PNNL-XXXXX

Biocellion 1.2 User Manual

Seunghwa Kang and Simon Kahan

November 15, 2016

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RLO1830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Acknowledgments

Support for this research was provided by the Extreme Scale Computing Initiative and the Fundamental and Computational Sciences Directorate, as part of the Laboratory Directed Research and Development Program at Pacific Northwest National Laboratory (PNNL). Portions of this work were conducted using PNNL Institutional Computing at PNNL. PNNL is operated by Battelle for DOE under contract DE-ACO5-76RLO 1830.

Changes

Changes from Version 1.1

Feature Updates

- We have modified *Biocellion*'s directory structure and Makefiles to facilitate enterprise installations in which users have read-only access to a shared instance of the *Biocellion* platform (pre-installed by a system administrator) and place and modify their own model files in their private workspace. See Section 6.2.1.

- *Biocellion* now supports multiple iterations of mechanical interaction evaluations per *baseline time step* and merges the *computeForceSpAgent()* and *computeExtraMechIntrctSpAgent()* model routines to *computeMechIntrctSpAgent()* (Section 4.4.2). The main motivation for supporting multiple iterations is to aid implementation of force propagation through junctions between agent pairs but *Biocellion* does not restrict users from using this extension for other purposes. The main motivation for the merging is to simplify the API considering that force evaluation can be implemented as (extra-)mechanical interaction evaluation by adding additional model specific mechanical interaction attributes for force.

- *Biocellion* now supports a grid resolution finer than the interface grid spacing for partial differential equation (PDE) solves. No longer is the maximum resolution of PDE solves constrained to be no finer than the maximum mechanical interaction distance between agent pairs.

- To generate adaptive mesh refinement (AMR) grid, users are asked to tag every interface grid unit boxes with a desired refinement level in the AMR hierarchy. In *Biocellion* 1.2, users can expand the tagged region in each level by a user provided amount to expand the regions covered by finer level boxes (See Section 2.4.14).

- *Biocellion* now asks users to set variable synchronization method for each attribute. See the descriptions about the sync_method_e and var_sync_method_e enumeration types in Section 2.3.

- *Biocellion* now allows users to output 3D vector variables (in addition to color, radius, and extra-scalar variables) for each agent. This feature is added to visualize ellipsoids (ellipsoids require two additional vectors for scaling and orientation). See Section 2.4.17.

- *Biocellion* now uses the vtkPolyData file format (.vtp) for agent visualization instead of the vtkUnstructuredGrid file format (.vtu). To visualize grid data with adaptive mesh refinement (AMR), *Biocellion* now uses the vtkOverlappingAMR file format (.vthb) instead of the vtkHierarchicalBoxDataSet file format (.vtm). See Sections 6.3.1 and 6.7.

- *Biocellion* now supports redirecting output (Section 3.4.2) messages to a file and time-stamping for output (Section 3.4.2), warning (Section 3.4.3), and error (Section 3.4.4) mes-

sages. See Section 6.3.2.

API Changes

- Classes are added, removed, or renamed.
  - MechModelVarInfo (Section 2.4.7) class is added.
  - The GridVelInfo class (Section 2.4.11) is added as a placeholder (for future computational fluid dynamics updates).
  - The AgentJunctionInfo class is renamed to JunctionData.
  - The AgentMechIntrctData class (which has VReal *force* and ExtraMechIntrctData *extraMechIntrctData* as member variables) is removed and the ExtraMechIntrctData class is renamed to MechIntrctData (Section 2.4.24). Now we consider force as one of many potential mechanical interaction attributes and *Biocellion* does not pre-specify any mechanical interaction attributes. Users specify the entire set of mechanical interaction attributes relevant to their model.
  - The IfGridUpdate class is removed. Users do not use this class.
  - The NbrBox class is removed and the UBEnv (Section 2.4.25), UBEnvModelVar (Section 2.4.26), NbrUBEnv (Section 2.4.27), NbrUBEnvModelVar (Section 2.4.28), and NbrUBAgentData (Section 2.4.29) classes are added.
- Enumeration types are added or updated.
  - The `PDE_TYPE_INCOMPRESSIBLE_NAVIER_STOKES_TIME_DEPENDENT` enumeration constant is added to the pde_type_e enumeration type (Section 2.3) as a placeholder (for future computational fluid dynamics updates).
  - The sync_method_e enumeration type (Section 2.3) is updated. In *Biocellion* 1.1, sync_method_e has `SYNC_METHOD_OVERWRITE`, `SYNC_METHOD_DELTA`, and `SYNC_METHOD_TRANSACTIONAL` as enumeration constants. In *Biocellion* 1.2, sync_method_e has `SYNC_METHOD_PER_ATTR` and `SYNC_METHOD_TRANSACTIONAL`. `SYNC_METHOD_TRANSACTIONAL` is currently not supported and just a placeholder. In *Biocellion* 1.2, synchronization method is selected for each attribute. A single variable can be updated in multiple model routine invocations within a single iteration (to evaluate mechanical interactions in *computeMechIntrctSpAgent()* or to update extracellular space state variables in *udpateIfGridVar()*). Synchronization method decides how to resolve conflicts if a single variable is updated multiple model routine invocations within a single iteration.
  - The var_sync_mehotd_e enumeration type (Section 2.3) is added. var_sync_method_e has `VAR_SYNC_METHOD_OVERWRITE`, `VAR_SYNC_METHOD_DELTA`, `VAR_SYNC_METHOD_MIN`, and `VAR_SYNC_METHOD_MAX` as enumeration constants. This enumeration type is used to set synchronization method for each attribute.

- The IF_GRID_VAR_TYPE_VEL enumeration constant is added to the if_grid_var_type_e enumeration type (Section 2.3) as a placeholder (for future computational fluid dynamics updates).

- The PARTICLE_OUTPUT_TYPE_PVTU in the particle_output_type_e enumeration type (Section 2.3) is renamed to PARTICLE_OUTPUT_TYPE_PVTP.

- The GRID_OUTPUT_TYPE_VTM in the grid_output_type_e enumeration type (Section 2.3) is renamed to GRID_OUTPUT_TYPE_VTHB.

- Class memember variables are added, removed, or renamed.
  - OptModelRoutineCallInfo (Section 2.4.6): S32 *numComputeMechIntrctIters* is added to set the number mechanical interaction evaluation iterations within a single *baseline time step*. S32 *numUpdateIfGridVarPreStateAndGridStepRounds* and S32 *numUpdateIfGridVarPostStateAndGridStepRounds* are renamed to *numUpdateIfGridVarPreStateAndGridStepIters* and *numUpdateIfGridVarPostStateAndGridStepIters*, respectively. The purpose of this renaming is to avoid naming conflict with the *round()* floating point rounding function. These variables should be updated in the *updateOptModelRoutineCallInfo()* model routine.

  - SpAgentInfo (Section 2.4.8): S32 *numExtramechIntrctModelReals* and S32 *numExtramechIntrctModelInts* are removed and Vector<MechModelVarInfo> *v_mechIntrctModelRealInfo* and Vector<MechModelVarInfo> *v_mechIntrctModelIntInfo* are added. These variables should be updated in the *updateSpAgentInfo()* model routine.

  - GridPhiInfo (Section 2.4.10): var_sync_method_e *syncMethod* is added. Now users set variable synchroniztion method for each attribute. In addition to PDE solves, molecular concentration variables can be updated in the *updateIfGridVar()* model routine. The synchronization method on conflict is set in the *updatePhiPDEInfo()* model routine.

  - PDEInfo (Section 2.4.14): S32 *pdeIdx*, S32 *ifLevel*, Vector<S32> *v_tagExpansionSize*, and GridVelInfo *gridVelInfo* are added. The *gridVelInfo* member variable is just a placeholder (for future computational fluid dynamics updates). These variables should be updated in the *updatePhiPDEInfo()* model routine.

  - IfGridModelVarInfo (Section 2.4.15): var_sync_method_e *syncMethod* is added. Now users set variable synchroniztion method for each attribute. Model specific REAL and S32 type variables associated with the interface grid are updated in the *updateIfGridVar()* model routine. The synchronization method on conflict is set in the *updateIfGridModelVarInfo()* model routine.

  - FileOutputInfo (Section 2.4.17): S32 *particleNumExtraOutputVars* is removed and Vector<std::string> *v_particleExtraOutputScalarVarName* and Vector<std::string> *v_particleExtraOutputVectorVarName* are added. This change allows users to set visualization output attribute names and also to output 3D vector (storing 3 REAL

values) variables.

  - SummaryOutputInfo (Section 2.4.18): Vector<std::string> *v_realName*, Vector<summary_type_e> *v_realType*, Vector<std::string> *v_intName*, and Vector<std::string> *v_intType* are removed. std::string *name* and summary_type_e *type* are added. A single instance of the SummaryOutputInfo class now configures summary output of a single attribute instead of configuring the entire set of attributes. These variables should be updated in the *updateSummaryOutputInfo()* model routine.

- Class member functions are added, removed, or renamed.
  - VIdx (Section 2.4.2): *shift()*, *diagShift()*, *coarsen()*, *min()*, *max()*, *setVal()*, *product()*, *lexLT()*, *lexGT()*, *mult()*, and *diagShift()* are added.

  - VReal (Section 2.4.3): *dotProduct()*, *length()*, *lengthSquare()*, *crossProduct()*, *rotate()*, and *normalize()* are added.

- *Biocellion* support functions are added, removed, or renamed.
  - *getParticleNumExtraOutputVars()* is removed and *getParticleNumExtraOutputScalarVars()* (Section 3.1.10) and *getParticleNumExtraOutputVectorVars()* (Section 3.1.11) are added.

  - *getSummaryInterval()* (Section 3.1.13) and *getRegriddingInterval()* (Section 3.1.13) are added.

  - The *WARNING()* macro (Section 3.4.3) is added.

- Model routines are added, removed, or renamed.
  - *updatePDEInfo()* is renamed to *updatePhiPDEInfo()* (Section 4.2.9). *updateVelPDEInfo()* will be added in the future (for future computational fluid dynamics updates).

  - *updateGlobalData()* is renamed to *initGlobal()* (Section 4.2.14).

  - *updateIfGridAdvectionVelocityInIfRegion()*, *updateIfGridAdvectionVelocityPDEBufferBdry()*, *updateIfGridAdvectionVelocityDomainBdry()*, *updatePDEBufferAdvectionVelocityInPDEBufferRegion()*, and *updatePDEBufferAdvectionVelocityDomainBdry()* are removed. These functions are never invoked in *Biocellion* 1.1 and 1.2. *Biocellion* does not support advection-reaction-diffusion partial differential equations, yet.

  - *initIfSubgridKappa()* (Section 4.5.2) is added. This function is relevant only when *v_gridPhiOutputDivideByKappa[elemIdx]* of the FileOutputInfo class is set to true in the *updateFileOutputInfo()* model routine.

  - *updateIfGridKappa()*, *updateIfGridAlpha()*, *updateIfGridBetaInIfRegion()*, *updateIfGridBetaPDEBufferBdry()*, *updateIfGridBetaDomainBdry()*, *updateIfGridRHSLinear()*, *adjustIfGridRHSTimeDependentLinear()*, and *updateIfGridRHSTimeDependentSplitting()* are renamed to *updateIfSubgridKappa()* (Section 4.5.4), *updateIfSubg*

ridAlpha() (Section 4.5.5), *updateIfSubgridBetaInIfRegion()* (Section 4.5.6), *update IfSubgridBetaPDEBufferBdry()* (Section 4.5.7), *updateIfSubgridBetaDomainBdry()* (Section 4.5.8), *updateIfSubgridRHSLinear()* (Section 4.5.9), *adjustIfSubgridRHSTi meDependentLinear()* (Section 4.5.10), and *updateIfSubgridRHSTimeDependentSplit ting()* (Section 4.5.11), respectively. PDE parameters are set in the interface subgrid (assume that the interface subgrid is imposed on the interface grid, and the interface subgrid spacing coincides with the finest level grid spacing in the AMR hierarchy to solve PDEs).

- Multiple model routines' input argument list are modified.
  - sync_method_e& *extraMechIntrctSyncMethod* in *updateSyncMethod()* is renamed to s ync_method_e& *mechIntrctSyncMethod*.
  - In *updateSummaryOutputInfo()*, SummaryOutputInfo& *summaryOutputInfo* is removed and Vector<SummaryOutputInfo>& *v_summaryOutputRealInfo* and Vector<Summa ryOutputInfo>& *v_summaryOutputIntInfo* are added.
  - REAL *a_gridPhi[3]* is replaced with Vector<REAL> *av_gridPhi[3]* in *updateIfGrid DirichletBCVal()* and *updateIfGridNeumannBCVal()*.
  - Vector<REAL> *v_extra* in *updateSpAgentOutput()* is replaced with Vector<REAL> *v_extraScalar* and Vector<VReal>& *v_extraVector*.
  - S32 *pdeIdx* is added to the input argument list of *updateIfSubgridKappa()*.
  - S32 *iter* is added to *computeMechIntrctSpAgent()*.
  - Vector<Vector<NbrBox<REAL>>>& *vv_gridPhiNbrBox*, Vector<Vector<NbrB ox<REAL>>>& *vv_gridModelRealNbrBox*, Vector<Vector<NbrBox<S32>>>& *vv_gridModelIntNbrBox* is replaced with NbrUBEnv& *nbrUBEnv*. This affects *spAge ntCRNODERHS()*, *updateSpAgentState()*, *spAgentSecretionBySpAgent()*, *updateSpAg entBirthDeath()*, *adjustSpAgent()*, *divideSpAgent()*, *updateIfGridVar()*, *updateIfGridA MRTags()*, and *updateSummaryVar()*.
  - Vector<REAL>& *v_gridPhi*, Vector<REAL>& *v_gridModelReal*, Vector<S32>& *v_gridModelInt* is replaced with UBEnv& *ubEnv*. This affects *initIfGridVar()*, *upd ateIfSubgridKappa()*, *updateIfSubgridAlpha()*, *updateIfSubgridBetaPDEBufferBdry()*, *updateIfSubgridBetaDomainBdry()*, and *updateIfSubgridRHSLinear()*.
  - Vector<REAL>& *v_gridPhi0*, Vector<REAL>& *v_gridModelReal0*, Vector<S32>& *v_gridModelInt0* is replaced with UBEnv& *ubEnv0* and Vector<REAL>& *v_gridPhi1*, Vector<REAL>& *v_gridModelReal1*, Vector<S32>& *v_gridModelInt1* is replaced with UBEnv& *ubEnv1*. This affects *updateIfSubgridBetaInIfRegion()* and *compute MechIntrctSpAgent()*.
  - Vector<REAL>& *v_gridModelReal*, Vector<S32>& *v_gridModelInt* is replaced with

UBEnvModelVar& *ubEnvModelVar*. This affects *adjustIfSubgridRHSLinear()*, *updateIfSubgridRHSTimeDependentSplitting()*, *updateIfGridDirichletBCVal()*, and *updateIfGridNeumannBCVal()*.

- NbrBox<const UBAgentData*>& *ubAgentDataPtrNbrBox* is replaced with NbrUBAgentData& nbrUBAgentData. This affects *updateIfGridVar()* (Section 4.5.3), *updateIfGridAMRTags()* (Section 4.5.12), and *updateSummaryVar()* (Section 4.6.2).

- VIdx& *subgridVOffset* is added to the input argument list of *updateIfSubgridKappa()*, *updateIfSubgridAlpha()*, *updateIfSubgridBetaPDEBufferBdry()*, *updateIfSubgridBetaDomainBdry()*, *updateIfSubgridRHSLinear()*, *adjustIfSubgridRHSTimeDependentLinear()*, and *updateIfSubgridRHSTimeDependentSplitting()*. VIdx& *subgridVOffset0* and VIdx& *subgridVOffset1* are added to *updateIfSubgridBetaInIfRegion()*.

- In the simulation configuration file (Section 6.3), *pvtu* is replaced with *pvtp* and *vtm* is replaced with *vthb*. *time_stamp* is added to *stdout* and *max_loadimbalance* is removed from *system*.

Miscellaneous

- Descriptions of *numBoolVars* and *v_odeNetInfo* member variables (SpAgentInfo class, Section 2.4.8) are missing in the *Biocellion* 1.1 User Manual. We added descriptions of these member functions in this manual.

- Descriptions of *getBoolVal()*, *getBoolValArray()*, *setBoolVar()*, and *setBoolVarArray()* member functions (SpAgentState class, Section 2.4.19) are missing in the *Biocellion* 1.1 User Manual. We added descriptions of these member functions in this manual.

- Modeling examples are added and removed. See Section 5.

Contents

List of Figures

1.0 Introduction

Many biological phenomena arise as the product of complex interactions in living systems of cells. These *emergent behaviors* are evident only once these systems grow to millions to trillions of cells. Understanding and predicting emergent behaviors is crucial to solving important problems in *energy* (Ha et al. 2011, Majors et al. 2008, Melnicki et al. 2013), *environment* (Daly 2000, Singh et al. 2011), *healthcare* (Hall-Stoodley et al. 2004, Weinberg 2007, Rubinstein et al. 2013) and other biotechnology areas. Computational biologists iterate modeling and simulation, refining their models until simulation results match those of experiments. *Biocellion* provides fast simulation of living systems needed to facilitate research into emergent behaviors.

Computer simulation of biological systems requires multiple steps. Computational biologists must acquire biological knowledge or generate hypotheses on various aspects of biological system behaviors; build a mathematical model based on this biological knowledge with detail sufficient to test their hypotheses; translate the mathematical model into a computer program; run the program on computers; and visualize and analyze the simulation results.

While an ongoing challenge to this process is to gather and integrate sufficient biological knowledge to build high fidelity models, biologists have been studying the behavior of individual cells and subcellular organelles for a long time and have accumulated a great amount of knowledge about how cells and subcellular organelles behave. The most natural way to build mathematical models based on this accumulated knowledge is to treat each cell (or a subcellular organelle) as a discrete simulation entity in combination with adoption of high resolution grids to model the environment. However, simulating these models—often referred to as discrete agent-based modeling—is considered by some to be computationally intractable for the number of cells needed to study emergent behaviors of large living systems (Kim et al. 2007, Stamatakos 2010). More optimistic researchers have envisioned such simulations would be feasible *"on a supercomputer when a proper parallel implementation is used (Jiao and Torquato 2011)."*

We side with the latter researchers. *Biocellion* is a software framework implemented by high-performance computing (HPC) experts targeting large-scale systems-biology simulations and running on parallel computers ranging from laptops to supercomputers. Efficient algorithms, careful performance tuning, and exploitation of parallelism at multiple levels increases computing capability well beyond that of general simulation frameworks, enabling simulation of high-fidelity models at large scale, often several orders of magnitude faster than on other simulation platforms. *Biocellion* relieves computational biologists of the burdens posed by performance and scalability, freeing them to focus instead on the modeling task itself.

Biological systems are diverse and highly complex, and there are not yet any standards for mathematical modeling flexible enough to express the expansive biological knowledge integrated into component models. Mathematical models depend on the specifics of the biological systems, questions being answered, required level of accuracy, availability of biological knowledge, and model developer's intellectual capability and experience. Software designed for a single mathematical model will have only a short life span. Each model is unique in its design. While the need for generality in modeling is therefore paramount, we have noticed that many multicellular system

models exhibit similar computational challenges. We have studied a wide range of biological systems and analyzed numerous biological system models to separate individual model specifics from common computational challenges. We have abstracted out and integrated into the *Biocellion* framework what is common to a wide spectrum of living systems, automating for example the parallel solution of partial differential equations and propagation of physical contact forces; while also providing developers flexible interfaces to express model specifics, such as growth rates and adhesion properties. ***Biocellion* provides model developers with flexibility in expressing model specifics while relieving them of the burdens of parallel computation and performance optimization.**

Today, modelers express these model specifics as program functions that *Biocellion* invokes when it runs. This provides modelers with the flexibility of a general purpose programming language when expressing model specifics without significant loss in computational efficiency. That is, **model developers express model specifics as sequential code** (`C++` code using only basic `C++` features, Section 8 explains `C++` concepts relevant to *Biocellion*). **The *Biocellion* core framework links to this `C++` model library at runtime, invoking functions as needed to integrate the model specifics.** Realizing that most computational biologists are not `C++` experts, we envision adding a productivity layer on top of the current version of *Biocellion* once mathematical modeling of biological systems matures sufficiently. Other possibilities include enabling modelers to use other languages (eg, Python) with which they may be more comfortable, at some cost to performance but no loss in scalability.

1.1 Simulation Components

***Biocellion* simulates biological systems with a large number of cells, capturing individual cell behaviors, interactions between cells, and interactions of cells with their biotic and abiotic environment.**

The current version of *Biocellion* maps each cell to a discrete agent. In future versions, *Biocellion* will allow users to map a cell to multiple discrete agents as illustrated in (Newman 2005, Sandersius et al. 2011). Discrete agents can also represent other non-cellular biological elements—*e.g.* Xavier *et al.* represent extracellular matrix using discrete agents in (Xavier et al. 2005). Each discrete agent has associated variables representing its position and state. *Biocellion* users set the number of discrete agent types and the number of discrete agent state variables (or attributes) for each discrete agent type. Each agent can be placed anywhere in the simulation domain (unless restricted by the mathematical model) to within the limits of floating point precision. Individual agents move, divide, disappear, and secrete new discrete agents. *Biocellion* considers each discrete agent as a sphere by default and each discrete agent has *radius* as a state variable. Optionally, attributes may be added to a discrete agent to define a non-spherical shape. Figure 1.1 provides an example. Four attributes (three for direction and one for height) are added to treat a discrete agent as a cylinder. The *Biocellion* core framework does not use radius or other discrete agent state variables internally to update the state of a simulated biological system.

Discrete agents may interact with each other directly and indirectly. Direct interaction includes short-range physico-mechanical interaction between neighboring discrete agents such as cell-cell adhesion and shoving; or immune cells engulfing other discrete agents (*phagocytosis*). We assume

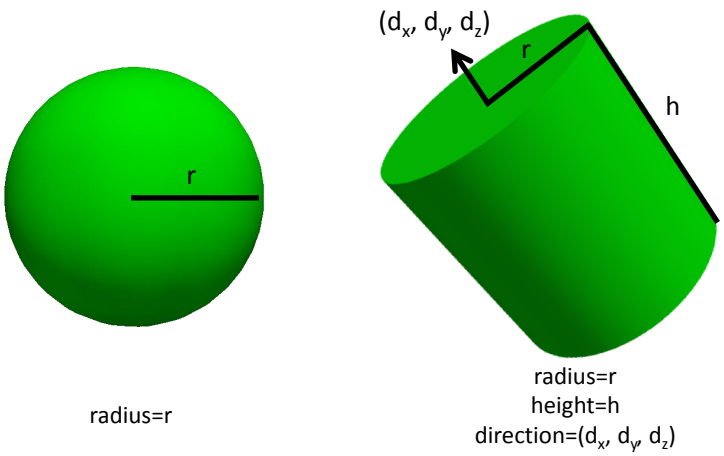

Figure 1.1. Discrete agent shapes.

the radius of direct interaction is relatively short (*e.g.* within a small multiple of cell diameter). Indirect interactions result from the secretion of diffusible molecules and cells uptaking the secreted diffusible molecules. Cell surface molecules (receptors) also bind to signaling molecules secreted by other cells.

Biocellion users set the maximum direct physico-mechanical interaction distance for each discrete agent type. The maximum interaction distance between two discrete agents with different types is computed by averaging the two maximum direct interaction distances for the two types. *Biocellion* invokes model routines that evaluate physico-mechanical interaction between a cell pair only for discrete agent pairs within the maximum direct interaction distance.

Physico-mechanical interactions between cell pairs often depend on past interactions. Once two cells form an adherens junction, pulling two cells apart requires stronger force. *Biocellion* users can explicitly form and break a junction between a pair of neighboring discrete agents (Figure 1.2). Each discrete agent has a set of junction ends (one for each junction), and each junction end has associated state variables. *Biocellion* asks users to set the number of junction end types and the number of attributes (or state variables) for each junction end type. Every junction automatically breaks if the distance between a pair of discrete agents exceeds the user provided maximum direct interaction distance for the pair.

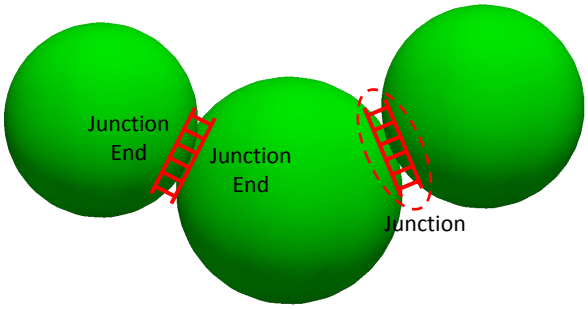

Figure 1.2. A junction between two discrete agents.

Associated with each discrete agent is a set of user-defined variables storing its temporary mechanical interaction state. These variables are updated repeatedly to reflect mechanical interactions exerted between every pair of interacting agents at a rate of zero or more times per *baseline time step* (time steps are explained in Section 1.3). Their values are used in turn to set discrete agent displacements and to update agent states based on model specific rules.

The surrounding environment influences biological system behavior. The surrounding environment affects the transport of molecules in the extracellular space and promotes and limits cell movement (*e.g.* *chemotaxis*, cells moving towards either the upward or downward gradient of molecular concentrations, promotes cell movement, and soil aggregate structure constrains cell movement paths (Resat et al. 2011)).

Biocellion provides two different mechanisms to update environment state variables (*e.g.* molecular concentrations or other extracellular space attributes). One is editing state variables by model specific rules and the other is by solving partial differential equations (PDEs). *Biocellion* users also mark sub-regions of the simulation domain as uninhabitable, and discrete agents cannot penetrate into an uninhabitable region (Figure 1.3).

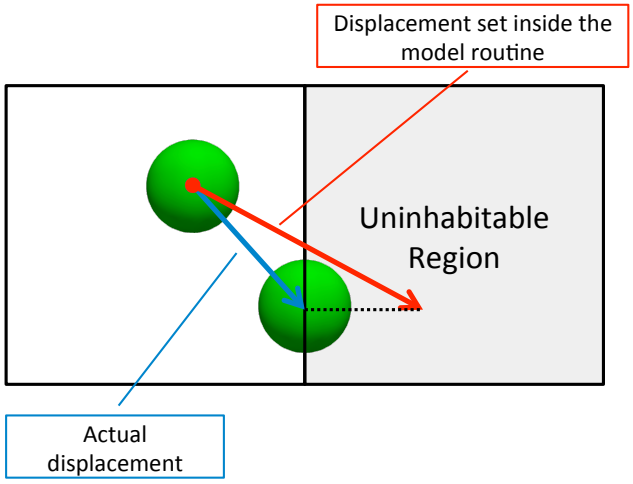

Figure 1.3. A discrete agent cannot move into an uninhabitable region. *Biocellion* considers only the position of a discrete agent in computing the displacement and does not internally use other discrete agent attributes (*e.g. radius*). By default, *Biocellion* allows any discrete agent of non-zero volume to overlap uninhabitable regions. If this is undesirable, the modeler can include code to test the agent's physical position and orientation, adjusting the displacement so as to move the agent fully out of uninhabitable regions.

Biocellion has three main computational modules to simulate the above simulation components. **The three modules 1) update the state of individual agents, 2) simulate short-range physico-mechanical interactions, and 3) update the state of the extracellular environment.** Computational modules communicate with other modules to simulate biological systems in their entirety.

1.2 Simulation Domain

Biocellion supports a three-dimensional rectangular simulation domain. *Biocellion* assumes a rectangular grid (the *interface grid*) with a fixed grid spacing (the *interface grid spacing*). **The interface grid spacing defines the default resolution of extracellular space representation.** *Biocellion* users set the interface grid spacing, h. Different computational modules communicate through this interface grid (See Figures 1.4 and 1.5). For each grid aligned $h \times h \times h$-box (*interface grid unit box*) *Biocellion* maintains a list of agents located within the box. **The interface grid spacing should be equal to or larger than the maximum direct physico-mechanical interaction distance between any two discrete agents.** This assures that a discrete agent directly interacts only with discrete agents in the same box or the neighboring 26 boxes.

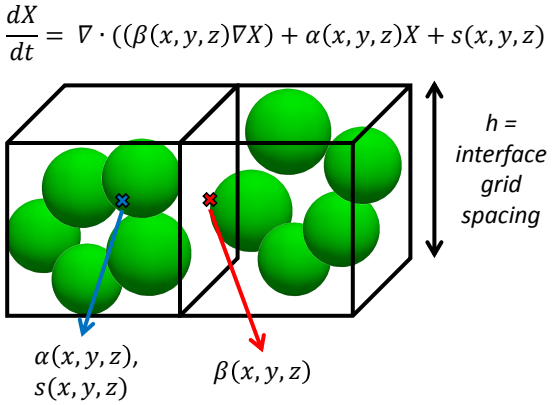

Figure 1.4. *Biocellion* asks users to set partial differential equation (PDE) parameters. *Biocellion* supports adaptive mesh refinement (AMR, Section 1.4) to solve PDEs. In *Biocellion* 1.1, the finest level resolution coincides with the interface grid resolution. From *Biocellion* 1.2, the finest level can have finer grid spacing than the interface grid spacing. *Biocellion* assumes that the interface subgrid is imposed on the interface grid, and the interface subgrid spacing coincides with the finest level grid spacing. *Biocellion* asks users to set PDE parameters in the interface subgrid. For reaction-diffusion PDEs, users set the decay rate (α) and the source term (s) for every interface subgrid unit box and the diffusion coefficient (β) for every face between two adjacent interface subgrid unit boxes sharing a face. User provided model routines set PDE parameters for an interface subgrid unit box based on the state of the interface grid unit box (which conincides with the interface subgrid unit box or contains the interface subgrid unit box) and the states of the discrete agents located in the interface grid unit box. For PDE parameters set for the face between two adjacent interface subgrid unit boxes, model routines set the parameters based on the states of the two interface grid unit boxes (each interface grid unit box coincides with or contains the corresponding interface subgrid unit box, and if the face is located strictly inside a single interface grid unit box, the two interface grid unit boxes coincide) and the states of the discrete agents residing in the two interface grid unit boxes.

Biocellion allows users to set sub-regions of the simulation domain as a partial differential equation (PDE) buffer; the PDE buffer is relevant only when solving PDEs. No discrete agent is allowed to

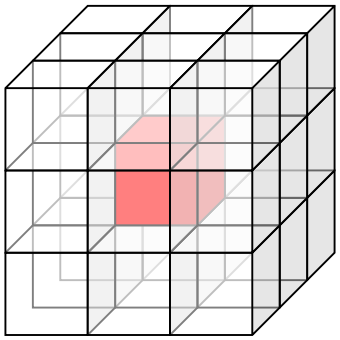

Figure 1.5. A model routine updating the state of a discrete agent can access the state of the interface grid unit box the agent is located in (red) and its 26 neighboring unit boxes (white).

reside in the PDE buffer. Say we are simulating a yeast colony growing on top of a thick agarose cylinder. The agarose cylinder holds nutrients and other molecules secreted by yeast cells. The agarose cylinder region needs to be included in simulation to faithfully track molecular concentration changes in the extracellular space. However, if no cells reside in the agarose cylinder region, maintaining data structures for all three computational modules at the resolution of the interface grid spacing can waste a significant amount of computing and memory. Users can set sub-regions of the simulation domain as a PDE buffer and adopt a coarser grid spacing for the buffer—*Biocellion* supports adaptive mesh refinement (AMR, Section 1.4) to solve PDEs and the PDE buffer grid spacing will be matched to the coarsest AMR level grid spacing. Figure 1.6 provides an example.

The simulation domain is decomposed into multiple partitions, multiple compute processes on a cluster computer work on different sets of partitions. Note that a single multithreaded compute process can exploit multiple cores on a compute node to work on a single partition. *Biocellion* users set the partition size. *Biocellion* outputs simulation results for different partitions in different files. If *Biocellion* restarts from checkpoint data, *Biocellion* reads separate checkpoint files corresponding to distinct partitions. Users can set each partition as a normal partition or a PDE buffer partition.

1.3 Time Step Sizes

Biocellion has three main computational modules with distinct time steps. Communication intervals also vary for different pairs of computational modules.

The *baseline time step* is the largest time step used by *Biocellion*. *Biocellion* uses this time step size to simulate direct physico-mechanical interactions between discrete agent pairs and to simulate discrete agent movement, birth, death, and secretion. The computational module simulating direct short range interactions communicates with the other two modules once every *baseline time step*.

Cell state changes affect the secretion and uptake rates of extracellular molecules. Molecular concentration changes in the extracellular space drive cell state changes as well. *Biocellion* allows

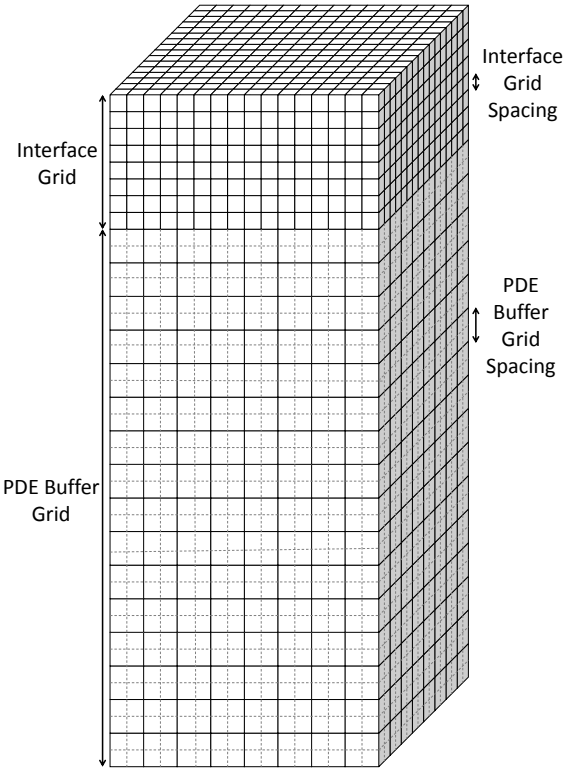

Figure 1.6. Simulation domain

users to couple these two processes more tightly. Users can split a *baseline time step* into one or more *state-and-grid time steps*. The cell state update module and the extracellular space state update module communicate once every *state-and-grid time step*.

A single *state-and-grid time step* can be further partitioned to solve partial differential equations (PDEs). The next section (Section 1.4) discusses time steps used to solve PDEs in more detail.

1.4 Partial Differential Equation (PDE) Solvers

The current version of *Biocellion* provides solvers for three different types of PDEs: steady-state linear reaction-diffusion PDEs, time-dependent linear reaction-diffusion PDEs, and time-dependent reaction-diffusion PDEs adopting the splitting scheme (Strang 1968) to update the reaction part. Future releases of *Biocellion* will support advection-reaction-diffusion and Navier-Stokes PDEs.

Biocellion uses CHOMBO (Colella et al. 2012) as an internal solver for PDEs and uses the Intel ordinary differential equation (ODE) solver (Intel Corporation 2008) to solve the reaction part of the splitting PDE. We briefly describe PDE solver features relevant to *Biocellion* users in this section and refer readers to (Colella et al. 2012, Intel Corporation 2008) for additional details.

The steady-state PDE and the time-dependent linear PDE update concentrations of a single molecular species. The time-dependent PDE based on the splitting scheme update the concentrations of

one or more molecular species reacting in the extracellular space.

Biocellion supports adaptive mesh refinement (AMR) to solve PDEs using CHOMBO (Colella et al. 2012). Users can set the number of AMR levels and select either 2 or 4 as the refinement ratio between two consecutive levels. Users mark the desired AMR level for each interface grid unit box. The coarsest AMR level is assumed for the PDE buffer. The actual AMR grid may use a finer grid spacing than the grid spacing specfied by the user for some regions to satisfy necessary conditions to maintain numerical efficiency (*e.g.* to limit the number of boxes, because processing a large number of tiny boxes can be computationally inefficient) and correctness (*e.g.* see the proper nesting condition in the CHOMBO manual). From *Biocellion* 1.2, there can be finer levels than the interface grid level.

A variant of the second order accurate L_0 stable Runge-Kutta scheme (see the CHOMBO manual and the Twizell *et al.*'s paper (Twizell et al. 1996) for additional details) is used for time stepping. This often allows *Biocellion* users to adopt a larger time step size than explicit methods for stiff PDEs. A single *PDE time step* can be set to a smaller value than the *state-and-grid time step* by partitioning a single *state-and-grid time step* to a positive integer number of *PDE time steps*.

To solve the splitting PDE, *Biocellion* solves ODEs for the reaction part for the duration of a single *PDE time step* first and sets the PDE source term based on the changes. Then the resulting linear reaction-diffusion PDE is solved for one or more *diffusion time steps*—a single *PDE time step* can be further partitioned to multiple *diffusion time steps* when the splitting scheme is used. *Biocellion* uses a variable time step size based on error estimation to solve ODEs (following the scheme used in the Intel ODE solver), and *Biocellion* users set the minimum ODE time step size. The single *PDE time step* size becomes the maximum time step size to solve ODEs.

1.5 Future Roadmap

Over time, we plan to update *Biocellion* to accommodate an ever broadening spectrum of biological system models. We plan to extend *Biocellion* to support mapping a single cell to multiple discrete agents and to provide advection-reaction-diffusion and Navier-Stokes PDE solvers.

Mapping a cell to more than one discrete agent will allow modeling of cell shapes and mechanical interactions between cells with higher fidelity. For example, mapping a budding yeast cell to two discrete agents more realistically reproduces the shape of a real budding yeast cell (Figure 1.7).

Updating individual cell states is largely a users' task at this point. *Biocellion* provides a solver for deterministic ordinary differential equation (ODE) systems, but there are many other approaches to modeling cell regulatory networks. We plan to add solvers for widely used regulatory network modeling approaches (*e.g.* Boolean network, stochastic ODE systems, and hybrids). For a cell mapped to multiple discrete agents, we can consider each discrete agent as a cell compartment or a subcellular organelle. Another path to extend *Biocellion* is to provide solvers to update the state of subcellular discrete agents or to support modeling of molecular transport across subcellular compartments.

Computing the flow rate of liquid in biological systems with time varying structures (*e.g.* water

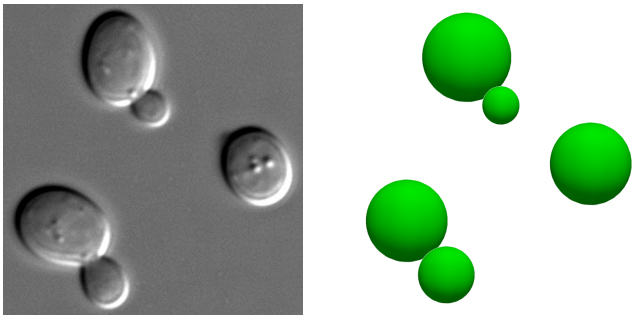

Figure 1.7. Modeling budding yeast using two discrete agents per cell. The budding yeast image (left) is reproduced from `http://en.wikipedia.org/wiki/Yeast`.

flow in soil aggregate or blood flow in human tissue) is another important computational task in building high fidelity models. In computing blood flow rate, our current interest is mainly in tracking long term average flow rate changes in capillaries (which affect the delivery rate of nutrients and vessel stabilization and regression) to model tissue or vascular tumor growth rather than modeling short term dynamics (*e.g.* flow changes within a single heart beat cycle) of blood flow in arteries which is more relevant in studying cardiovascular disease.

2.0 *Biocellion* Data Types

2.1 Basic Data Types

Biocellion redefines the following basic data types to promote portability[(a)] across different platforms and to support both single-precision and double-precision floating point arithmetic without code modification.

- **REAL**: Either single or double precision floating point number (float or double) based on compile time setup.
- **U8, U16, U32, U64**: 8, 16, 32, and 64-bit unsigned integers, respectively.
- **S8, S16, S32, S64**: 8, 16, 32, and 64-bit signed integers, respectively.
- **BOOL**: Boolean data type (true or false).

BOOL replaces the `C++` bool data type. *Biocellion* uses BOOL instead of bool to prevent the `C++` standard template library (STL) vector from packing multiple Boolean variables (up to 8) into a single byte. Such packing reduces memory consumption but requires additional bitwise operations to access and modify vector elements. If vector elements are concurrently updated by multiple threads, updating two different elements packed into the same byte requires locking (or using other data synchronization primitives); we expect users to write only sequential code, so data synchronization is not an issue for *Biocellion* users, but we define the BOOL type to avoid additional data type conversion overhead between user routines and the *Biocellion* framework code.

2.2 Extra Scalar Data Types

Biocellion defines additional scalar data types to maintain portability across different platforms and save memory.

- **agentType_t**: 16 bit unsigned integer by default. Every agent has an associated type. *Biocellion* users assign different numbers of model specific REAL type and S32 type variables (or discrete agent attributes) for different agent types.
- **junctionEndType_t**: 16 bit unsigned integer by default. Every junction end (Section 1.1) has an associated type. *Biocellion* users assign different numbers of model specific REAL type and S32 type variables (or junction end attributes) for different junction end types.
- **agentId_t**: 64 bit signed integer by default. Every discrete agent has a unique ID.
- **idx_t**: 16 bit signed integer by default. Three idx_t type variables are used to index an interface grid unit box (the size of a unit box is $h \times h \times h$ assuming that the interface grid

(a) The current version of *Biocellion* runs only on x86 compatible architectures. This may change in the future.

spacing is h) or an interface subgrid unit box within an interface grid unit box.

- **ubAgentIdx_t**: 16 bit unsigned integer by default. This data type is used to index discrete agents in a single interface grid unit box.

We assume that the default sizes of the above data types are large enough for most practical applications. If this does not hold, users need to redefine the above data types and recompile the entire *Biocellion* framework and model libraries. Advanced users can redefine the above data types to a smaller integer data type to save memory if the smaller data type is sufficient for their applications.

2.3 Enumerations

Biocellion has multiple enumerated data types which are used to provide model specifics (see Section 4 for the user interface to provide model specifics).

- **domain_bdry_type_e**: Used to set the simulation domain boundary types in the x, y, and z directions.
  - `DOMAIN_BDRY_TYPE_NONPERIODIC_HARD_WALL`: Non-periodic boundary condition is assumed. In solving PDEs, users can apply either Dirichlet or Neumann boundary conditions (for the low and high sides). Discrete agents cannot pass the domain boundary.
  - `DOMAIN_BDRY_TYPE_PERIODIC`: Periodic boundary condition is assumed for PDEs. Discrete agents passing the domain boundary will appear at the other end of the simulation domain.

- **pde_buffer_bdry_type_e**: Used to set the boundary type between an interface grid region and a PDE buffer region. This boundary type is only related to discrete agent movement and irrelevant to PDE solves.
  - `PDE_BUFFER_BDRY_TYPE_HARD_WALL`: This is the only valid option at this point. Discrete agents cannot penetrate into a PDE buffer region (PDE buffer regions are automatically marked as uninhabitable).

- **pde_type_e**: Used to set PDE types.
  - `PDE_TYPE_REACTION_DIFFUSION_STEADY_STATE_LINEAR`: Used to find a (quasi) steady state solution of a reaction-diffusion linear PDE. This option assumes that the PDE reaches steady state within a single *state-and-grid time step*.
  - `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`: Use time-stepping to find a solution of a reaction-diffusion linear PDE.
  - `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_SPLITTING`: Use time-stepping to find a solution of a reaction-diffusion PDE. The time splitting technique (Strang 1968) is applied to simulate reactions among multiple molecular species in the extracellular space.

- PDE_TYPE_ADVECTION_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR: Reserved for future use.

- PDE_TYPE_ADVECTION_REACTION_DIFFUSION_TIME_DEPENDENT_SPLITTING: Reserved for future use.

- PDE_TYPE_INCOMPRESSIBLE_NAVIER_STOKES_TIME_DEPENDENT: Reserved for future use.

- **bc_type_e**: Used to set PDE boundary condition types.
  - BC_TYPE_DIRICHLET_CONST: Apply Dirichlet boundary condition with a constant boundary value for the entire region of the domain boundary face at the low or high side in the x, y, or z direction.

  - BC_TYPE_DIRICHLET_MODEL: Apply Dirichlet boundary condition, and boundary values are set in the granularity of a grid unit box face in an adaptive mesh refinement (AMR) level.

  - BC_TYPE_NEUMANN_CONST: Apply Neumann boundary condition with a constant boundary value for the entire region of the domain boundary face at the low or high side in the x, y, or z direction.

  - BC_TYPE_NEUMANN_MODEL: Apply Neumann boundary condition, and boundary values are set in the granularity of a grid unit box face in an adaptive mesh refinement (AMR) level.

- **ode_stiff_e**: Inform the stiffness of the ODE system to solve. *Biocellion* uses different mathematical algorithms based on the provided information using the Intel ODE solver (Intel Corporation 2008).
  - ODE_STIFF_NORMAL: Indicate that the ODE system has low or medium stiffness. The dodesol_rkm9st function in the Intel ODE library (which is *"based on the fourth order Merson's method and the first order multistage method of up to and including 9 stages with stability control (Intel Corporation 2008)"*) is used to solve the ODE system.

  - ODE_STIFF_HIGH: Indicate that the ODE system is stiff. The dodesol_mk52lfn function in the Intel ODE library (which uses *"the implicit method based on L-stable(5,2)-method with the numerical Jacobi matrix, which is computed by the routine (Intel Corporation 2008)"*) is used to solve the ODE system.

- **sync_method_e**: *Biocellion* users can update temporary mechanical interaction state variables of a discrete agent in multiple function calls evaluating direct interactions between the discrete agent and its neighboring discrete agents (*computeMechIntrct()*)—one function call per pair of discrete agents. *Biocellion* provides a model routine to edit grid state variables using model specific rules (*udpateIfGridVar()*), and the routine is called once per interface grid unit box. A single invocation of the model routine can update variables associated with

an interface grid unit box and its neighboring 26 unit boxes. A single variable can be updated by multiple model routine calls. This requires a mechanism to coordinate the multiple updates (often referred as *data synchronization*). From *Biocellion* 1.2, users are asked to set the coordination method (or conflict resolution scheme) for every attribute. sync_method_e should be set to `SYNC_METHOD_PER_ATTR` in *Biocellion* 1.2 and var_sync_method_e is used to set the coordination method for each attribute when a single variable is updated in multiple function calls.

  - `SYNC_METHOD_PER_ATTR`: Set the coordination method for each attribute.
  - `SYNC_METHOD_TRANSACTIONAL`: Reserved for future use.

- **var_sync_method_e**: This enumeration type is used to set the coordination method for each attribute when a single varialbe is updated in multiple function calls.
  - `VAR_SYNC_METHOD_OVERWRITE`: If this option is selected, only one of the multiple updates is reflected—*e.g.* if three different model routines set the value of variable *a* to 3, 5, and 9, respectively, *a* is set to either 3, 5, or 9 with equal probability.
  - `VAR_SYNC_METHOD_DELTA`: If this option is selected, the differences from the initial value are summed—*e.g.* if three different model routines set the value of variable *a* to 3, 5, and 9, respectively, then *a* is set to 3 + 5 + 9 (assuming that the initial value is 0).
  - `VAR_SYNC_METHOD_MIN`: If this option is selected, the minimum value is selected– *e.g.* if three different model routines set the value of variable *a* to 3, 5, and 9, respectively, then *a* is set to 3.
  - `VAR_SYNC_METHOD_MAX`: If this option is selected, the maximum value is selected– *e.g.* if three different model routines set the value of variable *a* to 3, 5, and 9, respectively, then *a* is set to 9.

- **if_grid_var_type_e**: Used to set the type of a variable associated with an interface grid unit box.
  - `IF_GRID_VAR_TYPE_VEL`: Reserved for future use.
  - `IF_GRID_VAR_TYPE_PHI`: Set the type to a REAL variable storing a molecular concentration value which is mainly updated by solving PDEs. *Biocellion* updates variables of this type both when solving PDEs and when inside the model routine updating grid state variables based on model specific rules. Model specific REAL and S32 type variables are updated only inside the model routine updating grid state variables based on model specific rules.
  - `IF_GRID_VAR_TYPE_MODEL_REAL`: Set the type to a model specific REAL type variable.
  - `IF_GRID_VAR_TYPE_MODEL_INT`: Set the type to a model specific S32 type variable.

- **sim_init_type_e**: Used to set the simulation initialization method.

- SIM_INIT_TYPE_CODE: Initialize inside the model routines.

- SIM_INIT_TYPE_BINARY: Initialize using checkpoint data.

- **particle_output_type_e**: Used to set the output file format for discrete agents.
  - PARTICLE_OUTPUT_TYPE_PTVP: Set the output file format to the parallel vtkPolyData file format (.ptvp). This is the only valid option at this point.

- **grid_output_type_e**: Used to set the output file format for grid values.
  - GRID_OUTPUT_TYPE_VTHB: Set the output file format to the vtkOverlappingAMR file format (.vthb). This is the only valid option at this point.

- **summary_type_e**: *Biocellion* provides a summary (reduction) mechanism for interface grid summary attributes. Users define a fixed number of summary attributes, and *Biocellion* assigns summary variables to every interface grid unit box (one variable per attribute). Users update summary variables, and *Biocellion* reduces the entire set of vairables for each attribute visiting every interface grid unit box. summary_type_e is used to specify the reduction method for an attribute.
  - SUMMARY_TYPE_SUM: Ask to print the sum of the entire set of interface grid summary variables for an attribute.

  - SUMMARY_TYPE_AVG: Ask to print the average of the entire set of interface grid summary variables for an attribute.

  - SUMMARY_TYPE_MIN: Ask to print the minimum value of the entire set of interface grid summary variables for an attribute.

  - SUMMARY_TYPE_MAX: Ask to print the maximum value of the entire set of interface grid summary variables for an attribute.

- **rng_type_e**: *Biocellion* provides random number generators—internally using Intel math kernel library (MKL). rng_type_e is used to set the type of a random number generator.
  - RNG_TYPE_UNIFORM: Ask to generate random numbers with the uniform distribution.

  - RNG_TYPE_GAUSSIAN: Ask to generate random numbers with the Gaussian distribution.

  - RNG_TYPE_EXPONENTIAL: Ask to generate random numbers with the exponential distribution.

  - RNG_TYPE_GAMMA: Ask to generate random numbers with the gamma distribution.

2.4 Classes

2.4.1 Vector

Vector is a wrapper class of the C++ standard template library (STL) vector—STL vector is a data type for a variable size array. We create this wrapper to perform array index bound checks (similar to CHOMBO (Colella et al. 2012)) without re-compiling both the *Biocellion* core framework and a model library. Users can enable the boundary check for a model library without enabling the check for the core framework (See Section 6.2.1). Visit `http://www.cplusplus.com/reference/vector/vector` for additional details about the STL vector.

2.4.2 VIdx

VIdx is used to index a grid box in the simulation domain. VIdx has an internal array of three idx_t type variables.

Public static member variables

- VIdx `ZERO`: A constant three dimensional zero vector—*i.e.* (0, 0, 0).
- VIdx `UNIT`: A constant three dimensional unit vector—*i.e.* (1, 1, 1).
- VIdx `A_BASIS[3]`: A set of three dimensional basis vectors. `A_BASIS[0]` = (1, 0, 0), `A_BASIS[1]` = (0, 1, 0), and `A_BASIS[2]` = (0, 0, 1).

Public non-static member functions

- $+$, $-$, $\times$, $/$: Arithmetic operators.
- $==$, $!=$, $<$, $<=$, $>$, $>=$: Comparison operators.
- [*idx*]: An access operator—returns the idx^{th} element of the internal array holding three idx_t type variable values. *idx* should be 0, 1, or 2.
- *VIdx& shift(const S32 dim, const idx_t offset)*: Shift this VIdx instance by *offset* along the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) axis. Return a reference to this VIdx instance as well.
- *VIdx& shift(const VIdx& vOffset)*: Shift this VIdx instance by *vOffset[0]*, *vOffset[1]*, and *vOffset[2]* along the x, y, and z axes, respectively. Return a reference to this VIdx instance as well.
- *VIdx& diagShift(const idx_t offset)*: Shift this VIdx instance by *offset* along the x, y, and z axis. Return a reference to this VIdx instance as well.
- *VIdx& coarsen(const idx_t ratio)*: Coarsen this VIdx instance by *ratio*. Return a reference

to this VIdx instance as well.

- *VIdx& coarsen(const VIdx& vRatio)*: Coarsen this VIdx instance by *vRatio[0]*, *vRatio[1]*, and *vRatio[2]* in the x, y, and z dimensions, respectively. Return a reference to this VIdx instance as well.

- *VIdx& min(const VIdx& vIdx)*: Set the value of this VIdx instance in each dimension to the smaller value of this VIdx instance's value and the value of the input argument *vIdx*. Return a reference to this VIdx instance as well.

- *VIdx& max(const VIdx& vIdx)*: Set the value of this VIdx instance in each dimension to the bigger value of this VIdx instance's value and the value of the input argument *vIdx*. Return a reference to this VIdx instance as well.

- *void setVal(const S32 dim, const idx_t val)*: Set the VIdx instance's value in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) dimension to *val*.

- *S32 product(void) const*: Return the product of this VIdx instance's value in the x, y, and z dimensions.

- *BOOL lexLT(const VIdx& vIdx) const*: Lexicographically less than. Starting from the x dimension, retunrn true if this VIdx instance's value is less than the *vIdx*'s value and false if larger. On par, move to the next dimension. Retunr false if this VIdx instance and *vIdx* have identical values in the x, y, and z dimensions.

- *BOOL lexGT(const VIdx& vIdx) const*: Lexicographically greater than. Starting from the x dimension, return true if this VIdx instance's value is greater than the *vIdx*'s value and false if smaller. On par, move to the next dimension. Retunr false if this VIdx instance and *vIdx* have identical values in the x, y, and z dimensions.

Public static member functions

- *VIdx mult(const VIdx& vIdx, const idx_t scale)*: Return $vIdx \times scale$.

- *VIdx diagShift(const VIdx& vIdx, const idx_t offset)*: Return *vIdx.diagShift(offset)*.

- *VIdx coarsen(const VIdx& vIdx, const idx_t ratio)*: Return *vIdx.coarsen(ratio)*.

- *S64 getIdx3Dto1D(const VIdx& vIdx, const VIdx& regionSize)*: Return $vIdx[0] \times regionSize[1] \times regionSize[2] + vIdx[1] \times regionSize[2] + vIdx[2]$.

- *S64 getIdx3Dto1D(const VIdx& vIdx, const idx_t size)*: Return $vIdx[0] \times size \times size + vIdx[1] \times size + vIdx[2]$.

- *S64 getIdx3DTo1D(const idx_t i, const idx_t j, const idx_t k, const VIdx& regionSize)*: Return $i \times regionSize[1] \times regionSize[2] + j \times regionSize[2] + k$.

- *S64 getIdx3DTo1D(const idx_t i, const idx_t j, const idx_t k, const idx_t size)*: Return $i \times size \times size + j \times size + k$.

- *VIdx getIdx1DTo3D(const S64 idx, const VIdx& regionSize)*: Perform the reverse operation of *getIdx3DTo1D(const VIdx& vIdx, const VIdx& regionSize)*.

- *VIdx getIdx1DTo3D(const S64 idx, const idx_t size)*: Perform the reverse operation of *getIdx3DTo1D(const VIdx& vIdx, const idx_t size)*.

2.4.3 VReal

VReal is a container for three REAL type variables.

Public static member variables

- VReal `ZERO`: A constant three dimensional zero vector—*i.e.* (0.0, 0.0, 0.0).

- VReal `UNIT`: A constant three dimensional unit vector—*i.e.* (1.0, 1.0, 1.0).

- VReal `A_BASIS[3]`: A set of three dimensional basis vectors. `A_BASIS[0]` = (1.0, 0.0, 0.0), `A_BASIS[1]` = (0.0, 1.0, 0.0), and `A_BASIS[2]` = (0.0, 0.0, 1.0).

Public non-static member functions

- $+$, $-$, $\times$, $/$: Arithmetic operators.

- [*idx*]: An access operator—return the idx^{th} element of the internal array of three REAL type variables. *idx* should be 0, 1, or 2.

- *REAL length(void) const*: Return the length of this VReal instance (the distnace from the origin).

- *REAL lengthSquare(void) const*: Return the squared length of this VReal instance. This function is computationally cheaper than the above *length()* as this avoids computing a square root.

Public static member functions

- *VReal crossProduct(const VReal& vLeft, const VReal& vRight)*: Return the cross-product of *vLeft* and *vRight*.

- *VReal rotate(const VReal& vNormalizedAxisDir, const VReal& point, const REAL theta)*: Rotate *point* around the axis specified by *vNormalizedAxisDir* by *theta* (counter-clockwise, right-handed rule) and return the resulting VReal instance. This function expects *vNormalizedAxisDir* is already normalized.

- *VReal normalize(const REAL epsilon, const VReal& vUnnormalized)*: Normalize (*i.e.* rescale so the resulting VReal instance's length becomes 1.0) *vUnnormalized* and return the resulting VReal instance if the length of *vUnnormalized* is equal to or larger than *epsilon*. If *vUnnormalized*'s length is smaller than epsilon, return *VReal::ZERO*.

2.4.4 ODENetInfo

This class instance holds the information required to set-up a solver for a deterministic ordinary different equation (ODE) system modeling a discrete agent regulatory network.

Public non-static member variables

- S32 *numVars*: Set the number of variables in the ODE system.

- ode_stiff_e *stiff*: Set the stiffness of the ODE system (*Biocellion* uses different numerical algorithms based on this information).

- REAL *h*: Set the initial step size to $h \times$ *state-and-grid time step* size. $0.0 < h \leq 1.0$.

- REAL *hm*: Set the minimum time step size to $hm \times$ *state-and-grid time step* size. $0.0 < hm \leq h$.

- REAL *epsilon*: Set the relative error tolerance. The ODE solver cannot ensure required accuracy if *epsilon* is smaller than 1e-9.

- REAL *threshold*: If the absolute value of the solution is larger than *threshold*, relative error is controlled. Otherwise, absolute error (*threshold* $\times$ *epsilon*) is controlled.

- REAL *errorThresholdVal*: If any of the ODE system variables becomes smaller than *errorThresholdVal*, *Biocellion* prints an error message and aborts the program execution. *errorThresholdVal* should be equal to or smaller than 0.0. Set *errorThresholdVal* to REAL_MAX (defined in `$BIOCELLION_ROOT/include/base_type.h`) $\times$ -1.0 to turn this check off.

- REAL *warningThresholdVal*: If any of the ODE system variables becomes smaller than *warningThresholdVal* (but still no variable is smaller than *errorThresholdVal*), *Biocellion* prints a warning message (but does not abort the program execution). *warningThresholdVal* should be equal to or smaller than 0.0 and should be equal to or larger than *errorThresholdVal*. Set *warningThresholdVal* to REAL_MAX (defined in `$BIOCELLION_ROOT/include/base_type.h`) $\times$ -1.0 to turn this check off.

- BOOL *setNegToZero*: If *setNegToZero* is set to true, *Biocellion* automatically resets negative ODE variables (equal to or larger than *errorThresholdVal*) to 0.0. Set to false to turn this feature off.

2.4.5 TimeStepInfo

This class instance holds the information required to set the sizes of *baseline time step* and *state-and-grid time step*.

Public non-static member variables

- REAL *durationBaselineTimeStep*: Set the *baseline time step* size.

- S32 *numStateAndGridTimeStepsPerBaseline*: The *state-and-grid time step* size is set to *baseline time step* size divided by *numStateAndGridTimeStepsPerBaseline*.

2.4.6 OptModelRoutineCallInfo

Biocellion users evaluate mechanical interactions between agent pairs for 0 or more iterations per *baseline time step*. *Biocellion* users can update the interface grid state variables based on model specific rules. *Biocellion* optionally invokes the model routine updating interface grid state variables at the beginning and at the end of every *state-and-grid time step*. This class instance controls the optional model routine invocation.

Public non-static member variables

- S32 *numComputeMechIntrctIterations*: Evaluate mechanical interaction between agent pairs for *numComputeMechIntrctIterations* iterations per *baseline time step*. This variable should be zero or a positive integer.

- S32 *numUpdateIfGridVarPreStateAndGridStepIterations*: Call the *updateIfGridVar()* model routine (once for every interface grid unit box) for *numUpdateIfGridVarPreStateAndGridStepIterations* iterations at the beginning of every *state-and-grid time step*. This variable should be zero or a positive integer.

- S32 *numUpdateIfGridVarPostStateAndGridStepIterations*: Call the *updateIfGridVar()* model routine (once for every interface grid unit box) for *numUpdateIfGridVarPostStateAndGridStepIterations* iterations at the end of every *state-and-grid time step*. This variable should be zero or a positive integer.

2.4.7 MechModelVarInfo

This class instance holds the model specifics about a mechanical interaction state attrbiute.

Public non-static member variables

- var_sync_method_e *syncMethod*: Set the coordination scheme if a single variable is updated in multiple model routine invocations in a single iteration. See the descriptions about sync_method_e and var_sync_method_e in Section 2.3.

2.4.8 SpAgentInfo

This class instance holds the model specifics about a discrete agent type.

Public non-static member variables

- REAL *dMax*: Set the maximum direct physico-mechanical interaction distance. The value of this variable should be equal to or smaller than the interface grid spacing. The maximum direct interaction distance between two discrete agents having different types is computed by averaging the *dMax* values for the two types. In evaluating mechanical interactions between discrete agent pairs, every pair within the maximum interaction distance is enumerated.

- S32 *numBoolVars*: Set the number of model specific BOOL type state variables for this discrete agent type. Currently, updating these variables are solely users' responsibility.

- S32 *numStateModelReals*: Set the number of model specific REAL type state variables for this discrete agent type.

- S32 *numStateModelInts*: Set the number of model specific S32 type state variables for this discrete agent type.

- Vector<MechModelVarInfo> *v_mechIntrctModelRealInfo*: Provide the information about model specific REAL type temporary mechanical interaction state attributes for this discrete agent type.

- Vector<MechModelVarInfo> *v_mechIntrctModelIntInfo*: Provide the information about model specific S32 type temporary mechanical interaction state attributes for this discrete agent type.

- Vector<ODENetInfo> *v_odeNetInfo*: Provide the information about model specific deterministic ordinary differential equation (ODE) systems for this discrete agent type.

2.4.9 JunctionEndInfo

This class instance holds the model specifics about a junction end type.

Public non-static member variables

- S32 *numModelReals*: Set the number of model specific REAL type variables for this junction end type.

- S32 *numModelInts*: Set the number of model specific S32 type variables for this junction end type.

2.4.10 GridPhiInfo

This class instance holds the model specifics about the concentration values of a molecular species which are updated mainly by solving PDEs and optionally by the *udpateIfGridVar()* model routine.

Public non-static member variables

- S32 *elemIdx*: Set a unique index for this molecular species. *elemIdx* should be equal to or larger than 0 and smaller than the number of molecular species in the simulation instance.

- std::string *name*: Set the name of the molecular species.

- var_sync_method_e *syncMethod*: Set the coordination scheme if a single variable is updated in multiple model routine invocations in a single iteration. See the descriptions about sync_method_e and var_sync_method_e in Section 2.3.

- bc_tyep_e *aa_bcType[3][2]*: Boundary condition types for the x (*aa_bcType[0][]*), y(*aa_bcType[1][]*), and z(*aa_bcType[2][]*) directions and the low (*aa_bcType[][0]*) and high (*aa_bcType[][1]*) sides. Irrelevant if periodic boundary condition is applied.

- REAL *aa_bcVal[3][2]*: Set the constant boundary value for the low and high sides in the x, y, and z directions. Relevant only when the PDE boundary type is set to `BC_TYPE_DIRCHLET_CONST` or `BC_TYPE_NEUMANN_CONST`.

- REAL *errorThresholdVal*: If any of the PDE variables becomes smaller than *errorThresholdVal*, *Biocellion* prints an error message and aborts the program execution. *errorThresholdVal* should be equal to or smaller than 0.0. Set *errorThresholdVal* to REAL_MAX (defined in `$BIOCELLION_ROOT/include/base_type.h`) $\times$ -1.0 to turn this check off.

- REAL *warningThresholdVal*: If any of the PDE variables becomes smaller than *warningThresholdVal* (but still no variable is smaller than *errorThresholdVal*), *Biocellion* prints a warning message (but does not abort the program execution). *warningThresholdVal* should be equal to or smaller than 0.0 and should be equal to or larger than *errorThresholdVal*. Set *warningThresholdVal* to REAL_MAX (defined in `$BIOCELLION_ROOT/include/base_type.h`) $\times$ -1.0 to turn this check off.

- BOOL *setNegToZero*: If *setNegToZero* is set to true, *Biocellion* automatically resets negative PDE variables (equal to or larger than *errorThresholdVal*) to 0.0. Set to false to turn this feature off.

2.4.11 GridVelInfo

This is class is just a placeholder at this moment (*Biocellion* 1.2).

2.4.12 MGSolveInfo

This class instance holds the model specifics about the PDE multigrid solver.

Public non-static member variables

- S32 *numPre*: Set the number of smoothing steps before averaging in the multigrid solver. Set to 3 unless you wish to tune the multgrid solver based on your specific PDE problems.

- S32 *numPre*: Set the number of smoothing steps after averaging in the multigrid solver. Set to 3 unless you wish to tune the multgrid solver based on your specific PDE problems.

- S32 *numBottom*: Set the number of smoothing steps at the bottom level. Set to 3 unless you wish to tune the multgrid solver based on your specific PDE problems.

- BOOL *vCycle*: Set to true to use V-cycle or set to false to use W-cycle in the multigrid solver. Set to true unless you wish to tune the multgrid solver based on your specific PDE problems.

- S32 *maxIters*: Set the maximum number of V-cycles (or W-cycles). Se to 30 unless you wish to tune the multigrid solver based on your specific PDE problems (*e.g.* if your PDE converges very slowly, a larger value is more appropriate). If the PDE multigrid solver does not reach convergence within *maxIters* iterations, the solver exits prematurely and prints a warning.

- REAL *epsilon*: The PDE multigrid solver assumes solution convergence if the residual norm is equal to or smaller than the original norm multiplied by *epsilon*.

- REAL *hang*: Set the minimum required change in two consecutive V-cycles (or W-cycles). If the decrease in the norm (relative to the norm in the previous iteration) is less than *hang*, the PDE multigrid solver exits prematurely and prints a warning.

- REAL *normThreshold*: The PDE multigrid solver assumes solution convergence if the residual norm is equal to or smaller than *normThreshold*.

2.4.13 SplittingInfo

This class instance holds the information required to solve a PDE using the splitting scheme. A PDE can have multiple molecular species if the splitting scheme is used.

Public non-static member variables

- Vector<S32> *v_diffusionTimeSteps*: Set the number of diffusion time steps per *PDE time step* for each molecular species in the PDE.

- ode_stiff_e *odeStiff*: Set the stiffness of the ODE system for the PDE reaction part (*Biocel-*

lion uses different numerical algorithms based on this information).

- REAL *odeH*: Set the initial ODE time step size to *odeH* $\times$ *PDE time step* size. $0.0 < odeH \leq 1.0$.

- REAL *odeHm*: Set the minimum ODE time step size to *odeHm* $\times$ *PDE time step* size. $0.0 < odeHm \leq odeH$.

- REAL *odeEpsilon*: Set the relative error tolerance. The ODE solver cannot ensure required accuracy if *epsilon* is smaller than 1e-9.

- REAL *odeThreshold*: If the absolute value of the solution is larger than *threshold*, relative error is controlled. Otherwise, absolute error (*threshold* $\times$ *epsilon*) is controlled.

2.4.14 PDEInfo

This class instance holds the information required to solve a PDE.

Public non-static member variables

- S32 *pdeIdx*: Set a unique index for this PDE. *pdeIdx* should be equal to or larger than 0 and smaller than the number of PDEs in the simulation instance.

- pde_type_e *pdeType*: Set the type of the PDE.

- S32 *numLevels*: Set the number of adaptive mesh refinement (AMR) levels.

- S32 *ifLevel*: Set the interface grid level in the adpative mesh refinement (AMR) hierarchy.

- Vector<S32> *v_tagExpansionSize*: Set the amount to expand the tagged region in each level before generating an AMR hierarchy. *v_tagExpansionSize[0]* should be always 0.

- S32 *numTimeSteps*: Set the number of *PDE time steps* per *state-and-grid time step*.

- BOOL *callAdjustRHSTimeDependentLinear*: If set to true, *Biocellion* invokes the model routines (*adjustIfSubgridRHSTimeDependentLinear()* and *adjustPDEBufferRHSTimeDependentLinear()*) at the beginning of every *PDE time step* to fine tune the PDE reaction term. If set to false, *Biocellion* updates the PDE reaction part only once per *state-and-grid time step*. Relevant only when *pdeType* is set to `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`.

- MGSolveInfo *mgSolveInfo*: Set the PDE multigrid solver options. See Section 2.4.12 for additional details.

- AdvectionInfo *advectionInfo*: Reserved for future use. Irrelevant at this point.

- SplittingInfo *splittingInfo*: Set the information required to solve the PDE using the splitting scheme. Relevant only when *pdeType* is set to `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_SPLITTING`.

- Vector<GridPhiInfo> *v_gridPhiInfo*: Provide the information about molecular species in this PDE. If the splitting scheme is used, there can be more than one molecular species in a single PDE.

2.4.15 IfGridModelVarInfo

Biocellion allows users to assign a fixed number of model specific attributes to the interface grid (or a fixed number of REAL and S32 type state variables to every interface grid unit box). These variables are updated by model specific rules in the model routine (*udpateIfGridVar()*) optionally invoked at the beginning and the end of every *state-and-grid time step*. This class instance provides the information about a model specific attribute.

Public non-static member variables

- std::string *name*: Set the name of the attribute.

- var_sync_method_e *syncMethod*: Set the coordination scheme if a single variable is updated in multiple model routine invocations in a single iteration. See the descriptions about sync_method_e and var_sync_method_e in Section 2.3.

2.4.16 RNGInfo

Biocellion provides random number generators with the uniform, Gaussian, exponential, and gamma distributions. We assume that the probability density functions for the exponential and gamma distributions are $\frac{1}{\beta} \times e^{-x/\beta}$ and $\beta^{\alpha} \times \frac{1}{\Gamma(\alpha)} \times x^{\alpha-1} \times e^{-\beta \times x}$, respectively.

Public non-static member variables

- rng_type_e *type*: This variable sets the random number generator type.

- REAL *param0*: The minimum value if the uniform distribution is selected, the average value if the Gaussian distribution is selected, β if the exponential distribution is selected, and α if the gamma distribution is selected.

- REAL *param1*: The maximum value if the uniform distribution is selected, the standard deviation value if the Gaussian distribution is selected, the displacement value if the exponential distribution is selected, and β if the gamma distribution is selected.

- REAL *param2*: Relevant only when the gamma distribution is selected. The displacement value for the gamma distribution.

2.4.17 FileOutputInfo

This class instance holds the information related to file output of simulation results.

Public non-static member variables

- BOOL *particleOutput*: If set to true, *Biocellion* generates output files for discrete agent data.

- Vector<std::string> *v_particleExtraOutputScalarVarName*: This variable sets the number (*v_particelExtraOutputScalarVarName.size()*) and names of extra output scalar variables (in addition to the default *radius* and *color* variables) for particles representing discrete agents.

- Vector<std::string> *v_particleExtraOutputVectorVarName*: This variable sets the number (*v_particelExtraOutputVectorVarName.size()*) and names of extra output 3D vector (storing 3 REAL numbers) variables (in addition to the default *radius* and *color* variables) for particles representing discrete agents.

- Vector<BOOL> *v_gridPhiOutput*: Set to true to generate output files for each molecular species in the PDE.

- Vector<BOOL> *v_gridPhiOutputDivideByKappa*: Set to true to output molecular concentration values divided by κ values (Sections 4.5.2 and 4.5.4) for each molecular species in the PDE.

2.4.18 SummaryOutputInfo

This class instance holds the information related to a summary output attribute.

Public non-static member variables

- std::string *name*: Set the name of the summary attribute.

- summary_type_e *type*: Set the reduction method of the summary attribute.

2.4.19 SpAgentState

This class instance holds variables describing the state of a discrete agent and provides interface functions to access the variables. The entire set of discrete agent state variables are stored in an internal linear array. *Biocellion* does not allow direct access to the array which can be error-prone.

Public non-static member functions

- *agentType_t getType(void) const*: Return the type of the discrete agent.

- *void setType(const agentType_t type)*: Set the type of the discrete agent to *type*. **This**

function also resizes and resets the internal linear array. Users need to properly initialize discrete agent state variables after this function is invoked.

- *REAL getRadius(void) const*: Return the value of the radius variable (a default discrete agent state variable).

- *void setRadius(REAL radius)*: Set the value of the radius variable (a default discrete agent state variable) to *radius*.

- *BOOL getBoolVal(const S32 varIdx) const*: Return the *varIdx*$^{\text{th}}$ variable in the Boolean network.

- *Vector<BOOL> getBoolValArray(void) const*: Return a Vector holding the entire set of variable values in the Boolean network.

- *void setBoolVal(const S32 varIdx, const BOOL val)*: Set the value of the *varIdx*$^{\text{th}}$ variable in the Boolean network to *val*.

- *void setBoolValArray(const Vector<BOOL>& v_val)*: Set the entire set of variables in the Boolean network to *v_val*. *v_val*'s size should coincide with the number of variables in the Boolean network.

- *void getODEVal(const S32 odeNetIdx, const S32 varIdx) const*: Return the value of the *varIdx*$^{\text{th}}$ variable in the *odeNetIdx*$^{\text{th}}$ ordinary differential equation (ODE) system. A single discrete agent can have more than one ODE system.

- *Vector<REAL> getODEValArray(const S32 odeNetIdx) const*: Return a Vector holding the entire set of variable values in the *odeNetIdx*$^{\text{th}}$ ODE system.

- *void setODEVal(const S32 odeNetIdx, const S32 varIdx, REAL val)*: Set the value of the *varIdx*$^{\text{th}}$ variable in the *odeNetIdx*$^{\text{th}}$ ordinary differential equation (ODE) system to *val*.

- *void setODEValArray(const S32 odeNetIdx, const Vector<REAL>& v_val)*: Set the entire set of variables in the *odeNetIdx*$^{\text{th}}$ ODE system to *v_val*. *v_val*'s size should coincide with the number of variables in the ODE system.

- *REAL getModelReal(const S32 idx) const*: Return the value of the *idx*$^{\text{th}}$ model specific REAL type variable.

- *Vector<REAL> getModelRealArray(void) const*: Return a Vector holding the entire set of model specific REAL type variable values.

- *void setModelReal(const S32 idx, const REAL val)*: Set the value of the *idx*$^{\text{th}}$ model specific REAL type variable to *val*.

- *void setModelRealArray(const Vector<REAL>& v_val)*: Set the entire set of model spe-

cific REAL type variables to *v_val*. *v_val*'s size should coincide with the number of model specific REAL type variables for this discrete agent type.

- *void incModelReal(const S32 idx, const REAL inc)*: Increment the value of the idx^{th} model specific REAL type variable by *inc*.

- *S32 getModelInt(const S32 idx) const*: Return the value of the idx^{th} model specific S32 type variable.

- *Vector<S32> getModelIntArray(void) const*: Return a Vector holding the entire set of model specific S32 type variable values.

- *void setModelInt(const S32 idx, const S32 val)*: Set the value of the idx^{th} model specific S32 type variable to *val*.

- *void setModelIntArray(const Vector<S32>& v_val)*: Set the entire set of model specific S32 type variables to *v_val*. *v_val*'s size should coincide with the number of model specific S32 type variables for this discrete agent type.

- *void incModelInt(const S32 idx, const S32 inc)*: Increment the value of the idx^{th} model specific S32 type variable by *inc*.

2.4.20 JunctionEnd

This class instance holds variables describing the state of a junction end and provides interface functions to access the variables. The entire set of junction end state variables are stored in an internal linear array. *Biocellion* does not allow direct access to the array which can be error-prone.

Public non-static member functions

- *junctionEndType_t getType(void) const*: Return the type of the junction end.

- *void setType(const junctionEndType_t type)*: Set the type of the junction end to *type*. **This function also resizes and resets the internal linear array.** Users need to properly initialize junction end state variables after this function is invoked.

- *REAL getModelReal(const S32 idx) const*: Return the value of the idx^{th} model specific REAL type variable.

- *Vector<REAL> getModelRealArray(void) const*: Return a Vector holding the entire set of model specific REAL type variable values.

- *void setModelReal(const S32 idx, const REAL val)*: Set the value of the idx^{th} model specific REAL type variable to *val*.

- *void setModelRealArray(const Vector<REAL>& v_val)*: Set the entire set of model spe-

cific REAL type variables to *v_val*. *v_val*'s size should coincide with the number of model specific REAL type variables for this discrete agent type.

- *void incModelReal(const S32 idx, const REAL inc)*: Increment the value of the *idx*$^{\text{th}}$ model specific REAL type variable by *inc*.

- *S32 getModelInt(const S32 idx) const*: Return the value of the *idx*$^{\text{th}}$ model specific S32 type variable.

- *Vector<S32> getModelIntArray(void) const*: Return a Vector holding the entire set of model specific S32 type variable values.

- *void setModelInt(const S32 idx, const S32 val)*: Set the value of the *idx*$^{\text{th}}$ model specific S32 type variable to *val*.

- *void setModelIntArray(const Vector<S32>& v_val)*: Set the entire set of model specific S32 type variables to *v_val*. *v_val*'s size should coincide with the number of model specific S32 type variables for this discrete agent type.

- *void incModelInt(const S32 idx, const S32 inc)*: Increment the value of the *idx*$^{\text{th}}$ model specific S32 type variable by *inc*.

2.4.21 JunctionData

This class instance stores a list of junctions belong to this discrete agent and the state variables of the junction ends. A unique ID is assigned to every discrete agent particle. This ID is mainly used internally, but *Biocellion* allows users to read the ID if necessary. If an agent is created in the previous *baseline time step*, this class instance also holds the ID of the mother agent (*getPrevId()* returns the ID of the mother agent). Otherwise, *getPrevId()* and *getCurId()* return the same value.

Public non-static member functions

- *S64 getPrevId(void) const*: Return the ID of the mother agent if this discrete agent is newly created. Otherwise, this function and *getCurId()* return the same value.

- *S64 getCurId(void) const*: Return the ID of this agent.

- *S32 getNumJunctions(void) const*: Return the number of junctions between this agent and other neighboring agents.

- *S64 getOtherEndId(const S32 idx) const*: Return the ID of the discrete agent at the other end of the *idx*$^{\text{th}}$ junction.

- *const JunctionEnd& getJunctionEndRef(const S32 idx) const*: Return a constant reference of the *idx*$^{\text{th}}$ junction's end at this agent's side.

- *JunctionEnd& getJunctionEndRef(const S32 idx)*: Return a modifiable reference of the *idx*$^{\text{th}}$ junction's end at this agent's side.

- *BOOL isLinked(const JunctionData& otherData) const*: Return true if there is a junction between this agent and another agent assuming that *otherData* is the JunctionData type variable of the other discrete agent. Return false if there is no junction between the two discrete agents.

- *BOOL isLinked(const JunctionData& otherData, S32& idxThis, S32& idxOther) const*: Return true if there is a junction between this agent and another agent assuming that *otherData* is the JunctionData type variable of the other discrete agent. Return false if there is no junction between the two discrete agents. This function returns the indices to access the junction ends in addition. *idxThis* holds the index for the junction end at this discrete agent's side and *idxOther* holds the index for the junction end at the other discrete agent's side.

2.4.22 SpAgent

This class instance holds the position (offset from the center of the interface grid unit box this agent is located in) and the state of an agent mapped to a single particle.

Public non-static member variables

- SpAgentState *state*: A variable holding the state of this agent.

- JunctionData *junctionData*: A variable holding information about the entire set of junctions between this agent and neighboring agents.

- VReal *vOffset*: This variable holds the offset of this discrete agent from the center of the interface grid unit box this discrete agent is located in.

2.4.23 UBAgentData

This class instance holds a list of discrete agents in an interface grid unit box.

- Vector<SpAgent> *v_spAgent*: A collection of discrete agents in this interface grid unit box.

2.4.24 MechIntrctData

This class instance holds the temporary mechanical interaction state variables of a discrete agent. The entire set of extra temporary mechanical interaction state variables are stored in an internal linear array. *Biocellion* does not allow direct access to the array which can be error-prone.

Public non-static member functions

- *REAL getModelReal(const S32 idx) const*: Return the value of the *idx*$^{\text{th}}$ model specific

REAL type variable.

- *Vector<REAL> getModelRealArray(void) const*: Return a Vector holding the entire set of model specific REAL type variable values.

- *void setModelReal(const S32 idx, const REAL val)*: Set the value of the idx^{th} model specific REAL type variable to *val*.

- *void setModelRealArray(const Vector<REAL>& v_val)*: Set the entire set of model specific REAL type variables to *v_val*. *v_val*'s size should coincide with the number of model specific REAL type variables for this discrete agent type.

- *void incModelReal(const S32 idx, const REAL inc)*: Increment the value of the idx^{th} model specific REAL type variable by *inc*.

- *S32 getModelInt(const S32 idx) const*: Return the value of the idx^{th} model specific S32 type variable.

- *Vector<S32> getModelIntArray(void) const*: Return a Vector holding the entire set of model specific S32 type variable values.

- *void setModelInt(const S32 idx, const S32 val)*: Set the value of the idx^{th} model specific S32 type variable to *val*.

- *void setModelIntArray(const Vector<S32>& v_val)*: Set the entire set of model specific S32 type variables to *v_val*. *v_val*'s size should coincide with the number of model specific S32 type variables for this discrete agent type.

- *void incModelInt(const S32 idx, const S32 inc)*: Increment the value of the idx^{th} model specific S32 type variable by *inc*.

2.4.25 UBEnv

This class holds the environment state of an interface grid unit box.

Public non-static member functions

- *REAL getPhi(const S32 elemIdx) const*: Return the value of the $elemIdx^{th}$ molecular species concentration variable. If there are finer levels than the interface grid level, average the finest level values.

- *Vector<REAL> getPhiArray(void) const*: Return a vector array holding the entire set of molecular species concentration variable values. If there are finer levels than the interface grid level, average the finest level values.

- *REAL getSubgridPhi(const S32 subgridXOffset, const S32 subgridYOffset, const S32 sub-*

gridZOffset, const S32 elemIdx) const: Return the value of the *elemIdx*$^{\text{th}}$ molecular species concentration variable of an interface subgrid unit box. *subgridXOffset*, *subgridYOffset*, and *subgridZOffset* points an interface subgrid unit box within this interface grid unit box. If the refinement ratio between the interface grid level and the interface subgrid level is *ratio*, $0 \leq$ *subgridXOffset*, *subgridYOffset*, *subgridZOffset* $<$ *ratio* - 1.

- *REAL getSubgridPhi(const VIdx& subgridVOffset, const S32 elemIdx) const*: Use VIdx& *subgridVOffset* to index an interface subgrid unit box within an inteface grid unit box. Except for this, identical to the above *getSubgridPhi()*.

- *Vector<REAL> getSubgridPhis(const S32 elemIdx) const*: Return the molecular concentration values of the *elemIdx*$^{\text{th}}$ molecular species for the entire set of interface subgrid unit boxes in this interface grid unit box.

- *void setPhi(const S32 elemIdx, const REAL val)*: Set the value of the *elemIdx*$^{\text{th}}$ molecular species concentration variable to *val*. If there are finer levels than the interface grid level, set the molecular concentration values for the entire set of interface subgrid unit boxes in this interface grid unit box.

- *void setPhiArray(const Vector<REAL>& v_val)*: Set the entire set of molecular species concentration variable values. If there are finer levels than the interface grid level, set the molecular concentration values for the entire set of interface subgrid unit boxes in this interface grid unit box.

- *void setSubgridPhi(const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx, const REAL val)*: Set the value of the *elemIdx*$^{\text{th}}$ molecular species concentration variable of an interface subgrid unit box to *val*. *subgridXOffset*, *subgridYOffset*, and *subgridZOffset* points an interface subgrid unit box within this interface grid unit box. If the refinement ratio between the interface grid level and the finest level is *ratio*, $0 \leq$ *subgridXOffset*, *subgridYOffset*, *subgridZOffset* $<$ *ratio* - 1.

- *void setSubgridPhi(const VIdx& subgridVOffset, const S32 elemIdx, const REAL val)*: Use VIdx& *subgridVOffset* to index an interface subgrid unit box within an inteface grid unit box. Except for this, identical to the above *setSubgridPhi()*.

- *void setSubgridPhis(const S32 elemIdx, const Vector<REAL>& v_val) const*: Set the molecular concentration values of the *elemIdx*$^{\text{th}}$ molecular species for the entire set of interface subgrid unit boxes in this interface grid unit box. *v_val*'s size should conincide with the number of the interface subgrid unit boxes in this interface grid unit box.

- *void incPhi(const S32 elemIdx, const REAL inc)*: Increase the value of the *elemIdx*$^{\text{th}}$ molecular species concentration variable by *inc*. If there are finer levels than the interface grid level, increase the molecular concentration values for the entire set of interface subgrid unit boxes in this interface grid unit box.

- *void incSubgridPhi(const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgrid-*

ZOffset, const S32 elemIdx, const REAL inc): Increase the value of the *elemIdx*$^{\text{th}}$ molecular species concentration variable of an interface subgrid unit box by *inc*. *subgridXOffset*, *subgridYOffset*, and *subgridZOffset* points an interface subgrid unit box within this interface grid unit box. If the refinement ratio between the interface grid level and the interface subgrid level is *ratio*, $0 \leq$ *subgridXOffset*, *subgridYOffset*, *subgridZOffset* $<$ *ratio* - 1.

- *void incSubgridPhi(const VIdx& subgridVOffset, const S32 elemIdx, const REAL inc)*: Use VIdx& *subgridVOffset* to index an interface subgrid unit box within an inteface grid unit box. Except for this, identical to the above *incSubgridPhi()*.

- *REAL getModelReal(const S32 elemIdx) const*: Return the value of the *elemIdx*$^{\text{th}}$ model specific REAL type variable.

- *Vector<REAL> getModelRealArray(void) const*: Return a Vector holding the entire set of model specific REAL type variables.

- *void setModelReal(const S32 elemIdx, const REAL val)*: Set the value of the *elemIdx*$^{\text{th}}$ model specific REAL type variable to *val*.

- *void setModelRealArray(const Vector<REAL>& v_val)*: Set the entire set of model specific REAL type variables to *v_val*. *v_val*'s size should coincide with the number of model specific REAL type variables associated with an interface grid unit box.

- *void incModelReal(const S32 elemIdx, const REAL inc)*: Increase the value of the *elemIdx*$^{\text{th}}$ model specific REAL type variable by *inc*.

- *S32 getModelInt(const S32 elemIdx) const*: Return the value of the *elemIdx*$^{\text{th}}$ model specific S32 type variable.

- *Vector<S32> getModelIntArray(void) const*: Return a Vector holding the entire set of model specific S32 type variables.

- *void setModelInt(const S32 elemIdx, const S32 val)*: Set the value of the *elemIdx*$^{\text{th}}$ model specific S32 type variable to *val*.

- *void setModelIntArray(const Vector<S32>& v_val)*: Set the entire set of model specific S32 type variables to *v_val*. *v_val*'s size should coincide with the number of model specific S32 type variables associated with an interface grid unit box.

- *void incModelInt(const S32 elemIdx, const S32 inc)*: Increase the value of the *elemIdx*$^{\text{th}}$ model specific S32 type variable by *inc*.

2.4.26 UBEnvModelVar

This class is *UBEnv* less the member functions to access molecular concentration values (*getPhi()*, *getPhiArray()*, *getSubgridPhi()*, *getSubgridPhis()*, *setPhi()*, *setPhiArray()*, *setSubgridPhi()*,

setSubgridPhis(), *incPhi()*, and *incSubgridPhi()*).

2.4.27 NbrUBEnv

This class holds the environment state of an interface grid unit box and its 26 neighboring interface grid unit boxes. These variables are valid only for the boxes in a valid interface region. An offset variable in each dimension (*nbrXOffset*, *nbrYOffset*, *nbrZOffset*, *nbrVOffset[0]*, *nbrVOffset[1]*, or *nbrVOffset[2]*) should be -1, 0, or 1. (*nbrXOffset*, *nbrYOffset*, *nbrZOffset*) (or alternatively (*nbrVOffset[0]*, *nbrVOffset[1]*, *nbrVOffset[2]*)) = (0, 0, 0) indexes the center box.

Public non-static member functions

- *BOOL getValidFlag(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: If the indexed interface grid unit box is located in the valid interface grid region, return true. Otherwise, return false.

- *BOOL getValidFlag(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getValidFlag()*.

- *REAL getPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx) const*: Identical to invoking *getPhi()* of class UBEnv from the indexed interface grid unit box.

- *REAL getPhi(const VIdx& nbrVOffset, const S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getPhi()*.

- *Vector<REAL> getPhiArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: Identical to invoking *getPhiArray()* of class UBEnv from the indexed interface grid unit box.

- *Vector<REAL> getPhiArray(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getPhiArray()*.

- *REAL getSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx) const*: Identical to invoking *getSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *REAL getSubgridPhi(const VIdx& nbrVOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getSubgridPhi()*.

- *REAL getSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const VIdx& subgridVOffset, const S32 elemIdx) const*: Identical to invoking *getSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *REAL getSubgridPhi(const VIdx& nbrVOffset, const VIdx& subgridVOffset, const S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getSubgridPhi()*.

- *Vector<REAL> getSubgridPhis(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, S32 elemIdx) const*: Identical to invoking *getSubgridPhis()* of class UBEnv from the indexed interface grid unit box.

- *Vector<REAL> getSubgridPhis(const VIdx& nbrVOffset, S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getSubgridPhis()*.

- *void setPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const REAL val)*: Identical to invoking *setPhi()* of class UBEnv from the indexed interface grid unit box.

- *void setPhi(const VIdx& nbrVOffset, const S32 elemIdx, const REAL val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setPhi()*.

- *void setPhiArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const Vector<REAL>& v_val)*: Identical to invoking *setPhiArray()* of class UBEnv from the indexed interface grid unit box.

- *void setPhiArray(const VIdx& nbrVOffset, const Vector<REAL>& v_val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setPhiArray()*.

- *void setSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx, const REAL val)*: Identical to invoking *setSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *void setSubgridPhi(const VIdx& nbrVOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx, const REAL val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setSubgridPhi()*.

- *void setSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const VIdx& subgridVOffset, const S32 elemIdx, const REAL val)*: Identical to invoking *setSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *void setSubgridPhi(const VIdx& nbrVOffset, const VIdx& subgridVOffset, const S32 elemIdx, const REAL val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setSubgridPhi()*.

- *void setSubgridPhis(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const Vector<REAL>& v_val) const*: Identical to invoking *setSubgridPhis()* of class UBEnv from the indexed interface grid unit box.

- *void setSubgridPhis(const VIdx& nbrVOffset, const S32 elemIdx, const Vector<REAL>& v_val) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setSubgridPhis()*.

- *void incPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const REAL inc)*: Identical to invoking *incPhi()* of class UBEnv from the indexed interface grid unit box.

- *void incPhi(const VIdx& nbrVOffset, const S32 elemIdx, const REAL inc)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *incPhi()*.

- *void incSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx, const REAL inc)*: Identical to invoking *incSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *void incSubgridPhi(const VIdx& nbrVOffset, const S32 subgridXOffset, const S32 subgridYOffset, const S32 subgridZOffset, const S32 elemIdx, const REAL inc)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *incSubgridPhi()*.

- *void incSubgridPhi(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const VIdx& subgridVOffset, const S32 elemIdx, const REAL inc)*: Identical to invoking *incSubgridPhi()* of class UBEnv from the indexed interface grid unit box.

- *void incSubgridPhi(const VIdx& nbrVOffset, const VIdx& subgridVOffset, const S32 elemIdx, const REAL inc)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *incSubgridPhi()*.

- *REAL getModelReal(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx) const*: Identical to invoking *getModelReal()* of class UBEnv from the indexed interface grid unit box.

- *REAL getModelReal(const VIdx& nbrVOffset, const S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getModelReal()*.

- *Vector<REAL> getModelRealArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: Identical to invoking *getModelRealArray()* of class UBEnv from the indexed interface grid unit box.

- *Vector<REAL> getModelRealArray(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getModelRealArray()*.

- *void setModelReal(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const REAL val)*: Identical to invoking *setModelReal()* of class UBEnv from the indexed interface grid unit box.

- *void setModelReal(const VIdx& nbrVOffset, const S32 elemIdx, const REAL val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setModelReal()*.

- *void setModelRealArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const Vector<REAL>& v_val)*: Identical to invoking *setModelRealArray()* of class UBEnv from the indexed interface grid unit box.

- *void setModelRealArray(const VIdx& nbrVOffset, const Vector<REAL>& v_val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setModelRealArray()*.

- *void incModelReal(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const REAL inc)*: Identical to invoking *incModelReal()* of class UBEnv from the indexed interface grid unit box.

- *void incModelReal(const VIdx& nbrVOffset, const S32 elemIdx, const REAL inc)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *incModelReal()*.

- *S32 getModelInt(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx) const*: Identical to invoking *getModelInt()* of class UBEnv from the indexed interface grid unit box.

- *S32 getModelInt(const VIdx& nbrVOffset, const S32 elemIdx) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getModelInt()*.

- *Vector<S32> getModelIntArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: Identical to invoking *getModelIntArray()* of class UBEnv from the indexed interface grid unit box.

- *Vector<S32> getModelIntArray(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getModelIntArray()*.

- *void setModelInt(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const S32 val)*: Identical to invoking *setModelInt()* of class UBEnv from the indexed interface grid unit box.

- *void setModelInt(const VIdx& nbrVOffset, const S32 elemIdx, const S32 val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setModelInt()*.

- *void setModelIntArray(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const Vector<S32>& v_val)*: Identical to invoking *setModelIntArray()* of class UBEnv from the indexed interface grid unit box.

- *void setModelIntArray(const VIdx& nbrVOffset, const Vector<S32>& v_val)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *setModelIntArray()*.

- *void incModelInt(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset, const S32 elemIdx, const S32 inc)*: Identical to invoking *incModelInt()* of class UBEnv from the indexed interface grid unit box.

- *void incModelInt(const VIdx& nbrVOffset, const S32 elemIdx, const S32 inc)*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *incModelInt()*.

2.4.28 NbrUBEnvModelVar

This class is *NbrUBEnv* less the member functions to access molecular concentration values (*getPhi()*, *getPhiArray()*, *getSubgridPhi()*, *getSubgridPhis()*, *setPhi()*, *setPhiArray()*, *setSubgridPhi()*, *setSubgridPhis()*, *incPhi()*, and *incSubgridPhi()*).

2.4.29 NbrUBAgentData

This class instance holds constant pointers to the UBAgentData type variables associated with an interface grid unit box and its 26 neighboring interface grid unit boxes (one pointer per unit box). These variables are valid only for the boxes in a valid interface region. An offset variable in each dimension (*nbrXOffset*, *nbrYOffset*, *nbrZOffset*, *nbrVOffset[0]*, *nbrVOffset[1]*, or *nbrVOffset[2]*) should be -1, 0, or 1. (*nbrXOffset*, *nbrYOffset*, *nbrZOffset*) (or alternatively (*nbrVOffset[0]*, *nbrVOffset[1]*, *nbrVOffset[2]*)) = (0, 0, 0) indexes the center box.

Public non-static member functions

- *BOOL getValidFlag(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: If the indexed interface grid unit box is located in the valid interface grid region, return true. Otherwise, return false.

- *BOOL getValidFlag(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getValidFlag()*.

- *const UBAgentData* getConstPtr(const S32 nbrXOffset, const S32 nbrYOffset, const S32 nbrZOffset) const*: Return a pointer to the UBAgentData class instance associated with the

indexed interface grid unit box.

- *const UBAgentData* getConstPtr(const VIdx& nbrVOffset) const*: Use VIdx& *nbrVOffset* to index an interface grid unit box. Except for this, identical to the above *getConstPtr()*.

2.4.30 IfGridBoxData

This class instance holds a set of variables for a sub-region of the simulation domain (one value per one interface grid unit box in the partition). This class instance is often used to pass a set of variables for a single partition or a single partition plus the valid ghost region of the partition[(a)].

Public non-static member functions

- *T get(const VIdx& vIdx) const*: Return the value associated with the interface grid unit box at *vIdx*. The sub-region for this class instance should include *vIdx*.

- *void set(const VIdx& vIdx, const T val)*: Set the value associated with the interface grid unit box at *vIdx* to *val*. The sub-region for this class instance should include *vIdx*.

- *VIdx smallEnd(void) const*: Return the grid index for the interface grid unit box at the low end corner of the sub-region for this class instance.

- *VIdx size(void) const*: Return the size of the sub-region for this class instance.

- *idx_t size(const S32 dim) const*: Return the size of the sub-region for this class instance in either x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) dimension.

(a) The valid ghost region for a partition box can be obtained if we increase the size of the box by 1 in the $\pm$ x, y, and z directions (the box size in each direction is increased by 2) and exclude the unit boxes located in the original box, in the PDE buffer region, and outside the simulation domain

3.0 *Biocellion* Support Functions

3.1 Simulation Instance Information

3.1.1 *getCurBaselineTimeStep*

S32 getCurBaselineTimeStep(void)

Return the current *baseline time step*.

3.1.2 *getCurStateAndGridTimeStep*

S32 getCurStateAndGridTimeStep(void)

Return the current *state-and-grid time step* within the current *baseline time step*.

3.1.3 *getCurPDETimeStep*

S32 getCurPDETimeStep(const S32 pdeIdx)

Return the current *PDE time step* of the *pdeIdx*$^{\text{th}}$ PDE within the current *state-and-grid time step*.

3.1.4 *getGlobalDataRef*

const Vector<U8>& getGlobalDataRef(void)

Return a constant reference of the user global data. *Biocellion* invokes a model routine (*initGlobal()*, see Section 4.2.14) to set the global data during the simulation initialization process (this data can be read in any model routine once the data is set). This function returns a constant reference to the data.

3.1.5 *getRecentSummaryRealVal*

REAL getRecentSummaryRealVal(const S32 elemIdx)

Return the most recent value of the *elemIdx*$^{\text{th}}$ REAL type summary variable.

3.1.6 *getRecentSummaryIntVal*

REAL getRecentSummaryIntVal(const S32 elemIdx)

Return the most recent value of the *elemIdx*$^{\text{th}}$ S32 type summary variable.

3.1.7 *getDomainSize*

idx_t getDomainSize(const S32 dim)

Return the domain size in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) dimension. The domain size is set inside the simulation configuration file (see Section 6.3).

3.1.8 *getPartSize*

idx_t getPartSize(void)

Return the partition size set inside the simulation configuration file (see Section 6.3).

3.1.9 *getSimInitType*

sim_init_type_e getSimInitType(void)

Return the simulation initialization type set inside the simulation configuration file (see Section 6.3). See the explanation about sim_init_type_e in Section 2.3 for the valid simulation initialization types.

3.1.10 *getParticleNumExtraOutputScalarVars*

S32 getParticleNumExtraOutputScalarVars(void)

Return the number of extra scalar (REAL) output variables (in addition to the default radius and color variables) for each particle representing a discrete agent. This value is set inside the *updateFileOutputInfo()* model routine (see Section 4.2.12).

3.1.11 *getParticleNumExtraOutputVectorVars*

S32 getParticleNumExtraOutputVectorVars(void)

Return the number of extra 3D vector (storing 3 REAL values) output variables (in addition to the default radius and color variables) for each particle representing a discrete agent. This value is set inside the *updateFileOutputInfo()* model routine (see Section 4.2.12).

3.1.12 *getModelParam*

std::string getModelParam(void)

Return the model parameter string set inside the simulation configuration file (see Section 6.3).

3.1.13 *getSummaryInterval*

S32 getSummaryInterval(void)

Return the summary interval set inside the simulation configuration file (see Section 6.3).

3.1.14 *getRegriddingInterval*

S32 getRegriddingInterval(void)

Return the regridding interval set inside the simulation configuration file (see Section 6.3).

3.1.15 *getAMRRatio*

S32 getAMRRatio(void)

Return the refinement ratio between two consecutive adaptive mesh refinement (AMR) levels. The refinement ratio is set inside the simulation configuration file (see Section 6.3).

3.2 Random Number Generator

3.2.1 *getModelRand*

REAL Util::getModelRand(const S32 idx)

Return a random number generated using the idx^{th} random number generator. *Biocellion* allows users to assign a set of random number generators with different probability density functions.

3.3 Extra Support Functions

3.3.1 computeSphereUBVolOvlpRatio

void computeSphereUBVolOvlpRatio(const S32 maxLevel, const VReal& vOffset, const REAL radius, REAL aaa_ratio[3][3][3])
REAL computeSphereUBVolOvlpRatio(const S32 maxLevel, const VReal& vOffset, const REAL radius, const VIdx& ubVIdxOffset)

A discrete agent with a non-zero volume may span multiple interface grid unit boxes. Modelers may want to find the ratios of the overlapping volumes between the agent and different interface grid unit boxes over the entire volume of the agent. These functions find an approximate solution assuming that the discrete agent is a sphere (*vOffset* is the offset from the center of the unit box containing the discrete agent and points the center of the sphere, and *radius* is the radius of the sphere). The interface grid unit box containing the discrete agent and its 26 neighboring boxes are considered—the radius of a discrete agent cannot exceed one half of the interface grid spacing and the sphere can overlap with only these $3 \times 3 \times 3$ unit boxes. The first function finds the ratios for the entire set of $3 \times 3 \times 3$ unit boxes (*aaa_ratio* contains the solution), and the second function returns the ratio for only one of the $3 \times 3 \times 3$ unit boxes indexed by *ubVIdxOffset*. This function partitions the minimum bounding box of the sphere to $2^{maxLevel} \times 2^{maxLevel} \times 2^{maxLevel}$ sub-boxes. We consider that a sub-box belongs to an interface grid unit box if the sub-box center is located inside the sphere and the unit box. If the center is located exactly at the surface of the sphere, we consider the center is located inside the sphere. If the center is located exactly at the boundary of two unit boxes, we consider that the center is located at the unit box with a larger index. These functions compute the ratio by dividing the number of sub-boxes belong to an interface grid unit box by the number of sub-boxes belong to the $3 \times 3 \times 3$ boxes. *maxLevel* should be a non-negative integer equal to or smaller than 7. A larger value increases both accuracy and execution time. *vOffset* cannot point the location outside the unit box. *radius* should not exceed one half of the interface grid spacing. *ubVIdxOffset[idx]* (*idx* = 0, 1, or 2) should be -1, 0, or 1.

3.3.2 *changePosFormat2LvTo1Lv*

void changePosFormat2LvTo1Lv(const VIdx& vIdx, const VReal& vOffset, VReal& pos)

Particle position can be represented in a single level (with a single VReal varaible storing the offset from the low end corner of the simulation domain) or in two levels (with a single VIdx variable providing the unit box the particle belongs to and a single VReal variable holding the offset from the center of the unit box). This function changes the format from the two level format to the single level format (the updated position is stored in *pos*).

3.3.3 *changePosFormat1LvTo2Lv*

void changePosFormat1LvTo2Lv(const VReal& pos, VIdx& vIdx, VReal& vOffset)

Particle position can be represented in a single level (with a single VReal varaible storing the offset from the low end corner of the simulation domain) or in two levels (with a single VIdx variable providing the unit box index the particle belongs to and a single VReal variable holding the offset from the center of the unit box). This function changes the format from the single level format to the two level format (the updated position is stored in *vIdx* and *vOffset*).

3.4 Macros

3.4.1 *CHECK*

CHECK(expression)

If this macro is enabled in compile time (Section 6.2.1), abort when *expression* is false.

3.4.2 *OUTPUT*

OUTPUT(minVerbosity, msg)

Print *msg* to the standard output (or redirect to a file if the output redirection *path* is set in the simulation configuration file, see Section 6.3.2) if the verbosity set inside the simulation configuration file (see Section 6.3.2) is equal to or larger than *minVerbosity*.

3.4.3 *WARNING*

WARNING(msg)

Print *msg* to the standard output.

3.4.4 *ERROR*

ERROR(msg)

Print *msg* to the standard output and abort the program.

4.0 *Biocellion* Model Routines

Biocellion asks users to provide their model specifics by filling the function body of a set of predefined model routines. Users write only sequential code, but the *Biocellion* framework can concurrently invoke a single model routine. Users are disallowed to use any functions that are not thread-safe—*e.g.* the C rand() function is not thread safe, use *getModelRand()* (Section 3.2.1) instead.

We also strongly discourage users to access global variables inside a model routine with few exceptions. Reading constant global variables is acceptable. If global variables are initialized at the beginning of the simulation in the *init()* model routine (Section 4.2.15), reading the initialized variables is also acceptable with one caveat. If multiple MPI processes are created to run *Biocellion* on a system with multiple shared memory nodes (*e.g.* cluster computers and supercomputers), the *init()* model routine is called once in every MPI process. Model routines executed in different MPI processes access different instances of a global variable initialized in different *init()* invocations even when model routines (executed on different MPI processes) access using the same variable name. If users need to set shared global data across the entire set of MPI processes, update the data using the *initGlobal()* model routine (Section 4.2.14) which is invoked exactly once by just one of the MPI processes (the MPI process having rank 0). The data is copied to the remaining MPI processes by the *Biocellion* core framework. The data cannot be modified outside the *initGlobal()* model routine, and users can access the data through *getGlobalDataRef()* (Section 3.1.4).

Section 4.1 summarizes model routine invocation timings, and the remaining of this Section explains the model routines.

4.1 Model Routine Invocation Timings

Figures 4.1, 4.2, 4.3, and 4.4 summarizes model routine invocation timings for the initialization, main computation, PDE computation, and termination, respectively. Symbols after model routine names indicate the model routine invocation granularity. **S** indicates that the model routine is called only once per simulation even when there are more than one MPI processes. **M** indicates that the model routine is called once per MPI process. If there is only one (multi-threaded) process, **S** and **M** are identical. **P** indicates that the model routine is called once per partition. **IB** indicates that the model routine is called once per interface grid unit box. **IF** indicates that the model routine is called once per relevant interface unit box face. **ISB** indicates that the model routine is called once per interface subgrid unit box. **ISF** indicates that the model routine is called once per relevant interface subgrid unit box face—*e.g.* *updateIfSubgridBetaDomainBdry()* is called only for the finest level unit box faces in the interface region at the domain boundary. **PB** indicates that the model routine is called once per PDE buffer grid unit box, and **PF** indicates that the model routine is called once per relevant PDE buffer grid unit box face. **A** indicates that the model routine is called once per discrete agent, and **AP** indicates that the model routine is called once per discrete agent pair within the maximum direct interaction distance.

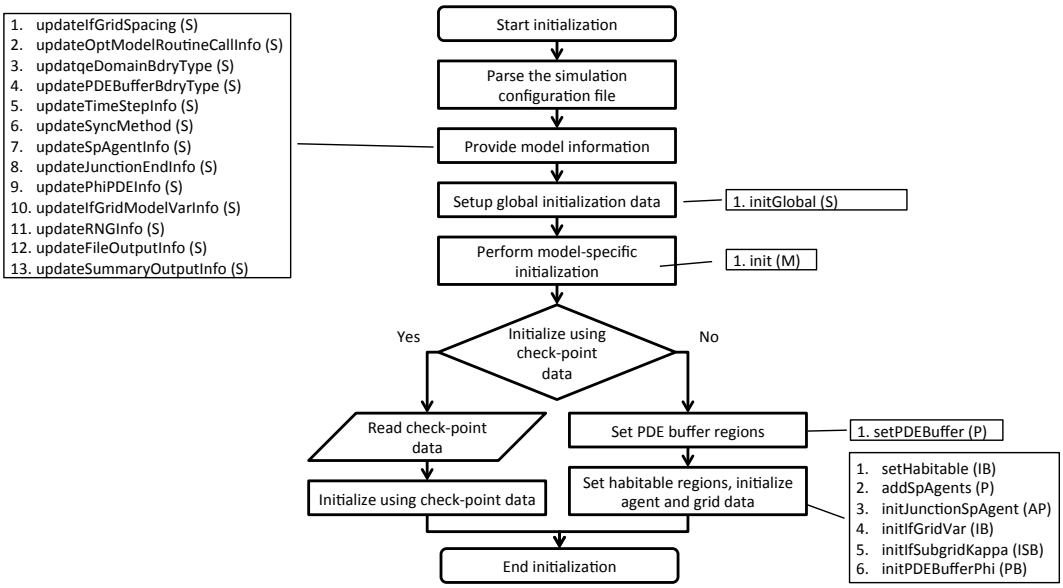

Figure 4.1. Initialization.

4.2 Model Configuration

4.2.1 *updateIfGridSpacing*

void updateIfGridSpacing(REAL& ifGridSpacing)

Set the interface grid spacing. This model routine sets *ifGridSpacing* to the desired interface grid spacing.

4.2.2 *updateOptModelRoutineCallInfo*

void updateOptModelRoutineCallInfo(OptModelRoutineCallInfo& callInfo)

Control invocation of optional model routines. This model routine updates *callInfo*. See Section 2.4.6 about the OptModelRoutineCallInfo class.

4.2.3 *updateDomainBdryType*

void updateDomainBdryType(domain_bdry_type_e a_domainBdryType[3])

Set the domain boundary types at the low and high sides in the x, y, and z directions. This model routine updates *a_domainBdryType[3]*. See the explanation about domain_bdry_type_e in Section 2.3 for the supported domain boundary types.

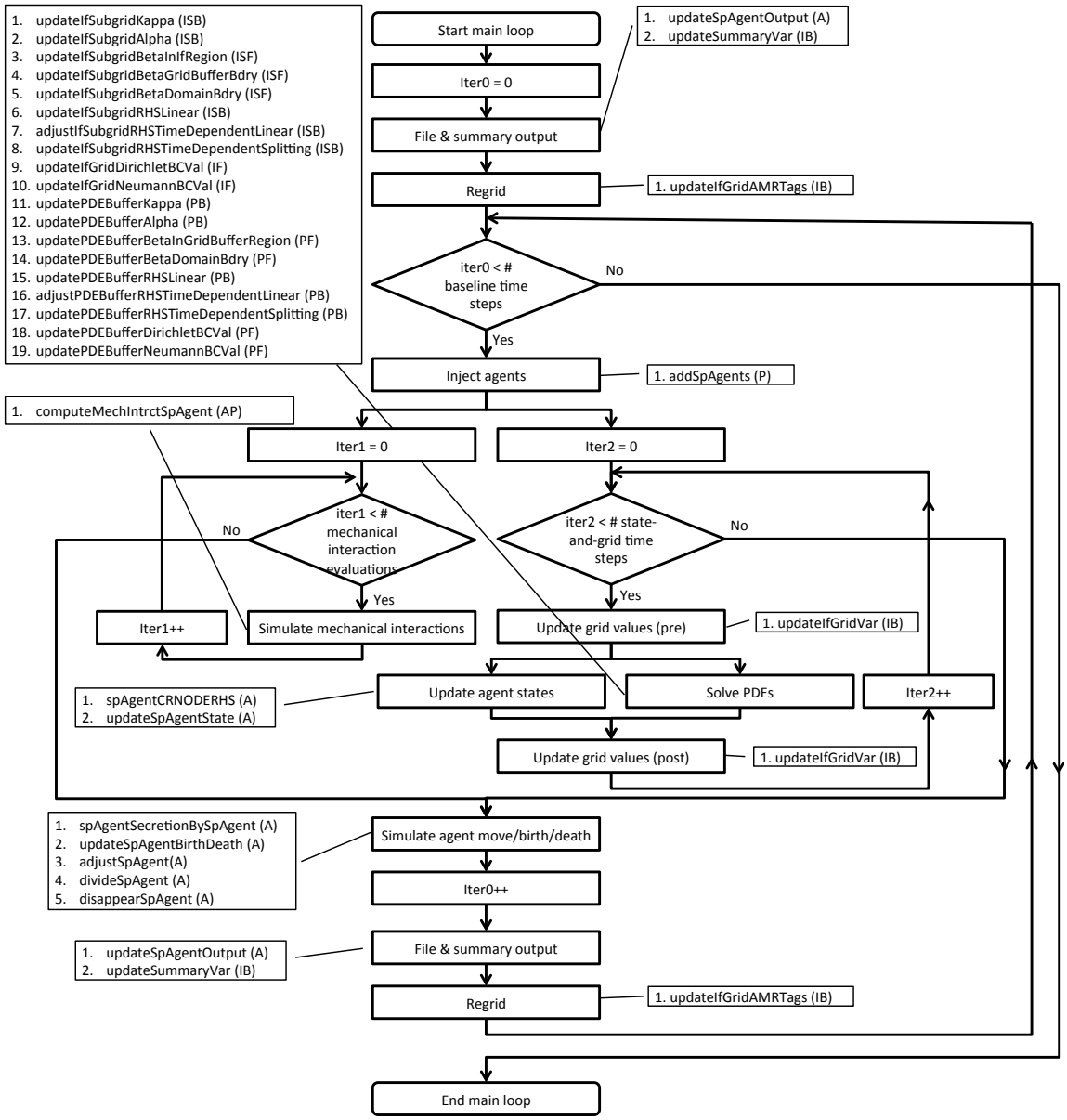

Figure 4.2. Main loop.

4.2.4 *updatePDEBufferBdryType*

void updatePDEBufferBdryType(pde_buffer_bdry_type_e& pdeBufferBdryType)

Set the boundary type between an interface grid partition and a PDE buffer partition. This model routine updates *pdeBufferBdryType*. See the explanation about pde_buffer_bdry_type_e in Section 2.3 for the supported PDE buffer boundary types.

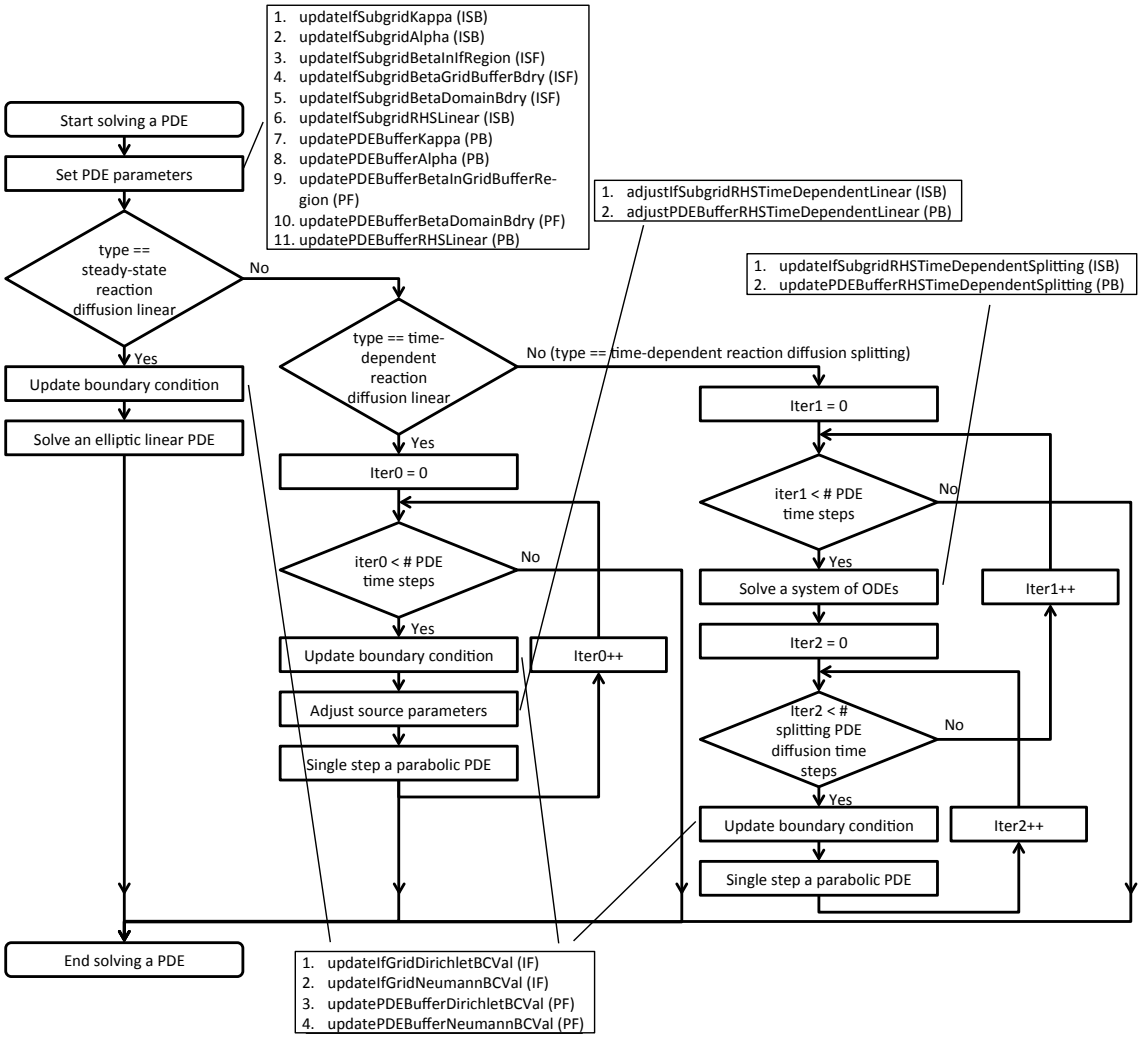

Figure 4.3. Solving a PDE.

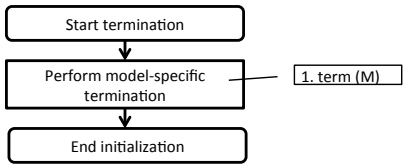

Figure 4.4. Termination.

4.2.5 *updateTimeStepInfo*

void updateTimeStepInfo(TimeStepInfo& timeStepInfo)

Set *baseline time step* size and *state-and-grid time step* size. This model routine updates *timeStepInfo*. See Section 2.4.5 to find details about the TimeStepInfo class.

4.2.6 *updateSyncMethod*

void updateSyncMethod(sync_method_e& mechIntrctSyncMethod, sync_method_e& updateIfGrid VarSyncMethod)

Set the synchronization method when a single variable is updated in multiple model routine calls. This model routine updates *mechIntrctSyncMethod* (for direct mechanical interactions between agent pairs, see *computeMechIntrctSpAgent()* in Section 4.4.2) and *updateIfGridVarSyncMetho d* (to update extracellular space state variables based on model specific rules, see *updateIfGrid-Var()* in Section 4.5.3). See the explanation about sync_method_e in Section 2.3 for the supported synchronization mechanisms.

4.2.7 *updateSpAgentInfo*

void updateSpAgentInfo(Vector<SpAgentInfo>& v_spAgentInfo)

Provide the information about the discrete agent types in the user model. This model routine updates *v_spAgentInfo*. The size of *v_spAgentInfo* sets the number of discrete agent types. Each vector element provides the information about a single discrete agent type. See Section 2.4.8 for additional details about the SpAgentInfo class.

4.2.8 *updateJunctionEndInfo*

void updateJunctionEndInfo(Vector<JunctionEndInfo>& v_junctionEndInfo)

Provide the information about the junction end types in the user model. This model routine updates *v_junctionEndInfo*. The size of *v_junctionEndInfo* sets the number of junction end types. Each vector element provides the information about a single junction end type. See Section 2.4.9 for additional details about the JunctionEndInfo class.

4.2.9 *updatePhiPDEInfo*

void updatePhiPDEInfo(Vector<PDEInfo>& v_pdeInfo)

Provide the information about the partial differential equations (PDEs) in the user model. This model routine updates *v_pdeInfo*. The size of *v_pdeInfo* sets the number of PDEs in the user model. Each vector element provides the information about a single PDE. See Section 2.4.14 for additional details about the PDEInfo class.

4.2.10 *updateIfGridModelVarInfo*

void updateIfGridModelVarInfo(Vector<IfGridModelVarInfo>& v_ifGridModelRealInfo, Vecto r<IfGridModelVarInfo>& v_ifGridModelIntInfo)

Provide the information about the model specific attributes for the extracellular space in the user model. This model routine updates *v_ifGridModelRealInfo* (for REAL type attributes) and *v_if-*

GridModelIntInfo (for S32 type variables). The sizes of *v_ifGridModelRealInfo* and *v_ifGridModelIntInfo* coincide with the number of REAL type and S32 type model specific attributes, respectively. Each vector element provides the information about a single model specific attribute. See Section 2.4.15 for additional details about the IfGridModelVarInfo class.

4.2.11 *updateRNGInfo*

void updateRNGInfo(Vector<RNGInfo>& v_rngInfo)

Provide the information about the random number generators used in the model. This model routine updates *v_rngInfo*. The size of *v_rngInfo* sets the number of random number generators. Each vector element provides the information about a single random number generator. See Section 2.4.16 for additional details about the RNGInfo class.

4.2.12 *updateFileOutputInfo*

void updateFileOutputInfo(FileOutputInfo& fileOutputInfo)

Provide the information about file output (used for visualization) of simulation results. This model routine updates *fileOutputInfo*. See Section 2.4.17 for additional details about the FileOutputInfo class.

4.2.13 *updateSummaryOutputInfo*

void updateSummaryOutputInfo(Vector<SummaryOutputInfo>& v_summaryOutputRealInfo, Vector<SummaryOutputInfo>& v_summaryOutputIntInfo)

Provide the information about summary output of simulation results. The size of *v_summaryOutputRealInfo* sets the number of REAL type summary attributes. The size of *v_summaryOutputIntInfo* sets the number of S32 type summary attributes. Each vector element provides the information about a single summary attribute. See Section 2.4.18 for additional details about the SummaryOutputInfo class.

4.2.14 *initGlobal*

void initGlobal(Vector<U8>& v_globalData)

Initialize the global data shared by the entire set of MPI processes. Once *v_globalData* is initialized in this model routine, the data is copied to every MPI process. *v_globalData* cannot be updated elsewhere. *getGlobalDataRef()* (Section 3.1.4) returns a constant reference to the updated data.

4.2.15 *init*

void init(void)

Users may perform model specific initialization inside this function.

4.2.16 *term*

void term(void)

Users may perform model specific clean-up inside this function.

4.2.17 *setPDEBuffer*

void setPDEBuffer(const VIdx& startVIdx, const VIdx& regionSize, BOOL& isPDEBuffer)

Mark whether this partition is an interface grid partition (set *isPDEBuffer* to false) or a PDE buffer partition (set *isPDEBuffer* to true). *startVIdx* points the low end corner of this partition, and *regionSize* is the size of this partition assuming the interface grid spacing.

4.2.18 *setHabitable*

void setHabitable(const VIdx& vIdx, BOOL& isHabitable)

Mark whether this unit box (indexed by *vIdx*) is habitable (set *isHabitable* to true) or not (set *isHabitable* to false). If this unit box is marked as uninhabitable, discrete agents cannot penetrate into this unit box.

4.3 Individual Agent Behavior

4.3.1 *addSpAgents*

void addSpAgents(const BOOL init, const VIdx& startVIdx, const VIdx& regionSize, const IfGrid BoxData<BOOL>& ifGridHabitableBoxData, Vector<VIdx>& v_spAgentVIdx, Vector<SpAge ntState>& v_spAgentState, Vector<VReal>& v_spAgentOffset)

Add discrete agents to this partition at the beginning of a simulation (if *init* is true) or at the beginning of a *baseline time step* (if *init* is false). *startVIdx* points the interface grid unit box at the low end corner of this partition, and *regionSize* is the size of this partition. *ifGridHabitableBoxData* informs whether a unit box in this partition is habitable or not. It is an error to add a discrete agent to an uninhabitable unit box. This model routine updates *v_spAgentVIdx*, *v_spAgentState*, and *v_spAgentOffset*. The sizes of *v_spAgentVIdx*, *v_spAgentState*, and *v_spAgentOffset* should be same—one vector element for one newly added discrete agent. An element of *v_spAgentVIdx* locates the interface grid unit box to place a discrete agent. An element of *v_spAgentState* stores the state of a discrete agent. An element of *v_spAgentOffset* sets the discrete agent position offset within a interface grid unit box (from the center of the unit box).

4.3.2 *spAgentCRNODERHS*

void spAgentCRNODERHS(const S32 odeNetIdx, const VIdx& vIdx, const SpAgent& spAgent, co nst NbrUBEnv& nbrUBEnv, const Vector<double>& v_y, Vector<double>& v_f)

Set the right hand side (the derivatives) of the *odeNetIdx*$^{\text{th}}$ ODE system for this discrete agent. *vIdx* indexes the interface grid unit box this discrete agent is located in. *spAgent* provides the information about this discrete agent. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. *v_y* stores the current values of the ODE system variables. This model routine updates *v_f* which stores the derivatives for the variables in the ODE system. The ODE systems for a discrete agent are updated before *updateSpAgentState()* is called for the agent.

4.3.3 *updateSpAgentState*

void updateSpAgentState(const VIdx& vIdx, const JunctionData& junctionData, const VReal& v Offset, const NbrUBEnv& nbrUBEnv, SpAgentState& state)

Update the state of this discrete agent. *vIdx* indexes the interface grid unit box this discrete agent is located in. *junctionData* stores the entire set of junctions between this discrete agent and its neighboring discrete agents. *vOffset* is the offset from the center of the unit box this discrete agent belongs to. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates *state* based on the provided information. *state* initially holds the state of this discrete agent. *updateSpAgentState()* is called after the ODE systems for the discrete agent are updated.

4.3.4 *spAgentSecretionBySpAgent*

void spAgentSecretionBySpAgent(const VIdx& vIdx, const JunctionData& junctionData, const V Real& vOffset, const MechIntrctData& mechIntrctData, const NbrUBEnv& nbrUBEnv, SpAgent State& state, Vector<SpAgentState>& v_spAgentState, Vector<VReal>& v_spAgentDisp)

Secrete new discrete agents from this discrete agent. *vIdx* indexes the interface grid unit box this discrete agent is located in. *junctionData* stores the entire set of junctions between this discrete agent and its neighboring discrete agents. *vOffset* is the offset from the center of the unit box this discrete agent belongs to. *mechIntrctData* stores the temporary mechanical interaction state of this discrete agent. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates this discrete agent's state (*state*) and the states (*v_spAgentState*) and the displacements (from this discrete agent, *v_spAgentDisp*) of newly secreted discrete agents. The displacement values cannot exceed the interface grid spacing. *v_spAgentState* and *v_spAgentDisp* should have same size—one element for one newly secreted

discrete agent. *state* initially holds the state of this discrete agent. *spAgentSecretionBySpAgent()* is called before *updateSpAgentBirthDeath()* within a single *baseline time step*.

4.3.5 *updateSpAgentBirthDeath*

void updateSpAgentBirthDeath(const VIdx& vIdx, const SpAgent& spAgent, const MechIntrctDa ta& mechIntrctData, const NbrUBEnv& nbrUBEnv, BOOL& divide, BOOL& disappear)

Mark whether this discrete agent will divide or disappear. *vIdx* indexes the interface grid unit box this discrete agent is located in. *spAgent* provides the information about this discrete agent. *mechIntrctData* stores the temporary mechanical interaction state of this discrete agent. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates *divide* and *disappear*. Set *divide* to true if this discrete agent is going to divide. *divideSpAgent()* (Section 4.3.7) will be called if *divide* is set to true. Set *disappear* to true if this discrete agent is subject to disappear. If neither *divide* nor *disappear* is set to true, *adjustSpAgent()* (Section 4.3.6) is called. It is an error to set both *divide* and *disappear* to true. *updateSpAgentBirthDeath()* is called after *spAgentSecretionBySpAgent()* within a single *baseline time step*.

4.3.6 *adjustSpAgent*

void adjustSpAgent(const VIdx& vIdx, const JunctionData& junctionData, const VReal& vOffset , const MechIntrctData& mechIntrctData, const NbrUBEnv& nbrUBEnv, SpAgentState& state, V Real& disp)

Adjust the state and position of this discrete agent. *vIdx* indexes the interface grid unit box this discrete agent is located in. *junctionData* stores the entire set of junctions between this discrete agent and its neighboring discrete agents. *vOffset* is the offset from the center of the unit box this discrete agent belongs to. *mechIntrctData* stores the temporary mechanical interaction state of this discrete agent. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates the state of this discrete agent (*state*) and the displacement from the original position (*disp*). *state* initially holds the state of this discrete agent. *disp* should not exceed the interface grid spacing.

4.3.7 *divideSpAgent*

void divideSpAgent(const VIdx& vIdx, const JunctionData& junctionData, const VReal& vOffse t, const MechIntrctData& mechIntrctData, const NbrUBEnv& nbrUBEnv, SpAgentState& mothe rState, VReal& motherDisp, SpAgentState& daughterState, VReal& daughterDisp, Vector<BOO L>& v_junctionDivide, BOOL& motherDaughterLinked, JunctionEnd& motherEnd, JunctionEn d& daughterEnd)

Divide this discrete agent to a mother discrete agent and a daughter discrete agent. *vIdx* indexes the interface grid unit box this discrete agent is located in. *junctionData* stores the entire set of

junctions between this discrete agent and its neighboring discrete agents. *vOffset* is the offset from the center of the unit box this discrete agent belongs to. *mechIntrctData* stores the temporary mechanical interaction state of this discrete agent. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates the states of the mother discrete agent and the daughter discrete agent (*motherState* and *daughterState*, respectively), the displacements from the original position for the mother and daughter discrete agents (*motherDisp* and *daughterDisp*, respectively), and a junction between the mother and daughter discrete agents. *motherState* initially stores the state of this discrete agent. *v_junctionDivide* specifies a subset of the junctions (if the *v_junctionDivide* element for a junction is set to true, the mother agent retains the junction) the mother agent will retain—the daughter agent inherits the remaining junctions. A junction is formed between the mother and daughter discrete agents if *motherDaughterLinked* is set to true with *motherEnd* and *daughterEnd* as the junction ends for the mother and daughter sides, respectively. *motherEnd* and *daughterEnd* are irrelevant if *motherDaughterLinked* is set to false.

4.4 Physico-Mechanical Interaction Between Agents

4.4.1 *initJunctionSpAgent*

void initJunctionSpAgent(const VIdx& vIdx0, const SpAgent& spAgent0, const VIdx& vIdx1, const SpAgent& spAgent1, const VReal& dir, const REAL& dist, BOOL& link, JunctionEnd& end0, JunctionEnd& end1)

Initialize a junction between a discrete agent pair within the maximum direct interaction distance. *vIdx0* indexes the interface grid unit box *spAgent0* is located in. *vIdx1* indexes the interface grid unit box *spAgent1* is located in. *dir* is a unit vector from the position of *spAgent1* to the position of *spAgent0*. *dist* is the distance between the positions of the two discrete agents. This model routine updates *link*, *end0*, and *end1*. Set *link* to true to form a junction between the two discrete agents. *end0* is the junction end in the *spAgent0* side and *end1* is the junction end in the *spAgent1* side. *end0* and *end1* are irrelevant if *link* is set to false.

4.4.2 *computeMechIntrctSpAgent*

void computeMechIntrctSpAgent(const S32 iter, const VIdx& vIdx0, const SpAgent& spAgent0, const UBEnv& ubEnv0, const VIdx& vIdx1, const SpAgent& spAgent1, const UBEnv& ubEnv1, const VReal& dir, const REAL& dist, MechIntrctData& mechIntrctData0, MechIntrctData& mechIntrctData1, BOOL& link, JunctionEnd& end0, JunctionEnd& end1, BOOL& unlink)

Evaluate mechanical interactions between a discrete agent pair acting on each agent; invoked 0 or more (*numComputeMechIntrctIters*, See Section 2.4.6) times for every agent pair within the direct mechanical interaction distance per *baseline time step*. *iter* is a non-negative integer specifying the iteration within the current *baseline time step* at the time of invocation. *vIdx0* indexes the interface grid unit box *spAgent0* is located in. *ubEnv0* stores the molecular concentration values and model specific REAL and S32 type attribute values for the interface grid unit box indexed by *vIdx0*. *vIdx1* indexes the interface grid unit box *spAgent1* is located in. *ubEnv1* stores the molec-

ular concentration values and model specific REAL and S32 type attribute values for the interface grid unit box indexed by *vIdx1*. *dir* is a unit vector from the position of *spAgent1* to the position of *spAgent0*. *dist* is the distance between the positions of the two discrete agents. This model routine updates the temporary mechanical interaction states for *spAgent0* and *spAgent1* which are *mechIntrctData0* and *mechIntrctData1*, respectively. Temporary mechanical interaction state variables are reset to zero at the beginning of every *baseline time step*. Users also form and break a junction between a discrete agent pair in this model routine. If the two discrete agents are linked and *link* is set to true, the junction ends in the *spAgent0* and *spAgent1* sides are replaced with *end0* and *end1*, respectively—*isLinked* (Section 2.4.21) returns whether two discrete agents are linked or not. If *spAgent0* and *spAgent1* are linked and *unlink* is set to true, the *Biocellion* framework breaks the junction. If there is no junction between *spAgent0* and *spAgent1* and *link* is set to true, a new junction is formed with junction ends *end0* and *end1*. If there is no junction and *unlink* is set to true or both *link* and *unlink* is set to false, no change occurs. It is an error to set both *link* and *unlink* to true.

4.5 State Changes in the Extra-cellular Space

4.5.1 *initIfGridVar*

void initIfGridVar(const VIdx& vIdx, const UBAgentData& ubAgentData, UBEnv& ubEnv)

Initialize the state variables associated with an interface grid unit box. *vIdx* indexes an interface grid unit box. *ubAgentData* holds the discrete agents located in this interface grid unit box. This model routine updates *ubEnv* which stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box.

4.5.2 *initIfSubgridKappa*

void initIfSubgridKappa(const S32 pdeIdx, const VIdx& vIdx, const VIdx& subgridVOffset, const UBAgentData& ubAgentData, const UBEnv& ubEnv, REAL& gridKappa)

Initialize the ratio of the available volume in this interface subgrid unit box (PDE parameter κ of the $pdeIdx^{\text{th}}$ PDE, Section 4.5.4 explains κ). *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the discrete agents located in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. This function is relevant only when *v_gridPhiOutputDivideByKappa[elemIdx]* of the FileOutputInfo class instance is set to true in the *updateFileOutputInfo()* model routine for at least one molecular species update by the $pdeIdx^{\text{th}}$ PDE.

4.5.3 *updateIfGridVar*

void updateIfGridVar(const BOOL pre, const S32 iter, const VIdx& vIdx, const NbrUBAgentDat a& nbrUBAgentData, NbrUBEnv& nbrUBEnv)

Update the state variables associated with an interface grid unit box and its 26 neighboring boxes.

It is an error to access or update values associated with a unit box location outside the simulation domain or a unit box location in a PDE buffer region. *pre* indicates whether this routine is called at the beginning of a *state-and-grid time step* (if *pre* is true) or at the end of a *state-and-grid time step* (if *pre* is false). *iter* indicates the iteration in which this model routine is called—users are allowed update extracellular space state variables in multiple iterations (Section 2.4.6). *vIdx* indexes this interface grid unit box. *nbrUBAgentData* holds pointers for UBAgentData class instances for this interface grid unit box and its 26 neighboring unit boxes. This model routine updates *nbrUBEnv* which stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes.

4.5.4 *updateIfSubgridKappa*

void updateIfSubgridKappa(const VIdx& vIdx, const VIdx& subgridVOffset, const UBAgentData & ubAgentData, const UBEnv& ubEnv, REAL& gridKappa)

Set the ratio of the available volume in this interface subgrid unit box in computing diffusion flux. Molecular concentration in *Biocellion* is the total mass of a molecular species in a unit box divided by the entire volume of the unit box. However, in computing diffusion, the direction and rate of diffusion is determined by the effective molecular concentration in the space available for molecular diffusion. If one unit box is heavily occupied by cells and another unit box has no cell, and the mathematical model assumes that only the molecules in the extracellular space is available for diffusion, then the net amount of diffusion to the empty unit box can be positive even when the molecular concentration (ignoring cell volume exclusion) in the empty unit box is higher. This model routine sets *gridKappa* to consider the available volume in computing diffusion flux. Say there are two unit boxes (*box0* and *box1*) sharing a face. ϕ_0 and ϕ_1 are molecular concentrations for *box0* and *box1*, respectively. κ_0 and κ_1 are *gridKappa* values set for *box0* and *box1*, respectively. β is the diffusion coefficient set for the common face. Then, the diffusion flux from *box0* to *box1* becomes $\beta \times (\frac{\phi_0}{\kappa_0} - \frac{\phi_1}{\kappa_1}) \times \frac{1}{h}$, where h is the length of a box side. *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. $0.0 < gridKappa \leq 1.0$. Set *gridKappa* to 1.0 if the entire unit box volume is available for diffusion. *gridKappa* is assumed to be 1.0 outside the simulation domain.

4.5.5 *updateIfSubgridAlpha*

void updateIfSubgridAlpha(const S32 elemIdx, const VIdx& vIdx, const VIdx& subgridVOffset, c onst UBAgentData& ubAgentData, const UBEnv& ubEnv, REAL& gridAlpha)

Set the decay rate for the $elemIdx^{\text{th}}$ molecular species. *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. This model routine updates *gridAlpha*. $-1.0 < gridAlpha \leq 0.0$. Set *gridAlpha* to 0.0 to ignore decay.

4.5.6 *updateIfSubgridBetaInIfRegion*

void updateIfSubgridBetaInIfRegion(const S32 elemIdx, const S32 dim, const VIdx& vIdx0, const VIdx& subgridVOffset0, const UBAgentData& ubAgentData0, const UBEnv& ubEnv0, const VId x& vIdx1, const VIdx& subgridVOffset1, const UBAgentData& ubAgentData1, const UBEnv& ub Env1, REAL& gridBeta)

Set the diffusion coefficient for the *elemIdx*$^{\text{th}}$ molecular species for an interface subgrid unit box face strictly inside the interface grid region. The interface subgrid unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. *vIdx0* and *subgridVOffset0* indexe the interface subgrid unit box at the low side of the face and *vIdx1* and *subgridVOffset1* indexe the interface subgrid unit box at the high side of the face. *ubAgentData0* and *ubAgentData1* hold discrete agents in the interface grid unit boxes at the low and high side unit boxes, respectively. *ubEnv0* and *ubEnv1* store the molecular concentration values and model specific REAL and S32 type attribute values for the interface grid unit boxes indexed by *vIdx0* and *vIdx1*, respectively. This model routine updates *gridBeta*.

4.5.7 *updateIfSubgridBetaPDEBufferBdry*

void updateIfSubgridBetaPDEBufferBdry(const S32 elemIdx, const S32 dim, const VIdx& vIdx, c onst VIdx& subgridVOffset, const UBAgentData& ubAgentData, const UBEnv& ubEnv, REAL& gridBeta)

Set the diffusion coefficient for the *elemIdx*$^{\text{th}}$ molecular species for an interface subgrid unit box face between an interface grid partition and a PDE buffer partition. The interface subgrid unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. This model routine updates *gridBeta*.

4.5.8 *updateIfSubgridBetaDomainBdry*

void updateIfSubgridBetaDomainBdry(const S32 elemIdx, const S32 dim, const VIdx& vIdx, con st VIdx& subgridVOffset, const UBAgentData& ubAgentData, const UBEnv& ubEnv, REAL& gri dBeta)

Set the diffusion coefficient for the *elemIdx*$^{\text{th}}$ molecular species for an interface subgrid unit box face at the domain boundary. The interface subgrid unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. This model routine updates *gridBeta*.

4.5.9 *updateIfSubgridRHSLinear*

void updateIfSubgridRHSLinear(const S32 elemIdx, const VIdx& vIdx, const VIdx& subgridVOff set, const UBAgentData& ubAgentData, const UBEnv& ubEnv, REAL& gridRHS)

Update the source term for the *elemIdx*th molecular species. This model routine is relevant only for linear PDEs (`PDE_TYPE_REACTION_DIFFUSION_STEADY_STATE_LINEAR` and `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`). *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnv* stores the molecular concentration values and model specific REAL and S32 type attribute values for the indexed interface grid unit box. This model routine updates *gridRHS*.

4.5.10 *adjustIfSubgridRHSTimeDependentLinear*

void adjustIfSubgridRHSTimeDependentLinear(const S32 elemIdx, const VIdx& vIdx, const VIdx & subgridVOffset, const UBAgentData& ubAgentData, const UBEnvModelVar& ubEnvModelVa r, const REAL gridPhi, REAL& gridRHS)

Adjust the source term for the *elemIdx*th molecular species. This model routine is invoked in every *PDE time step* and is relevant only for time-dependent linear PDEs (`PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`). This model routine is called only when users configure to call this function (Section 2.4.14). *vIdx* and *subgridVOffset* index an interface subgrid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *ubEnvModelVar* stores the model specific REAL and S32 type attribute values for the indexed interface grid unit box. *gridPhi* is the concentration of the *elemIdx*th molecular species in this interface subgrid unit box at the beginning of the *PDE time step*. *gridRHS* is initially set to the *gridRHS* value set inside the *updateIfSubgridRHSLinear()* model routine (if *PDE time step* is 0) or the *gridRHS* in the previous *PDE time step* (if *PDE time step* is not 0) for this molecular species. Users may change the value based on model specifics.

4.5.11 *updateIfSubgridRHSTimeDependentSplitting*

void updateIfSubgridRHSTimeDependentSplitting(const S32 pdeIdx, const VIdx& vIdx, const VI dx& subgridVOffset, const UBAgentData& ubAgentData, const UBEnvModelVar& ubEnvModel Var, const Vector<double>& v_gridPhi, Vector<double>& v_gridRHS)

Set the reaction term of the *pdeIdx*th PDE which is solved using the splitting scheme. *vIdx* indexes an interface grid unit box and *subgridVOffset* indexes an interface subgrid unit box within the interface grid unit box. *ubAgentData* holds the entire set of discrete agents in this interface grid unit box. *v_gridPhi* holds molecular concentrations **for the molecular species in this PDE—** the size of the vector is equal to the number of molecular species in this PDE, and the first vector element holds the molecular concentration for the molecular species in this PDE with the smallest molecular species index. *ubEnvModelVar* stores the model specific REAL and S32 type attribute values for the indexed interface grid unit box. This model routine updates *v_gridRHS* which holds the reaction term values **for the molecular species in this PDE**.

4.5.12 *updateIfGridAMRTags*

void updateIfGridAMRTags(const VIdx& vIdx, const NbrUBAgentData& nbrUBAgentData, con st NbrUBEnv& nbrUBEnv, Vector<S32>& v_finestLevel)

Set the desired adaptive mesh refinement (AMR) level for this interface grid unit box. *vIdx* indexes an interface grid unit box. *nbrUBAgentData* holds pointers for UBAgentData class instances for this interface grid unit box and its 26 neighboring unit boxes. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates *v_finestLevel* which holds the desired finest AMR level for this interface grid unit box **for the PDEs in this simulation**—index the vector using a PDE index, not a molecular species index.

4.5.13 *updateIfGridDirichletBCVal*

void updateIfGridDirchletBCVal(const S32 elemIdx, const VReal& pos, const S32 dim, const BO OL lowSide, const UBEnvModelVar a_ubEnvModelVar[3], const Vector<REAL>& av_gridPhi, REAL& bcVal)

Set the model specific Dirichlet boundary value for the *elemIdx*$^{\text{th}}$ molecular species for a point at the domain boundary face specified by *pos*, *dim*, and *lowSide*. This model routine is valid only when `BC_TYPE_DIRICHLET_MODEL` is selected. *pos* locates the point at the domeian boundary face. The domain boundary face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. The domain boundary face is at the low side (if *lowSide* is true) or the high side (if *lowSide* is false) of the simulation domain. *a_ubEnvModelVar* stores the modoel specific REAL and S32 type attribute values for the three interface grid unit boxes in the valid inteface grid region along the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) axis starting from the interface grid unit box including *pos* and facing the domain boundary. *av_gridPhi* holds the *elemIdx*$^{\text{th}}$ molecular species concentration values for the entire set of interface subgrid unit boxes contained in the three interface grid unit boxes. This model routine updates *bcVal*.

4.5.14 *updateIfGridNeumannBCVal*

void updateIfGridNeumannBCVal(const S32 elemIdx, const VReal& pos, const S32 dim, const B OOL lowSide, const UBEnvModelVar a_ubEnvModelVar[3], const Vector<REAL>& av_gridPhi , REAL& bcVal)

Set the model specific Neumann boundary value for the *elemIdx*$^{\text{th}}$ molecular species for a point at the domain boundary face specified by *pos*, *dim*, and *lowSide*. This model routine is valid only when `BC_TYPE_NEUMANN_MODEL` is selected. *pos* locates the point at the domeian boundary face. The domain boundary face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. The domain boundary face is at the low side (if *lowSide* is true) or the high side (if *lowSide* is false) of the simulation domain. *a_ubEnvModelVar* stores the modoel specific REAL and S32 type attribute values for the three interface grid unit boxes in the valid inteface grid region along the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) axis starting from the interface grid unit box including *pos* and facing the domain boundary. *av_gridPhi* holds the

elemIdx$^{\text{th}}$ molecular species concentration values for the entire set of interface subgrid unit boxes contained in the three interface grid unit boxes. This model routine updates *bcVal*.

4.5.15 *initPDEBufferPhi*

void initPDEBufferPhi(const VIdx& startVIdx, const VIdx& pdeBufferBoxSize, Vector<REAL> & v_gridPhi)

Initialize molecular concentrations of a PDE buffer unit box. *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. This model routine updates *v_gridPhi*.

4.5.16 *updatePDEBufferKappa*

void updatePDEBufferKappa(const S32 pdeIdx, const VIdx& startVIdx, const VIdx& pdeBufferB oxSize, REAL& gridKappa)

Set the ratio of the available volume (used in computing diffusion flux) for a unit PDE buffer box of the *pdeIdx*$^{\text{th}}$ PDE. *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. This model routine updates *gridKappa*. $0.0 < gridKappa \leq 1.0$. Set *gridKappa* to 1.0 to assume that 100% of the volume is available for diffusion. *gridKappa* is assumed to be 1.0 outside the simulation domain.

4.5.17 *updatePDEBufferAlpha*

void updatePDEBufferAlpha(const S32 elemIdx, const VIdx& startVIdx, const VIdx& pdeBufferB oxSize, REAL& gridAlpha)

Set the decay rate for the *elemIdx*$^{\text{th}}$ molecular species. *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. This model routine updates *gridAlpha*. $-1.0 < gridAlpha \leq 0.0$. Set *gridAlpha* to 0.0 to ignore decay.

4.5.18 *updatePDEBufferBetaInPDEBufferRegion*

void updatePDEBufferBetaInPDEBufferRegion(const S32 elemIdx, const S32 dim, const VIdx& s tartVIdx0, const VIdx& startVIdx1, const VIdx& pdeBufferBoxSize, REAL& gridBeta)

Set the diffusion coefficient for the *elemIdx*$^{\text{th}}$ molecular species for a PDE buffer grid unit box face strictly inside a PDE buffer region. The unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. *startVIdx0* points the low end of the unit box at the low side of the face and *startVIdx1* points the low end of the unit box at the high side of the face assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. This model routine updates *gridBeta*.

4.5.19 *updatePDEBufferBetaDomainBdry*

void updatePDEBufferBetaDomainBdry(const S32 elemIdx, const S32 dim, const VIdx& startVId x, const VIdx& pdeBufferBoxSize, REAL& gridBeta)

Set the diffusion coefficient for the *elemIdx*$^{\text{th}}$ molecular species for a PDE buffer unit box face at the domain boundary. The unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. This model routine updates *gridBeta*.

4.5.20 *updatePDEBufferRHSLinear*

void updatePDEBufferRHSLinear(const S32 elemIdx, const VIdx& startVIdx, const VIdx& pdeB ufferBoxSize, const REAL gridPhi, REAL& gridRHS)

Update the source term for the *elemIdx*$^{\text{th}}$ molecular species. This model routine is relevant only for linear PDEs (`PDE_TYPE_REACTION_DIFFUSION_STEADY_STATE_LINEAR` and `PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`). *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. *gridPhi* holds the molecular concentration of the *elemIdx*$^{\text{th}}$ molecular species for this PDE buffer grid unit box. This model routine updates *gridRHS*.

4.5.21 *adjustPDEBufferRHSTimeDependentLinear*

void adjustPDEBufferRHSTimeDependentLinear(const S32 elemIdx, const VIdx& startVIdx, con st VIdx& pdeBufferBoxSize, const REAL gridPhi, REAL& gridRHS)

Adjust the source term for the *elemIdx*$^{\text{th}}$ molecular species. This model routine is invoked in every *PDE time step* and is relevant only for time-dependent linear PDEs (`PDE_TYPE_REACTION_DIFFUSION_TIME_DEPENDENT_LINEAR`). This model routine is called only when users configure to call this function (Section 2.4.14). *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. *gridPhi* is the concentration of the *elemIdx*$^{\text{th}}$ molecular species in the unit box. *gridRHS* is initially set to the *gridRHS* value set inside the *updatePDEBufferRHSLinear()* model routine (if *PDE time step* is 0) or the *gridRHS* in the previous *PDE time step* (if *PDE time step* is not 0) for this molecular species. Users may change the value based on model specifics.

4.5.22 *updatePDEBufferRHSTimeDependentSplitting*

void updatePDEBufferRHSTimeDependentSplitting(const S32 pdeIdx, const VIdx& startVIdx, co nst VIdx& pdeBufferBoxSize, const Vector<double>& v_gridPhi, Vector<double>& v_gridRHS)

Set the derivative values for the reaction term for the *pdeIdx*$^{\text{th}}$ PDE which is solved using the

splitting scheme. *startVIdx* points the low end corner of a PDE buffer unit box assuming the interface grid spacing. *pdeBufferBoxSize* is the size of a PDE buffer unit box assuming the interface grid spacing. *v_gridPhi* holds molecular concentrations **for the molecular species in this PDE—** the size of the vector is equal to the number of molecular species in this PDE, and the first vector element holds the molecular concentration for the molecular species in this PDE with the smallest molecular species index. This model routine updates *v_gridRHS* which holds the derivative values **for the molecular species in this PDE**.

4.5.23 *updatePDEBufferDirichletBCVal*

void updatePDEBufferDirichletBCVal(const S32 elemIdx, const VReal& startPos, const VReal& pdeBufferFaceSize, const S32 dim, const BOOL lowSide, REAL& bcVal)

Set the model specific Dirichlet boundary value for the *elemIdx*$^{\text{th}}$ molecular species for a PDE buffer unit box face at the domain boundary. This model routine is valid only when `BC_TYPE_DIRICHLET_MODEL` is selected. *startPos* points the low end corner of the unit box face. *pdeBufferFaceSize* is the size of a PDE buffer unit box face. The unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. The face is at the low side (if *lowSide* is true) or the high side (if *lowSide* is false) of the simulation domain. This model routine updates *bcVal*.

4.5.24 *updatePDEBufferNeumannBCVal*

void updatePDEBufferNeumannBCVal(const S32 elemIdx, const VReal& startPos, const VReal& pdeBufferFaceSize, const S32 dim, const BOOL lowSide, REAL& bcVal)

Set model specific Neumann boundary values for the *elemIdx*$^{\text{th}}$ molecular species for a PDE buffer unit box face at the domain boundary. This model routine is valid only when `BC_TYPE_NEUMANN_MODEL` is selected. *startPos* points the low end corenr of the unit box face. *pdeBufferFaceSize* is the size of a PDE buffer unit box face. The unit box face is orthogonal to a unit vector in the x (if *dim* is 0), y (if *dim* is 1), or z (if *dim* is 2) direction. The face is at the low side (if *lowSide* is true) or the high side (if *lowSide* is false) of the simulation domain. This model routine updates *bcVal*.

4.6 Simulation Output

4.6.1 *updateSpAgentOutput*

void updateSpAgentOutput(const VIdx& vIdx, const SpAgent& spAgent, REAL& color, Vector<REAL>& v_extraScalar, Vector<VReal>& v_extraVector)

Set the output variables for a discrete agent. *vIdx* indexes the interface grid unit box this discrete agent (*spAgent*) is located in. This model routine updates the color variable (*color*) and the model specific extra scalar and 3D vector output variables (*v_extraScalar* and *v_extraVector*, respectively). The numbers of model specific scalar and 3D vector output variables for a discrete agent are set inside *updateFileOutputInfo()*.

4.6.2 *updateSummaryVar*

void updateSummaryVar(const VIdx& vIdx, const NbrUBAgentData& nbrUBAgentData, const NbrUBEnv& nbrUBEnv , Vector<REAL>& v_realVal, Vector<S32>& v_intVal)

Update summary variable values for this interface grid unit box. The entire set of summary variables (one variable per interface grid unit box) for each attribute are reduced to get a single summary value (one summary value per attribute). *vIdx* indexes an interface grid unit box. *nbrUBAgentData* holds pointers for UBAgentData class instances for this interface grid unit box and its 26 neighboring unit boxes. *nbrUBEnv* stores the state information (molecular concentration values and model specific REAL and S32 type attribute values) for the indexed interface grid unit box and its 26 neighboring unit boxes. This model routine updates the REAL (*v_realVal*) and S32 (*v_intVal*) type summary attribute values for this interface grid unit box.

5.0 Sample Implementations

Two sample models are included in the *Biocellion* 1.2 release.

5.1 Sorting

A property of some cell types is preferential adhesion to cells of the same type. When migrating cells of different type but sharing this property are intermixed, they segragate over time into clumps of same-type cells. Providing an introductory and abstract application of *Biocellion*, we have included a model of this kind of cell-sorting. You can find source in `$BIOCELLION_USER/model-sorting`.

5.2 Gut

Demonstrating application of *Biocellion* to an anatomical example, we have included a model of the intestial crypt-villus architecture based upon an independently published article (Cockrell et al. 2014) describing the authors' spatially explicit general-purpose model of enteric tissue (SEGMEnT). You can find the *Biocellion* model in `$BIOCELLION_USER/model-gut`.

6.0 *Biocellion* Installation and Execution

6.1 System Requirements

Biocellion runs on a spectrum of parallel computers ranging from multicore PCs and workstations to cluster computers, Cloud computers, and supercomputers—users do not need to change their model code to run on different platforms. The current version of *Biocellion* runs only on x86 compatible systems (a great majority of computers with Intel or AMD CPUs are x86 compatible). *Biocellion* runs on 64-bit Linux operating systems. Compiling *Biocellion* model code requires a `C++` compiler—we have tested using the GNU gcc compiler and the Intel icc compiler. *Biocellion* uses the Intel Thread Building Blocks library for multi-threading, and the library is freely available from the Thread Building Blocks homepage (`http://threadingbuildingblocks.org`). *Biocellion* 1.2 has been tested using Intel Thread Building Blocks version 2017 update 2, and we ask users to use Intel Thread Building Blocks 2017 update 2 or later.

6.2 Compiling *Biocellion*

6.2.1 Compiling Model Code

Users first copy the `$BIOCELLION_RELEASE_ROOT/biocellion-user` directory in the *Biocellion* release file to their private space (`$BIOCELLION_USER`) or create their own `$BIOCELLION_USER` directory, copy `$BIOCELLION_RELEASE_ROOT/$BIOCELLION_ROOT/libmodel/Makefile.model` and the `$BIOCELLION_RELEASE_ROOT/$BIOCELLION_ROOT/libmodel/model` directory (this directory contains empty template model routine files) to `$BIOCELLION_USER`. In either case, users need to set `BIOCELLION_ROOT` in `$BIOCELLION_USER/Makefile.model` to the full path of the `$BIOCELLION_ROOT` directory (which points the directory where `$BIOCELLION_RELEASE_ROOT/biocellion` is copied). We expect the model code to be placed under `$BIOCELLION_USER/model`. If the user copied the `$BIOCELLION_RELEASE_ROOT/biocellion-user` directory in the release file, one may pick one sample model directory (*e.g.* model-sorting) and rename to `model` or create a soft-link (*e.g.* under the `$BIOCELLION_USER` directory, type *ln -s model-sorting model*).

Users may update several additional variables in `$BIOCELLION_USER/Makefile.model` to use a different compiler or different compilation flags—the default is to use GNU gcc. We strongly encourage users to enable checks on model routine outputs and input arguments of *Biocellion* support functions by setting `CHECK_FLAG` to "-DENABLE_CHECK=1" at the early stage of model development. Once the model code is well verified, users may disable the checks (set `CHECK_FLAG` to "-DENABLE_CHECK=0") for faster simulation (enabling the check significantly increases the execution time).

Once `Makefile.model` has been revised, the user can compile by typing "make" under the `$BIOCELLION_USER/model` directory. When compilation completes successfully, `libmodel.DP.SPAGENT.so` will be created under `$BIOCELLION_USER/model`.

6.2.2 Compiling the *Biocellion* Core Framework

Advanced users may want to recompile the *Biocellion* core framework to use a different compiler with different optimization flags or to link the *Biocellion* core framework to a different flavor of MPI library (note that the release file does not include the *Biocellion* frameowkr source code and we do not intend to provide the source code). Please contact us (`http://biocellion.com`) if you need a *Biocellion* framework with a different configuration.

6.3 Simulation Configuration File

Biocellion asks users to provide specifics of a simulation instance in an XML file. Examples are located in the same directories as the sample models referred to in Sections 5.1 and 5.2.

6.3.1 Required Elements

- *time_step*: Set the number of *baseline time steps* to execute. For example, $<$*time_step num_baseline=''60000''*/$>$ asks *Biocellion* to run for 60,000 *baseline time steps*.

- *domain*: Set the domain sizes in the x, y, and z directions. For example, $<$*domain x=''128'' y=''128'' z=''4928''*/$>$ sets the domain size to 128 $\times$ 128 $\times$ 4928 assuming the interface grid spacing. Domain sizes in the x, y, and z directions should be equal to or larger than the minimum partition size (which is 4). If adaptive mesh refinement (AMR) is used to solve PDEs, domain sizes in the x, y, and z directions should be a positive integer multiple of $\textit{AMR ratio}^{\textit{interface grid level}}$ and domain sizes in the x, y, and z directions should be equal to or larger than $\textit{AMR ratio}^{\textit{interface grid level}} \times 2$.

- *init_data*: Set the initialization method, partition size, and initialization file type and path. Initialization method can be either "code" or "file". If the initialization method is set to "file", *Biocellion* users need to set file type ("binary" is the only valid option at this point) and path. For example, $<$*init_data partition_size=''32'' src=''file'' file_type=''binary'' path=''/data/input''*/$>$ sets *Biocellion* to set partition size to 32 and initialize using check point data under `/data/input`. Partition size should be equal to or larger than the minimum partition size (which is 4). If adaptive mesh refinement (AMR) is used to solve PDEs, partition size should be a positive integer multiple of $\textit{AMR ratio}^{\textit{interface grid level}}$ and partition size should be equal to or larger than $\textit{AMR ratio}^{\textit{interface grid level}} \times 2$. Note that the number of partitions in the simulation domain should be equal to or larger than the number of MPI processes in the target system to avoid idling MPI processes—if the amount of computing in each partition varies significantly, more partitions are necessary to load-balance.

- *output*: Set the output path, interval, and file formats ("pvtp" is the only valid option for discrete agent and "vthb" is the only valid option for molecular concentrations at this point). For example, $<$*output path=''/data/output'' interval=''100'' particle=''pvtp'' grid=''vthb''*/$>$ sets the file output path to `/data/output`, the file output interval to once per every 100 *baseline time steps*, the particle output format to the parallel vtkPolyData format, and the molecular concentration output format to the vtkOverlappingAMR format. Setting

"interval" to 0 turns off file output.

6.3.2 Optional Elements

- *model*: Set the model specific parameter string. Model routines can access the parameter string using *getModelParam()* (Section 3.1.12). For example, *<model param=''test''/>* sets the model specific parameter string to "test". The model specific parameter string is set to "" by default.

- *stdout*: Control the output redirection path, verbosity, and time stamping. If output redirection path is set, *Biocellion* redirects output (Section 3.4.2) messages to files. If not set, *Biocellion* prints output messages to the standard output. Verbosity can have a value between 0 and 5 (5 for the highest level of verbosity). If time stamping is enabled, *Biocellion* prints a time stamp before output (Section 3.4.2), warning (Section 3.4.3), and error (Section 3.4.4) messages. For example, *<stdout path=''.'' verbosity=''3'' time_stamp=''yes'' />* asks to redirect output messages to a file under `''.''`, sets the output verbosity level to 3, and enables time stamping. By default, output messages are printed to the standard output. The default verbosity level is 1. Time stamping is disabled by default.

- *system*: Set the number of node groups, number of nodes per node group, number of sockets per node, and the number of threads per MPI process. Set the number of sockets to the number of sockets per shared memory node for non-uniform memory access (NUMA) binding. Set the number of sockets per node to 1 to disable NUMA binding. For example, *<system num_node_groups=''1'' num_nodes_per_group=''1'' num_sockets_per_node=''1'' num_threads=''8''/>* sets the number of node groups, the number of nodes per group, and the number of sockets per node to 1, and sets the number of threads per MPI process to 8. The default number of node groups is 1. The default value for the number of nodes per node group is equal to the number of MPI processes in the MPI communicator. The default value for the number of sockets per node is 1 (disable NUMA binding). The default number of threads per MPI process is the number of cores (or hardware threads in a system with hardware multi-threading support) in the target system. Note that the number of node groups $\times$ the number of nodes per group $\times$ the number of sockets per node should coincide with the number of MPI processes in the MPI communicator.

- *super_partition*: Set the super partition size in the x, y, and z directions. This is relevant when running *Biocellion* on a large cluster with a large number of nodes and hierarchical interconnection network. *Biocellion* allows users to group closely located nodes. Multiple partitions in a single super partition is processed by the MPI processes in a single node group, and this reduces communication between distant nodes. The sizes of a super partition in the x, y, and z directions should be a positive integer multiple of the partition size. Note that the number of super partitions in the simulation domain should be equal to or larger than the number of node groups in the target system to avoid idling node groups—if the amount of computing in each super partition varies significantly, more super partitions are necessary to load-balance. *<super_partition x=''128'' y=''128'' z=''128''/>* sets the super partition sizes in the x, y, and z directions to 128. The default super partition size in the x, y, or z direction is the smallest positive integer multiple of the partition size that is equal to or larger

than the simulation domain size in the x, y, or z direction so the entire simulation domain is contained in a single super partition.

- *interval*: Set the intervals for summary output, load-balancing, AMR regridding, and checkpoint data output. For example, *<interval summary=''1'' load_balance=''100'' regriddi ng=''100'' checkpoint=''600''/>* sets *Biocellion* to print the summary every *baseline time step*, load balance once per every 100 *baseline time steps*, regrid AMR hierarchy once per every 100 *baseline time steps*, and generate checkpoint data every 600 *baseline time steps*. The load balancing interval should be a positive integer multiple of the regirdding interval. Setting the interval to 0 turns off summary, load balancing, regridding, or checkpoint data generation. Default values for summary output and checkpoint data generation are 0 and load balancing and regirdding are 100.

- *amr*: Set the refinement ratio between two consecutive AMR levels, and the desired fill ratio for AMR boxes. The fill ratio is the number of user tags in an AMR box divided by the number of unit boxes in the AMR box. If fill ratio is high, *Biocellion* tends to create a large number of tiny AMR boxes, which lowers the efficiency of computation. If fill ratio is set to a low value, *Biocellion* tends to create fewer boxes, but a larger percentage of the simulation domain is covered by fine grids. The default refinement ratio is 2, and the refinement ratio should be either 2 or 4. The default fill ratio is 0.5 and $0.0 \leq fill\ ratio \leq 1.0$.

6.4 Execution on Desktop PCs and Workstations

Biocellion usees Intel Thread Building Blocks (version 2017 update 2 or later) for multi-threading, and users need to set the `LD_LIBRARY_PATH` Linux environment variable to include the TBB library directory (in TBB 2017 update 2, this is `$TBB_ROOT/lib/intel64/gcc4.1`, `$TBB_ROOT/lib/intel64/gcc4.4`, or `$TBB_ROOT/lib/intel64/gcc4.7`) and the model directory (`$BIOCELLION_USER/model`). *Biocellion* executables are located under `$BIOCELLION_ROOT/framework/main`. `biocellion.DP.SPAGENT.DEBUG` is a *Biocellion* executable compiled with debugging support, and `biocellion.DP.SPAGENT.OPT` is an executable that delivers high-performance (DP stands for double precision, use the executables with SP instead of DP if single-precision floating point arithmetic is sufficient). Users execute *Biocellion* by typing `./biocellion.DP.SPAGENT.OPT simulation_configuration_file_name`.

6.5 Execution on Clusters

Use `biocellion.DP.SPAGENT.MPI.OPT` (or `biocellion.DP.SPAGENT.MPI.DEBUG` for debugging support) to run on clusters with the Ethernet interconnect and MPICH2 (`http://www.mpich.org/`). For clusters with the InfiniBand Interconnect (or other proprietary interconnects) or different flavors of MPI libraries, users need to recompile the *Biocellion* framework. `mpiexec -n number_of_MPI_processes --machinefile machine_file_name ./biocellion.DP.SPAGNT.MPI.OPT simulation_configuration_file_name` launches multiple MPI processes to run a large simulation using multiple nodes. Refer to MPI manuals for additional details. To run *Biocellion* on a cluster with a job scheduler, refer to a job scheduler manual or contact a system administrator.

6.6 Debugging Support

Biocellion installation includes binaries with debugging support. Users can enable checks on model routine outputs and input arguments to *Biocellion* support functions by enabling the check flag on compile time as stated in Section 6.2.1.

Users can set *Biocellion* to monitor ordinary differential equation (ODE) and partial differential equation (PDE) solver outputs to assure that output values are equal to or larger than the user specified minimum valid value (Sections 2.4.4 and 2.4.10). If PDE solver output values contain a value smaller than the specified minimum value, *Biocellion* prints the PDE parameters and the molecular concentrations of the unit box with the erroneous value and the neighboring boxes in the $\pm$ x, y, and z directions at the beginning of the PDE time step along with the erroneous value after the PDE time step which produced the erroneous value. *Biocellion* aborts after printing the error message.

6.7 Visualization Using Paraview

We briefly summarize visualization instructions assuming Paraview 5.2.0. See the Paraview homepage and manual for advanced visualization.

6.7.1 Visualizing Discrete Agents

1. Click the "File" menu and select the "Open" item.
2. Open a set of files in the parallel vtkPolyData format(.pvtp).
3. Click the "Properties" tab at the bottom left side and click the "Apply" icon.
4. Select the "Filters" menu, the "Alphabetical" sub-menu, and the "Glyph" item.
5. Select the "Properties" tab.
6. Set "Scalars" to "radius", "Glyph Type" to "Sphere", "Radius" to 1.0, "Scale Mode" to "Scalar", "Set Scale Factor" to 1.0, and "Maximum Number of Points" to "All point".
7. Click "Apply."
8. Click the "Display" tab.
9. Set "Color by" to "color".
10. Click "Zoom To Data." This maps discrete agents to a set of spheres.

6.7.2 Visualizing Molecular Concentrations

1. Click the "File" menu and select the "Open" item.

2. Open a set of files in the vtkOverlappingAMR file format (.vthb).

3. Click the “Properties” tab at the bottom left side, set the “Default Number of Levels” to 2, and click the “Apply” icon.

4. Select the “Filters” menu, the “Alphabetical” sub-menu, and the “Slice” item.

5. Select the “Property” tab and adjust the plane to slice, and click the “Apply” icon.

6. Select the “Display” tab.

7. Set “Color by” to phi.

8. Click “Zoom To Data.” This displays the molecular concentrations at the slice plane.

7.0 Tips and Caveats

- We strongly encourage *Biocellion* users to enable the checks on model routine outputs and input arguments to *Biocellion* support routines (Section 6.2.1) if the simulation does not work properly. This often guides users to find errors on their model routines.

- The relative and absolute error norm thresholds for multigrid iterations (Section 2.4.12) have significant impact on the accuracy and speed of simulation. If the norm thresholds are too large, users may see no change at all in molecular concentrations even though the source terms in PDEs are set to non-zero values. Setting these values to unnecessarily small values can significantly slow down the simulation on the other hand. Be cautious when setting these parameters.

- Note that double (instead of REAL) is used in *spAgentCRNODERHS()*, *updateIfSubgridRHSTimeDependentSplitting()*, and *updatePDEBufferRHSTimeDependentSplitting()*—this is mainly because the Intel ODE solver does not support single-precision arithmetic.

- Be cautious about setting Neumann boundary values when κ (see Sections 4.5.4 and 4.5.16) is set to a value smaller than 1.0. *Biocellion* does not consider κ in setting boundary values (κ is used only in computing diffusion flux). For example, users may expect there is no net diffusion if the Neumann boundary value for a boundary unit box face is set to 0.0. *Biocellion* sets the molecular concentration of the box outside the simulation domain to the molecular concentration of the box inside the simulation domain if the Neumann boundary value is 0.0—the gradient ignoring κ becomes 0.0. *Biocellion* assumes that κ outside the simulation domain is 1.0 and if the κ value for the box inside the simulation value is smaller than 1.0, $\frac{\phi_0}{\kappa_0} - \frac{\phi_1}{\kappa_1}$ can have a non-zero value. Users may set the diffusion coefficient at the boundary to 0.0 instead to apply the zero flux boundary condition.

8.0 C++ Basics Relevant to *Biocellion*

We provide several links for users who are new to C++.

- http://www.cplusplus.com/doc/tutorial/classes/ explains C++ class, class object (or instance), and access specifiers (private and public).
- http://www.cplusplus.com/reference/vector/vector explains C++ standard template library (STL) vector.
- http://en.wikipedia.org/wiki/Class_variable explains static member variables and functions.
- http://en.wikipedia.org/wiki/C%2B%2B_template explains C++ template
- http://en.wikipedia.org/wiki/Reference_(C%2B%2B) explains the concept of reference in C++.

9.0 Bibliography

Cockrell C, S Christley, and G An. 2014. "Investigation of inflammation and tissue patterning in the gut using a spatially explicit general-purpose model of enteric tissue (SEGMEnT)." *PLoS Comput Biol* 10(3):e1003507.

Colella P, DT Graves, JN Johnson, HS Johansen, ND Keen, TJ Ligocki, DF Martin, PW McCorquodale, D Modiano, PO Schwartz, TD Sternberg, and BV Straalen. 2012. *Chombo Software Package for AMR Applications Design Document*. Lawrence Berkeley National Laboratory.

Daly MJ. 2000. "Engineering radiation-resistant bacteria for environmental biotechnology." *Current Opinion in Biotechnology* 11(3):280–285.

Ha SJ, JM Galazka, SR Kim, JH Choi, X Yang, JH Seo, NL Glass, JHD Cate, , and YS Jin. 2011. "Engineered Saccharomyces cerevisiae capable of simultaneous cellobiose and xylose fermentation." *Proceedings of the National Academy of Sciences* 77(16):5822–5825.

Hall-Stoodley L, JW Costerton, and P Stoodley. 2004. "Bacterial biofilms: from the Natural environment to infectious diseases." *Nature Reviews Microbiology* 2:95–108.

Intel Corporation. 2008. *Intel Ordinary Differential Equation Solver Library Reference Manual*.

Jiao Y and S Torquato. 2011. "Emergent Behaviors from a Cellular Automaton Model for Invasive Tumor Growth in Heterogeneous Microenvironments." *PLoS Computational Biology* 7(12):e1002314.

Kang S, S Kahan, and B Momeni. TBP. *Methods in Molecular Biology*, Ch. Simulating Microbial Community Patterning using Biocellion. Springer.

Kim Y, MA Stolarska, and HG Othmer. 2007. "A hybrid model for tumor spheroid growth in vitro I: theoretical development and early results." *Mathematical Models and Methods in Applied Sciences* 17(1):1773–1798.

Majors PD, JS McLean, and JC Scholten. 2008. "NMR bioreactor development for live in-situ microbial functional analysis." *Journal of Magnetic Resonance* 192(1):159–166.

Melnicki MR, GE Pinchuk, EA Hill, LA Kucek, SM Stolyar, JK Fredrickson, AE Konopka, and AS Beliaev. 2013. "Feedback-controlled LED photobioreactor for photophysiological studies of cyanobacteria." *Bioresource Technology* 134:127–133.

Momeni B, KA Brileya, MW Fields, and W Shou. 2013. "Strong inter-population cooperation leads to partner intermixing in microbial communities." *eLife* p. 00230.

Newman TJ. 2005. "Modeling Multicellular Systems Using Subcellular Elements." *Mathematical Biosciences and Engineering* 2(3):611–622.

Resat H, V Bailey, LA McCue, and A Konopka. 2011. "Modeling Microbial Dynamics in Heterogeneous Environments: Growth on Soil Carbon Sources." *Microbial Ecology*.

Rubinstein MR, X Wang, W Liu, Y Hao, G Cai, and YW Han. 2013. "Fusobacterium nucleatum Promotes Colorectal Carcinogenesis by Modulating E-Cadherin/-Catenin Signaling via its FadA Adhesin." *Cell Host & Microbe*.

Sandersius SA, CJ Weijer, and TJ Newman. 2011. "Emergent cell and tissue dynamics from subcellular modeling of active biomechanical processes." *Physical Biology* 8(4):045007.

Singh JS, P Abhilash, H Singh, RP Singh, and D Singh. 2011. "Genetically engineered bacteria: An emerging tool for environmental remediation and future research perspectives." *Gene* 480(1–2):1–9.

Stamatakos GS. 2010. *Multiscale Cancer Modeling*, Ch. In Silico Oncology Part I–Clinically Oriented Cancer Multilevel Modeling Based on Discrete Event Simulation. CRC Press.

Strang G. 1968. "On the construction and comparison of difference schemes." *SIAM Journal on Numerical Analysis* 5(3):506–517.

Twizell EH, AB Gumel, and MA Arigu. 1996. "Second-order, L_0-stable methods for the heat equation with time-dependent boundary conditions." *Advances in Computational Mathematics* 6:333–352.

Weinberg RA. 2007. *The biology of cancer*. Garland Science.

Xavier JB, C Picioreanu, and MCM van Loosdrecht. 2005. "A framework for multidimensional modelling of activity and structure of multispecies biofilms." *Environmental Microbiology* 7(8):1085–1103.