PhysiCell: an open source physics-based multicellular simulator

# Tutorial

Ahmadreza Ghaffarizadeh Samuel H. Friedman Shannon M. Mumenthaler Paul Macklin*

Revision: July 7, 2016

This tutorial will teach you how to download, install and run a series of biological problems in PhysiCell. Please note that this tutorial will be periodically updated. Users should check PhysiCell.MathCancer.org for the latest version. PhysiCell manuscript in preparation for peer review; please see the the citation information below.

## 1 Citing PhysiCell

If you use PhysiCell in your project, please cite PhysiCell and the version number, such as below:

> *We solved the diffusion equations using PhysiCell (Version 1.0.0) [1]*
>
> [1] A Ghaffarizadeh, SH Friedman, SM Mumenthaler, and P Macklin, PhysiCell: an Open Source Physics-Based Cell Simulator for Multicellular Systems, 2016 (in preparation).
>
> Because PhysiCell extensively uses BioFVM, we suggest you also cite BioFVM as below:
>
> *We implemented and solved the model using PhysiCell (Version 1.0.0) [1], with BioFVM [2] to solve the transport equations.*
>
> [1] A Ghaffarizadeh, SH Friedman, SM Mumenthaler, and P Macklin, PhysiCell: an Open Source Physics-Based Cell Simulator for Multicellular Systems, 2016 (in preparation).
>
> [2] A Ghaffarizadeh, SH Friedman, and P Macklin, BioFVM: an efficient parallelized diffusive transport solver for 3-D biological simulations, Bioinformatics 32(8): 1256-8, 2016. DOI: 10.1093/bioinformatics/btv730

## 2 Preparing to use PhysiCell

### 2.1 Downloading PhysiCell

BioFVM is available at PhysiCell.MathCancer.org and at PhysiCell.sf.net. Because we aim for cross-platform compatibility and simplicity, we designed PhysiCell to minimize external dependencies. As of Version 1.0, PhysiCell doesn't directly need any external library, however, since PhysiCell is built on the top of BioFVM, pugixml is needed for compiling the code (included in the download).

## 2.2 Supported platforms

PhysiCell should successfully compile and run on any C++11 or later compiler that supports OpenMP. We recommend using a 64-bit compiler for best results. We target g++ (Version 4.8.4 or later) on Linux and OSX, and MinGW-W64(gcc version 4.9.0 or later) on Windows for this version (testing and support are planned for the Intel C++ compiler in the future versions).

## 2.3 Including PhysiCell in a project

PhysiCell does not require any form of installation for use in a project. Instead, extract all its cpp and h files in your project directory. All PhysiCell source files begin with the prefix "`PhysiCell_`". If your project uses makefiles, you'll want to include the following lines:

```
CC     := g++       # replace with your compiler
ARCH := core2 # a reasonably safe default for most CPUs since 2007
# ARCH := corei7
# ARCH := corei7-avx # earlier i7
# ARCH := core-avx-i # i7 ivy bridge or newer
# ARCH := core-avx2 # i7 with Haswell or newer
# ARCH := nehalem
# ARCH := westmere
# ARCH := sandybridge
# ARCH := ivybridge
# ARCH := haswell
# ARCH := broadwell
# ARCH := bonnell
# ARCH := silvermont
# ARCH := nocona #64-bit pentium 4 or later

CFLAGS := -march=$(ARCH) -O3 -s -fomit-frame-pointer -mfpmath=both -fopenmp -m64 -std=c++11
# replace CFLAGS as you see necessary, but make sure to use -std=c++11 -fopenmp

BioFVM_OBJECTS := BioFVM_vector.o BioFVM_matlab.o BioFVM_utilities.o BioFVM_mesh.o \
BioFVM_microenvironment.o BioFVM_solvers.o BioFVM_basic_agent.o \
BioFVM_agent_container.o BioFVM_MultiCellDS.o

PhysiCell_OBJECTS := PhysiCell_cell_container.o PhysiCell_cell.o PhysiCell_standard_models.o \
PhysiCell_digital_cell_line.o PhysiCell_utilities.o

pugixml_OBJECTS := pugixml.o

COMPILE_COMMAND := $(CC) $(CFLAGS)

BioFVM_vector.o: BioFVM_vector.cpp
$(COMPILE_COMMAND) -c BioFVM_vector.cpp

BioFVM_agent_container.o: BioFVM_agent_container.cpp
$(COMPILE_COMMAND) -c BioFVM_agent_container.cpp

BioFVM_mesh.o: BioFVM_mesh.cpp
$(COMPILE_COMMAND) -c BioFVM_mesh.cpp
```

```
BioFVM_microenvironment.o: BioFVM_microenvironment.cpp
$(COMPILE_COMMAND) -c BioFVM_microenvironment.cpp

BioFVM_solvers.o: BioFVM_solvers.cpp
$(COMPILE_COMMAND) -c BioFVM_solvers.cpp

BioFVM_utilities.o: BioFVM_utilities.cpp
$(COMPILE_COMMAND) -c BioFVM_utilities.cpp

BioFVM_basic_agent.o: BioFVM_basic_agent.cpp
$(COMPILE_COMMAND) -c BioFVM_basic_agent.cpp

BioFVM_matlab.o: BioFVM_matlab.cpp
$(COMPILE_COMMAND) -c BioFVM_matlab.cpp

BioFVM_MultiCellDS.o: BioFVM_MultiCellDS.cpp
$(COMPILE_COMMAND) -c BioFVM_MultiCellDS.cpp

PhysiCell_digital_cell_line.o: PhysiCell_digital_cell_line.cpp
$(COMPILE_COMMAND) -c PhysiCell_digital_cell_line.cpp

PhysiCell_cell.o: PhysiCell_cell.cpp
$(COMPILE_COMMAND) -c PhysiCell_cell.cpp

PhysiCell_cell_container.o: PhysiCell_cell_container.cpp
$(COMPILE_COMMAND) -c PhysiCell_cell_container.cpp

PhysiCell_standard_models.o: PhysiCell_standard_models.cpp
$(COMPILE_COMMAND) -c PhysiCell_standard_models.cpp

PhysiCell_utilities.o: PhysiCell_utilities.cpp
$(COMPILE_COMMAND) -c PhysiCell_utilities.cpp

pugixml.o: pugixml.cpp
$(COMPILE_COMMAND) -c pugixml.cpp
```

We have listed the common values the can be used for `ARCH`. Please note that we used core2 as the default one, however, you need to consider choosing your CPU architecture settings from the list for a better performance. Your compiler flags will require `-fopenmp` for OpenMP (for parallelization across processor cores) and `-std=c=++11` (to ensure compatibility with C++11 or later, and should include `-m64` (to compile as a 64-bit application with greater memory address space).

More sophisticated IDEs may require additional steps to "import" the PhysiCell source; see your software's user documentation for further details.

# 3 Your first PhysiCell application: a simple multicellular system starting from a single cell

We will now create a basic PhysiCell application that creates a spheroid of cells starting from a single cell placed in a microenvironment simulated by BioFVM. As mentioned earlier, PhysiCell is built on the top of BioFVM,

so if you are not familiar with creating a BioFVM application, please first read the tutorial on BioFVM here biofvm.mathcancer.org.

We create this example using the microenvironment in tutorial3_BioFVM where we used Dirichlet condition for the boundaries of our microenvironment. Now we are going to place a cell at the center of this microenvironment and let it grow and divide. The first step in creating any experiment in PhysiCell is to declare and initialize the cell_container: a data structure that extends the BioFVM_agent_container and is the backbone of PhysiCell. We need to provide the voxel size we need for the cell container. We discussed how we set this value in supplementary material of the PhysiCell paper. Voxel size is set to 30 $\mu$m in this experiment.

```
Cell_Container* cell_container = new Cell_Container;
cell_container->initialize(minX, maxX, minY, maxY, minZ, maxZ, voxel_size );
```

Now we can define the cell we want to place in the microenvironment. However, before creating the cell, we first need to define the cell line that our cell takes its phenotype from. PhysiCell_digital_cell_line.cpp provides some default cell lines that can be used. We will later describe the essential fields you may want to change to define your own digital cell line or a specific phenotype within a cell line. We use the two following lines of code to create a sample cell line for cancer cells:

```
Cell_Line cancer;
set_cancer_cell_line_MCF7( cancer );
```

Each cell line is composed of multiple phenotypes. For this example, we just use the first phenotype of cancer cell line (at index 0). Once we created our cell line, we need to modify it to make it consistent with the number of substrates we have in our microenvironment. Specifically, we need to change the secretion rates, uptake rates, and saturation densities of the cell line based on the substrates in the microenvironment and the type of the cell we are working with. For this example, we have only one substrate (oxygen) that is uptaken by cells. Our cells in this experiment do not secrete any substrate, so the values for secretion rate and saturation density should be zero. We set the oxygen uptake rate for the cells to 10 in this example.

```
cancer.phenotypes[0].secretion_rates.rates.resize( microenvironment.number_of_densities(), 0.0 );
cancer.phenotypes[0].uptake_rates.rates.resize( microenvironment.number_of_densities(), 0.0 ) ;
cancer.phenotypes[0].saturation_densities.densities.resize(microenvironment.number_of_densities()
                                                                                    , 0.0 );
cancer.phenotypes[0].uptake_rates.rates[0] = 10;
```

Now if we call display_information function from our cell line, it provides a summary of the cell line.

```
Cell line: MCF7
phenotype 0: viable
phenotype 1: physioxic
phenotype 2: hypoxic
phenotype 3: necrotic
phenotype 4: hypoxic_glycolytic
phenotype 5: physioxic_glycolytic
phenotype 6: normoxic_glycolytic
```

To create a cell, we use create_cell method from PhysiCell_cell.cpp and then assign the position of the cell.

```
Cell* pCell = create_cell();
pCell->assign_position(500, 500, 500);
```

Cells need to have access to the microenvironment they are located in, so we associate the cell with the microenvironment we defined earlier. We also set the phenotype of the cell based on cell line that we defined earlier.

```
pCell->register_microenvironment(&microenvironment);
pCell->set_phenotype(cancer.phenotypes[0]);
```

Now we need to provide cells with a model that manages their life and death cycle. `Physicell_standard_models` has implemented some known models including basic KI-67, advanced KI-67, live and dead, and total cell models (see the the main PhysiCell paper for more details).

```
pCell->advance_cell_current_phase= ki67_advanced_cycle_model;
```

We also need to specify at what phase in the cycle model the cell is:

```
pCell->phenotype.set_current_phase(PhysiCell_constants::Ki67_negative);
```

Finally, similar to `basic_agents` in BioFVM, we need to call `set_internal_uptake_constants` for the cells.

```
pCell->set_internal_uptake_constants(dt);
```

To run this experiment, we should call the `update_all_cells` function from the `cell_container` within the main loop of simulation. Note that we also called `simulate_cell_sources_and_sinks` in the main loop since our cells are uptaking oxygen.

```
double t;
double t_max= 60 * 24 * 3; // 3 days
while( t < t_max )
{
    microenvironment.simulate_bulk_sources_and_sinks( dt );
    microenvironment.simulate_cell_sources_and_sinks( dt );
    microenvironment.simulate_diffusion_decay( dt );
    ((Cell_Container *)microenvironment.agent_container)->update_all_cells(dt);
    t += dt;
}
microenvironment.write_to_matlab( "final_concentration.mat" );
```

`PhysiCell_utilities` provides some useful methods to log the output of PhysiCell. The main loop can be modified a bit to keep the track of the cells:

```
double t_output_interval = 60.0; // every 1 hour
double t_next_output_time = 0;
int output_index =0;
```

```
BioFVM::RUNTIME_TIC();
BioFVM::TIC();

std::ofstream report_file ("report_spheroid.txt");
report_file<<"simulated time\tnum cells\tnum division\tnum death\twall time"<<std::endl;
while( t < t_max )
{
    if(  fabs( t - t_next_output_time ) < 0.0001 )
    {
        log_output(t, output_index, microenvironment, report_file);
        t_next_output_time += t_output_interval;
    }
    microenvironment.simulate_cell_sources_and_sinks( dt );
    microenvironment.simulate_diffusion_decay( dt );
    ((Cell_Container *)microenvironment.agent_container)->update_all_cells(t,
                                                    cell_cycle_dt, mechanics_dt);
    t += dt;
    output_index++;
}
log_output(t, output_index, microenvironment, report_file);
report_file.close();
```

The above code keeps the record of the simulation time, number of cells removed from the system (death), number of cells that divided, total number of cells, and wall time at each iteration (logged in report.txt). It also records the detailed description of each cell in `output` folder at each output interval.

To compile and run this example, add the following line to the makefile (if it is not already there):

```
physicell_example1: $(pugixml_OBJECTS) $(BioFVM_OBJECTS) $(PhysiCell_OBJECTS) $(UTILITY_OBJECTS)
    <your_file_name> $(COMPILE_COMMAND) -o PhysiCell_example1 $(BioFVM_OBJECTS)
    $(PhysiCell_OBJECTS) $(UTILITY_OBJECTS) $(pugixml_OBJECTS) <your_file_name>
```

Then in the command line (or terminal in Unix based systems) type `make physicell_example1`. This will create an executable file that you can run on your system.

As listed in Table XXXXXXX of supplementary materials, each cell has a set of function pointers that are set to default function pointers. There are some other function pointers that are left to users to set. We used one of these functions in the above experiment: `advance_cell_current_phase`. Use of the function pointers make it easy to disable a behavior that we are not interested in for an experiment. For example, if in the above experiment, we want our cells to not move, we can set `pCell->update_velocity` to `do_nothing`; or as another example, if we do not want our cells to update their volume, we can easily set it by:

```
pCell->update_volume = do_nothing;
```

The source code for this example is provided in `examples/tutorial1_PhysiCell.cpp`.

## 4 Example 2: cells growing inside a duct

Ductal Carcinoma *in situ* (DCIS) is one of the most well-studied types of cancer that starts within ducts of breast. In this example we show how you can create a simulation of DCIS using PhysiCell. Most part of this example is

similar to what we did in previous section. The main difference is that in this example, we need to impose the duct structure. All we need to do is to crete a function that returns the distance $d$ of a cell (passed as a pointer) to the duct walls and assign it to the `distance_to_membrane` function pointer of the cells. We use a level set function (a signed distance function d(x)) to represent the breast duct. In this method, $d < 0$ indicates that the cell is inside the duct, $d = 0$ indicates that the cell is on the duct wall, and $d > 0$ indicated that the cell is located outside the duct (in the stroma), $|d|$ is the distance to the duct wall, and $\nabla d(x)$ points outward from the duct wall, oriented from the closest point on the duct wall to the cell. See **??** for further details.

The following function implements the level set function we discussed for the duct shown in Figure 1. The code first checks if the cell located inside the cap of the duct (within the semi-sphere) or within the cylinder part of the duct. Based on the result of the test, code executes instruction either at PART1 or PART2 (see the code below). PART1 first computes the distance of the cell from the x-axis, sets the displacement vector accordingly, and returns the distance of the cell from the wall as the difference of duct radius and the distance to x-axis. If the cell is inside the cap of the duct, PART2 computes the distance of the cell to the origin, sets the displacement vector accordingly, and returns the distance of the cell from the cap walls as the difference of semi-sphere radius (same as the duct radius) and the distance to the origin.

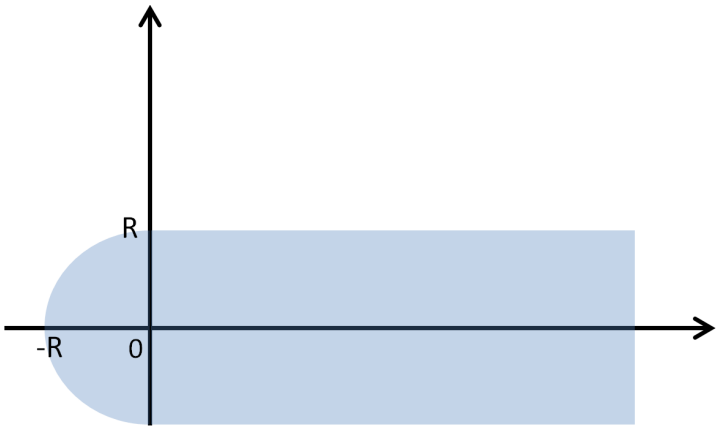


Figure 1: The duct cross-section at z = 0.

**NOTE: any method that is assigned to the `distance_to_membrane` function pointer of the cells must set the displacement vector in addition to returning the distance of the cell from the membrane.**

```
double distance_to_membrane_duct(Cell* pCell)
{
    //Note that this function assumes that duct cap center is located at <0, 0, 0>
    double epsillon= 1e-7;
    if(pCell->position[0]>=0) // Cell is within the cylinder part of the duct
    {
        */ ########################## PART1 ##########################*/
        double distance_to_x_axis= sqrt(pCell->position[1]* pCell->position[1] +
                                        pCell->position[2]*pCell->position[2]);
        distance_to_x_axis = max(distance_to_x_axis, epsillon); // prevents division by zero
        pCell->displacement[0]=0;
        pCell->displacement[1]= -pCell->position[1]/ distance_to_x_axis;
        pCell->displacement[2]= -pCell->position[2]/ distance_to_x_axis;
        return fabs(duct_radius- distance_to_x_axis);
    }

    // Cell is inside the cap of the duct
    */ ########################## PART2 ##########################*/
```

```
    double distance_to_origin= dist(pCell->position, {0.0,0.0,0.0});  // distance to the origin
    distance_to_origin = max(distance_to_origin, epsillon);   // prevents division by zero
    pCell->displacement[0]= -pCell->position[0]/ distance_to_origin;
    pCell->displacement[1]= -pCell->position[1]/ distance_to_origin;
    pCell->displacement[2]= -pCell->position[2]/ distance_to_origin;
    return fabs(duct_radius- distance_to_origin);
}
```

Once the function is implemented, you can assign it to `distance_to_membrane` when you are initializing the cells (see the initiation phase in the previous example).

```
pCell->distance_to_membrane= distance_to_membrane_duct;
```

The rest of the code is similar to the previous example. This example is very similar to the DCIS simulation we have presented in the paper. If you start with 1000 cells and let the simulation run for about 30 days (about 16 hours on a quad-core machine with hyperthreading), and visualize the output, it should generate something similar to Figure **??** in the paper.

The source code for this example is provided in `examples/tutorial2_PhysiCell.cpp`.

# 5 Visualizing outputs

PhysiCell stores its output in two formats: txt and pov. The txt format file at each time step has all the needed information that can be used later for continuing the simulation from that point. For each cell in the txt file, PhysiCell stores cell ID, cell position, cell volume and all subvolumes, cell phase ID, and the elapsed time in the current phase. The pov file is the ready-to-render file that can be rendered using POV-Ray software (download for free from `http://www.povray.org/download/`). Please note that the PhysiCell needs `header.inc` and `footer.inc` for rendering the output pov files. The default files are available in XXXX folder of ant PhysiCell download. `header.inc` keeps the information related to the position of camera, the direction of the camera, sources of light, and the color of each cell phase. You may need to manually change this file to get the desired rendering.

We also have provided a Matlab script file that can read a txt format file and create a cross-section of the system. This script can be found at XXXXXX. Note that this script does not rewrite `header.inc` and `footer.inc` files, so you need to have them from the PhysiCell output.

Please note that the microenvironment can be visualized using BioFVM Matlab functions available in any BioFVM download.