BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations

Tutorial

Ahmadreza Ghaffarizadeh, Samuel H. Friedman, Paul Macklin*

Revision: October 15, 2015

This tutorial will teach you how to download, install and run a series of biological problems in BioFVM. Please note that this tutorial will be periodically updated. Users should check BioFVM.MathCancer.org for the latest version. BioFVM manuscript is currently under review by *Bioinformatics* journal; please see the the citation information below.

1 Citing BioFVM

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

> *We solved the diffusion equations using BioFVM (Version 1.00) [1]*
>
> [1] A. Ghaffarizaeh, S.H. Friedman, and P. Macklin, BioFVM: an efficient parallelized diffusive transport solver for 3-D biological simulations, Bioinformatics, 2015 (submitted).

2 Preparing to use BioFVM

2.1 Downloading BioFVM

BioFVM is available at BioFVM.MathCancer.org and at BioFVM.sf.net. Because we aim for cross-platform compatibility and simplicity, we designed BioFVM to minimize external dependencies. As of Version 1.0, the only external library is pugixml (included in the download).

2.2 Supported platforms

BioFVM 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 BioFVM in a project

BioFVM 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 BioFVM source files begin with the prefix "`BioFVM_`". 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

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

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 BioFVM source; see your software's user documentation for further details.

3 Your first BioFVM application: diffusion + point sources + point sinks

We will now create a basic BioFVM application that solves for diffusion and decay of a substrate (e.g., a signaling factor) secreted by 500 random point sources, and consumed / uptaken by 500 different random point sinks (cells). Throughout our tutorial, we shall add complexity and features to this baseline example to introduce the functions of BioFVM. The source for this tutorial can be found in `./tutorials/first_BioFVM.cpp`

In this example, we use Neumann (no flux) conditions on the computational boundary (see `BioFVM_microenvironment.h` for the list of other possible boundary conditions).

To begin, download the BioFVM package and go to the parent directory. Create a new cpp file (`tutorial1_BioFVM.cpp`) and begin by including the BioFVM library and relevant header files, defining the number of OpenMP threads (usually your number of CPU cores, multiplied by 2 if your CPU supports hyperthreading), and starting the main function:

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <ctime>
#include <omp.h>

#include "BioFVM.h"

int omp_num_threads = 8; // set number of threads for parallel computing
// set this to # of CPU cores x 2 (for hyperthreading)

int main( int argc, char* argv[] )
{
 omp_set_num_threads(omp_num_threads);


 return 0;
}
```

The remainder of the commands are inserted after the "`omp_set_num_threads`" line and prior to the final "`return 0;`".

Next, create a microenvironment, set its units, and set the settings of the default substrate:

```
Microenvironment microenvironment;
microenvironment.name="substrate scale";

microenvironment.set_density(0, "substrate1" , "dimensionless" );
microenvironment.spatial_units = "microns";
microenvironment.mesh.units = "microns";
microenvironment.time_units = "minutes";
```

Note that more densities can be added by `add_density("density_name", "density_unit")`.

Next, we resize the domain to a 1000 micron cube at 20 micron resolution, and display information about our microenvironment. (Note: For best performance, we recommend always setting up the substrates prior to resizing the mesh or domain.)

```
microenvironment.resize_space_uniform(0.0,1000.0, 0.0,1000.0, 0.0,1000.0,  10.0);
microenvironment.display_information( std::cout );
```

Let's compile and run the example. Create a makefile similar to one in the Section 2.3 and then add the following lines to it:

```
tutorial1: $(BioFVM_OBJECTS) $(pugixml_OBJECTS) tutorial1_BioFVM.cpp
$(COMPILE_COMMAND) -o tutorial1 tutorial1_BioFVM.cpp $(BioFVM_OBJECTS) $(pugixml_OBJECTS)
```

Then at the command prompt, type `make tutorial1`, and then run it as `./tutorial1` (`tutorial1.exe` in Windows). The output should look like this:

```
Microenvironment summary: substrate scale:

Mesh information:
type: uniform Cartesian
Domain: [0,1000] microns x [0,1000] microns x [0,1000] microns
   resolution: dx = 10 microns
   voxels: 1000000
   voxel faces: 0
   volume: 1e+009 cubic microns
Densities: (1 1 total)
   substrate1:
     units: dimensionless
     diffusion coefficient: 0 microns^2 / minutes
     decay rate: 0 minutes^-1
     diffusion length microenvironment: 0 microns
```

Now, let's set the initial condition to a Gaussian profile:

```
std::vector<double> center(3);
center[0] = 500; center[1] = 500; center[2] = 500;
double stddev_squared = -100.0 * 100.0;
std::vector<double> one( microenvironment.density_vector(0).size() , 1.0 );
#pragma omp parallel for
for( int i=0; i < microenvironment.number_of_voxels() ; i++ )
{
    std::vector<double> displacement = microenvironment.voxels(i).center - center;
    double distance_squared = norm_squared( displacement );
    double coeff = distance_squared;
    coeff /=  stddev_squared;
    microenvironment.density_vector(i)[0]= exp( coeff );
}
```

`norm_squared` is a built-in function in BioFVM that calculates the squared l^2 norm of a vector. Now if we save the microenvironment variable in a Matlab output format:

```
microenvironment.write_to_matlab( "initial_concentration.mat" );
```

and use Matlab to draw the middle cross-section of microenvironment (see Section 7 for a list of provided Matlab functions), the output would be similar to Figure 1.

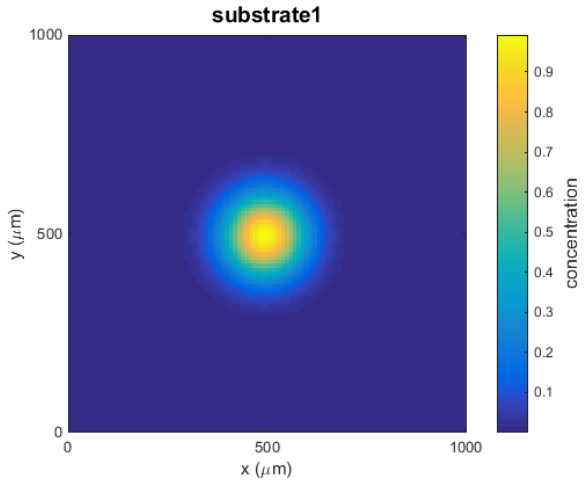


Figure 1: Initial condition Gaussian profile.

Now, let's set the diffusion constant and decay rate for the substrate, and register the diffusion-decay solver with the LOD 3-D solver.

```
microenvironment.diffusion_coefficients[0] = 100000;
microenvironment.decay_rates[0] = 0.1;
microenvironment.diffusion_decay_solver = diffusion_decay_solver__constant_coefficients_LOD_3D;
```

Note that you should use `diffusion_decay_solver__constant_coefficients_LOD_2D` in the case your problem is 2D.

Now if we use the `display_information` again and run the code, the output would display the diffusion coefficient, decay rate, and diffusion length as follows:

```
Mesh information:
type: uniform Cartesian
Domain: [0,1000] microns x [0,1000] microns x [0,1000] microns
   resolution: dx = 10 microns
   voxels: 1000000
   voxel faces: 0
   volume: 1e+009 cubic microns
Densities: (1 1 total)
   substrate1:
     units: dimensionless
     diffusion coefficient: 1000 microns^2 / minutes
     decay rate: 0.01 minutes^-1
     diffusion length microenvironment: 316.228 microns
```

Next, we'll add 500 random point sources (cells secreting `substrate1` at rate 10.0) throughout the domain, using the `basic_agent` class.

```
double dt=0.01;
for(int i=0; i< 500;i++)
{
    std::vector<double> tempPoint(3,0.0);
    for( int j=0; j < 3 ; j++ )
    { tempPoint[j] = UniformRandom()*1000; }

    Basic_Agent * temp_point_source = create_basic_agent();
    temp_point_source->register_microenvironment(&microenvironment);
    temp_point_source->assign_position(tempPoint);
    (*temp_point_source->secretion_rates)[0]=10;
    (*temp_point_source->saturation_densities)[0]=1;
    temp_point_source->set_internal_uptake_constants(dt);
}
```

Note that the method `set_internal_uptake_constants(dt)` should be called when an agent is defined for the first time or whenever that the volume of the agent is updated. Since this method needs to adjust the solver constant with respect to `dt` (the interval for solving solver equations), we define `dt` before defining agents.

Lastly, we'll add 500 random point sinks (cells uptaking `substrate1` at rate 0.8), also using the `basic_agent` class.

```
for(int i=0; i< 500;i++)
{
    for( int j=0; j < 3 ; j++ )
    { tempPoint[j] = UniformRandom()*1000; }
    Basic_Agent * temp_point_sink = create_basic_agent();
    temp_point_sink->register_microenvironment(&microenvironment);
    temp_point_sink->assign_position(tempPoint);
    (*temp_point_sink->uptake_rates)[0]=0.8;
    temp_point_sink->set_internal_uptake_constants(dt);
}
```

Now that the problem definition is completed, we can run the code in the following way:

```
double t;
double t_max=5;
while( t < t_max )
{
    microenvironment.simulate_cell_sources_and_sinks( dt );
    microenvironment.simulate_diffusion_decay( dt );
    t += dt;
}
microenvironment.write_to_matlab( "final_concentration.mat" );
```

If we draw the middle cross-section, the output would be similar to Figure 2.

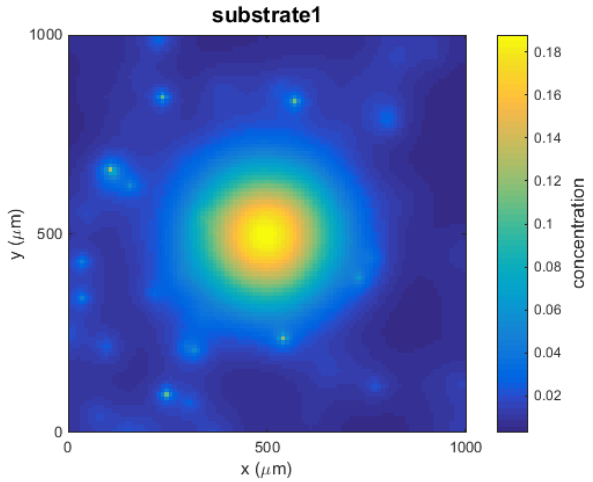


Figure 2: Concentration of substrate1 after 5 minutes of simulation for example 1.

4 Second BioFVM application: diffusion+ bulk sources + bulk uptakes

Many biological problems include bulk sources/uptakes of substrates. This example shows how to model such a problem with BioFVM.

To create a bulk source or a bulk uptake in BioFVM, you need to define functions that provide the density/rate information when called. BioFVM calls this function to adjust the concentration of substrates during simulation. A bulk source needs two functions: a source (supply) function and a saturation function. These functions need to have a signature similar to following:

```
void sample_supply_function( Microenvironment* microenvironment, int voxel_index,
    std::vector<double>* write_here )
void sample_supply_target_function( Microenvironment* microenvironment, int voxel_index,
    std::vector<double>* write_here )
```

Each function receives a pointer to the microenvironment instance, the index of voxel that it should provide the value for, and a pointer to a vector for writing the value in. Having functions like this enables users to model bulk sources that their supply rate and saturation density changes during simulation.

In this example we create a computational domain in which there are bulk sources at the boundary of the domain. We first initialize the domain with a constant value for all the voxels.

```
#pragma omp parallel for
for( int i=0; i < microenvironment.number_of_voxels() ; i++ )
{
    microenvironment.density_vector(i)[0]= 1.0;
}
```

Suppose that the width of the bulk source boundary is 40 μm. The supply function checks the position of each voxel and sets the supply value of the voxel to a predefined value if the distance of the voxel from the edge is less than 40.

```
void tutorial2_supply_function( Microenvironment* microenvironment, int voxel_index ,
    std::vector<double>* write_here )
{
    double domain_upper_bound = 1000;
    double domain_lower_bound =0;
    double strip_width=40;
    (*write_here)[0] = 0;

    if( abs(domain_upper_bound - microenvironment->voxels(voxel_index).center[0]) < strip_width ||
        abs(microenvironment->voxels(voxel_index).center[0] - domain_lower_bound)< strip_width  ||
        abs(domain_upper_bound - microenvironment->voxels(voxel_index).center[1]) < strip_width ||
        abs(microenvironment->voxels(voxel_index).center[1] - domain_lower_bound)< strip_width  ||
        abs(domain_upper_bound - microenvironment->voxels(voxel_index).center[2]) < strip_width ||
        abs(microenvironment->voxels(voxel_index).center[2] - domain_lower_bound)< strip_width )
        {
            (*write_here)[0] = substrate1_supply_rate;
        }

    return;
}
```

We assume that saturation density of `substrate1` is constant all over the domain, so we set a fixed value for the saturation density of each voxel regardless of its position.

```
void tutorial2_supply_saturation_function( Microenvironment* microenvironment, int voxel_index,
    std::vector<double>* write_here )
{
    (*write_here)[0] = substrate1_saturation_density;
    return ;
}
```

Similar to bulk sources, the bulk sinks also need a rate function. In this example, we assume that a there is a tumor spheroid with the radius of 200 μm located at the center of computational domain ($< 500, 500, 500 >$) and each sink voxel uptakes the substrate with a predefined rate.

```
void tutorial2_uptake_function( Microenvironment* microenvironment, int voxel_index,
    std::vector<double>* write_here )
{
    double spheroid_radius=100;
```

```
    std::vector<double> center(3);
    center[0] = (microenvironment->mesh.bounding_box[0]+microenvironment->mesh.bounding_box[3])/2;
    center[1] = (microenvironment->mesh.bounding_box[1]+microenvironment->mesh.bounding_box[4])/2;
    center[2] = (microenvironment->mesh.bounding_box[2]+microenvironment->mesh.bounding_box[5])/2;
    if(sqrt( norm_squared(microenvironment->voxels(voxel_index).center - center))<spheroid_radius)
     (*write_here)[0] = substrate1_uptake_rate;
    return ;
}
```

Once we defined our functions, we should point the microenvironment's function pointers to these functions.

```
microenvironment.bulk_supply_rate_function =  tutorial2_supply_function;
microenvironment.bulk_supply_target_densities_function = tutorial2_supply_saturation_function;
microenvironment.bulk_uptake_rate_function = tutorial2_uptake_function;
```

Now we can run the simulation similar to the previous example:

```
while( t < t_max )
{
    microenvironment.simulate_bulk_sources_and_sinks( dt );
    microenvironment.simulate_diffusion_decay( dt );
    t += dt;
}
```

Note that `simulate_cell_sources_and_sinks` is replaced with `simulate_bulk_sources_and_sinks` in this example. The middle cross section of the domain after 5 minutes of simulation is visualized in Figure 3.

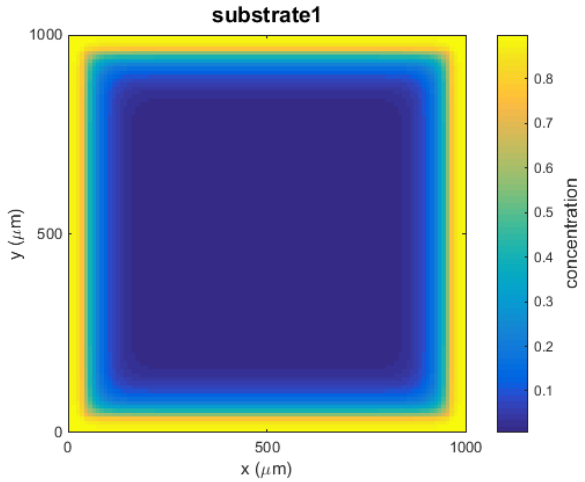


Figure 3: Concentration of substrate1 after 5 minutes of simulation for example 2.

The source code for this example is provided in `examples/tutorial2_BioFVM.cpp`.

5 Third BioFVM application: diffusion+ Dirichlet boundary conditions + bulk uptake

To model some biological phenomena, we need a more complex boundary condition like Dirichlet condition. To address this need, BioFVM has built-in methods for defining this type of boundary condition.

This example reimplements the previous example with only one change: instead of having bulk sources at the boundaries, we use Dirichlet conditions. The following lines of code set the boundary condition at the boundaries.

```
std::vector<double> dirichlet_one( 1 , 1.0 );
double min_x=0, max_x=1000;
double min_y=0, max_y=1000;
double min_z=0, max_z=1000;
double strip_width=40;

for( int i=0; i < microenvironment.number_of_voxels() ; i++ )
{
    if( abs(max_x-microenvironment.voxels(i).center[0]) < strip_width ||
         abs(microenvironment.voxels(i).center[0]- min_x)< strip_width||
         abs(max_y-microenvironment.voxels(i).center[1]) < strip_width||
         abs(microenvironment.voxels(i).center[1]- min_y)< strip_width||
         abs(max_z-microenvironment.voxels(i).center[2]) < strip_width||
         abs(microenvironment.voxels(i).center[2]- min_z)< strip_width )
        {
            microenvironment.add_dirichlet_node( i , dirichlet_zero );
        }
    }
}
```

Since each Dirichlet node should have a value for each of the substrates in the simulation, we need to provide a vector with the size equal to the number of substrates for `add_dirichlet_node` Each vector element specifies to the value the Dirichlet node will take when computing the corresponding substrate. The middle cross section of output for this example is shown in Figure 4. The source code for this example can be found in `examples/tutorial3_BioFVM.cpp`.

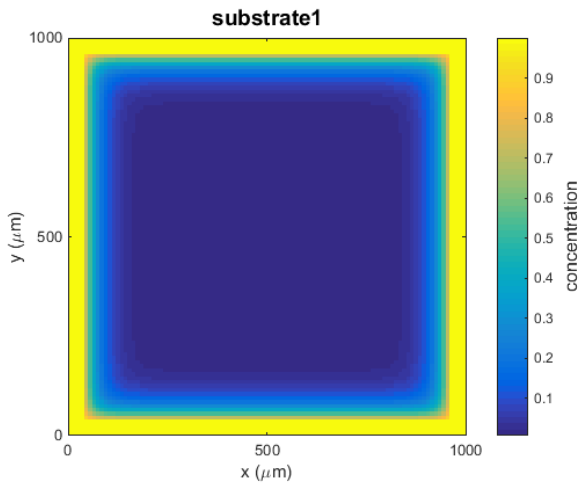


Figure 4: Concentration of substrate1 after 5 minutes of simulation for example 3.

6 Convergence testing and scaling analysis

The source code for all the convergence testing and scaling analysis problems of Appendix 1 is provided in `examples` directory. Table 1 summarizes all the information needed to find, compile, and run these examples.

Test name	File name	Command line parameters	Sample command
Example 1: 1-D diffusion	convergence_test1.cpp convergence_test1_analyitical.cpp	dt (sec), t_max(min), dx (μm) t_max(min), dx (μm)	conv_test1.exe 0.01 2 20 conv_test1_analyitcal.exe 2 20
Example 2: 3-D diffusion-reaction with bulk sources	convergence_test2.cpp	dt (sec), t_max(min), dx (μm)	conv_test2.exe 0.01 2 20
Example 3: 3-D diffusion-reaction with bulk sources, grid-aligned cell uptake	convergence_test3.cpp	dt (sec), t_max(min), dx (μm)	conv_test3.exe 0.01 2 20
Example 4: 3-D diffusion-reaction with bulk sources, off-lattice cell uptake	convergence_test4_1.cpp convergence_test4_2.cpp	dt (sec), t_max(min), dx (μm) dt (sec), t_max(min), dx (μm)	conv_test4_1.exe 0.01 2 20 conv_test4_2.exe 0.01 2 20
Example 5: 3-D diffusion-reaction with bulk sources, off-lattice cell uptake, and point sources	convergence_test5.cpp	dt (sec), t_max(min), dx (μm)	conv_test5.exe 0.01 2 20
Performance scaling with number of substrates	performance_test_substrates.cpp	number of substrates	perf_test_substrates.exe 25
Performance scaling with number of voxels	performance_test_voxels.cpp	half-width of a side of a cubic domain (μm)	perf_test_voxels.exe 400
Performance scaling with number of cells (uptake/source terms)	performance_test_numcells.cpp	number of cells	perf_test_numcells.exe 250000

Table 1: List of corresponding files, command line parameters, and sample commands for each problem used in Appendix.

Please note that `file_name.exe` commands in the last column of Table 1 should be replaced with `./filename` in Linux/Unix.

7 Matlab functions for postprocessing the outputs

BioFVM is designed to work well with Matlab input/outputs. To read data from a `mat` file you can simply use `read_matlab(filname)` :

```
std::vector< std::vector<double> > cells_position = read_matlab( "cells.mat" );
```

For now, BioFVM just supports "Level 4 MAT-File Format". To make sure that your data is readable by BioFVM, you need to save your data in Matlab as `save -v4 filename data`. A sample mat file is provided in `examples` directory (test_4_cells.mat). Please see the example `convergence_test4_2` for an example of how this file is used to store the position of cells.

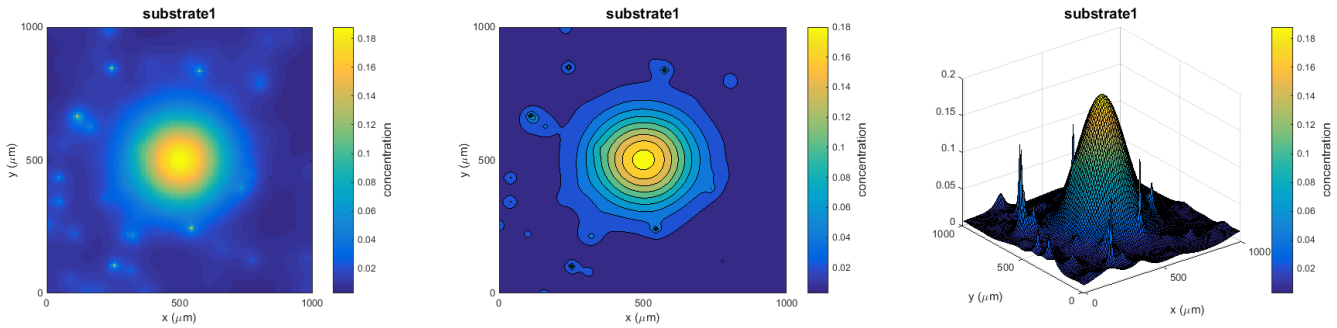
As you have already seen in earlier example, BioFVM has a built-in function to write meshes to a file (as a 3-D matrix):

```
microenvironment.write_to_matlab( "filename.mat" );
```

To ease working with output files of BioFVM, three script files are provided in `matlab` directory to visualize the output: `cross_section_colormap, cross_section_contour` and `cross_section_surface`. These files get the name of a mat file as the input and draw the middle cross-section of the domain as a colormap, contour, or surface. Users can choose the needed planes for the cross section (please see the source files). Sample outputs of these codes are shown in Figure 5.

8 Explicit solver

In addition to the implicit 3-D LOD solver, BioFVM also has an explicit solver for the cases that a user needs a much slower but more accurate solution. To use this solver, you need to register it with the microenvironment similar to the main solver:



(a) (b) (c)

Figure 5: The visualization of middle cross section of the computational domain in Example 3 using provided Matlab scripts. (a) Colormap using cross_section_colormap, (b) contour using cross_section_contour, and (c) surface using cross_section_surface.

```
microenvironment.diffusion_decay_solver = diffusion_decay_explicit_uniform_rates
```

Note that the explicit solver needs a sufficiently small `dt` to be stable.