MMCSS tutorial(jp)

Takanori Saiki

last update : 2025 年 2 月 2 日

1 Introduction

MMCSS (Multicellular Molecular Communication System Simulator) is a general-purpose simulator for simulating multicellular molecular communication systems. Since the basic program for the simulation is provided, users can easily implement their own simulation models by changing parameters or extending their own simulation models.

2 Preparation

This section describes the procedure for installing MMCSS. The environment used is assumed to be Mac OS or Linux, However, it is also possible to run MMCSS on Windows with a few tools.

2-1 Download

Download the software from the MMCSS page on GitHub (`https://github.com/saikiRA1011/CellNetwor`
The download location can be anywhere, but be sure to place it on a drive with plenty of space because image and video files will be created after the simulation is run.

2-2 Setting

MMCSS uses several external tools to create and run simulations. Below is an example of some of these tools and versions.

- gcc 11.3.0 (Versions of C++11 available)
- Python 3.9.12
- pip 22.1.2
- make 3.81
- yaml-cpp 0.8

The Makefile uses gcc-14, but you should change it to match the version of gcc you have installed. Also, since OpenMP is used, other compilers should be used with caution.

Libraries used in Python can be installed at once by the following command.

```
1  pip install -r requirements.txt
```

3 Execution

This section describes how to run the simulation; MMCSS allows you to use make to build, run the simulation, and generate video files.

To build the simulator, simply execute `make`. SimMain will be generated as an executable file. Also, when running the simulation, execute `make run`. The simulation results will be output as a text file in the "result" directory for each step. The simulation results are output as a text file in the "result" directory, step by step. Next, run `make convert` to

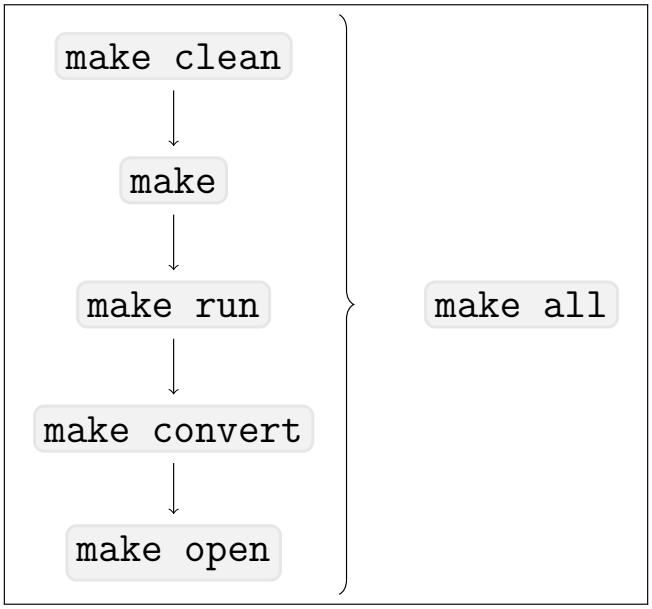

図 1　Sequence of steps with the make command

generate a video file from the simulation results. The final product out.mp4 will be output to the video directory, and the intermediate product images for each step will be output to the image directory. Finally, When you want to check the generated video, execute `make open`.

In addition, these series of processes can be executed together with `make all`. Figure reffig:appMake shows the sequence of steps.

4　Self-creation of simulations

Users can create their own simulations by writing their own programs (or by changing parameters). If the user wishes to return the contents of the file to the default, he/she can initialize it by executing `make reset`, which is described in 5-2. This section describes the knowledge required to create your own simulation for each file that the user edits.

4-1　config.yml, SimulationSettings.hpp(.cpp)

In config.yml and SimulationSettings.hpp(.cpp), define the parameters needed for the simulation. For example, the number of simulation steps, the number of Cells, and the unit time per step of the simulation. The description of each parameter is written as a comment in the file concerned, so please refer to them.

You can also add more parameters if necessary. Add the simulation parameters you feel necessary to SimulationSettings.hpp, and add the code for reading from the yaml file to SimulationSettings.cpp in the try block. These parameters are defined as static variables, so be sure to initialize them at the end of SimulationSettings.cpp.

If you want to set parameters, write the key and value in config.yaml. If you have defined

your own parameters, do not forget to add them to this file as well.

*The parameters provided by default cannot be deleted.

4-2 UserSimulation.hpp(.cpp)

In UserSimulation.hpp(.cpp), you can define the model and initial conditions of the simulation (distribution of Cells, etc.). Currently, the only functions that can be defined are Cell initialization (initCells), step pre-process (stepPreprocess), a mechanistic model of cell-cell interactions (calcCellCellForce), step post-process (stepEndProcess), and the calculation of cell-cell interactions (calcCellCellForce) only.

The cell types and molecule types used in the simulation are also defined here.

Like SimulationSettings.hpp, this file also allows users to define their own variables and functions. Please add or modify functions according to the model you wish to implement.

4-3 UserCell.hpp(.cpp)

In UserCell.hpp(.cpp), you can define the behavior of cells.

For example, you can define when cells die, when they divide, and how their metabolism changes their energy and molecular concentrations.

The most difficult thing for users to implement is how to manage the list of existing cells. Since MMCSS implements this tedious management in Cell.hpp(.cpp), the user only needs to implement when and under what circumstances the cells increase or decrease. situation can be reproduced.

4-4 UserMoleculeSpace.hpp(.cpp)

In UserMoleculeSpace.hpp(.cpp), you can define the system of molecules you want to treat in the simulation.

The parent class, MoleculeSpace, implements several boundary conditions (Neumann, Dirichlet, and periodic boundary conditions), so no additional processing is required by the user when using these boundary conditions.

MoleculeSpace is also implemented with references to each other so that it can interact with Cell. This feature is inherited by UserMoleculeSpace and UserCell, so users do not need to implement cumbersome class references even when simulating complex interactions between cells and molecules.

5 Appendix A

5-1 File of simulation results

When the simulation is run, several text files with the file name cells_step number will be output in the result directory. These files are the output of the simulation results.

The contents of the text files are separated by tab characters and contain the simulation results in the following format.

Listing 1 format of the result file

```
<Header Line>
ID typeID X Y Z Vx Vy Vz R N_contact Contact_ID_1 Contact_ID_2 ...
```

Each of the data is described below:

- ID : Identifier of Cell. A unique value is assigned.
- typeID : Identifier of the type of Cell. Different behaviors can be defined for each type.
- X : Position of X coordinate of Cell
- Y : Position of Y coordinate of Cell
- Z : Position of Z coordinate of Cell
- Vx : Velocity of Cell in X direction
- Vy : Velocity of Cell in Y direction
- Vz : Velocity of Cell in Z direction
- R : Radius of Cell
- N_contact : Number of Cells in contact
- Contact_ID : ID of the contacting Cell

5-2 make commands

MMCSS provides several commands in addition to the make command, which is involved in running the simulation and checking the results. The following is an explanation of these commands:

- `make` : Build a program.
- `make run` : Run a simulation.
- `make convert` : Convert simulation result to movie file.
- `make open` : Check the simulation result movie file.
- `make clean` : Delete all object files.
- `make all` : Delete all object files, then build, run simulation, generate movie and check it all.
- `make reset` : Reset the simulation user configuration files to default.

- `make data-cleanup` : Delete all simulation products (text, images, videos).
- `make data-archive` : Zip archive the currently stored simulation results.
- `make archive-restore date=YYYYYMMDD_HHmm` : Restore the archive contents by specifying the date and time of the archive file.
- `make archive-cleanup` : Delete all archive files.
- `make help` : Display usage and explanation of the commands registered in the Makefile.

6 Appendix B : Samples

6-0.1 The Cluster Formation Model

Listing 2 calcCellCellForce()

```
Vec3 UserSimulation::calcCellCellForce(std::shared_ptr<UserCell> c) const
    noexcept
{
    auto aroundCells = cellList.aroundCellList(c);
    Vec3 force = Vec3::zero();

    for (auto i : aroundCells) {
        if (cells[i]->getCellType() == CellType::WORKER) {
            force += Simulation::calcRemoteForce(c, cells[i]);
        }
    }
    force = force.normalize();

    for (auto i : aroundCells) {
        if (cells[i]->getCellType() != CellType::NONE) {
            force += Simulation::calcVolumeExclusion(c, cells[i]);
        }
    }

    return force.timesScalar(SimulationSettings::DELTA_TIME);
}
```

6-1 The Network Formation Model

Listing 3 UserSimulation.hpp

```
  // add member variables
  std::vector<std::vector<bool>> bondMatrix;
  const double dMax = 100.0;
  const double dMin = 40.0;
```

```
5  const double dEx = 20.0;
6  const double lambda = 20.0;
```

Listing 4 UserSimulation()

```
1  UserSimulation::UserSimulation(/* args */)
2  {
3      bondMatrix.resize(SimulationSettings::CELL_NUM, std::vector<bool>(
           SimulationSettings::CELL_NUM, false));
4  }
```

Listing 5 stepEndProcess()

```
1  void UserSimulation::stepEndProcess() noexcept
2  {
3      for (int i = 0; i < cells.size(); i++) {
4          cells[i]->clearAdhereCells();
5          for (int j = 0; j < cells.size(); j++) {
6              if (i == j) {
7                  continue;
8              }
9
10             const Vec3 diff = cells[i]->getPosition() - cells[j]->
                   getPosition();
11             const double dist = diff.length();
12             if (dist <= dMin) {
13                 bondMatrix[i][j] = true;
14             }
15
16             if (bondMatrix[i][j] && dist >= dMax) {
17                 bondMatrix[i][j] = false;
18             }
19
20             if (bondMatrix[i][j]) {
21                 cells[i]->adhere(*cells[j]);
22             }
23         }
24     }
25 }
```

Listing 6 calcCellCellForce()

```
1  Vec3 UserSimulation::calcCellCellForce(std::shared_ptr<UserCell> c) const
       noexcept
```

```
{
    Vec3 force = Vec3::zero();

    for (auto cell : cells) {
        if (c->id == cell->id)
            continue;

        const Vec3 diff = c->getPosition() - cell->getPosition();
        const double dist = diff.length();

        force += -diff.normalize().timesScalar(std::exp(-dist / lambda)).
            timesScalar(1.0);

        if (bondMatrix[c->id][cell->id]) {
            force += -diff.normalize().timesScalar(std::max((dist - dMin) /
                 (dMax - dMin), 0.0)).timesScalar(30.0);
        }

        force += diff.normalize().timesScalar(std::max((dEx - dist) / dEx,
            0.0)).timesScalar(30.0);
    }

    force = force.timesScalar(SimulationSettings::DELTA_TIME);

    return force;
}
```

6-1.1 The Cell Cluster Growth Model

Listing 7 UserSimulation.hpp

```
// add member variables
private:
std::vector<std::vector<bool>> bondMatrix;
  const double dMax = 150.0;
  const double dMin = 80.0;
  const double dCont = 20.0;
  const double lambda = 20.0;
```

Listing 8 UserSimulation()

```
UserSimulation::UserSimulation(/* args */)
{
```

```
3     bondMatrix.resize(SimulationSettings::CELL_NUM, std::vector<bool>(
          SimulationSettings::CELL_NUM, false));
4   }
```

Listing 9 initCells()

```
1   void UserSimulation::initCells() noexcept
2   {
3     std::uniform_real_distribution<double> rand_r(0, 150);
4     std::uniform_real_distribution<double> rand_theta(0, 2 * M_PI);
5     for (int i = 0; i < SimulationSettings::CELL_NUM; i++) {
6         double r = rand_r(rand_gen);
7         double theta = rand_theta(rand_gen);
8
9         double x = r * std::cos(theta);
10        double y = r * std::sin(theta);
11
12        UserCell c(CellType::WORKER, x, y, 10);
13        cells.push_back(std::make_shared<UserCell>(c));
14    }
15  }
```

Listing 10 stepPreprocess()

```
1   void UserSimulation::stepPreprocess() noexcept
2   {
3       int32_t preCellCount = cells.size();
4
5       for (int i = 0; i < preCellCount; i++) {
6           cells[i]->initForce();
7       }
8
9       for (int i = 0; i < preCellCount; i++) {
10          cells[i]->clearAdhereCells();
11          for (int j = 0; j < preCellCount; j++) {
12              if (i == j) {
13                  continue;
14              }
15
16              const Vec3 diff = cells[i]->getPosition() - cells[j]->
                    getPosition();
17              const double dist = diff.length();
18              if (dist <= dCont) {
19                  bondMatrix[i][j] = true;
```

```
            }

            if (bondMatrix[i][j] && dist >= dMax) {
                bondMatrix[i][j] = false;
            }

            if (bondMatrix[i][j]) {
                cells[i]->adhere(*cells[j]);
            }
        }
    }
}
```

Listing 11 calcCellCellForce()

```
Vec3 UserSimulation::calcCellCellForce(std::shared_ptr<UserCell> c) const
    noexcept
{
    Vec3 force = Vec3::zero();

    for (auto cell : cells) {
        if (c->id == cell->id)
            continue;

        const Vec3 diff = c->getPosition() - cell->getPosition();
        const double dist = diff.length();

        if (bondMatrix[c->id][cell->id]) {
            if (c->adhereCells.size() <= 3) {
                force += diff.normalize().timesScalar((dMax - dist) / dMax).
                    timesScalar(2.0);
            } else {
                force += -diff.normalize().timesScalar(std::max(0.0, (dist
                    - dMin) / (dMax - dMin))).timesScalar(2.0);
            }
        }

        force += -diff.normalize().timesScalar(std::exp(-dist / lambda)).
            timesScalar(0.05);

        if (dist < dCont) {
            force += diff.normalize().timesScalar((dCont - dist) / dCont).
                timesScalar(10.0);
```

```
24          }
25      }
26
27      force = force.timesScalar(SimulationSettings::DELTA_TIME);
28
29      return force;
30  }
```

6-1.2 The Cell Cluster Rotation Model

Listing 12 calcCellCellForce()

```
1  Vec3 UserSimulation::calcCellCellForce(std::shared_ptr<UserCell> c) const
       noexcept
2  {
3      auto aroundCells = cellList.aroundCellList(c);
4      Vec3 force = Vec3::zero();
5      const Vec3 center = Vec3(0, 0, 0);
6      const Vec3 diff_from_center = c->getPosition() - center;
7      Vec3 force_cont = Vec3::zero();
8
9      constexpr double COEFFICIENT = 1.0;
10     constexpr double REPUlSION_C = 0.20;
11     constexpr double REPULSION_LEN = 15;
12     constexpr double BONDING_LEN = 5;
13
14     force += -diff_from_center.normalize().timesScalar(COEFFICIENT);
15
16     for (auto i : aroundCells) {
17         auto cell = cells[i];
18
19         if (c->id == cell->id)
20             continue;
21
22         const Vec3 diff = c->getPosition() - cell->getPosition();
23         const double dist = diff.length();
24
25         if (dist < REPULSION_LEN) {
26             force += diff.timesScalar(1.0 / dist).timesScalar(REPUlSION_C).
                   timesScalar((REPULSION_LEN - dist) / REPULSION_LEN);
27         }
28
29         const Vec3 v = cell->getVelocity();
```

```
30          if (dist < BONDING_LEN) {
31              force_cont += v;
32          }
33      }
34
35      force += force_cont.normalize().timesScalar(0.2);
36      force = force.timesScalar(SimulationSettings::DELTA_TIME);
37
38      return force;
39  }
```

6-2 The Signal Molecular Diffusion Model

Listing 13 stepEndProcess()

```
1  void UserSimulation::stepEndProcess() noexcept
2  {
3      for (int32_t i = 0; i < (int32_t)cells.size(); i++) {
4          if (cells[i]->getCellType() == CellType::DEAD || cells[i]->
               getCellType() == CellType::NONE) {
5              continue;
6          }
7
8          Vec3 pos = cells[i]->getPosition();
9          cells[i]->updateState(moleculeSpaces[0]->getMoleculeNum(pos));
10     }
11 }
```

Listing 14 calcCellCellForce()

```
1  Vec3 UserSimulation::calcCellCellForce(std::shared_ptr<UserCell> c) const
       noexcept
2  {
3      return Vec3::zero();
4  }
```

Add the following to member variables and member functions of class UserCell:

Listing 15 class UserCell(UserCell.hpp)

```
1      private:
2      double cAMP = 0;
3      double activeReceptor = 0.0;
4      double diffCamp = 0;
5      double diffActiveReceptor = 0;
```

```
6       static constexpr double C = 10;
7       static constexpr double K1 = 0.036;
8       static constexpr double K2 = 0.666;
9       static constexpr double L1 = 10;
10      static constexpr double L2 = 0.005;
11      static constexpr double Q = 4000;
12      static constexpr double SIGMA = 6.0;
13      static constexpr double Ki = 1.7;
14      static constexpr double Ke = 5.4;
15      static constexpr double Kt = 0.9;
16      static constexpr double ALPHA = 1;
17      static constexpr double H = 5;
18      static constexpr double THETA = 0.01;
19      static constexpr double LAMBDA = 0.01;
20      static constexpr double EPSILON = 1;
21
22      public:
23      double emitMolecule(int32_t moleculeId) noexcept override;
24      double calcSynthesis(double extracellularCAMP) const noexcept;
25      double f1(double cAMP) const noexcept;
26      double f2(double cAMP) const noexcept;
27      double calcInactiveReceptor(double extracellularCAMP) const noexcept;
28      double calcActiveReceptor(double extracellularCAMP) const noexcept;
29
30      void setDiffState(double extracellularCAMP) noexcept;
31      void updateState(double extracellularCAMP) noexcept;
```

The added member function definitions for class UserCell are shown below:

Listing 16 class UserCell(UserCell.cpp)

```
1  double UserCell::emitMolecule(int32_t moleculeId) noexcept
2  {
3      return cAMP * Kt / H;
4  }
5
6  double UserCell::calcSynthesis(double extracellularCAMP) const noexcept
7  {
8      const double y = (activeReceptor * extracellularCAMP) / (1.0 +
           extracellularCAMP);
9      const double num = ALPHA * (LAMBDA * THETA + EPSILON * y * y);
10     const double den = 1 + ALPHA * THETA + EPSILON * y * y * (1 + ALPHA);
11     const double phi = num / den;
12
```

```
13      return Q * SIGMA * phi / (Ki + Kt);
14  }
15
16  double UserCell::f1(double cAMP) const noexcept
17  {
18      return (K1 + K2 * cAMP) / (1 + cAMP);
19  }
20
21  double UserCell::f2(double cAMP) const noexcept
22  {
23      return (K1 * L1 + K2 * L2 * C * cAMP) / (1 + C * cAMP);
24  }
25
26  void UserCell::setDiffState(double extracellularCAMP) noexcept
27  {
28      diffCamp = calcSynthesis(extracellularCAMP);
29      diffActiveReceptor = (-activeReceptor * f1(extracellularCAMP) + (1.0 -
            activeReceptor) * f2(extracellularCAMP));
30  }
31
32  void UserCell::updateState(double extracellularCAMP) noexcept
33  {
34      cAMP = diffCamp;
35      activeReceptor += diffActiveReceptor * SimulationSettings::DELTA_TIME;
36      activeReceptor = std::max(0.0, activeReceptor);
37  }
```

Listing 17 class UserMoleculeSpace(UserMoleculeSpace.hpp)

```
1  private:
2      const double _D = 0.024 * 1000000.0;
```

Listing 18 calcConcentrationDiff()

```
1  void UserMoleculeSpace::calcConcentrationDiff() noexcept
2  {
3      constexpr double hydrolysisCoefficient = 5.4;
4
5      for (u_int32_t x = 1; x <= width; x++) {
6          for (u_int32_t y = 1; y <= height; y++) {
7              for (u_int32_t z = 1; z <= depth; z++) {
8                  deltaMoleculeSpace[x][y][z] = diffuse(x, y, z);
9                  deltaMoleculeSpace[x][y][z] += -hydrolysisCoefficient *
                       moleculeSpace[x][y][z];
```

```
            }
        }
    }

    const int32_t width = SimulationSettings::FIELD_X_LEN;
    const int32_t height = SimulationSettings::FIELD_Y_LEN;
    const int32_t depth = SimulationSettings::FIELD_Z_LEN;

    for (u_int32_t i = 0; i < cells.size(); i++) {
        std::shared_ptr<UserCell>& cell = cells[i];

        int32_t x = (int32_t)((cell->getPosition().x + width / 2)) + 1;
        int32_t y = (int32_t)((cell->getPosition().y + height / 2)) + 1;
        int32_t z = (int32_t)((cell->getPosition().z + depth / 2)) + 1;

        deltaMoleculeSpace[x][y][z] += cell->emitMolecule(ID);
    }

    for (u_int32_t i = 0; i < cells.size(); i++) {
        int32_t x = (int32_t)((cells[i]->getPosition().x + width / 2)) + 1;
        int32_t y = (int32_t)((cells[i]->getPosition().y + height / 2)) + 1;
        int32_t z = (int32_t)((cells[i]->getPosition().z + depth / 2)) + 1;

        cells[i]->setDiffState(moleculeSpace[x][y][z]);
    }
}
```

Listing 19 config.yaml

```
cell:
  cell_seed: 10 # Seed value used to set cell location, etc.
  cell_num: 500000 # initial number of cells. It becomes very heavy when the
      number of cells is increased. Find out why.
  position_update_method: EULER # select from AB4, AB3, AB2, EULER

simulation:
  simulation_step: 100000 # Select from AB4, AB3, AB2, EULER
  sim_step: 100000 # Total number of simulation steps
  output_interval: 100 # Output interval
  field_x_len: 1024 # Length of the x-coordinate of the simulation field
  field_y_len: 1024 # length of the y-coordinate of the simulation field
  field_z_len: 0 # length of the simulation field in z-coordinate
  delta_time: 0.001 # time in simulation per step
```

```
14
15 molecule:
16   default_molecule_nums: # initial number of each molecule (expressed as a
         list)
17       - 3000
18   delta_time: 0.001
19   field_x_len: 1024 # length of x-coordinate in field for molecule management
         >= 1
20   field_y_len: 1024 # length of y-coordinate of the molecular control field >=
          1
21   field_z_len: 1 # length of the z-coordinate of the molecular control field
         >= 1
22   boundary_condition: # Boundary condition for the molecular control field
23       xy1: NEUMANN
24       xy2: NEUMANN
25       yz1: NEUMANN
26       yz2: NEUMANN
27       zx1: NEUMANN
28       zx2: NEUMANN
29
30 # Not used in this case, so not relevant
31 cell_list:
32   use_cell_list: false # whether cell list is used or not
33   grid_size_mag: 32 # grid division factor in cell list. Minimum must be 1 and
          value must be 2^n.
34   search_radius: 256 # cells within this radius (difference of position) are
         used for force calculation.
```