CyCells User Manual

Christy Warrender
christy@cs.unm.edu

September 17, 2004

1 Introduction

This documentation is for CyCells version 0-3.

CyCells is a three-dimensional, discrete-time simulator for studying intercellular interactions through extracellular molecular signals. Cells are represented explicitly; molecules are represented by their concentration. Cell actions, including secretion of molecules, are influenced by the local concentration of extracellular molecules. The behavior of different types of cells and molecules is specified at run time, as described below. There is significant flexibility in the way cell types can be defined, and a number of different cell and molecule types can be combined in any given model. As a result, a large variety of conceptual models that fit this general framework can be implemented and executed with the simulator.

The code is written in C++ and has been tested under linux. There are two versions of the simulator. CyCells runs in batch mode, executed from the command line. wxCyCells is an interactive version that uses a GUI developed with wxWindows. Both can be compiled using the makefile included with the code. wxCyCells requires wxWindows and OpenGL libraries, available as Debian packages glutg3-dev and libwxgtk-dev.

Running a simulation requires three steps: model definition, model initialization, and the actual execution. These are explained in sections 2, 3, and 4. Simulator outputs are explained in section 5.

2 Model Definition

A model definition file like the one shown in Figure 1 specifies the behavior of each molecule and cell type used in a simulation. The definition file must begin with the line

```
#DefFormat 8.
```

The file consists of a list of cell type names, followed by a block for each molecule type and a block for each cell type. There is no preset limit on the number of cell and molecule types that can be defined.

```
#DefFormat 8

cell_names { macrophage cycling tissue }

molecule_type CSF {
decay_rate 1e-4
diffusion_rate 100
}

cell_type macrophage {
radius 10
attribute cmax lognormal 3.58 0.4 lognormal 3.58 0.4
attribute b fixed 0 fixed 0
attribute S fixed 0 uniform 0 700000
attribute sr gaussian 700000 50000 gaussian 700000 50000
attribute time fixed 0 fixed 0
attribute tc gaussian 43200 1800 gaussian 43200 1800
sense b consume-indiv CSF cmax 1.3E-13
process S update linear b 1 0
action change cycling gte_var S sr
action die calc_prob inhibiting b 1e-5 0.37
}

cell_type cycling {
radius 10
attribute cmax lognormal 3.58 0.4 lognormal 3.58 0.4
attribute b fixed 0 fixed 0
attribute S fixed 0 uniform 0 700000
attribute sr gaussian 700000 50000 gaussian 700000 50000
attribute time fixed 0 uniform 0 43200
attribute tc gaussian 43200 1800 gaussian 43200 1800
sense b consume-indiv CSF cmax 1.3E-13
process time update fixed 1
action divide macrophage gte_var time tc
}

cell_type tissue {
action secrete CSF fixed 2 always
}
```

Figure 1: Sample model definition file.

Molecule types are defined by giving them a name and specifying the appropriate decay and/or diffusion rates. Decay rates are in units of sec^{-1}; diffusion rates are in microns^2/sec. Note that cells can also change the molecular concentration by secreting or binding molecules. Although there is only one molecule type in the example, multiple molecule types are allowed.

A cell type definition specifies the attributes all cells of that type have, how those attributes should be initialized, and how they should be updated or used to make decisions each time step. The actual values of those attributes are stored for each individual cell. In the example shown in Figure 1, macrophages have 6 attributes. Each attribute is defined by a line of the form:

```
attribute attr init_flag init_param1 [init_param2]
                   rand_flag rand_param1 [rand_param2]
```

attr is the name of the attribute; this is used in other parts of the model definition to index the appropriate value. The rest of the line defines two initialization schemes for this value. The first initialization scheme is used to create two new daughter cells when a cell divides; the second is used to create cells when the simulation is initialized (in which case older cells may have different parameter values than newly divided cells). In each case, the flags specify the distribution to use; the choices are: `fixed`, `uniform`, `gaussian`, or `lognormal`. A `fixed` flag is followed by only a single value, which will be used for all cells. The parameters following a `uniform` flag specify lower and upper bounds; a value will be chosen randomly for each cell from within this range. The `gaussian` parameters are the mean and standard deviation of the distribution from which values should be drawn. Specification for `lognormal` is similar, but the standard deviation is that of the underlying gaussian distribution.

There are also a couple of parameter values that are defined for all cells of each type. One of these is cell radius, as shown in the example. It may be omitted; the default radius is 5 microns. The other is speed, specified in microns/sec; the default is 0. Note that speed determines how fast a cell moves when there are no obstacles; interactions with other cells will affect its movement.

Cells are represented as spheres for convenience, but those spheres are allowed to overlap to account for the fact that real cells are deformable. There is an ad hoc repulsive force when cells do overlap that will tend to move them apart, space permitting, and that will prevent cells from moving directly through each other.

The simulator treats sensing, intracellular signalling and actions as different kinds of cell functions that use or update cell attributes. Cells may die, divide, differentiate, migrate, or secrete molecules. Some of these actions are constitutive, others depend on cell state. For the latter, intracellular processing functions determine how a cell's current state and its perception of the local environment are combined to update the cell state. They are an abstraction of the complex signalling that goes on inside real cells. Sensing functions update cell variables in accordance with the current molecular concentration and may remove molecules from the environment in the process The model definition includes a line for each such function used by each cell type, with the relevant parameter values.

2.1 Sensing functions

Cells may sense molecules in their neighborhood or nearby cells. A sense function must be defined for each molecule type and cell type that a cell senses. All sensing functions are defined using a line of the form:

```
sense attr sense_type [parameters]
```

attr is the name of the cell attribute where the resulting value will be stored. The various kinds of *sense_type* are explained below.

2.1.1 Sensing molecular concentrations

There are four routines available that implement cell sensing of the molecular environment. The simplest form of molecular sensing is just to copy the local concentration into a local cell variable:

```
sense attr copy_conc mol_type
```
where *mol_type* specifies which molecular species is to be sensed. This is not a very good representation for what cells actually do, because they are not infinitely sensitive to all possible molecular concentrations. However, it may be adequate, depending on the model objectives. The following routines incorporate more of what is known about the mechanisms of receptor-ligand binding.

Reversible binding, in which ligand binds to cell surface receptors, but can also dissociate, can be included by using the following definition:
```
sense attr bind_rev mol_type k_f k_r R
```
k_f is the (forward) rate of binding, k_r is the (reverse) rate of dissociation, and R is the number of receptors this cell type has available for binding. These, together with the local molecular concentration (L), are used to determine the current number of bound receptor-ligand complexes C according to:

$$C_t = C_{t-1} + \Delta t(k_f(R - C)L - k_r C) \tag{1}$$

The number of bound complexes C is stored in the cell value indexed by *attr*. Note that this value is both read and updated each time step. Changes to the number of bound receptors also affect changes to the local molecular concentration; newly bound molecules are removed from the extracellular environment, and newly dissociated molecules are returned to the extracellular environment.

In some cases, internalization of bound receptor-ligand complexes dominates dissociation. In this case, we use either
```
sense attr consume mol_type c_max c_half
```
or
```
sense attr consume_indiv mol_type c_max_attr c_half
```
In the first version, the maximum consumption rate c_{max} is specified directly; in the second, it is stored in a cell variable. The latter allows heterogeneity between cells of the same type. The value stored at *attr* represents the rate c of binding new molecules, which is assumed to represent the signal a cell gets from the molecular environment:

$$c_t = \frac{c_{max} L}{c_{half} + L} \tag{2}$$

where c_{half} is the half-saturation constant (the value of L at which c will be half-maximal).

2.1.2 Sensing other cells

There are two routines for sensing other cells. The simplest is simply to detect whether a cell of a certain type is nearby, which uses the following definition:
```
sense attr cognate target_type distance
```
target_type is the name of the target cell type. *distance* is the maximum distance at which the sensing cell can detect the target cell. Cells are treated as spheres for movement purposes, but real cells aren't spheres, so specifying a distance above reflects the assumption that a cell might extend a pseudopod or use some

other means to sense beyond its spherical radius. However, *distance* represents a measure between cell centers, so you need to take the respective cell sizes into account in determining what *distance* should be. Each time this function is executed, if there is a cell of the target type within the specified distance, *attr* will be set to 1; if not, it will be set to 0.

The second sensing function implements phagocytosis of one cell by another (as in macrophage phagocytosis of bacteria). The appropriate definition is:

```
sense attr phag target_type distance R_attr R_thr
```

target_type and *distance* have the same meanings as in the last sensing function (cognate). *R_attr* specifies a cell attribute that is assumed to store the number of receptors the cell has for this target. This value will be compared to the threshold value *R_thr* to determine whether phagocytosis occurs. During each time step of the simulation, this function will cause a cell to 'examine' one cell in its neighborhood—defined by *distance*—at random, assuming there are any. If the cell is of the target type, and the value stored in *R_attr* is greater than *R_thr*, the target cell will be removed from the extracellular environment, and the *attr* value, which is assumed to represent the number of ingested target cell types, will be incremented by one.

2.2 Processing functions

Most cell variables are assumed to represent real-valued quantities (and all are stored as real values). The first set of processing functions described here provide a number of ways to manipulate these values mathematically. Processing functions can be used either to incrementally update cell variables (in which case the current value is used in determining the next value) or replace them (in which case the current value is not used). The general definition form is:

```
process attr process_type rate_desc [min max]
```

attr identifies the cell value Y to be changed. *process_type* is either `replace`, `update`, or `update_bounded`. *rate_desc* specifies how to calculate a value V based on current cell state (explained in section 2.3); *min* and *max* are only used with `update_bounded`. The *process_type* determines how V will be used to update Y:

`replace`: $Y = V$.
`update`: $Y \mathrel{+}= V\Delta t$
`update_bounded`:
    if $V < min$, $Y = min$
    else if $V > max$, $Y = max$
    else $Y = V\Delta t$

Another form allows an attribute that only takes on two values to switch between those values according to certain conditions (explained in section 2.5). The general form for these process functions is:

```
process attr process_type low_value high_value cond1 cond2
```

If the attribute currently is equal to *low_value* (*high_value*) and *cond1* (*cond2*) is true, *attr* will be set to *high_value* (*low_value*). A *process_type* of `toggle` indicates that the values are specified as actual numbers; a *process_type* of `toggle_var` indicates that the values are specified as indices of cell attributes.

2.2.1 Intracellular pathogen dynamics

There are two special-purpose processing functions that implement birth-death dynamics on cell variables representing intracellular pathogens. In this case, the cell values are kept as whole numbers. Changes are based on pathogen birth (division) and death rates. These rates (in units of sec^{-1}) can either be specified directly:

```
process bd attr birth_rate death_rate
```

or stored as cell variables:

```
process bd_var attr birth_attr death_attr
```

These rates are converted to probabilities in the simulator, and the value in *attr* is updated stochastically. It is assumed that the rates and simulator time step are small enough that the number of pathogens is either incremented by one, decremented by one, or remains the same in any given time step.

2.3 Rate functions

The simulator does not parse arbitrary mathematical equations. Instead, it provides a few functions that I have found useful for manipulating cell state variables. These are actually function objects, combining parameter values with specific mathematical equations. Originally, all of the values calculated by these functions really were rates, and were multiplied by Δt. This is no longer true, but the name remains.

The simplest rate 'functions' are really just ways to store a value in a way that a process function can access it:

```
fixed value
```

stores the specified value;

```
var attr
```

stores an index to an attribute containing the required value.

A linear function of a value X stored in *attr* is defined by:

```
linear attr slope y-intercept
```

Other functions are defined similarly, with the value X assumed to be stored in *attr*:

```
sigmoid attr thr sigma
```

implements the function

$$V = \frac{1}{1 + e^{-\sigma(X - thr)}} \tag{3}$$

which returns values between 0 and 1, with 0.5 returned at $X = thr$, and σ determining how steeply V changes from 0 to 1.

```
saturating attr v_max v_half
```

implements the function

$$V = \frac{v_{max} X}{v_{half} + X} \tag{4}$$

`inhibiting` *`attr v_max v_half`*
implements the function

$$V = \frac{v_{max} v_{half}}{v_{half} + X} \tag{5}$$

There are three functions that operate on two cell values X_1 and X_2:
`rel_sat` *`attr1 attr2 v_max v_half f`*
implements the function

$$V = \frac{v_{max} X_1}{X_1 + f X_2 + v_{half}} \tag{6}$$

`rel_inh` *`attr1 attr2 v_max v_half f`*
implements the function

$$V = \frac{v_{max} X_1 v_{half}}{X_1 + f X_2 + v_{half}} \tag{7}$$

`synergy` *`attr1 attr2 v_max v_half f`*
implements the function

$$V = \frac{v_{max} X_1 (1 + f X_2)}{X_1 (1 + f X_2) + v_{half}} \tag{8}$$

2.4 Cell Actions

The general form for defining a cell action is
`action` *`action_type`* `[`*`parameters`*`]` *`condition`*
condition determines when an action will be triggered; these are explained in section 2.5. First, we list the various action types possible and the parameters they require.

2.4.1 Cell Death

`action die` *`condition`*
requires no parameters. When the condition is met, the cell is removed from the simulation. Note that this may be used to represent cells leaving the simulated compartment, as well as actual death.

2.4.2 Cell Differentiation

`action change` *`new_cell_type condition`*
causes a cell to change its type, as in differentiation, or infection. As far as the simulation goes, this provides a way for a cell to keep its state information, but change the rules by which it functions. Note that in order for this to work, the two cell types involved should have the same set of attributes.

2.4.3 Cell Division

`action divide` *`new_cell_type condition`*
specifies that when a cell divides, it produces two new cells of the type specified in *new_cell_type*. This could be the same type as the original cell; the option to create a different type is provided to allow differentiation or maturation—represented by a change in cell function—to accompany division. Again, if the cell types are different, they should have the same set of attributes. Daughter cell attributes are initialized according to the first initialization scheme in the attribute definition. The two new cells are given the same y and z coordinate as the original cell but a slightly different x coordinate; one cell is offset by 0.1 microns and the other by -0.1 microns. (Rather arbitrary, but it's worked so far.)

2.4.4 Cell Influx

`action admit` *`new_cell_type init_flag distance condition`*
allows a new cell of the type specified to enter the simulated compartment. This is under the control of a cell already in the simulation, because it is assumed that endothelial and/or epithelial cells generally control entry of cells from circulation into local tissues, usually in response to their molecular environment. So this action would be part of the definition of such a 'gateway' endothelial or epithelial cell. The initialization scheme to use for the new cell can be specified - `init` means use the first initialization scheme for each attribute; `rand` means use the second initialization scheme for each attribute. Different initialization schemes cannot be used for different attributes of the same cell. The new cell is positioned at the specified distance away from the 'gateway' cell in a randomly chosen direction.

`action admit_gradient` *`new_cell_type init_flag molecule_type distance condition`*
is similar, but the direction for the offset of the new cell is chosen in the direction of the local molecular gradient of *molecule_type*. If the gradient is not detectable, the offset direction is chosen randomly.

2.4.5 Release of Intracellular Pathogens

The release of intracellular pathogens is similar to influx of new cells in that new simulated cells must be created and placed into the compartment, and this function is under the control of an existing simulated cell.

`action admit_mult` *`new_cell_type init_flag distance rate`*
creates multiple cells of the specified type, with the number determined by *rate* (often this is just equal to an internal cell attribute value). Their positions are radially distributed around the host cell's position at the specified distance.

2.4.6 Cell Movement

There are several actions that specify how cells move. It is important to realize that these really control a cell's choice of the direction in which it is going to *try* to move; collisions with neighboring cells may affect the actual displacement of a cell. The choices allow either random or chemotactic movement in either 3 or 2 dimensions. In 2-dimensional movement, the z component of the velocity is always 0. The form for random movement definition is just

`action move_randomly` *`condition`*

or

`action move_randomly2D` *`condition`*

For chemotactic movement, a molecule type and minimum detectable concentration must also be specified:

`action move_chem` *`mol_type min_conc condition`*

or

`action move_chem2D` *`mol_type min_conc condition`*

The cell will move up the local gradient of this molecule if there is one and if the concentration is greater than *min_conc*; otherwise it will choose a direction randomly. The condition specifies how often a cell changes direction.

2.4.7 Secretion

There are also several options for specifying molecular secretion.

`action secrete_burst` *`mol_type attr condition`*

causes a cell to secrete a fixed number of molecules, specified by the value stored in *attr*, when the condition is met. The other forms take a secretion rate; the amount secreted will be the product of this rate and the time step size:

`action secrete_fixed` *`mol_type rate condition`*

takes a fixed secretion rate *rate*;

`action secrete_var` *`mol_type attr condition`*

uses the value stored in *attr* as the rate; and

`action secrete` *`mol_type rate_desc condition`*

calculates the rate using a rate function as described in section 2.3.

2.4.8 Composite Actions

Finally, there is a means for specifying that two actions should triggered by the same condition:

`action composite` *`action1 action2 condition`*

(these could be nested to allow more actions to be combined in the same way).

2.5 Conditions

There are two primary types of conditions. The first is probabilistic, in which case a random number is compared to some probability to determine whether the condition is met; the second is a threshold condition on some cell variable.

2.5.1 Probabilistic Conditions

There are three forms of probabilistic conditions:

```
fixed value
```

specifies a fixed probability;

```
var_prob attr
```

specifies a cell attribute from which to read the probability; and

```
calc_prob rate
```

takes a rate function as specified in section 2.3. If the calculated value is less than 0, a probability of 0 is used; if the calculated value is greater than 1, a probability of 1 is used. It is up to you to make sure that the rate function returns values appropriate for your model.

2.5.2 Threshold Conditions

There are four forms of threshold conditions:

```
gte attr value
```

becomes true if the value stored in *attr* is greater than or equal to *value*.

```
gte_var attr1 attr2
```

becomes true if the value stored in *attr1* is greater than or equal to that stored in *attr*2. Similarly,

```
lte attr value
```

becomes true if the value stored in *attr* is less than or equal to *value*.

```
lte_var attr1 attr2
```

becomes true if the value stored in *attr1* is less than or equal to that stored in *attr*2.

2.5.3 Composite Conditions

Also, conditions can be combined using either

```
composite cond1 cond2
```

which ANDs the two conditions, or using

```
or cond1 cond2
```

which ORs the two conditions.

3 Model Initialization

An initialization file, like the one shown in Figure 2, specifies the simulation geometry, initial molecular concentration(s), and initial numbers of each cell type. Cell positions can be specified if desired; in this example, cells are positioned randomly. The file must begin with the line

```
#InitFormat 4
```

```
geometry
1000x1000x1000 microns; mol_res:  0  cell_res:  0

molecule_uniform:  CSF 6E-15 0

cell_count:  tissue 1000
cell_count:  cycling 30
cell_count:  macrophage 970
```

Figure 2: Sample model initialization file.

3.0.4 Geometry

The simulation works with a three-dimensional volume, specified in microns. That space may be divided into regular cubes, or patches, where the patch size determines spatial resolution. Spatial resolution for cells and molecules are independent of each other. The geometry is specified by a line in the initialization file of the form:

`geometry` xxyxz `microns; mol_res` *`mol_res`* `; cell_res` *`cell_res`*

(This is shown as two lines in the example; the actual arrangement of white space does not matter). For molecules, *mol_res* determines the granularity of stored molecular concentrations; a concentration is stored for each patch of size *mol_res*^{3}. For cells, *cell_res* determines how cells are grouped by position to facilitate finding neighbors. In either case, a resolution value of 0 means that spatial effects are not important, and the volume does not need to be divided; the molecular environment is assumed to be homogeneous, and/or relative locations of cells should not matter. When spatial effects are important, either the molecular resolution or the cell resolution or both may be set accordingly. Each is specified in microns, and must divide evenly into each of the three length dimensions defining the simulation volume. Smaller molecule grid sizes increase the run time, as each grid cell must be updated on each time step. In contrast, small cell resolutions can decrease the run time by decreasing the number of potential neighbors the simulator must check when moving cells or looking for neighbors with which a cell might interact. Too small a resolution will make this search useless, though; a good rule of thumb is to set the cell resolution to the diameter of the largest cell type. If the cells do not move or try to find neighbors, then *cell_res* should be set to 0.

3.0.5 Molecular concentrations

Molecular concentrations are usually set by specifying a mean value and standard deviation:

`molecule_uniform:` *`mol_type mean std_dev`*

The value at each patch will be set by choosing a value from the gaussian distribution described. The standard deviation value can be 0.

It is also possible to specify that the concentration be reset periodically:

`molecule_reset:` *`mol_type interval mean std_dev`*

At each time divisible by *interval*, the concentration will be reset as above.

If no initialization information is given for a molecular species defined in the definition file, it is assumed to be 0 everywhere.

3.0.6 Cell numbers and positions

Cells can be added either by specifying a number and letting the simulator place them randomly, by positioning each cell individually, or by adding cells in certain pre-defined patterns. To add n cells of one type randomly, use either

`cell_count:` *`cell_type n`*

or

`cell_count2D:` *`n cell_type z`*

For the latter option, z specifies the z-coordinate of each cell's position. For multiple cell types, use a separate line for each cell type.

Cells can be individually positioned by using

`cell_list:` *`n`*

followed by n lines of the form

`cell_type x y z`

where x, y, and z specify each cell's position. The cell list does not need to be ordered by type or position.

There are a few ways to pack cells into a plane.

`cell_sheet:` *`cell_type z`*

fills the available space with a rectangular array of cells of type *cell_type*, each with a z-coordinate of z. Similarly,

`cell_hexsheet:` *`cell_type z`*

fills the available space with a hexagonal array of cells. Two cell types can be mixed in a hexagonal array using

`cell_hexmix:` *`cell_type1 cell_type2 frac1 z`*

where the fraction *frac1* specifies what fraction of the total will be of type *cell_type1*.

There is also an option to add cells in a grid like that shown in figure 3. The 2D version is:

`cell_grid2D:` *`cell_type size z`*

where *size* specifies the distance between rows or columns of cells in microns, measuring from cell centers. This distance should be a multiple of the cell radius. The 3D version just extends the same pattern into the z-dimension; it does not fill any plane $z = value$.

All of the options for adding batches of cells use the second of the two initialization schemes for each attribute, on the assumption that these are not necessarily new daughter cells.

3.0.7 Other

Initial time may be set using

`timestamp:` *`time`*

and a seed for the random number generator may be specified using

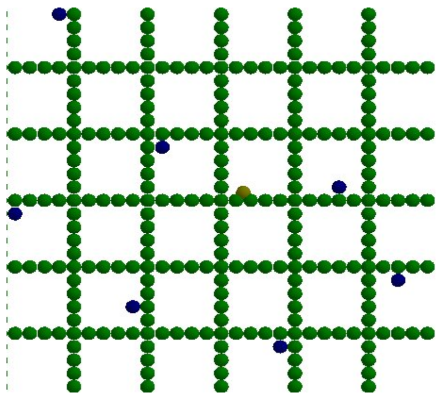


Figure 3: Grid arrangement of cells.

`seed:` *`value`*
The default initial time is 0; if no seed is specified, one will be generated randomly.

4 Simulation Execution

4.1 Running in batch mode

Start the simulator in batch mode from the command line with:

```
CyCells [-d def_file] [-i init_file] [-o output_file] [-s seed]
[-t duration] [-e stepsize] [-f detail_file] [-w history_stepsize]
[-v detail_stepsize] [-c max_cells]
```

def_file is the name of the definition file. The default is test.def.

init_file is the name of the initialization file. The default is test.init. Note that the detailed output file described in section 5.2 can be used as an initialization file.

output_file is the name of the file to which time course data should be written. The default is test.history. Time course data is explained in section 5.1.

seed is the seed for the random number generator. If none is provided, a seed will be generated randomly. Note that if you also specify a seed in the initialization file, that seed will be used in place of the command line seed.

duration is the duration in seconds for the simulation to run. The default is 10 seconds (just enough to make sure there are no problems with model definition or initialization). This can be overridden by *max_cells*.

stepsize is the size of each time step in seconds. The default is 1 second. Both duration and time step sizes can be real-valued, although I have not needed to use time steps less than 1 second. Your choice of time step should take into account the various rates in your model definition. The only rates that the

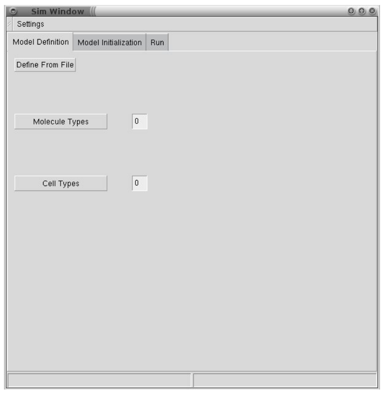


Figure 4: wxCyCells.

simulator automatically adjusts for are molecular diffusion rates; the diffusion routine will run multiple times per time step if necessary.

detail_file is the base file name to use when writing detailed information for a single time step. The default is test.detail. The time will appended to the end of this name. For example, data from time = 100 would be written into a file named test.detail.100. Detailed output data is explained in section 5.2.

history_stepsize defines how often data will be written to the history file. This can be set separately from the simulation time step to avoid generating huge output files on long simulations. The default is 1 second or the simulation time step, whichever is larger.

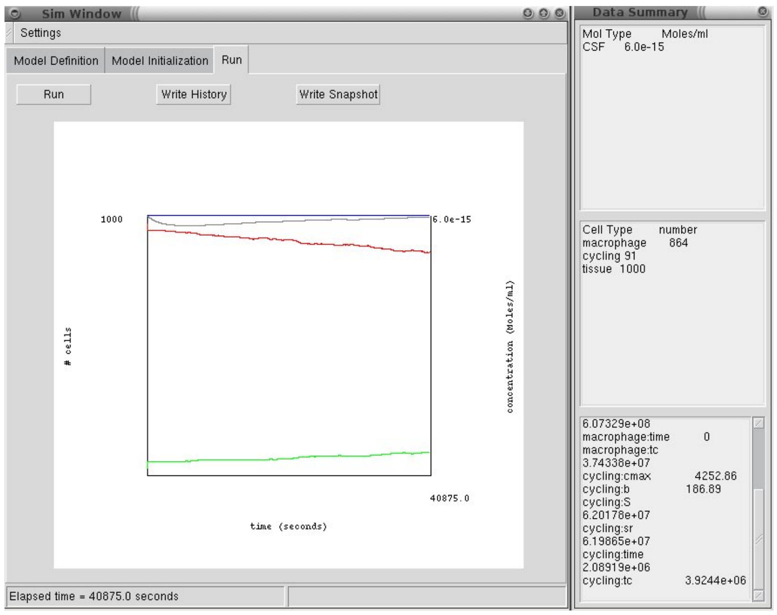
detail_stepsize defines how often detailed data will be written. The default is not to produce detailed output.

max_cells is the maximum number of cells that the simulation will allow; if the actual number instantiated exceeds this number, the simulation will terminate even if the specified duration has not been reached.

4.2 Running interactively

Start the interactive version by typing `CyCells` at the command line. This will bring up the user interface shown in figure 4. Interactive model definition and model initialization have not been developed yet; you must use the user interface to load in model definition and initialization files.

From the settings menu, you can set the simulation duration and time step, and select the model view to use and how often it is updated. The duration is really a maximum duration; the simulation can be started and stopped as often as you like by pressing the Run/Stop button on the Run page. Only a few options are currently available for time step in the interactive version. For model view, you may select 'history view', which plots cell counts and average concentration over time as the simulation runs; '3D view', which shows



```
Elapsed time = 40875.0 seconds
```

```
Mol Type        Moles/ml
CSF     6.0e-15
```

```
Cell Type       number
macrophage      864
cycling 91
tissue  1000
```

```
6.07329e+08
macrophage:time         0
macrophage:tc
3.74338e+07
cycling:cmax            4252.86
cycling:b               186.89
cycling:S
6.20178e+07
cycling:sr
6.19865e+07
cycling:time
2.08919e+06
cycling:tc              3.9244e+06
```

Figure 5: wxCyCells history view and data summary.

a graphical image of the cells in the simulated compartment; or ‘none’, which runs faster. An example of the 3D view was shown in 3. The history view is shown in 5. Note that there is also a data summary showing the current concentrations, cell counts, and cell attributes. The latter are shown as totals from all cells of each type.

The ‘Write History’ and ‘Write Snapshot’ data allow you to write the time course and detailed output files, respectively.

4.3 Main simulation algorithm

One time step of the simulation consists of the following sequence of activities:

1. Molecular diffusion and decay
2. Update of each cell according to sense, process, and act functions
3. Cell movement (if applicable)

5 Outputs

5.1 Time course output

For each time that data is written, a line of tab-separated data is added to the history file. The first column is the time. This is followed by a column for the

```
#InitFormat 4

geometry
1000x1000x1000 microns;  mol_res: 50  cell_res: 10

timestamp:  10

rnginfo:  31 7 0 83026927 584163512 725632082 414406863 413444359 977228253 8498
83336 369849816 528234276 217847404 515951109 733228325 175785633 330925614 5561
3486 92332524 780810725 562687387 699569707 453524480 380207747 558144610 511024
713 48502957 810944398 153794088 722471470 610013658 855086146 383971984 6909294
44 738797523 670080061 377034171 504320296 786753422 468969012 54777986 44986285
8 296124543 60580449 253297575 265364630 825182676 353855757 52819119 954054302
530998671 673409198 887141788 440360814 758641679 817191042 748241266 71764105

molecule_detail: CSF
5.99941e-15

cell_detail:  4
type 1  (220.191, 842.712, 508.188)   (0.00374799, 0.0144182, -0.00989944)   (-0
.218404, 0.795706, 0.564936)  0.398667 2 1.6e-05 3.5056e-05
type 3  (514.578, 588.619, 658.534)   (0, 0, 0)   (0, 0, 0)  9.53322e-15
type 2  (605.216, 372.008, 621.942)   (0, 0, 0)   (0, 0, 0)
type 3  (492.46, 508.159, 673.696)   (0, 0, 0)   (0, 0, 0)  4.60612e-15
```

Figure 6: Sample detail output file.

spatial average concentration of each molecule type defined, and then a column for the number of cells of each cell type defined. Molecule and cell data is listed in the order in which it was defined. For the model definition example given in figure 1, there would be five columns: time, CSF concentration, number of macrophages, number of cycling cells, and number of tissue cells.

5.2 Detailed data for one time step

For each time that detailed data is written, a new file is created. The last part of the file name is the simulation time that this data represents. A sample detail file (abbreviated) is shown in figure 6. The general format is the same as that used for initialization files, so that the output of one simulation can be used as the starting point for another.

rnginfo lists the values stored by the random number generator that need to be reloaded in order to resume the correct position in the pseudo-random number sequence.

Molecular concentrations are shown for each molecule type, with a value given for each patch (only one used in the example).

For each cell, the file lists the cell type index, position, velocity, orientation and attribute values. Indices start at 0; in this example, type 2 corresponds to the tissue cells from the model definition example in figure 1. Attributes are listed in the order they were defined.

5.3 Action tallies

The number of times certain actions are invoked is tallied and written to a file with the same name as the history file plus the extension '.actions'.