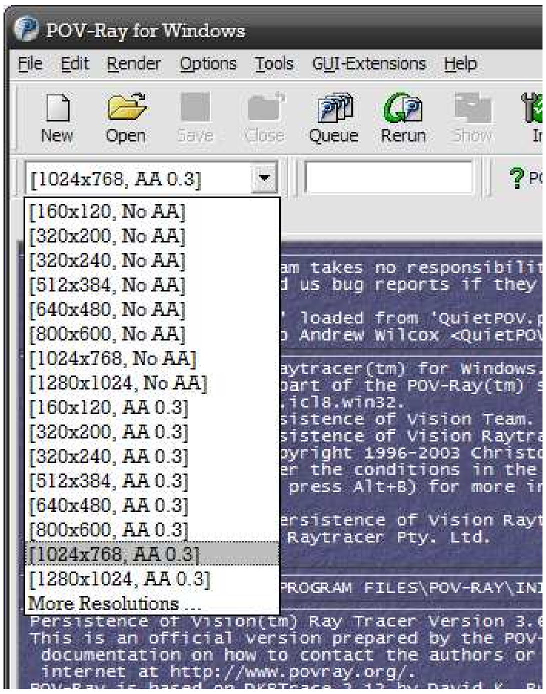
NOTE: Some people have reported trouble with using POV-Ray version 3.6 and QuietPOV on Windows Vista. If you are running Vista, it seems you must use POV-Ray version 3.5. Install the software in a folder *other than* Program Files, and before trying to run the software, browse to the installed directory and into the 'bin' directory (e.g. `C:\idynomics\POV-Ray for Windows v3.5\bin`). Then right-click on the file **pvengine.exe**, look in the menu 'Compatibility' and select 'Make compatible for Win89/ME', and allow all users full access. This procedure has solved the problem for some users.

QuietPOV is a GUI Extension for POV-Ray for Windows that allows the user to start and stop renders in POV-Ray from the command-line. As the name suggests, it stops POV-Ray from popping up the POV-Ray window in the foreground upon starting a new render. It is not necessary but makes using POV-Ray more tolerable. To install QuietPOV, run the **QuietPOV-install.exe** file. Then start POV-Ray from the Start menu, and check under the 'GUI-Extensions' menu that QuietPOV is enabled (as shown below). Finally, choose a default resolution to render your images (also shown below) and close POV-Ray. You can still perform your simulations without QuietPOV, recent releases of POV-Ray seem to be missing this GUI extension.



[800x600, AA 0.3]

```
DISCLAIMER : The POV-Team takes no responsibility for the operation of third-party GUI Extensions.
             DO NOT send us bug reports if they cause things to go haywire.

GUI Extension 'QuietPOV' loaded from 'quietpov.pge'
```



[1024x768, AA 0.3]

[160x120, No AA]
[320x200, No AA]
[320x240, No AA]
[512x384, No AA]
[640x480, No AA]
[800x600, No AA]
[1024x768, No AA]
[1280x1024, No AA]
[160x120, AA 0.3]
[320x200, AA 0.3]
[320x240, AA 0.3]
[512x384, AA 0.3]
[640x480, AA 0.3]
[800x600, AA 0.3]
[1024x768, AA 0.3]
[1280x1024, AA 0.3]
More Resolutions ...

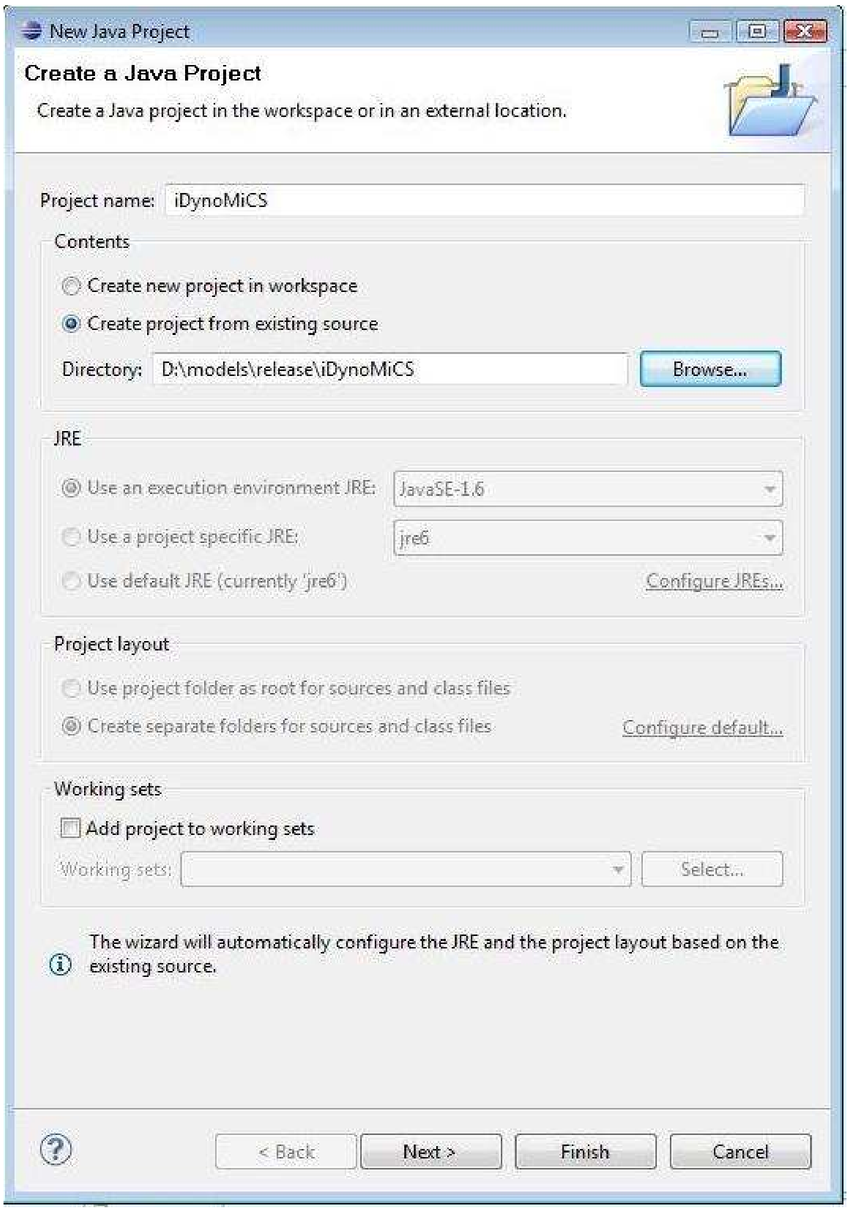
```
am takes no responsibilit
d us bug reports if they

' loaded from 'QuietPOV.p
o Andrew Wilcox <QuietPOV

aytracer(tm) for Windows.
part of the POV-Ray(tm) s
.icl8.win32.
sistence of Vision Team.
sistence of Vision Raytra
pyright 1996-2003 Christo
er the conditions in the
 press Alt+B) for more in

ersistence of Vision Rayt
 Raytracer Pty. Ltd.

PROGRAM FILES\POV-RAY\INI

Persistence of Vision(tm) Ray Tracer Version 3.6
This is an official version prepared by the POV-
 documentation on how to contact the authors or
 internet at http://www.povray.org/.
```

II.5 Eclipse

The Eclipse software is really only necessary if you are interested in exploring or further developing the iDynoMiCS source code. Download the newest stable release from http://www.eclipse.org/ and install Eclipse into the directory of your choice (the local directory holding the iDynoMiCS software is fine). At the root of the eclipse directory, there is an executable file called **eclipse.exe** that you will use to launch Eclipse. To create a desktop shortcut for this executable, right-click on the .exe file and choose ‘Send to -> Desktop’. You should also add a shortcut to your Start menu folder for convenience.

When you first start Eclipse, the software will ask you to define your workspace; it is best to choose something simple like ‘`C:\java`’. After choosing the directory, close the introduction screen.

Now we need to create a new Java project based on the iDynoMiCS source code. To do this, click on ‘File -> New -> Java Project’; this will open a window where you will define the new project. Write ‘iDynoMiCS’, select the ‘Create project from existing source’ option, and browse to the ‘iDynoMiCS’ directory provided. (See the image below for an example.) Finally, click the ‘Finish’ button. Now in the left panel there should be a project with the ‘src’ and other directories contained inside.


New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: iDynoMiCS

Contents

Create new project in workspace [✓ Create project from existing source]

Directory: D:\models\release\iDynoMiCS — Browse...

JRE

[✓ Use an execution environment JRE:] JavaSE-1.6

Use a project specific JRE: jre6

Use default JRE (currently 'jre6') — Configure JREs...

Project layout

Use project folder as root for sources and class files [✓ Create separate folders for sources and class files] — Configure default...

Working sets

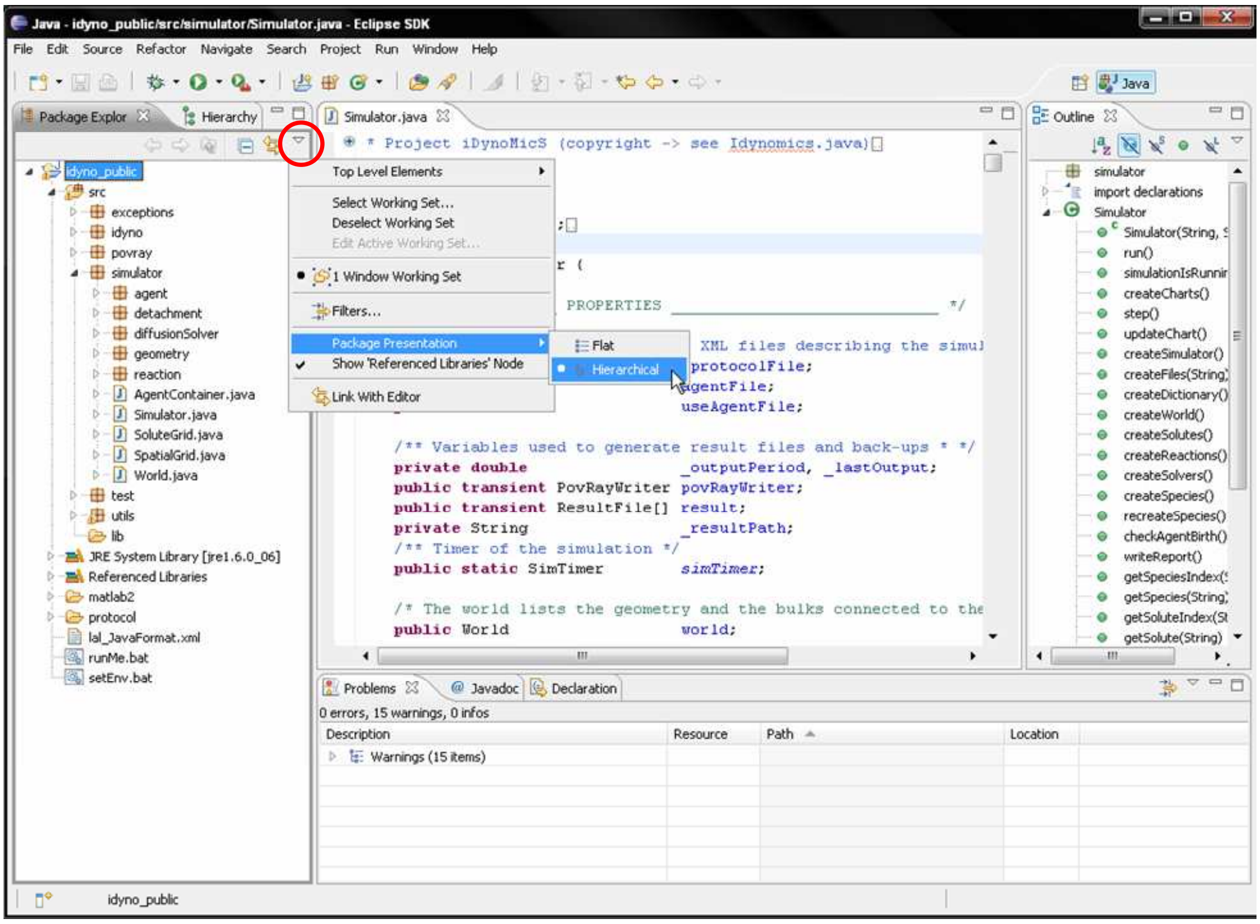
☐ Add project to working sets

Working sets: — Select...

The wizard will automatically configure the JRE and the project layout based on the existing source.

< Back | Next > | Finish | Cancel

There is one additional setting to change in order to make use of Eclipse friendlier. From the 'Package Explorer' perspective on the left, click on the small downward arrow (called the *View Menu*, shown in a red circle on the next figure) and choose 'Package Presentation -> Hierarchical'. Then on the left side of the screen you have a browser to navigate easily through the packages, and on the right side you can find an outline which summarizes all the methods of the current class.



```
* Project iDynoMicS (copyright -> see Idynomics.java)

PROPERTIES ________________________ */
XML files describing the simul
protocolFile;
agentFile;
useAgentFile;

/** Variables used to generate result files and back-ups * */
private double                     _outputPeriod, _lastOutput;
public transient PovRayWriter povRayWriter;
public transient ResultFile[] result;
private String                     _resultPath;
/** Timer of the simulation */
public static SimTimer             simTimer;

/* The world lists the geometry and the bulks connected to the
public World                       world;
```

0 errors, 15 warnings, 0 infos

Description	Resource	Path	Location
Warnings (15 items)			

II.6 *Installing ImageMagick*

ImageMagick is a set of tools for manipulation of images, and is used by several Matlab routines in order to render simulation output and create animations of the biofilm state during a simulation. You may download an installer for the ImageMagick suite from the http://www.imagemagick.org website. Look for the "Binary Releases" link on the left, and you can just choose their suggested download for your particular platform. It is okay to install to the default location.

III Running a Simulation in iDynoMiCS

The iDynoMiCS package provided has all files located in the directory 'iDynoMiCS' and its subdirectories. For those just exploring the use iDynoMiCS, this structure is perfectly adequate; however, we recommend that users who run many iDynoMiCS simulations store their protocol files in an entirely separate directory as simulations will produce GB of data so it is better to separate the code from the simulation output. For example, if you store data on drive "D:", you could make a directory "D:\eclipse\iDynoMiCS" to keep the source code (within the Eclipse workspace if you are editing the source code with eclipse), and another directory "D:\runs" to put your protocol files so all simulation output will end up in subdirectories of "D:\runs".

Multiple simulations and random numbers

At the end of each simulation the state of the random number generator (described in more detail in the Supporting Information of Lardon et al. 2011) is output as a "random.state" file in the same directory as the protocol file. Whenever a simulation starts, iDynoMiCS will look first for a random.state file in the same directory as the protocol file used to start iDynoMiCS, and will use it as the random seed if it is present; this is to allow multiple replicates of a simulation (with identical settings in the protocol file) to run with non-overlapping random numbers, being drawn from consecutive regions of the cyclic random number stream. Care should be taken that random.state files are not read in when the user does not desire them to be read in: a simple step is to store protocols in separate subdirectories.

There are two (main) ways to run iDynoMiCS simulations, depending on whether you want to use the Eclipse environment or not. We will show how to run iDynoMiCS using both methods.

III.1 Using the RunIdyno.py File

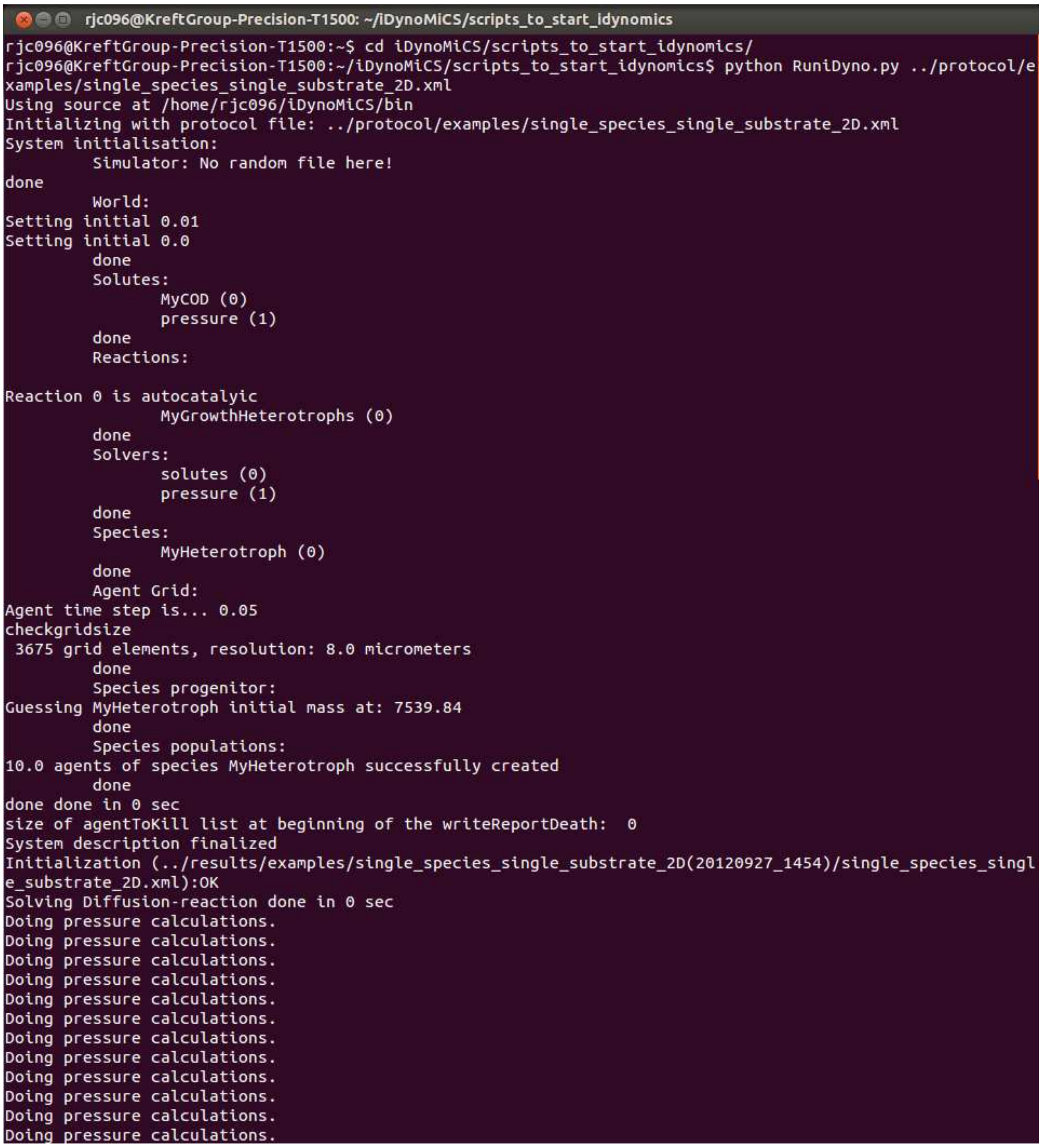
In the provided 'iDynoMiCS' directory (also known as a folder) you can find scripts to run a simulation in iDynoMiCS in the directory 'scripts_to_start_idynomics'. The file called **RunIdyno.py** is a python script that will launch a simulation from command line. It is the recommended method of launching iDynoMiCS because it is Operating System independent but requires python to be installed, two alternate scripts (*a ".bat"* batch script for Windows and a shell script "*.sh*" for Linux or Mac) are provided. These are less flexible and general, and the paths to the iDynoMiCS classes have to be edited to match the paths to your installation.
Open a command prompt window directly (from the Start menu choose 'Run' and type '`cmd`'), and then navigate using '`cd`' to the directory containing the **RunIdyno.py** file, as shown below.

Then call the RunIdyno.py file followed by the path where the protocol is located. Example protocol files are provided in the 'iDynoMiCS\protocol\examples' directory.

You can supply several protocol files on the command line if you want to execute several simulations in the sequence given.

The simulation should start, printing output into the command window until the simulation finishes. The 'single_species_single_substrate_2D.xml' protocol simulates 3 days of growth and with the default timestep runs for 72 iterations (one per hour), which should take around 10 minutes or less to compute on a modern computer. The simulation may be stopped at any time by closing the command line window or pressing Ctrl+C twice.

Simulation results for protocols located in the 'iDynoMiCS\protocol' directory are by default stored in 'iDynoMiCS\results', and this directory will recreate the directory structure of the 'protocol' directory.


rjc096@KreftGroup-Precision-T1500: ~/iDynoMiCS/scripts_to_start_idynomics

```
rjc096@KreftGroup-Precision-T1500:~$ cd iDynoMiCS/scripts_to_start_idynomics/
rjc096@KreftGroup-Precision-T1500:~/iDynoMiCS/scripts_to_start_idynomics$ python RuniDyno.py ../protocol/e
xamples/single_species_single_substrate_2D.xml
Using source at /home/rjc096/iDynoMiCS/bin
Initializing with protocol file: ../protocol/examples/single_species_single_substrate_2D.xml
System initialisation:
         Simulator: No random file here!
done
         World:
Setting initial 0.01
Setting initial 0.0
         done
         Solutes:
                MyCOD (0)
                pressure (1)
         done
         Reactions:

Reaction 0 is autocatalyic
                MyGrowthHeterotrophs (0)
         done
         Solvers:
                solutes (0)
                pressure (1)
         done
         Species:
                MyHeterotroph (0)
         done
         Agent Grid:
Agent time step is... 0.05
checkgridsize
 3675 grid elements, resolution: 8.0 micrometers
         done
         Species progenitor:
Guessing MyHeterotroph initial mass at: 7539.84
         done
         Species populations:
10.0 agents of species MyHeterotroph successfully created
         done
done done in 0 sec
size of agentToKill list at beginning of the writeReportDeath:  0
System description finalized
Initialization (../results/examples/single_species_single_substrate_2D(20120927_1454)/single_species_singl
e_substrate_2D.xml):OK
Solving Diffusion-reaction done in 0 sec
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
Doing pressure calculations.
```

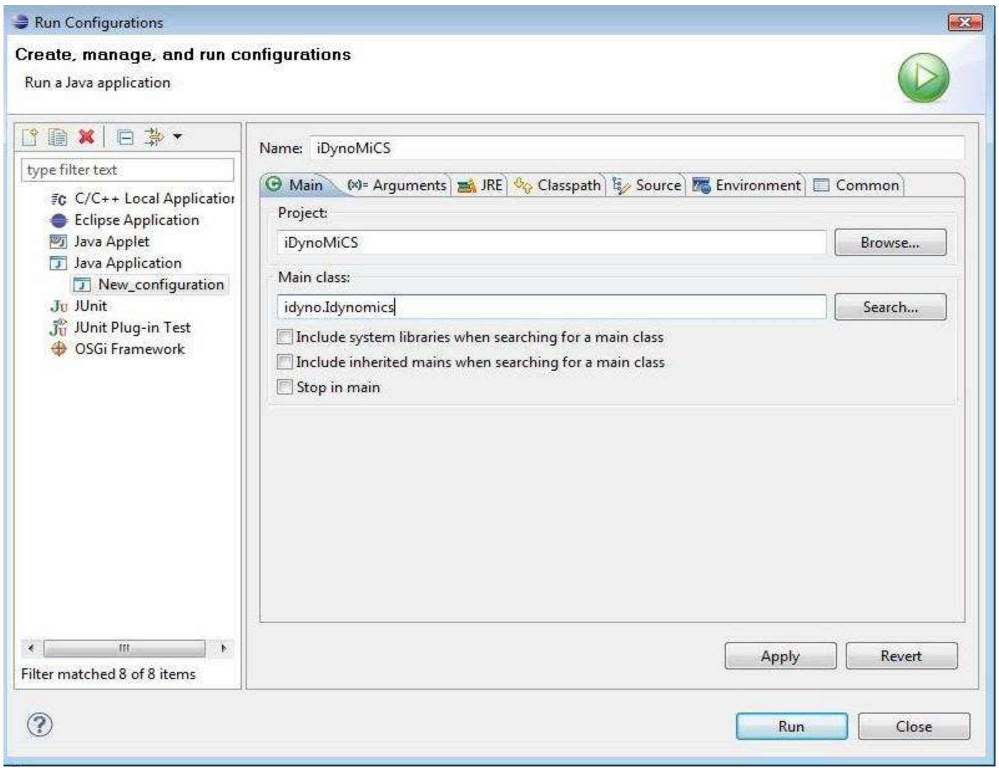
Replicate Simulations

You can also run replicate simulations using RunIdyno.py. For that simply call the script as before, but include the option --multiples= followed by the number of replicates desired. Each replicate run is started with a different random number by reading in the "random.state" file as described at the beginning of this section.

To those who wish to develop iDynoMiCS, and so may have different versions of the source code saved on their system, we recommend that RunIdyno.py be customised to point to a specific source directory. On line 10, replace 'set_to_source_directory' to the location of the desired directory; this can be overridden by calling the command line option --src= followed by an alternate source directory.

III.2 Using the Eclipse Environment

If you are viewing or developing the iDynoMiCS source code using Eclipse, you may run simulations through this environment as well. To do this, from the 'Run' menu select 'Run Configurations'. In the left-side list highlight 'Java Application', and then click the upper-left icon (blank page with a plus symbol) to create a new application. You may name the application whatever you like, though it's often best to have it match the project name. Then fill in the 'Project' field with your project name and the 'Main class' field with 'idyno.Idynomics'. (The next figure illustrates how the fields should be set.)



Run Configurations

Create, manage, and run configurations

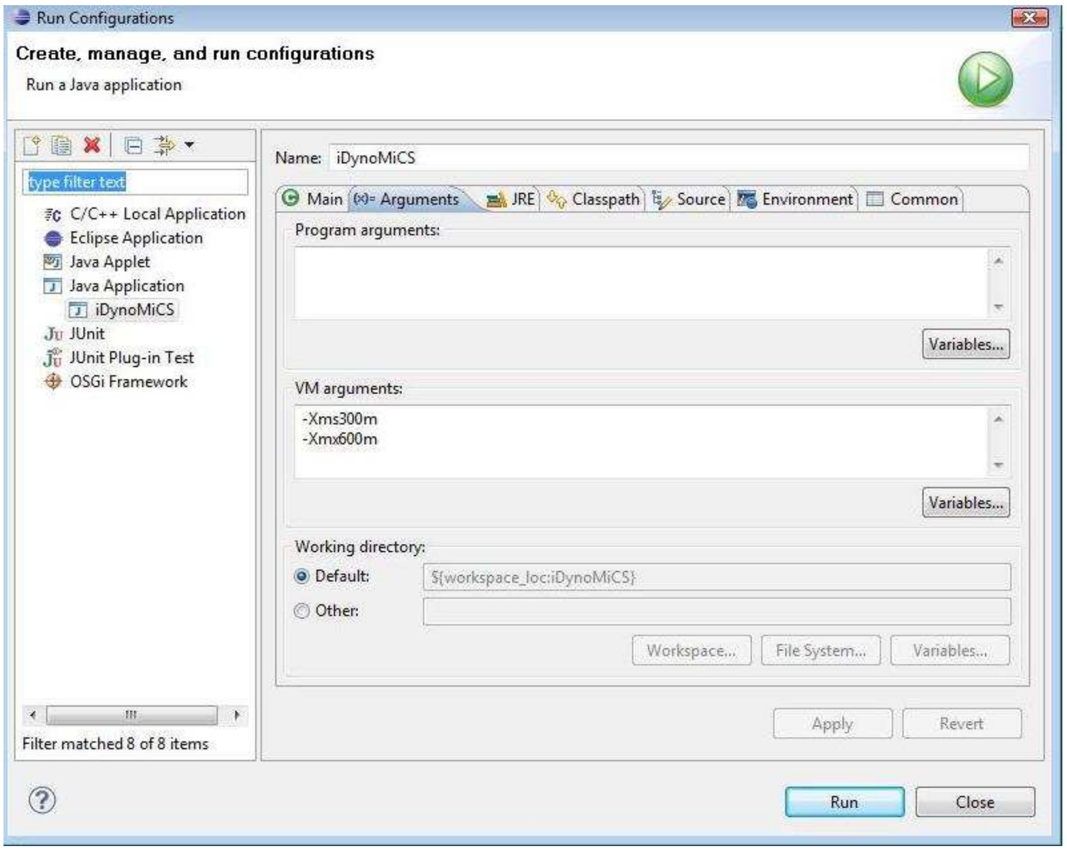
Run a Java application

Name: iDynoMiCS

Project: iDynoMiCS

Main class: idyno.Idynomics

After creating the application, we will set some runtime options. To do this, select the 'Arguments' tab and in the 'VM arguments' box enter the following: `-Xms300m -Xmx600m` (shown in the following figure). These options set the iDynoMiCS memory usage: 300 MB minimum and 600 MB maximum; you may wish to change these values depending on the performance capabilities of your system.


Run Configurations

Create, manage, and run configurations

Run a Java application

type filter text

- C/C++ Local Application
- Eclipse Application
- Java Applet
- Java Application
  - iDynoMiCS
- JUnit
- JUnit Plug-in Test
- OSGi Framework

Filter matched 8 of 8 items

Name: iDynoMiCS

Main | Arguments | JRE | Classpath | Source | Environment | Common

Program arguments:

VM arguments:

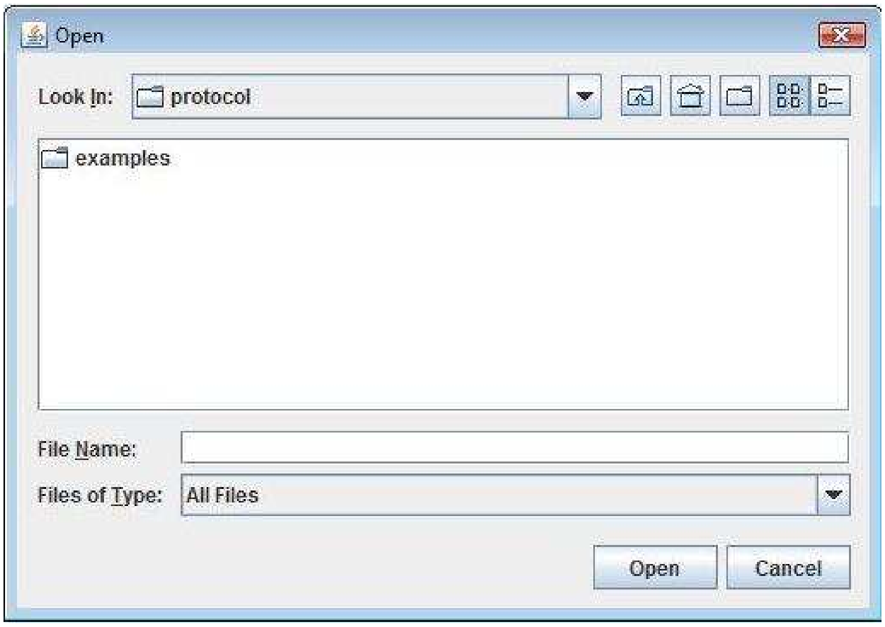
```
-Xms300m
-Xmx600m
```

Working directory:

(●) Default: ${workspace_loc:iDynoMiCS}

( ) Other:

No argument is required in the ‘Program arguments’ box, because when run iDynoMiCS will ask you to choose a protocol file.


Open

Look In: protocol

examples

File Name:

Files of Type: All Files

III.3 Queuing and Running Multiple Different Simulations

Multiple simulations are used either to produce replicate simulations (so using the same protocol file but a different random number seed) or to produce a set of different simulations started with different protocol files, e.g. in order to sweep parameters. In this case it can be useful to specify several simulations to run one after another. To do this, merely collect all the relevant protocol files into one directory, and when choosing a protocol file in the file browser select that directory of interest rather than a particular file. Then all simulations inside this directory will be run one after the other. Remember the advice given at the beginning of this section about random.state files: simulations run in sequence will be started with a different random number state, rather than the same random number state that the simulations would be started with if they were started independently.

IV Protocol Files

iDynoMiCS uses plain-text XML files in order to define the simulation to run; these files are called *protocol files*. In a protocol file, an XML mark-up is used to define the simulation parameters, including details such as run time, simulated solutes, simulated agents, and included reactions. In this section we describe how this mark-up is used. The overall simulation information is contained within the `<idynomics>` and `</idynomics>` opening and closing tags, and comments are specified with the following syntax: `<!-- this is a comment -->`.

We will go through each part of an example protocol file and explain what each part of the mark-up does. For more example protocol files useful as templates when defining your own simulations, look in the 'iDynoMiCS\protocol' directory. The included 'example_2D.xml' and 'example_3D.xml' files are also useful references in creating your own protocol files because they include extensive documentation of each part of the file. Note also in those files that many user-definable labels are prefixed with 'My' so that it is clear which labels are reserved and which are free; we follow that convention in this tutorial.

IV.1 Defining the simulation

The **simulator** mark-up defines the parameters controlling the overall simulation.

```
<simulator>
  <param name="quietMode">false</param>
  <param name="restartPreviousRun">false</param>
  <param name="randomSeed">42</param>
  <param name="outputPeriod" unit="hour">6</param>

  <timeStep>
    <param name="adaptive">true</param>
    <param name="timeStepIni" unit="hour">0.25</param>
    <param name="timeStepMin" unit="hour">0.025</param>
    <param name="timeStepMax" unit="hour">1</param>
    <param name="endOfSimulation" unit="day">15</param>
  </timeStep>

  <param name="agentTimeStep" unit="hour">0.05</param>
  <param name="chemostat">true</param>
  <param name="invComp">false</param>
</simulator>
```

The **quietMode** parameter allows the user to suppress output to the log file in order to save storage space and make simulations run a little faster. We recommend that this be used only once preliminary tests have been performed and the user is confident that their protocol is correct.

The **restartPreviousRun** parameter allows one to restart a previously-completed run, for instance to extend the time of simulation. Restarting in this manner will preserve the numbering of output files. To restart a previous run, merely edit the protocol file contained in the relevant results directory: change this parameter from *false* to *true*, change any other relevant parameters (such as ending time), and when running **RunSimulation.bat** choose the newly modified protocol file from the output directory.

The **randomSeed** parameter allows you to specify a value for the seed of the random number generator. To assess the effect of stochasticity on your model, you should run several simulations with different seeds and look at how the simulation results differ for each case. Note that if a random.state file is present in the same directory as the protocol file then this will override the randomSeed (see Section III).

The **outputPeriod** parameter gives the simulation time-interval between the generation of output files (state, summary and povray files). Low values (more frequent outputs) will lead to more files and higher use of disk space, while high values (fewer outputs) may not have sufficient time resolution to visualize transient behaviour. Therefore this parameter should be set based on the expected dynamics of your system.

The **timeStep** mark-up specifies all parameters for the time stepping of the simulation. This mark-up is analysed by the 'idyno.SimTimer' constructor. If you want to use an adaptive timestep, set the parameter **adaptive** to *true* and assign values for **timeStepMin** and **timeStepMax** (minimum and maximum timestep value, respectively). Otherwise, set the parameter **adaptive** to *false*, then the timeStep will always be equal to value of **timeStepIni**. The parameter **endOfSimulation** defines the overall length of the simulation.

It is possible to specify a smaller timestep for agent dynamics using the **agentTimeStep** parameter, which means that the agents will be stepped several times between relaxations of the Diffusion-Reaction problem. If this mark-up is not present, **agentTimeStep** is equal to the overall simulation timestep.

The **chemostat** parameter must be set to true when simulating a chemostat scenario. In such simulations, the agents' positions are ignored and are not changed and all agents have the same growth conditions in order to simulate a well-mixed chemostat.

It may be convenient for the simulation to stop automatically once there is only one microbial species remaining in the system. This is often the case in invasion or competition contests, and so by setting the **invComp** flag to true the user can save time.

IV.2 Starting from previous state files

The **input** mark-up allows you to use existing state files as the initial condition for a new simulation.

```
  <input>
    <param name="useAgentFile">true</param>
    <param name="inputAgentFileURL">agent_State(last).xml"</param>
    <param name="useBulkFile">false</param>
    <param name="inputBulkFileURL">env_Sum(last).xml"</param>
  </input>
```

This is different from the **restartPreviousRun** parameter above, because these inputs are meant for use in a new simulation starting from time 0 rather than for restarting a previous simulation. Setting the flag to *true* or *false* will use the input file as needed; just be sure that the path to the input file is correct (it is best to copy the input file to the local directory). Note that the input files MUST be consistent with the agent and solute descriptions defined later in the protocol file, or otherwise the simulation will give an error.

IV.3 Defining solutes and particle types

The **solute** mark-ups list and describe all the solute compounds used by the model, along with their molecular diffusivity in water. At this point we do not specify concentration information; that information will be specified in the section for bulk compartments.

```
  <solute domain="biofilm" name="MyO2">
    <param name="diffusivity" unit="m2.day-1">2e-4</param>
  </solute>
  <solute domain="biofilm" name="MyNH4">
    <param name="diffusivity" unit="m2.day-1">1.7e-4</param>
  </solute>
  <solute domain="biofilm" name="MyNO3">
    <param name="diffusivity" unit="m2.day-1">1.6e-4</param>
  </solute>
  <solute domain="biofilm" name="MyCOD">
    <param name="diffusivity" unit="m2.day-1">1.6e-4</param>
  </solute>
  <solute domain="biofilm" name="pressure">
    <param name="diffusivity" unit="m2.day-1">1</param>
  </solute>
```

The ‘pressure solute’ is not actually a physical solute, but is used in computing biomass spreading effects. It is best to leave the pressure mark-up as-is. However, during a **chemostat** run any solutes not involved in the reactions in which the agents participate must not be listed (i.e. do not use the ‘pressure solute’ for chemostat simulations).

The **particle** mark-ups refer to the different compartments used to describe agent species (which is done later in the protocol file). When defining a species, one has to list which of the possible compartments are used by that species. At this level of the protocol file, the compartments are defined independently of any species and described by their name and density.

```
  <particle name="biomass">
    <param name="density" unit="g.L-1">200</param>
  </particle>
  <particle name="inert">
    <param name="density" unit="g.L-1">200</param>
  </particle>
  <particle name="capsule">
    <param name="density" unit="g.L-1">75</param>
  </particle>
```

These three **particle** types are very common (active biomass, inert biomass, and capsular compounds, respectively), and most simulations will employ all three.

IV.4 Defining the world

The **world** mark-up collects the description of all bulks and computation domains defined in the simulation.

```
  <world>
    <bulk name="MyTank">
    <!-- replace "MyTank" with "chemostat" for a chemostat run -->
         ...
    </bulk>
    <computationDomain name="MyBiofilm">
         ...
    </computationDomain>
  </world>
```

Only one **world** may be defined, but this **world** may contain several **bulk** compartments and **computationDomain** domains, each with a different name. Though when simulating a **chemostat** scenario, the name of the bulk MUST be "chemostat" regardless of the corresponding **computationDomain** name.

a) Bulk Compartments

A **bulk** is a perfectly mixed liquid compartment of the system, usually with a larger size than the extent of the simulated biofilm domain. For example, in a wastewater reactor the **bulk** would refer to the liquid volume being treated. The **bulk** will usually have a fixed volume, but the volume is not specified here; instead, the definition of the **computationDomain** addresses the **bulk** compartment volume.

```
<bulk name="MyTank">
<!-- replace "MyTank" with "chemostat" for a chemostat run -->
  <param name="isConstant">false</param>
  <param name="D" unit="h-1">0.6</param>

  <solute name="MyO2">
    <param name="Sbulk" unit="g.L-1">10e-3</param>
    <param name="Sin" unit="g.L-1">10e-3</param>
    <param name="isConstant">true</param>
  </solute>
  <solute name="MyNH4">
    <param name="Sbulk" unit="g.L-1">5e-3</param>
    <param name="Sin" unit="g.L-1">5e-3</param>
  </solute>
  <solute name="MyNO3">
    <param name="Sbulk" unit="g.L-1">0</param>
    <param name="Sin" unit="g.L-1">0</param>
  </solute>
  <solute name="MyCOD">
    <param name="Sbulk" unit="g.L-1">10e-3</param>
    <param name="Sin" unit="g.L-1">10e-3</param>
    <param name="Spulse" unit="g.L-1">20e-3</param>
    <param name="pulseRate" unit="h-1">0.5</param>
  </solute>
</bulk>
```

The **isConstant** parameter sets whether the concentration in the bulk is assumed to be constant or is affected by the mass balance between the reactions occurring in the biofilm and the feed flow. The two values for **isConstant** lead to two different **bulk** compartment behaviours:

1. When **isConstant** is set to *true*, the concentration of each solute in the **bulk** will not vary in time, and will remain at its initial concentration (**Sbulk**, described below) for the entire simulation.
2. When **isConstant** is set to *false*, the solute concentrations in the **bulk** will vary in time. In this case, you must specify the reactor dilution rate (parameter **D**) and the concentration of each solute in the feed flow (parameter **Sin**). The initial concentration of each solute will be set to the value specified in **Sbulk**. In addition, any of the solutes may be affected by a **pulseRate** parameter, described below.

The solutes listed in the **bulk** mark-up are taken from the previously-defined set of all solutes in the simulation, and any solutes not listed here are assumed to be zero in this **bulk**. Each solute is described by the **Sbulk**, **Sin**, **isConstant** and possibly the **Spulse** and **pulseRate** parameters. **Sbulk** sets the initial bulk concentration of that solute, and **Sin** the feed flow concentration of that solute. If **isConstant** is set to *true* the concentration of the solute will remain constant even if the **bulk isConstant** parameter is set to *false*. The **pulseRate** parameter may be used to periodically spike the concentration to **Spulse** at the given rate.

Any solute called **pressure** is kept constant in the bulk because **pressure** is meant to be used for biomass spreading within the biofilm, and thus has no relevance in the bulk compartment.

b) The Computation Domain

Following definition of all **bulk** compartments, the **computationDomain** is defined.

```
<computationDomain name="MyBiofilm">
  <grid nDim="2" nI="33" nJ="33" nK="1"/>
  <param name="resolution" unit="um">8</param>
  <param name="boundaryLayer" unit="um">40</param>
  <param name="biofilmDiffusivity">0.8</param>
  <param name="specificArea" unit="m2.m-3">80</param>

  <boundaryCondition>
         ...
  </boundaryCondition>
         ...
</computationDomain>
```

The **computationDomain** may be either 2D or 3D, as desired. (Note though that 3D simulations are more computationally intensive and will require more time to run.) The spatial domain is broken into rectilinear regions that each occupy a small space, and definition of the **computationDomain** requires defining how these regions are set up.

The parameter **nDim** controls the number of simulation dimensions: 2 or 3. The parameters **nI**, **nJ**, and **nK** set the number of grid elements in each of the x, y, and z directions, respectively. Note that in iDynoMiCS the y and z directions represent the two horizontal directions and x is always the vertical direction – whether 2D or 3D – in order to preserve the substratum location in 2D simulations (when the z direction is ignored). The algorithm used to solve for the solute concentration fields requires that the number of gridpoints be a power of two plus 1, i.e. 33, 65, etc. In the biofilm scenario the **resolution** parameter (units of micrometers) defines the width of each side of each grid element, so that in the first dimension the spatial extent of the computational

domain will be **nI*resolution**, and will be **nJ*resolution** and **nK*resolution** in the second and third dimensions, respectively.
In the chemostat scenario there is only one cubic grid element, and the **resolution** parameter will determine the volume of the reactor and hence the size of the system being simulated - the higher the volume, the more agents can exist for a given set of growth and substrate conditions.

When defining a 2D simulation, merely set **nDim** to 2 and set **nK=1**; the third dimension will be assumed to have the length of the resolution.

In a **chemostat** run there is only one grid element and the resolution is used to calculate the volume of a cube with that length. This is the real volume being simulated, and the number of agents being simulated is determined by the size of the chemostat. Nevertheless, substrate and biomass concentrations in steady state given a set of growth conditions remain unchanged despite the volume being simulated. Bear in mind that big volumes will simulate thousands of agents, which in turn renders the simulations much slower. The volume of the chemostat is noted in the first few lines of the log file.

The next few parameters refer to more physical aspects of the domain. The **boundaryLayer** parameter gives the thickness of the boundary layer between the biofilm and the bulk. This boundary layer forms part of the pure liquid region within the domain, and in this region only diffusion governs the local concentration. Beyond the boundary layer region the liquid is assumed to be well-mixed, and so the solute concentrations are kept equal to their concentration in an attached **bulk** compartment.

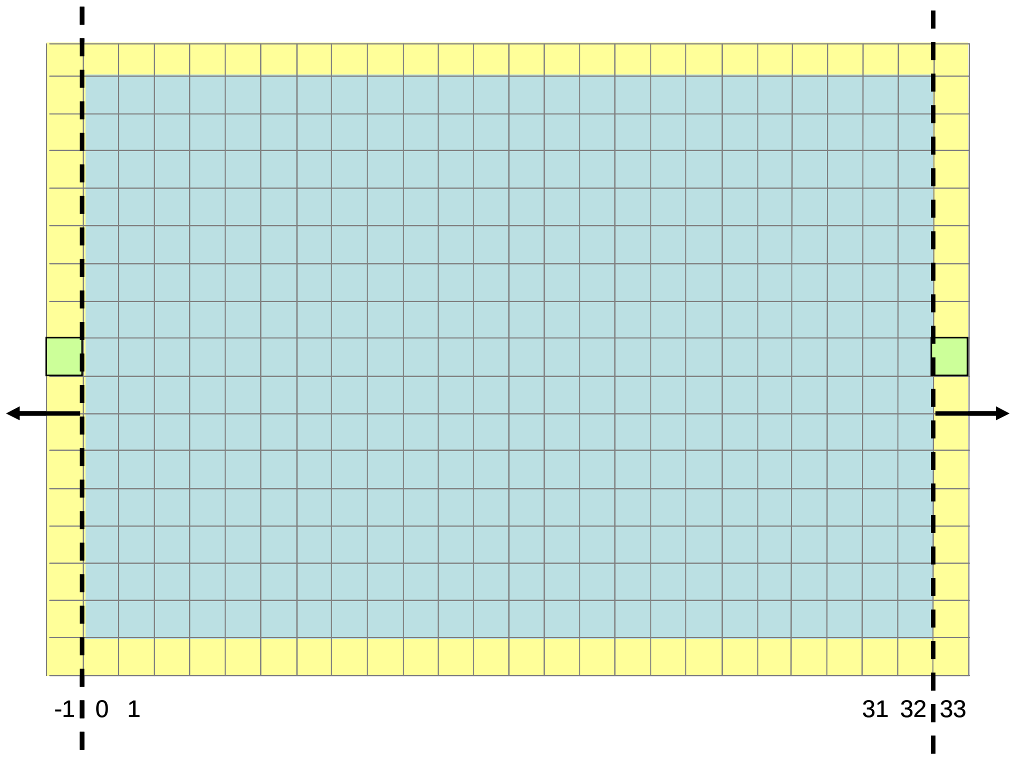
The **biofilmDiffusivity** parameter is a factor used to decrease solute diffusivity inside the biofilm. It is a multiplicative factor applied to the diffusivity in water. Values around 0.8 are common; values lower than 0.5 are rare.

The **specificArea** parameter is used to describe the ratio between the carrier surface (substratum on which the biofilm grows) and the **bulk** compartment volume; this is where the physical volume of the system appears in the simulation definition. This parameter is expressed in units of m^2/m^3.

It is also necessary to define the behaviour at the boundaries of the **computationDomain**. A 2D simulation requires 4 boundary conditions and a 3D system requires 6. iDynoMiCS at this point includes only planar boundaries, but the extensible nature of the code means different shapes may be introduced (see the source code package ***simulator.geometry.shape*** for details). The planar shape of a particular boundary is defined by the following mark-up:

```
  <shape class="Planar">
    <param name="pointIn" x="-1" y="0" z="0"/>
    <param name="vectorOut" x="-1" y="0" z="0"/>
  </shape>
```

Because a plane is fully defined by a point and a normal vector, these two properties must be specified for each boundary; the normal vector by convention points outside the domain. For a particular boundary plane, the parameter **pointIn** defines a gridpoint which exists on that domain boundary (just outside the domain itself), and the parameter **vectorOut** specifies the direction of the planar normal vector. Note that each of these two parameters are described in the XML file using discrete coordinates, and so the **pointIn** parameter should be given in terms of gridpoint index. Because the gridpoint indices run from 0 to N-1, the **pointIn** parameter will take value -1 or N in the relevant direction. For reference, consider the following image of a cyan **computationDomain** with indexed gridpoints and dashed boundaries:



In this image, the green elements represent the points belonging to the boundary plane and the two arrows are the planar normal vectors. The left green element will have the plane description given above, while the green element on the right will be described by the following:

```
  <shape class="Planar">
    <param name="pointIn" x="33" y="0" z="0"/>
    <param name="vectorOut" x="1" y="0" z="0"/>
  </shape>
```

In addition to the boundary shape, one must also specify the behaviour of the boundary when defining a boundary condition. The class used to create a boundary condition is defined by the attribute **class** of the mark-up **boundaryCondition**. iDynoMiCS has several possible boundaries already defined, but once again more may be defined by adding source code for different behaviours (see the package **simulator.geometry.boundaryConditions**). In addition to the **class** attribute of each **boundaryCondition** mark-up, one must specify via the **name** attribute which face of the cubic **computationDomain** is being described. For example, the y-z plane for x = 0 will be called ‘`y0z`’ while the opposing side will be called ‘`yNz`’; these two faces describe the bottom and top of the domain, respectively. The remaining four faces are called ‘`x0y`’, ‘`xNy`’, ‘`x0z`’, and ‘`xNz`’. We now show an example of each currently usable boundary condition type.

The zero-flux boundary condition defines a boundary that is impermeable to both agents and solutes (the substratum is usually of this type):

```
<boundaryCondition class="BoundaryZeroFlux" name="y0z">
  <shape class="Planar">
    <param name="pointIn" x="-1" y="0" z="0"/>
    <param name="vectorOut" x="-1" y="0" z="0"/>
  </shape>
</boundaryCondition>
```

The bulk boundary condition connects the domain to a previously-defined **bulk** compartment:

```
<boundaryCondition class="BoundaryBulk" name="yNz">
  <param name="bulk">MyTank</param>
  <!-- replace "MyTank" with "chemostat" for a chemostat run -->
  <shape class="Planar">
    <param name="pointIn" x="33" y="0" z="0"/>
    <param name="vectorOut" x="1" y="0" z="0"/>
  </shape>
</boundaryCondition>
```

The solute concentrations at this boundary are set equal to the concentrations in the connected **bulk** compartment. This boundary condition is generally used for the top of the domain.

A constant boundary condition is similar to the **BoundaryBulk** type, but in this case the bulk concentration is not updated dynamically:

```
<boundaryCondition class="BoundaryConstant" name="yNz">
  <param name="bulk">MyTank</param>
  <shape class="Planar">
    <param name="pointIn" x="33" y="0" z="0"/>
    <param name="vectorOut" x="1" y="0" z="0"/>
  </shape>
</boundaryCondition>
```

The **BoundaryBulk** type is more generally applicable than **BoundaryConstant** because it may too handle constant bulks, but this condition is here if needed.

Cyclic boundaries are useful in that they allow simulation of larger domains by assuming that the **computationDomain** region is replicated indefinitely; in this treatment it is assumed that variations in the laterally-repeated direction are less important than vertical variations within the domain. The two connected boundaries are specified together when creating cyclic boundary conditions, and so one must define two planes within the **boundaryCondition** mark-up:

```
<boundaryCondition class="BoundaryCyclic" name="x0z">
  <shape class="Planar">
    <point name="pointIn" x="0" y="-1" z="0"/>
    <vector name="vectorOut" x="0" y="-1" z="0"/>
  </shape>
  <shape class="Planar">
    <param name="pointIn" x="0" y="33" z="0"/>
    <param name="vectorOut" x="0" y="1" z="0"/>
  </shape>
</boundaryCondition>
```

Solute concentrations are kept constant across cyclic boundaries, and any agent crossing one of the cyclic boundaries will be instantly moved to the connected boundary. These boundaries are generally used for the sides of the domain.

Finally, a **BoundaryGasMembrane** boundary condition is provided in order to simulate systems where one or more boundaries exhibit selective permeability (e.g. growth on an oxygen-permeable membrane):

```
<boundaryCondition class="BoundaryGasMembrane" name="y0z">
  <param name="bulk">MyGasMembrane</param>
  <param name="isPermeableTo" detail="MyO2" unit="m2.day-1">1</param>
  <shape class="Planar">
    <param name="pointIn" x="-1" y="0" z="0"/>
    <param name="vectorOut" x="-1" y="0" z="0"/>     </shape>
</boundaryCondition>
```

This boundary condition acts like a zero-flux boundary for the agents and all solutes, except for any solutes for which the **isPermeableTo** parameter is given; in that case, the solute permeability governs the behaviour of that solute at the boundary. This boundary condition will generally be used for the domain bottom (substratum).

IV.5 Specifying Reactions

Reactions in which solutes or biomass are produced or consumed mediate the dynamics of the entire iDynoMiCS simulation. Any reaction may be defined via both stoichiometric and kinetic forms, and both of these representations are used in defining reactions in the protocol file. Consider for example a reaction in which a compound S_1 is partially oxidized to produce a solute product S_2 and biomass, with 80% of the biomass produced consisting of active biomass (X) and 20% of the biomass produced consisting of extracellular compounds (EPS). Assume also that the kinetic reaction has a Monod-form dependence on S_1 and O_2, and that solute S_2 inhibits the forward reaction via simple inhibition. We may then write the stoichiometric and kinetic equations for this reaction:

$$\begin{array}{cccccc|c} S_1 & +O_2 & \rightarrow S_2 & +X & +EPS & & \multirow{2}{*}{$r = \mu_{\max} \dfrac{S_1}{K_1^S + S_1} \dfrac{O_2}{K_O^S + O_2} \dfrac{K_2^J}{K_2^J + S_2} X$} \\ 2 & 0.5 & 0.5 & 0.8 & 0.2 & & \end{array}$$

In the stoichiometric equation on the left, the coefficients balance when mass units of oxygen demand (COD) are used (oxygen has a COD mass of -1, so both sides of the equation above have coefficients sums of 1.5). The kinetic equation on the right is composed of several multiplicative terms, each of which is readily specified in the protocol file.

The following **reaction** mark-up specifies the above reaction, and the mark-up for other reactions is similar.

```
<reaction catalyzedBy="biomass" class="ReactionFactor" name="MyGrowthAut">
  <param name="muMax" unit="hour-1">0.0417</param>
  <kineticFactor class="MonodKinetic" solute="MyNH4">
    <param name="Ks" unit="g.L-1">1.5e-3</param>
  </kineticFactor>
  <kineticFactor class="MonodKinetic" solute="MyO2">
    <param name="Ks" unit="g.L-1">0.5e-3</param>
  </kineticFactor>
  <kineticFactor class="SimpleInhibition" solute="MyInhibitingSolute">
    <param name="Ki" unit="g.L-1">0.05e-3</param>
  </kineticFactor>
  <yield>
    <param name="MyNH4" unit="g.g-1">-2</param>
    <param name="MyO2" unit="g.g-1">-0.5</param>
    <param name="MyInhibitingSolute" unit="g.g-1">0.5</param>
    <param name="biomass" unit="g.g-1">0.8</param>
    <param name="capsule" unit="g.g-1">0.2</param>
  </yield>
</reaction>
```

In general, reactions are specified using the **ReactionFactor class** because this allows the most general multi-factor reaction expression (though you may look at the iDynoMiCS source under the package ***simulator.reaction*** for additional forms). Like for the solute and particle names, you may choose any name you would like for the reaction's **name** attribute. You must also specify which **particle** type facilitates the reaction using the **catalyzedBy** attribute; during the simulation, this reaction will only occur if this **particle** type is present.

The first parameter describing the reaction is the **muMax** parameter, which is the maximum specific rate at which the reaction may occur, in units of 1/*hour*. This parameter will multiply each factor that follows in order to yield a net reaction rate.

Following the rate parameter are several **kineticFactor** mark-ups, which comprise the multiplicative terms that make up the entire reaction kinetic. In the example above, we have two Monod kinetic terms and a simple inhibition term; each of these terms acts on a particular solute and has a particular controlling parameter. iDynoMiCS includes the following suite of **kineticFactor** types:

- **MonodKinetic**: specify the solute of interest as an attribute and the half-maximum concentration (**Ks**) as a parameter. Formula: $\mu = \frac{S}{K_S + S}$.

```
<kineticFactor class="MonodKinetic" solute="MyCOD">
  <param name="Ks" unit="g.L-1">1.5e-3</param>
</kineticFactor>
```

- **SimpleInhibition**: specify the solute of interest as an attribute and the half-maximum concentration (**Ki**) as a parameter. Formula: $\mu = \frac{K_i}{K_i + S}$.

```
<kineticFactor class="SimpleInhibition" solute="MyInhibitingSolute">
  <param name="Ki" unit="g.L-1">0.05e-3</param>
</kineticFactor>
```

- **HaldaneKinetic**: specify the solute of interest as an attribute and the concentrations (**Ki**) and (**Ks**) as parameters. Formula: $\mu = \frac{S}{K_S + S + \frac{S^2}{K_i}}$.

```
<kineticFactor class="HaldaneKinetic" solute="MySolute">
  <param name="Ks" unit="g.L-1">0.05e-3</param>
  <param name="Ki" unit="g.L-1">0.25e-3</param>
</kineticFactor>
```

- **HillKinetic**: specify the solute of interest as an attribute and the half-maximum concentration (**Ks**) and exponent (**h**) as parameters. Formula: $\mu = \frac{S^h}{K_S^h + S^h}$.

```
<kineticFactor class="HillKinetic" solute="MySolute">
  <param name="Ks" unit="g.L-1">0.05e-3</param>
  <param name="h" unit="-">0.5</param>
</kineticFactor>
```

- **FirstOrderKinetic**: no parameters or attributes to specify; the reaction rate depends only on the **muMax** parameter.

```
<kineticFactor class="FirstOrderKinetic"/>
```

The final component of the **reaction** mark-up is the **yield** mark-up. Within this mark-up are specified the yield coefficients of all components affected by the reaction; these coefficients are taken from the original stoichiometric equation defined at the beginning of this section. To define a coefficient, you merely add a new parameter to the mark-up, specify the affected solute or particle using the **name** attribute, and give the coefficient as the parameter value. Negative values represent consumption of the component, and positive values represent production. Note that there is NO control on the chemical balance of the chosen stoichiometric coefficients: it is up to YOU to ensure that your model simulation matches reality, and that there is a mass balance in the equations. Also note that when defining the reactions in the simulation, it is necessary to use solutes and particles already defined; any syntax mistake in their use will cause an error.

IV.6 Setting up the Solute Fields and Pressure Solvers

In iDynoMiCS simulations, it is assumed that the solute fields are always in steady-state with respect to biomass growth because solute consumption/production and diffusion occur very fast (on the order of seconds) compared to growth of biomass (which occurs on the order of hours to days). Therefore, at each time step it is possible to compute the steady-state solute profile within the computation domain. iDynoMiCS allows one to use different solvers to compute these fields, but by default uses an iterative multi-grid method. In the protocol file, a **solver** mark-up defines parameters for the solver.

```
<solver class="Solver_multigrid" name="MySoluteSolver" domain="MyBiofilm">
  <param name="active">true</param>
  <param name="preStep">150</param>
  <param name="postStep">150</param>
  <param name="coarseStep">1500</param>
  <param name="nCycles">5</param>

  <reaction name="MyGrowthAut"/>
  <reaction name="MyMaintenanceAut"/>
  <reaction name="MyGrowthHet"/>
  <reaction name="MyMaintenanceHet"/>
</solver>
```

In this mark-up, you must specify the type of solver to use (via the **class** attribute), a name for the solver (the **name** attribute) and the computation domain in which the solver is being used (the **domain** attribute). Other parameters are particular to the solver being used.

The example above shows the implementation for the multi-grid solver. It is usually not necessary to change the particular controlling parameters, but we describe them here for completeness. The **active** parameter defines whether the solver will actually be used; the **preStep** and **postStep** parameters define the number of iterations made before and after a resolution change; the **coarseStep** parameter defines the number of iterations done at the coarsest resolution; and **nCycles** defines the maximal number of cycles to be carried out by the solver.

After defining solver parameters, you must also specify the reactions that will be used in computing the solute fields; this is done using the **reaction** mark-up. In general, all reactions that involve a solute (both production and consumption) will appear here, while reactions that involve only biomass (e.g. conversion of active to inert biomass) will not be included. Be sure that the names entered here match those entered in the **reaction** mark-ups above, for otherwise a syntax error will result and the simulation will fail.

In addition to using a solver for computing the solute fields, iDynoMiCS uses a similar method to compute the pressure field that was defined similar to the solutes; this is done using the following mark-up:

```
<solver class="Solver_pressure" name="MyPressureSolver" domain="MyBiofilm">
  <param name="active">true</param>
</solver>
```

Note that there is only one parameter (**active**), as the pressure solver includes by design all reactions affecting biomass components. If you do not include the pressure field, this section may be removed.

To simulate bacteria growth in a chemostat we use the solver class **Solver_chemostat**, for which two parameters can be set in the protocol file: **rtol** stands for the relative tolerance of the calculated error and **hmax** is maximum internal step of the solver. Further details can be found in the class documentation, but in general it is not necessary to change the default values.

```
  <solver class="Solver_chemostat" name="MyChemostatSolver"
domain="MyBiofilm">
    <param name="rtol">1e-2</param>
    <param name="hmax">1e-3</param>
    <param name="active">true</param>

  <reaction name="MyGrowthAut"/>
  <reaction name="MyMaintenanceAut"/>
  <reaction name="MyGrowthHet"/>
  <reaction name="MyMaintenanceHet"/>
  </solver>
```

IV.7 Setting up the agentGrid

All agents in biofilm simulations are stored and tracked using a grid called the **agentGrid**, which is similar to the grid for the solutes but serves a different purpose unique to agents. There are several parameters that need to be specified for the grid, and this is done with the **agentGrid** mark-up.

```
<agentGrid>
  <param name="computationDomain">MyBiofilm</param>
  <param name="resolution" unit="um">8</param>

  <detachment class="DS_Quadratic">
    <param name="kDet" unit="um-1.hour-1">1e-6</param>
    <param name="maxTh" unit="um">500</param>
  </detachment>

  <param name="sloughDetachedBiomass">true</param>

  <param name="shovingMaxNodes">2e6</param>
  <param name="shovingFraction">0.025</param>
  <param name="shovingMaxIter">250</param>
  <param name="shovingMutual">true</param>
</agentGrid>
```

The first parameter (**computationDomain**) specifies the domain in which the grid is to be used, and the second parameter (**resolution**) specifies the size of each grid element in physical units. The resolution should be chosen to match the size of the individuals (should be near the diameter of the largest species) and the resolution of the solute grid. It is best to choose a size that is a factor of two different from the size of the solute grid, i.e. if the solute grid has a resolution of 8 μm, the **agentGrid** should have a resolution of 4, 8, or 16 μm. If unsure, use 8 μm.

The **detachment** mark-up defines one way in which the outside world may affect growth of the biofilm system via erosion acting at the surface of the growing biofilm. There are two detachment types currently available in iDynoMiCS: 'DS_Quadratic' and 'DS_Biomass'. The desired form of detachment is set using the **class** attribute. The 'DS_Quadratic' type (illustrated above) sets the erosion strength to be a parameter k_{Det} times the square of the local biofilm height, and the parameter **kDet** must be specified in the mark-up. You must also set the maximum thickness that the biofilm may reach (**maxTh**), which should be used as a safety catch to ensure that the biofilm region remains within the computation domain; any biomass growing above this height will be removed from the system, or in order to keep the biofilm to a certain thickness below the top of the domain. The second detachment **class** is 'DS_Biomass', which is similar to the 'DS_Quadratic' **class** but with the added property that the erosion is scaled by the local biomass concentration; this means that higher concentrations of biomass will be less susceptible to detachment. The mark-up looks as follows:

```
  <detachment class="DS_Biomass">
    <param name="kDet" unit="fg.um-4.hour-1.">2e-4</param>
    <param name="maxTh" unit="um">300</param>
  </detachment>
```

(Recall the conversion 1 fg/μm^3 = 1 g/L, which you might use in relating this detachment parameter to the height-based parameter above.)

As the biofilm grows, it sometimes happens that decay and erosion outpace growth in the lower reaches of the biofilm. When this occurs, agents may be removed because they have become too small, and if this happens for many agents it is possible that large chunks of biofilm may no longer be connected to the substratum. The **sloughDetachedBiomass** parameter sets whether you would like biofilm pieces that are no longer connected to the substratum to be removed from the domain. The default value is *true*, to remove those pieces, but some applications might require the pieces to remain; in those cases, set the flag to *false*.

Finally, the remaining parameters set the behaviour of the shoving algorithm, whose purpose is to move agents as they grow and compete for space in the biofilm. In general, it is not necessary to modify these values, but again for completeness we describe their purpose. The **shovingFraction** parameter sets the allowed cut-off for moving agents; when the fraction of still-moving agents is below this value, the agent positions are accepted as equilibrated and the simulation continues. The **shovingMaxIter** parameter sets the maximal number of iterations allowed to reach the final agent positions. Finally, the **shovingMutual** parameter sets whether the shoving motion is applied to both agents or only to one when two agents overlap.

IV.8 Specifying Species

The final part of the protocol file is used to define the species involved in the simulation using **species** mark-ups. Each species is defined from one of several different classes, and each is given a name and has some defining parameters. In the iDynoMiCS source it is possible to introduce nearly any type of species one would like, but at this point there are a few select species types that are possible to use: *Bacterium*, *BactEPS*, *BactAdaptable*, or *ParticulateEPS*, all found in the package ***simulator.agent.zoo*** in the iDynoMiCS source. We will describe each of the available **species** in turn.

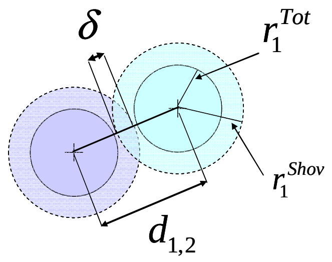
a) Bacterium

The *Bacterium* species is the most commonly-used type, and shown here is an example mark-up for a typical *Bacterium*.

```
<species class="Bacterium" name="MyAutotroph">
  <!-- all particles go here -->
  <particle name="biomass">
    <param name="mass" unit="fg">1200</param>
  </particle>
  <particle name="inert">
    <param name="mass" unit="fg">0</param>
  </particle>
  <particle name="capsule" class="MyAutotrophEPS">
    <param name="mass" unit="fg">0</param>
  </particle>

  <!-- all common parameters go here -->
  <param name="color">red</param>
  <param name="computationDomain">MyBiofilm</param>
  <param name="divRadius" unit="um">4</param>
  <param name="deathRadius" unit="um">0.5</param>
  <param name="babyMassFrac">0.5</param>

  <param name="shoveFactor" unit="um">1.25</param>
  <param name="shoveLimit" unit="um">0</param>

<!-- setting the CV (coefficient of variation) to zero will stop the
simulation from making random deviations from the protocol specified
parameter values. Use this to turn a stochastic cell division or death into
a deterministic one. The default CV is 0.1. -->
  <param name="divRadiusCV">0</param>
  <param name="deathRadiusCV">0</param>
  <param name="babyMassFracCV">0</param>

  <!-- all parameters unique to a species go here -->
  <param name="epsMax">0.1</param>

  <!-- all reactions go here -->
  <reaction name="MyGrowthAut" status="active"/>
  <reaction name="MyMaintenanceAut" status="active"/>
  <reaction name="MyInactivationAut" status="active"/>

  <!-- initial locations go here -->
  <initArea number="25">
    <param name="birthday" unit="hour">0</param>
    <coordinates x="0" y="0" z="0"/>
    <coordinates x="1" y="528" z="0"/>
  </initArea>
</species>
```

The first part of the species mark-up lists the different **particle**s composing this species; these particles form 'compartments' within the agent that signify the types of biomass that the agent contains. The possible **particle** types that may be used are taken from the **particle** mark-ups defined early in the protocol file, generally consisting of 'biomass', 'inert', and 'capsule'. The initial mass of each **particle** may be given (in units of femtograms, 1 fg = 10^{-15} g), but if you set the value to zero the simulator will assign a reasonable random initial value for the mass. For *Bacterium* species, only one **particle** type is actually necessary to include, but it is common to include several types. In the example mark-up above, the *Bacterium* contains active biomass, inert biomass, and EPS compartments (the 'biomass', 'inert', and 'capsule' compartments, respectively); in this species, inert and capsule compartments exhibit special behaviour: inert biomass will accumulate in the agent, while a capsule will be excreted if it accumulates to take up too much of the agent's mass. Note that the 'capsule' particle also requires you to specify its type via the **class** attribute; when the capsule has become too large and needs to be excreted, an agent of that **class** will be created (this means that it is necessary to have *previously defined* a species with that **name**; in the above example, the 'MyAutotrophEPS' species MUST be defined before 'MyAutotroph').

After the **particle** mark-ups, several different species parameters are defined; these few parameters are common to most types of agents. The **color** parameter defines the color used to draw agents of this species in POV-Ray images; possible color names include: red, blue, cyan, darkgray, lightgray, gray, green magenta, orange, pink, yellow, black, and white. The **computationDomain** parameter merely states which domain contains this particular species. The **divRadius** parameter defines how large an agent is allowed to grow before dividing into two agents. The **deathRadius** parameter defines how small an agent may become before being removed from the simulation. The **babyMassFrac** parameter defines what fraction of the parent agent's biomass the newly created agent will inherit at division. The next two parameters define how neighbouring agents interact, as illustrated in the following diagram:



In this figure, two agents (defined by solid lines) overlap in their region of influence (dashed lines), and will thus shove one another apart. The **shoveFactor** parameter is a factor multiplying the agent radius that is used to determine the region of influence in neighbour shoving; in the image, the shoving radius r_1^{Shov} is computed from the agent radius r_1^{Tot} using this factor. The **shoveLimit** parameter sets the minimal distance between two agents, and is usually set to zero; if two agents are closer than this limit (overlap δ is positive), then shoving will occur.

The three additional parameters **divRadiusCV, deathRadiusCV** and **babyMassFracCV** can be used to define the degree of stochasticity affecting the division and death processes. By default the CV (coefficient of variation) of these parameters is set to 0.1 (see the **LocatedParam** class in the source code). Setting them to zero allows one to simulate an almost deterministic version of iDynoMiCS and can be useful to assess the effect of stochasticity in a particular system.

Following the location, size, shoving, and coefficient of variation parameters, one now defines any parameters particular to this **class** of species. For the **class** *Bacterium* shown above, the only additional parameter to specify is **epsMax**, which defines the maximum volume fraction of the agent that may be taken up by the EPS capsule before EPS excretion will occur. Other species types generally have other parameters that are defined here, as will be explained in the following sections.

The next item to define is the set of reactions in which this species is involved. These reactions are taken from the **reaction** mark-ups that were defined earlier, and once again you must take care that for each reaction the correct **name** is used in order to avoid a syntax error. For each reaction you may also set the **status** attribute to 'active' or 'inactive'; some species types include mechanisms for switching reactions on or off during a simulation.

The final entry in the **species** mark-up is the **initArea** mark-up. The previous parameters in the mark-up defined properties of the species, but this last mark-up will actually create individuals of the species. The **number** attribute specifies how many agents of this species are to be created, and the parameter **birthday** specifies the point of the simulation at which they will be introduced (with 0 hours meaning at the start of the simulation). The two **coordinates** parameters specify two opposing corners of the rectilinear region within which the agents will be randomly placed. In 2D simulations, this region is a rectangle and the **z** value is not read, while in 3D simulations the region is a rectangular prism.

b) BactEPS

The *BactEPS* species behaves just like the *Bacterium* species, with one small change: capsular EPS is excreted continuously at a defined rate rather than during discrete events like for a *Bacterium*. During excretion, an agent will distribute any excreted EPS to neighbouring EPS agents of that type, or will create a new EPS agent if none are nearby. This new functionality requires specification of one additional parameter: the hydrolysis rate **kHyd**, which controls the rate at which capsular EPS is excreted to neighboring EPS particles.

```
<species class="BactEPS" name="MyAutotrophToo">
         ...
  <!-- all parameters unique to a species go here -->
  <param name="epsMax">0.1</param>
  <param name="kHyd" unit="hr-1">0.007</param>
         ...
</species>
```

Besides this change the mark-up for *BactEPS* is the same as for *Bacterium*.

c) BactAdaptable

The *BactAdaptable* species derives from the *BactEPS* species, but adds the ability to change its set of active reactions based on definable conditions. The mechanism for this switching behavior is an ON/OFF state switch, with each state having a particular set of active reactions that are set as inactive when the switch is in the opposite state. You may use local solute concentrations or agent biomass amounts as switch conditions; this example mark-up uses the local oxygen (MyO2) concentration as the switch condition.

```
<species class="BactAdaptable" name="MyAdaptableAutotroph">
         ...
  <reaction name="MyGrowthAerobic" status="active"/>
  <reaction name="MyGrowthAnaerobic" status="inactive"/>
         ...
  <reactionSwitch>
     <whenOff>
        <reaction name="MyGrowthAerobic" status="active"/>
        <param name="switchLag" unit="hour">0</param>
        <param name="color">white</param>
     </whenOff>
     <whenOn>
        <reaction name="MyGrowthAnaerobic" status="active"/>
        <param name="switchLag" unit="hour">3</param>
        <param name="color">black</param>
     </whenOn>
     <onCondition type="solute" name="MyO2">
        <param name="switch">lessThan</param>
        <param name="concentration" unit="g.L-1">0.25e-3</param>
     </onCondition>
  </reactionSwitch>
         ...
</species>
```

The first part of the **species** mark-up proceeds as for the other types. After all possible reactions have been defined, the **reactionSwitch** mark-up is used to define the behavior of the switch. The first two sub-mark-ups, **whenOff** and **whenOn**, define which reactions will be active for those respective switch states. (Note that it is not necessary to explicitly set an on-state reaction as 'inactive' when the switch is in the off-state; the turning on and off of reactions is taken care of automatically.) Each sub mark-up also defines the lag time (**switchLag**) before the switch is actually put in the relevant position, and is meant to capture microbial metabolic delays. In the example above, when conditions are such that the switch should be put in the ON state, the switch will not actually occur until after a delay of 3 hours; in contrast, the switch will move to the OFF position with no delay because the parameter **switchLag** is 0 for the **whenOff** state. (Note that if a switch is triggered but not yet activated because of the lag, a reversal of conditions will cancel the switch request.) Finally, the ON/OFF states each include a **color** definition, which set what color the agent should be in the POV-Ray output; setting a different color for each switch state is useful in analyzing simulation output.

The **onCondition** mark-up defines the condition for which the switch will be moved to the ON state, with the opposite condition moving the switch to the OFF state. The attribute **type** may be either 'solute' or 'biomass', and sets whether the condition is based on a local solute concentration or the agent's biomass amount. The **name** attribute defines the solute or particle type that is used to decide the switch state. For the **onCondition** mark-up, the **switch** parameter may be set to either '*lessThan*' or '*greaterThan*', and for solutes there is a **concentration** parameter that is used to set the threshold value. In the example above, the switch will be moved to ON when the local oxygen (MyO2) concentration is below 0.25 mg/L, and will be moved to OFF when the concentration is above that value; these states lead to anaerobic and aerobic growth, respectively. For a switch based on a biomass amount, the **onCondition** mark-up looks as follows:

```
      <onCondition type="biomass" name="biomass">
         <param name="switch">lessThan</param>
         <param name="mass" unit="fg">2000</param>
      </onCondition>
```

The **name** attribute must refer to one of the **particle** types defined previously (usually *biomass*, *inert*, or *capsule*), and the **mass** parameter is specified in femtograms (1 fg = 10^{-15} g).

d) ParticulateEPS

The final species currently defined in the general iDynoMiCS code is *ParticulateEPS*. This species is composed entirely of the *capsule* particle type, and includes the parameters common to all species.

```
<species class="ParticulateEPS" name="MyAutotrophEPS">
  <particle name="capsule">
    <param name="mass" unit="fg">0</param>
  </particle>

  <param name="color">pink</param>
  <param name="computationDomain">MyBiofilm</param>
  <param name="divRadius" unit="um">5</param>
  <param name="deathRadius" unit="um">0.1</param>
  <param name="shoveFactor" unit="um">1</param>
  <param name="shoveLimit" unit="um">0</param>

  <reaction name="MyHydrolysisEPS" status="active"/>
</species>
```

This species often includes a hydrolysis reaction, which converts COD bound in EPS into soluble COD. Note that if a bacterium species includes a capsule of this species of *ParticulateEPS*, the EPS species MUST be defined first in the protocol file in order to avoid an error; because of this the example protocol files included in iDynoMiCS list *ParticulateEPS* as the first defined **species**.

V Result Files

During the course of a simulation, iDynoMiCS will save output files describing the current solute fields and agent states. These output files will be written at the interval specified by the **outputPeriod** parameter in the **simulator** mark-up. After each output period, six different XML files are written: **agent_State(nFile).xml**, **agent_Sum(*nFile*).xml**, **agent_StateDeath(nFile).xml**, **agent_SumDeath(*nFile*).xml**, **env_State(*nFile*).xml**, and **env_Sum(*nFile*).xml**. In each file name, *nFile* represents the iteration number at which the file was written. The **agent_State** and **agent_Sum** files describe the state of the agents in the system; the **agent_State** file describes each agent in detail, while the **agent_Sum** file summarizes the agents on the species level. The ***Death*** version of these files lists all agents that have been removed from the system at the given iteration and includes the reason for the agent's death: too small, detached, over board (i.e. killed by the boundary bulk for trespassing) or washed away by dilution in the chemostat case. Similarly, the **env_State** and **env_Sum** files describe, respectively, the overall state of the solute fields and a more summarized version. After being written, each result file is also copied to a zip archive kept for each type, with the most recently written state always being accessible in the 'lastIter' directory. Note that these files can take up several megabytes each, so be careful when choosing the **outputPeriod** parameter to avoid taking up too much disk space.

In addition to the numerical description, a .pov file describing a picture of the agents in the biofilm is also written. This file can be processed using the POV-Ray software.

We now describe the output files in a little more detail.

a) agent_State and agent_Sum Files

The **agent_State** files describe all the agents, sorted by species. The information included in the agent description depends on the species class (*e.g.* **Bacterium**, **ParticulateEPS**, etc.). A species mark-up provides the list of descriptors in the attribute **header**, followed by an array where each line represents a different agent and each column a different agent property.

```
<idynomics>
  <simulation iterate="24" time="12.0">
    <grid resolution="8.0" nI="33" nJ="33" nK="1"/>
    <species name="EPS_Aut"
header="family,genealogy,generation,birthday,biomass,inert,capsule,growthRat
e,volumeRate,locationX,locationY,locationZ,radius,totalRadius">
0,0,0,0.0,0.0,0.0,48.46,0.0,0.0,0.49,7.12,0.0,0.0,0.20;
0,0,0,0.0,0.0,0.0,51.27,0.0,0.0,18.90,279.36,0.0,0.0,0.21;
0,0,0,0.0,0.0,0.0,63.32,0.0,0.0,18.18,323.13,0.0,0.0,0.23;
        ...
    </species>
    <species name="Aut" header="family,genealogy,generation,
birthday,biomass,inert,capsule,growthRate,volumeRate,locationX,locationY,loc
ationZ,radius,totalRadius">
112,16,5,19.75,1159.2,10.9,30.2,0.0,0.0,25.6,118.6,0.0,0.70,0.72,0,0,0.0,0.0
;
109,18,5,21.25,1152.9,11.57,18.0,0.0,0.0,22.4,101.8,0.0,0.70,0.71,0,0,0.0,0.
0;
63,10,4,21.25,846.2,11.0,41.8,0.0,0.0,9.3,317.4,0.0,0.60,0.63,0,0,0.0,0.0;
        ...
    </species>
  </simulation>
</idynomics>
```

The **agent_Sum** files are similar, but contain information about each species as a whole rather than information about each individual agent:

```
<idynomics>
  <simulation iterate="24" time="12.0">
    <grid resolution="8.0" nI="33" nJ="33" nK="1"/>
    <species name="EPS_Aut" header="population,mass,growthRate">
7.0,206.21,-2.71
    </species>
    <species name="Aut" header="population,mass,growthRate">
107.0,21301.21,190.71
    </species>
  </simulation>
</idynomics>
```

There are some general Matlab routines provided that allow the result files to be read in for analysis. To use these files, you must first set the Matlab path as described early in this tutorial, and then change your working directory to where the result files are located. Then you may read in a particular agent state using the command

```
a = loadAgents(nFile);
```

where *nFile* is the index of the file you want to load and analyze (*i.e.* its filename is **agent_State(*nFile*).xml**). If the file exists it will be loaded, and if it does not exist, the program will search into the zip file, decompress the file, and then load it. By using instead the syntax

```
a = loadAgents(nFile,true);
```

the program will delete the file it has decompressed after having loaded it into the Matlab workspace. The `loadAgents` routine returns an indexed structure (‘`a`’ in the above examples), which means that `a(1)` stores a structure describing the first species, `a(2)` a structure for the second species, and so on. Each structure includes several fields, including the name of the species in **name**, a **header** describing the saved data, and a **data** matrix that contains a description of all agents that belong to this species. For example, to extract the spatial coordinates of an agent of the first species, use the following command:

```
xyz = a(1).data(:,[10 11 12]);
```

This routine copies from the **data** matrix all rows, but only columns 10, 11, and 12, which you can see in the **agent_State** mark-up above are the columns containing the location data (Matlab begins counting at 1 for indices, unlike Java which begins counting at 0). Note that you may easily access the last state by changing into the ‘lastIter’ directory and entering -1 for the *nFile* parameter.

b) env_State and env_Sum Files

The **env_State** files contain a description of the concentration grid (2D or 3D) for different simulation components that are represented on a grid, including: each solute, the current rate of each reaction, the total biomass in each grid element, and a measure of the total biomass in the system. Also included in the output files are the mean, standard deviation, and maximum thickness of the biofilm/liquid interface.

```
<idynomics>
 <simulation iterate="3" time="3.0" unit="hour">
<thickness domain="MyBiofilm" unit="um">
  <mean>6.8</mean>
  <stddev>3.8157568056677826</stddev>
  <max>12.0</max>
</thickness>
     <solute name="MyCOD" unit="g.L-1" resolution="8.0" nI="33" nJ="33"
nK="1">
0.0;
0.007984741958479553;
0.008212787453421614;
0.008192453215480797;
     </solute>

     <solute name="pressure" unit="g.L-1" resolution="8.0" nI="33" nJ="33"
nK="1">
0.0;
10.453449374255657;
0.0;
5.658756459327816;
12.087282908434052;
     </solute>

     <solute name="MyGrowthHeterotrophs-rate" unit="g.L-1" resolution="8.0"
nI="33" nJ="33" nK="1">
0.0;
36.85145201293968;
0.0;
11.847738644031354;
45.1715474628225;
0.0;
     </solute>

     <solute name="MyHeterotroph" unit="g.L-1" resolution="8.0" nI="33"
nJ="33" nK="1">
0.0;
0.0;
56.13524052200083;
0.0;
18.0340393493944;
     </solute>
     <solute name="totalBiomass" unit="g.L-1" resolution="8.0" nI="33"
nJ="33" nK="1">
0.0;
56.135240522000835;
0.0;
18.0340393493944;
     </solute>
  </simulation>
</idynomics>
```

The units for each concentration are included in the **unit** attribute. The values within each **solute** mark-up describe the 1D array of *K*-values for each *I* and *J* index. In general, you do not have to work with these files directly because there are Matlab routines pre-made for reading these values in.

To load the solute grids into Matlab, you may follow the same procedure as for the agents, but instead use the following function:

```
s = loadSolute(nFile,true);
```

As before, the variable `s` is an array of structures, with the **name**, **resolution**, and **data** fields. A routine to display the solutes is also available:

```
displaySolute(s);
```

This routine will create plots displaying the concentration for each solute, the reaction rates, and the biomass amounts. If the grids represent 3D simulations, two 2D plots will be created, one averaging the y-direction, the other one averaging the z-direction. In addition, another plot displaying the profile along the x-direction is created.

The **env_Sum** files are similar to the **env_State** files, but contain information about the solute concentrations in the different bulk compartments:

```
<idynomics>
  <simulation iterate="24" time="12.0">
    <thickness unit="um">
      <mean>43.07987</mean>
      <stddev>5.87328</stddev>
      <max>54.0</max>
    </thickness>
    <bulk name="tank">
      <solute name="MyCOD" unit="g.L-1">0.02</solute>
        ...
      <uptake_rate name="MyCOD" unit="g.L-1.hour-1">0.01E-02</uptake_rate>
        ...
    </bulk>
  </simulation>
</idynomics>
```

For each **bulk** mark-up, there are **solute** and **uptake_rate** mark-ups describing the concentration and rate of change for each solute in that bulk.

c) POV-Ray Files

The .pov files written by iDynoMiCS may be rendered into images individually by right-clicking on the file in an explorer window and selecting the 'Render' command. (That this command is available of course requires that POV-Ray was installed correctly.) You may render multiple files sequentially in the same manner simply by selecting more than one command before selecting 'Render'.

However, these files may also be rendered using a few Matlab routines. To use these routines, you must first open the *getProgramPath.m* file located in the 'iDynoMiCS\results_analysis_matlab\routines_2d\utilities' directory and edit the paths to the Pov-Ray, QuietPOV, and ImageMagick software installations to match your computer's configuration (where you installed the software). The example locations already in the file should give you an idea what the paths on your system might be.

After editing this Matlab file, you may create images from the .pov files using the `createPics` command, for example with a call such as:

```
createPics('all');
```

Passing in the 'all' argument will render images for all .pov files in the current directory, and if none are present will first unzip the 'povray.zip' file to create the files. This routine will also accept 'last' or a number in order to render the last iterate or a particular iterate, respectively.

To make a movie from the .pov files, use the `createMovie` routine. For example, use the following command:

```
createMovie('myMovie.gif');
```

where *myMovie.gif* is the name of the output movie file. Running the command like this will first create images from the .pov files (using `createPics('all')`), but if you have previously rendered the images you may instead pass in 'true' for the second argument to use the already existing files:

```
createMovie('myMovie.gif', 'true');
```

This routine will produce movie file formats ending in '.gif', '.avi', and '.mpg'. To produce a different output format simply change the name passed into the routine, for example:

```
createMovie('myMovie.avi');
```

If a simulation has been run and you would like to change the color of one or more species in the POV-Ray files (for example if you mistakenly set two species to be the same color), you may edit the 'sceneheader.inc' file and change the RGB components of the colors declared for each species.

VI Suite of Matlab Routines to Analyse 2D Simulations

In addition to the general routines for reading in and visualizing output from simulations, iDynoMiCS includes a basic set of routines useful for post-processing 2D simulations in particular. These routines are located in the 'iDynoMiCS\results_analysis_matlab\routines_2d' directory. The following routines are available for you to examine simulation outputs.

`showRunInfo(nFile)`: This routine will print out information about the simulation, including the names of bulk compartments, solutes, agent species, and reactions, and will also print a list of the iterates for which outputs were saved. If *nFile* is not given, the last iterate is used.

`plotAgents(nFile)`: This routine will plot all the agents for a particular iterate using the 'agent_State' file, useful for examining the location of each species within the biofilm. Each agent is represented by a circle colored for each species, but note that the sizes of the circles DO NOT correspond to the actual size of the agents. If you want a plot of the agents with the correct agent sizes, use the `makePic` command described above to create a rendered file with POV-Ray.

`plotContour(nFile,fieldName)`: This routine plots a contour for a particular field given in the 'env_State' file, thus may be used to plot solute concentration fields, biomass concentration fields, and fields for reaction rates. The routine also recognizes two additional options for **`fieldName`**: 'allbio' will plot a contour for all biomass in the system, and 'interface' will plot a contour that shows the location of the biofilm/liquid interface.

`plotMix(nFile,fieldName)`: This routine is a mix of the **`plotAgents`** and **`plotContour`** routines. It will plot the field given by **`fieldName`**, and then plot the agents on top of the contour so that you may see how the agents affect the gradients in the particular field.

`plotProfile(nFile,hloc,vtop)`: This routine plots a 1D profile of the solute concentration fields, reaction rates, and biomass concentrations perpendicular to the substratum. *nFile* once again specifies the iterate to plot. *hloc* specifies the horizontal position at which the profile is to be plotted (distance in micrometers from the origin), but *hloc* may also be used to plot instead the horizontal average of each quantity (*hloc* = 'avg') or the average along with standard deviation (*hloc* = 'std'). *vtop* is optional and may be used to specify the vertical limit of the plotted region; setting *vtop* to 'edge' limits the plot to show only the region within the biofilm.

`plotTime(type)`: This routine plots one of several different quantities as a function of time. The possible arguments for *type* are:

- *bulks*: plots the bulk concentration in each bulk compartment
- *bulk_rates*: plots the rate of change of bulk concentrations in each bulk compartment
- *N*: plots the numbers of each species

- *interface*: plots the mean and maximum biofilm thickness
- *simpsonIndex*: plots the diversity index (0 is more homogeneous, 1 is more diverse)
- *simpsonIndex_noeps*: also plots the diversity index, but excludes any species with 'EPS' in the name

For performance reasons, it is useful to manually unzip the 'agent_Sum.zip' and 'env_Sum.zip' files before using this routine.

Note that typing '`help`' and then the command name (eg. '`help plotAgents`') in Matlab will also give you a description for what each argument does.

The above routines are all built using the more base-level routines located in the 'iDynoMiCS\results_analysis_matlab\routines_2d\utilities' directory. Each file has a short description you may access using the '`help`' command in Matlab, and if you have additional analyses you would like to carry out these files might be helpful as you construct your own post-processing routines. Of particular note are three routines useful for extracting data from the simulation output files:

`getAgentData(nFile,type)`: this file is similar to the previous `loadAgents` routine, but using the *type* argument allows one to load the agent information as individuals, on the species level, placed onto a solute-style grid, or placed onto a solute-style grid but separated into active, inert, and EPS components. Type '`help getAgentData`' for a more complete description.

`getEnvData(nFile)`: this is similar to the previous `loadSolute` routine, but returns the information in the solute output files split by type (solute, reaction, biomass, etc.). This routine may also return biofilm height and biomass pressure information. Again, type '`help getEnvData`' for a more complete description.

`getBulkData(nFile)`: this routine loads bulk concentration information that is stored in the 'env_Sum' files, and may also return biofilm height information. Remember: '`help`' is your friend.

If you examine the above 2D plotting scripts you will see that they make extensive use of these `get*Data` routines, and that you may make similar use of the scripts in constructing your own post-processing routines.

VII Converting XML output into plain columns of numbers

iDynoMiCS outputs files in XML form. This is fine for reading data into Matlab, but is difficult to get working with Octave, R, spreadsheets or other software. Thus a program has been written by Chinmay Kanchi in python to convert the XML files into comma-separated values (CSV) text files that can be read by basically any software. Usage of this file is very simple: open a command prompt window (Start → run → "cmd"). Call the `xml2csv.exe` executable, and pass it the *directory* containing the **agent_State.zip** and **env_State.zip** files. It will then create the CSV files in the current directory for you.
There is a second, optional, argument, which is the character to be used for commenting out the text header in the CSV file, which otherwise consists solely of numbers. This allows you to

comment out text headers if these would interfere with reading in the data in whatever software you use. For Matlab or Octave, use a percentage sign (%).If you are going to use R for processing, the comment character is a hash (#). For spreadsheets, you can leave this argument blank. However, if you use R, you will probably not want to comment out the header line as R will automatically assign the columns to the variable names it finds in the first line of the file.

If you are running this on Unix, or you wish to run the python script rather than the executable on Windows, please note that the “beautifulsoup” and “NumPy” packages are required.

One point of note: if a particular species does not have one of the fields (columns) that the other species have, this column will be left blank. This preserves similarity with the XML file, and prevents problems from different software packages having different notations for missing values such as NA (not available) or NaN (not any number).

VIII Suite of R Routines to Analyse 2D Simulations

A number of general routines for visualization and post-processing have been written for R – thanks to Susanne Schmidt and Edd Miles. These scripts make use of several packages that you may need to install first, using the `install.packages('packagename')` syntax. The packages required are: vegan, ReadImages and stringr. The R routines are stored in the 'results_analysis_R' folder.

The R scripts read in the data from the csv files (see above) rather than XML files as the Matlab scripts do, therefore some of the information is harder to obtain and at the moment has to be supplied by hand. First, the simulation time information is lost and will have to be calculated manually. Second, many of the plots use the number of the output file in a sorted list of output files rather than the iteration number coded in the file name. This means manual transformation is needed (e.g. if there were three output files, agent_State(0).csv, agent_State(20).csv and agent_State(40).csv, the data in these for species abundance would be plotted as 1,2,3). Third, the names of the species and bulks are not read in from the CSV files and must be manually converted to and from the identification number.

The first functions read in the data. These are `readStates('path_to_csv_files')` and `readEnv('path_to_env_csv_files'),` which will read in all the agent_State and env_State files in the folder given by path (by default, this is “.”, i.e. the current working directory), and output them as a matrix of lists.
Examples:
```
aStates<- readStates()
eStates<- readEnv('~/Results/2010')
```

plotAgents(agent_state_data). This routine will plot all the agents for a particular iterate using the ‘agent_State’ file, useful for examining the location of each species within the biofilm. Each agent is represented by a circle coloured for each species, but note that the sizes of the circles DO NOT correspond to the actual sizes of the agents. If you want a plot of the agents with the correct agent sizes, use POV-Ray to render the pov file outputted by iDynoMiCS. You can pass this function either the number of the iteration file you want read in (in the current folder), or you can pass it the variable (a list) containing the data you have already read (If you read all the files in using the

readState or readEnv functions, this can be achieved by `plotAgents(Name_of_agent_variable[iteration_no,])`.
Examples:
```
plotAgents(aStates[1,])
plotAgents(0)
```

plotTimeCourseAgents(agent_state_data). This will plot the abundance of each species at each iteration. If you pass this no arguments, it will read in all the agent state CSV files in the current folder. Otherwise, it will work with the list of states you pass to it. It will output not only the graph, but also the species abundances, that can be fed into simpsonIndex() as described below.
Examples:
```
abundances <- plotTimeCourseAgents(aStates)
abundances <- plotTimeCourseAgents()
```

simpsonIndex(abundance) will plot the diversity of the community at each iteration and return the numbers as well.
Examples:
```
diversity <- simpsonIndex(abundances)
```

plotTimeCourseAbund(AgentStates). This routine plots the sum of all the abundances of all species at each iteration. If you pass this no arguments, it will read in all the agent state CSV files in the current folder. Otherwise, it will work with the list of states you pass to it.
Examples:
```
plotTimeCourseAbund(aStates)
plotTimeCourseAbund()
```

plotContour(iterate, solute) This routine draws a contour graph of the solute concentration field at iteration iterate. Iterate can either be the number of the file you want to read in, or a list object already present, solute should be the numeric ID of the solute you want the contours for.
Examples
```
ctours <- plotContour(aStates[1,], 1)
ctours <- plotContour(0,1)
```

overlay(picture, contours) This routine takes an image rendered in POV-Ray and plots the contour data on top. Picture is the path to the image (image must be in jpg format), and contours is the data returned by plotContour().
Example:
```
Overlay('~/it(0).jpg', ctours)
```

IX Future Development

iDynoMiCS is under constant development by several different teams working in the realm of microbial ecology. If you are interested in contributing to the further development of this software, be this through the addition of new species types, new routines for post-processing simulations, or deep improvements to the numerical algorithms, please contact one of the current research partners listed below.

X The iDynoMiCS team (including past members)

- Andreas Dötsch (andreas.doetsch@helmholtz-hzi.de), Chronic Pseudomonas Infections Group, Helmholtz Centre for Infection Research (Germany)
- Jan-Ulrich Kreft (j.kreft@bham.ac.uk), Centre for Systems Biology, School of Biosciences, University of Birmingham (UK)
- Laurent Lardon (lardonl@supagro.inra.fr), DTU Environment, Technical University of Denmark (Denmark) & Laboratory of Environmental Biotechnology, INRA (France)
- Sónia Martins (SCM808@bham.ac.uk), Centre for Systems Biology, School of Biosciences, University of Birmingham (UK)
- Brian Merkey (bvm@northwestern.edu), DTU Environment, Technical University of Denmark (Denmark) & Department of Engineering Sciences and Applied Mathematics, Northwestern University (USA)
- Cristian Picioreanu (C.Picioreanu@tudelft.nl), The Biofilm Research Group, Technical University of Delft (Netherlands)
- Susanne Schmidt (s.schmidt@bham.ac.uk), Centre for Systems Biology, School of Biosciences, University of Birmingham (UK)
- Barth F Smets (bfsm@env.dtu.dk), DTU Environment, Technical University of Denmark (Denmark)
- João Xavier (xavierj@mskcc.org), Computational Biology Research, Memorial Sloan-Kettering Cancer Center, New York (USA)
- Robert Clegg (rjc096@bham.ac.uk), Centre for Systems Biology, School of Biosciences, University of Birmingham (UK)