ContinuumColony for 3DdFBA Simulations: A User's Guide

John A. Cole

March 5, 2015

Contents

1 Introduction

The continuumColony code is a GPU-accelerated program for simulating the growth and metabolism of colonies of cells. Compiling it will require a linux or Mac OSX machine, the CUDA Toolkit 5.5 or later, and running it will require a CUDA-capable GPU (compute capability 2.0 or higher). Included with this user guide should be the source code (continuumColony_15.3.cu), a make file, and some other files intended to be used as inputs for a test simulation.

2 Compilation

In the terminal, navigate to the directory where the source code and everything else is located. Assuming you are using a 64-bit machine with the CUDA toolkit installed in the usual location, simply typing "make" and pressing enter should compile the code using CUDA compute capability

3.0. If it does not, the make file will need to be edited. Open the make file using a text editor and change the paths to the compiler (nvcc), the included headers (-I/usr/local/cuda/include) and libraries (-L/usr/local/cuda/lib64), as needed. Using "make compute_2" will compile using compute capability 2.0, which may be necessary for earlier GPUs.

As the code compiles, it may print a series of warnings—its totally normal, so don't worry. Once compiled, you might want to edit your .bashrc file in order to alias the path of the executable to something you can easily remember.

3 Test Simulation

You can launch a test simulation with the command:

```
user@machine:/workingPath$ ./continuumColony -i test.inp
```

This simulation should run for a few minutes, printing stuff to the terminal console as it goes, then finish leaving a file called test.tar.gz in the same folder that the test.inp file was located.

Untaring the output file with:

```
user@machine:/workingPath$ tar -xzvf test.tar.gz
```

you will see the output of the simulation is actually a directory with several nested subdirectories. On the top level is a copy of the input file, and a file called time. This time file actually contains the final time of the simulation in machine-readable (binary) format (the file's size is 4 bytes—the same size as the single-precision float used in the program to keep track of time). Feel free to look around. In the subdirectory named "cells" you will find additional subdirectories "0", "1", and "totalVolumeFraction". Inside each are files that contain information about the volume fraction and metabolic fluxes of cell type 0, cell type 1, and the total volume fraction of all cell types throughout the simulation volume. These files are named by the time they represent in the simulation. A similar system is used in naming and organizing the different diffusing chemicals in the simulation. This information is organized in the subdirectory "chems" within the top level directory. By looking into "/test/chems/" you might surmise that the test simulation we performed had five different chemical species in it labeled 0 through 4.

In order to get a better idea of what the different cell types are and what the different chemical species are, go ahead and open the file test.inp in a text editor.

At the top you will see a few comments, including my name (John), the date I prepared this file, and a brief description of the file. Below you will see a list of parameters for the test simulation. They are formatted, one parameter to a line, as:

```
parameterName1:<tab><value>
parameterName2:<tab><value>
.
.
.
parameterNameN:<tab><value>
```

where "parameterName" might be maxT (the total time to be simulated), or pullbackT (the times at which the state of the simulation will be "pulled back" from the GPU and written out to the output directory), etc. These "universal" parameters are detailed in Table 1.

Table 1: Universal continuumColony Parameters

Parameter	Description	Type
maxT	length of the simulation	float
pullbackT	time interval at which the simulation state is saved	float
checkpointT	time interval at which the simulation state is tared as a checkpoint	float
chemEqT	equilibration time for the diffusing species	float
growT	time interval equilibrated growth rates are used to project forward	float
initialChemEqT	equilibration time before the simulation starts	float
cellMass	average cell mass (g)	float
cellVolume	average cell volume (m^3)	float
xl	number of lattice sites (in x-direction) of simulation box	int
yl	number of lattice sites (in y-direction) of simulation box	int
zl	number of lattice sites (in z-direction) of simulation box	int
lambda	lattice spacing (m)	float
fbaTau	time step between applications of the uptake and FBA kernels	float
agarHeight	height (in lattice sites) of the agar in the simulation	int
maxVolFrac	maximum volume fraction of cells in a given lattice site	float

After the universal parameters, several other types of parameters appear in this input file. These describe the different cell and substrate types, as well as organize parameters for how the cells take up substrates or can change type in response to their environments. We will cover all this in order.

The line:

```
cellType0:<tab>glucivorous
```

simply tells the program that there is a cell type 0 in the simulation. The word gluciverous is just a comment to ourselves so that we can remember what behavior cell type 0 does—it is gluciverous; it eats glucose. Cell type 0 has its own FBA lookup table, the name of which is specified by the line:

```
cellType0 fbaFilename:<tab>fbaTable_glc104_o2318.dat
```

and which is located in the same directory as the input file. Cell type zero will also be present at the beginning of the simulation, hence:

```
cellType0 starterFlag:<tab>1
```

and because it will be present at the beginning of the simulation, we need to know how many cells there are initially and where they are :

```
cellType0 initX:<tab>96
cellType0 initY:<tab>96
cellType0 initNumber:<tab>1
```

This simulation contains another type of cells—cell type 1, the acetivorous (or acetate-eating) cells. They also have their own FBA lookup table (fbaTable_ac16_o2318.dat) but are not present

yet at the start of the simulation.

Below the cellType definitions, we have several chemType definitions. The first—chemType0—describes extracellular oxygen. It has different diffusion rates in water and in agar (2.6×10^{-9} and 2.47×10^{-9} m^2 s^{-1}, respectively) and can have different starting concentrations in the agar and in the air (although in this case they have the same concentration, 2.6×10^{-4} M, the equilibrium oxygen concentration in water calculated from Henry's law).

There is a flag to tell the program if a chemical species experiences hindered diffusion:

```
chemType1 isHindered:<tab>1
```

which takes values either 0 or 1 (when not specified the default is 0). Although oxygen does not use this flag, the next chemical type 1—extracellular glucose—does. This is used to account for the fact that glucose cannot diffuse through cells but must diffuse around them, giving rise to a crowded environment and a correspondingly slowed diffusion rate.

There is another flag:

```
chemType2 isIntracellular:<tab>1
```

which also takes values either 0 or 1 (when not specified the default is zero), that tells the program if a chemical species is intracellular or extracellular. Those that are intracellular are localized to the interior of cells and cannot diffuse away. Chemical types 2 and 4 (intracellular glucose for cell type 0 and cell type 1, respectively) both use this flag. The index of the cell type that the chemical is localized inside (0 and 1, respectively) is then specified with:

```
chemType2 cellIdxChemLivesIn:<tab>0
```

and

```
chemType4 cellIdxChemLivesIn:<tab>1
```

respectively.

Finally there are flags to tell the program if the floor and walls of the simulation box have reflective (Neumann) boundary conditions for a given species. These flags look like:

```
chemTypeX refFloorBcFlag:<tab>1
chemTypeX refWallBcFlag:<tab>1
```

When not specified (as in the case of our test simulation), the default behavior is fixed-concentration (Dirichlet) boundary conditions.

Following all the chemical type definitions, we have a single Michaelis-Menten-type uptake reaction for glucose. This has parameters for the number of transporters per cell (E), their rate parameters (km and kcat), which cell type actually does this reaction (cellType), which chemical is the reactant (reactantIndex, in this case 1, meaning extracellular glucose), and which chemical is the product (2, meaning intracellular glucose for cell type 0). Together these all look like:

```
rxnType0:<tab>extracellularGlc + E <=>extracellularGlcE --> intracellularGlc + E
rxnType0 km:<tab>3.7033e-4
rxnType0 kcat:<tab>448635
rxnType0 E:<tab>1
rxnType0 cellType:<tab>0
rxnType0 reactantIndex:<tab>1
rxnType0 productIndex:<tab>2
```

After this we find a description of how the cells can change their regulatory state. regType0, for example, describes how glucivorous cells (reactantCellType = 0) can change to acetivorous

cells (productCellType = 1) in response to the local glucose (chemOneIndex = 1) and acetate (chemTwoIndex = 3) concentrations:

```
regType0:<tab>glucivorous -> acetivorous
regType0 chemOneIndex:<tab>1
regType0 chemTwoIndex:<tab>3
regType0 reactantCellType:<tab>0
regType0 productCellType:<tab>1
regType0 multipliers:<tab>0.0<tab>-0.00777<tab>0.00888<tab>0.0<tab>0.025<tab>
                     0.0<tab>0.0<tab>0.0<tab>0.0<tab>0.0
regType0 chemCognate0:<tab>2<tab>4
```

This change occurs at a rate described by:

$$\begin{aligned} k_{i\to j}(\phi_m(\mathbf{x},t),\phi_n(\mathbf{x},t)) = \max(0,&\alpha_0 + \alpha_1\phi_m(\mathbf{x},t) + \alpha_2\phi_n(\mathbf{x},t) \\ &+\alpha_3\phi_m(\mathbf{x},t)^2 + \alpha_4\phi_n(\mathbf{x},t)^2 + \alpha_5\phi_m(\mathbf{x},t)\phi_n(\mathbf{x},t)) \\ &+\alpha_6\phi_m(\mathbf{x},t)^3 + \alpha_7\phi_n(\mathbf{x},t)^3 + \alpha_8\phi_m(\mathbf{x},t)^2\phi_n(\mathbf{x},t)) \\ &+\alpha_9\phi_m(\mathbf{x},t)\phi_n(\mathbf{x},t))^2 \end{aligned} \tag{1}$$

where the parameters $\{\alpha_i\}$ are given to the program via the "multipliers" parameter. Since we are allowing cells of one type turn into cells of another type, we need to make sure we can account for whats inside one cell type to turn into whats inside another cell type. As cells of type 0 become cells of type 1, cell type 0's intracellular glucose (chemType 2) must become cell type 1's intracellular glucose (chemType 4). This effect, it turns out, is not super important for glucose because it is metabolized so quickly, but down the road it might be important for other metabolites, so its a nice feature to have. In the case of our test simulation, this is specified using the "chemCognate" parameter; here chemType 2 becomes chem type 4 under regulation type 0, and is listed as such. Note that under the reverse regulation (regType1) the order of these cognates is reversed.

Our test simulation uses 2 cell types, and accounts for 3 diffusing chemical species and two more intracellular chemical species. The code currently allows for up to 10 cell types, 10 total chemical species, 10 Michaelis-Menten uptake reactions, 10 forms of regulation (each dependent on up to 2 chemical species, and each accounting for up to 10 chemical cognates)

4 FBA Tables

At this time you should have a pretty good idea of what the test simulation was doing, but there are still a few things you need to know before you can do your own simulations. The most important is how to prepare an FBA lookup table. Opening up the the file "fbaTable_glc104_o2318.dat" in a text editor, you will see it is mostly a long list of numbers with a few parameters defined at the top of the file:

```
nInputs:<tab>2
nOutputs:<tab>4
inputIndexes:<tab>2<tab>0
outputIndexes:<tab>2<tab>3<tab>0
numberOfTabledValues:<tab>53<tab>160
inputStride:<tab>0.20<tab>0.20
fbaTable:
```

0.000000<tab>0.000000<tab>0.000000<tab>0.000000
.
.
.

The parameters at the top tell the program how to interpret that long list of numbers. The first line is the number of inputs—which is two, the glucose and oxygen maximum uptake rates. The second is the number of outputs—4, the predicted glucose and oxygen uptake rates (not necessarily the same as their maximum uptake rates) the predicted acetate efflux rate, and the growth rate. After these we have the chemType indexes which allow us to connect the tabled fluxes with the chemicals that appear in "test.inp". For the inputs we see "inputIndexes:<tab>2<tab>0", corresponding to the intracellular glucose of cell type 0 (chemType2), and the extracellular oxygen (chemType0). For the outputs we see "outputIndexes:<tab>2<tab>3<tab>0", corresponding to the intracellular glucose of cell type 0 (chemType2), and the extracellular acetate (chemType3) and the extracellular oxygen (chemType0). Not listed here is the 4th output, the growth rate, which is ALWAYS last. Based on these indexes, the rows of the FBA table contained in this file are ordered—predicted glucose flux, predicted acetate flux, predicted oxygen flux, predicted growth rate.

The next few lines tell how to index into this table. "numberOfTabledValues:<tab>53<tab>160" says the table contains the fluxes predicted for 53 different maximum glucose uptake rates and 160 different maximum oxygen uptake rates, for a total of $53 \times 160 = 8480$ total rows. Finally the inputStride tells how far apart these input uptake rates are—in this case each maximum oxygen or glucose uptake rate is separated by 0.2 mmol gDwt^{-1} hr^{-1}.

Included with this distribution should be a file called "makeFbaTable.m". Its a Matlab function that produces an FBA table like the ones I have given you. It has a number of dependencies, including the COBRA Toolbox (freely available at: http://opencobra.sourceforge.net/openCOBRA/Welcome.html) (Schellenberger *et al* (2011)). It also includes in embedded function called pfbaJAC which is a simplified version of the pFBA function within the COBRA toolbox. This makeFbaTable takes two arguments—a constraint-based COBRA model (in irreversible form—see COBRA documentation), and a struct, dataStruct, with several parameters. These parameters are listed in Table 2:

Table 2: dataStruct Parameters for makeFbaTableGPU.m

Parameter	Description	Type
dataStruct.filename	the name of the output file	string
dataStruct.inputPtclIndexes	the indexes from "test.inp" for the input metabolites	int array
dataStruct.outputPtclIndexes	the indexes from "test.inp" for the input metabolites	int array
dataStruct.maxUpperBounds	the maximum flux upper bounds for the different input metabolites	float array
dataStruct.inputStride	The input stride for the different input metabolites	float array
dataStruct.inputBackwardFluxIndexes	the indexes in the COBRA model for each input metabolite's reverse flux	int array
dataStruct.outputForwardFluxIndexes	the indexes in the COBRA model for each output metabolite's forward flux	int array
dataStruct.outputBackwardFluxIndexes	the indexes in the COBRA model for each output metabolite's reverse flux	int array
dataStruct.grIndex	the index in the COBRA model of the biomass term being used	int

In order to make an abridged version of "fbaTable_glc104_o2318.dat", open a Matlab session, and load the *iJO1366 E. coli* metabolic model (Orth *et al* (2011); Chang *et al* (2013)). Initialize the COBRA Toolbox, then convert the model to irreversible form using:

```
>> initCobraToolbox;
>> irrModel = convertToIrreversible(iJO1366);
```

Now create a struct of parameters:

```
>> dataStruct.filename = 'testFbaTable.dat';
>> dataStruct.inputPtclIndexes = [2 0];
>> dataStruct.outputPtclIndexes = [2 3 0];
>> dataStruct.maxUpperBounds = [11.0 32.0];
>> dataStruct.inputStride = [1.0 1.0];
>> dataStruct.inputBackwardFluxIndexes = [320 496];
>> dataStruct.outputForwardFluxIndexes = [319 63 495];
>> dataStruct.outputBackwardFluxIndexes = [320 64 496];
>> dataStruct.grIndex = 8;
```

and simply run the script:

```
>> makeFbaTable(irrModel, dataStruct);
```

After a few minutes, the script will produce an FBA table file in your working directory called "testFbaTable.dat"

Making an FBA table can be time-consuming, but it can be used again and again to run many different simulations. Its also embarrassingly parallelizable, so if you have access to a cluster, its fairly straightforward to divide up the work then at the end put several small tables together.

5 Command Line Arguments

In the console you can launch the program without any arguments:

```
user@machine:/workingPath$ ./continuumColony
Proper Usage:
continuumColony -i <Input Filename>

optional flags:
-d <device ID> specifies device
-r specifies restart
-Xslice <int> writes trajectory files for x = int slice of lattice
-Yslice <int> writes trajectory files for y = int slice of lattice
-Zslice <int> writes trajectory files for z = int slice of lattice

NOTE: to make output trajectories once a simulation is finished use both -r and -<XYZ>slice
```

You will see that the program does not run, it just prints out its own usage instructions. On systems where multiple GPUs are available, the flag -d <device ID> tells the program to use GPU # <device ID> to do the simulation. The flag -r specifies a restart. This will start from the last frame in an existing <InputFilename>.tar.gz and run from there until maxT. The flags -Xslice <x>, -Yslice <y>, and -Zslice <z> tell the program to write out the a 2-D slice through the simulation volume in a plain text format that can be easily read by Matlab or other plotting software. These -<XYZ>slice commands can be combined with the -r command in order to produce plain text output from a completed simulation. Try the command :

```
user@machine:/workingPath$ ./continuumColony -r -i test.inp -Xslice 96
```

It will produce text files that contain the concentrations, cell volume fractions, and fluxes at the x = 96 plane within the simulation.

The included Matlab function, readContSimData.m, will enable you to easily read the plain-text data you have just produced. In the Matlab console simply type:

```
>> test = readContSimData('<path to working directory>/test_chem2_x96.traj');
```

and press enter. Your Matlab workspace will now include a struct called "test" that has the times and the glucose concentrations in the x = 96 plane at each output frame of the simulation. You can view the t = 120 slice using the commands:

```
>> figure;
>> imagesc(test.slice(:, :, 2));
```

This should be pretty boring; you would have to run the simulation for about 15 to 20 hours before anything particularly exciting happened (if you did you would see several different metabolic behaviors within the simulated colony—some cells out at the edge of the colony would be growing very fast, while others would be engaged in a form of symbiotic acetate crossfeeding (Cole *et al* (2015))! Very cool!).

6 Arbitrary Starting Configurations

There is one final feature that has not been mentioned yet. It is possible to define an arbitrary starting configuration for your simulated colony. The program can read a tab-delimited text file with the x-y-z coordinates and volume fractions for a given cell type and initialize the simulation with that configuration. Included with this distribution is a file called arbitraryDisk.xyzp. You can look at its contents using a text editor. It has a few lines of comments, then a list of locations and non-zero volume fractions. The file test_2.inp is identical to test.inp except it uses this arbitraryDisk.xyzp for a starting configuration instead of a single cell at the center. Open it up and look at it. You will see a only a few differences. The "cellType0 starterFlag" is still 1, but instead of initial x- and y-coordinates and an initial number of cells being supplied, the name of the starting configuration file is given:

```
cellType0 arbitraryVolumeFractionFilename:<tab>arbitraryDisk.xyzp
```

You can launch this simulation is the usual way and have a peek at what comes out afterward using the -<XYZ>slice command line arguments we used before. You should be able to see a disk of cells in the middle of your simulation volume. Some cells on the edge should be growing quickly, while others in the interior should be somewhat anoxic and slip into a fermentative acetate-producing behavior (investigate "test_2_cell0_growthRate_x96.traj" and "test_2_cell0_chem3flux_x96.traj" to see the spatial distribution of growth rates and acetate production, respectively, within this disk-shaped colony).

7 Contact Information

For further questions, feel free to e-mail John Cole at cole15@illinois.edu.

References

Chang RL, Andrews K, Kim D, Li Z, Godzik A, Palsson BØ (2013) Structural systems biology evaluation of metabolic thermotolerance in *Escherichia coli*. *Science* **340**: 1220–1223

Cole JA, Kohler L, Hedhli J, Luthey-Schulten Z (2015) Spatially-Resolved Metabolic Cooperativity Within Dense Bacterial Colonies. *BMC Syst Biol In Press*

Orth JD, Conrad TM, Na J, Lerman JA, Nam H, Feist AM, Palsson BØ (2011) A comprehensive genome-scale reconstruction of *Escherichia coli* metabolism—2011. *Molecular systems biology* **7**

Schellenberger J, Que R, Fleming RM, Thiele I, Orth JD, Feist AM, Zielinski DC, Bordbar A, Lewis NE, Rahmanian S, *et al* (2011) Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox v2.0. *Nat Protoc* **6**: 1290–1307