Technical specification

Overview

Morpheus is built out of the following building blocks:

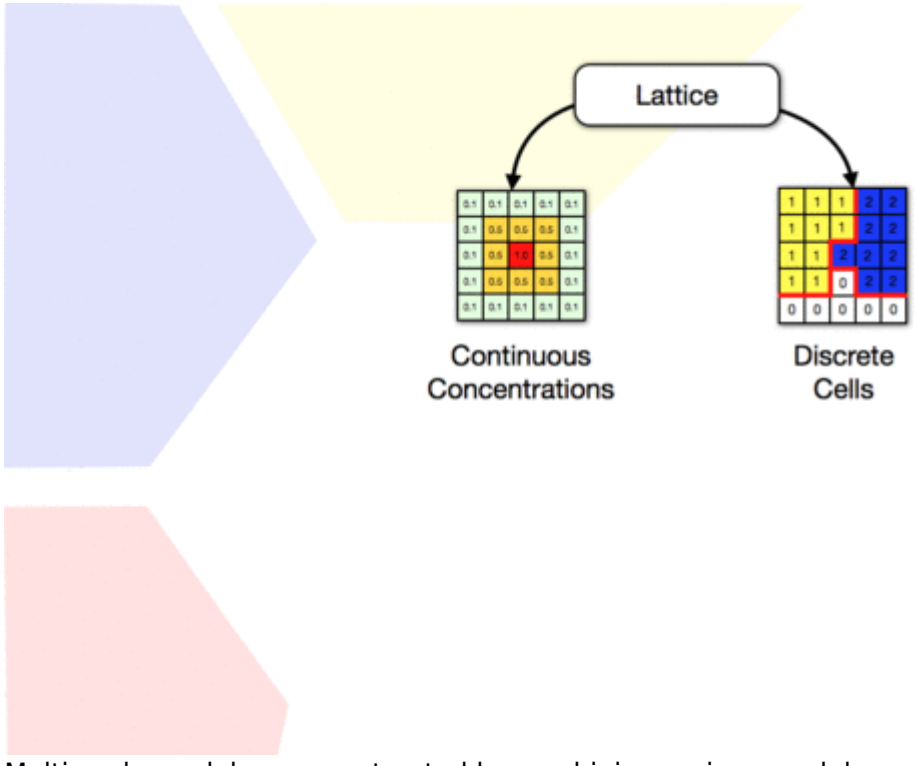
- **Equation solvers**: numerical solvers for systems of (differential) equations (ODE, SDE, PDE)
- **Lattice**: space discretization for both PDEs and discrete cell models (e.g. CPM)
- **Diffusion**: redistribution of concentration of species in PDE
- **Motility**: cell shape and motility of discrete cells in CPM

These components can be combined into complex models using

- **Automated model integration**: scheduling and spatial mapping

New features can be added through implementation of

- **Plugins**: framework for extensibility, based on various plugin interfaces



Multi-scale models are constructed by combining various modules.

Solvers

Ordinary differential equations

Morpheus implements various explicit finite different solvers of numerical integration of **ODEs**:

- Euler's method: 1st order
- Heun's method (a.k.a. midpoint): 2nd order

- Runge-Kutta's method: 4th order

Stochastic differential equations

In addition, Morpheus implements a method to scale the noise amplitude in stochastic ODEs (**SDEs**):

- Euler-Maruyama's method.

The latter is used automatically when using Euler or Heun method with a noise term. **Note**: Maruyama's method is not available when using Runge-Kutta's solver.

Limitations

Solvers with **adaptive time stepping**, such as the Runge-Kutta-Fehlberg method, are not available. Currently, we use fixed time stepping methods, because this greatly simplifies temporal scheduling of updates for automated model integration. Incorporation of adaptive time stepping is planned for a future release, though (see ODEint below).

Although these implemented explicit solvers are general and flexible, they are not applicable to **stiff systems**. Stiff systems are ODE systems that can change so rapidly that explicit solvers will result in numerical instability, unless the time step is extremely small. This situation may occur in systems where very large and small values are multiplied.

Planned: ODEint integration

Currently, Morpheus implements its own solvers. This was chosen because of the requirements of

- parsing of text-entered math expressions using muparser, and
- control over precise scheduling of updates.

However, we are planning to convert to the use of the extremely flexible numerical solver package ODEint that will soon be integrated into the well-known boost library.

This will extend the available methods for numerical integration of ODE solvers. For instance, adaptive time stepping solvers will become available. Implicit solvers, to solve stiff systems, may also become available (but will require the specification of the Jacobian).

Lattice

Lattice structure

Morpheus is a lattice-based simulation platform. This means that the spatial models are discretized on a lattice.

The following lattices are available:

- **linear** 1D lattice
- **square** 2D (orthogonal) lattice
- **hexagonal** 2D lattice
- **cubic** 3D (orthogonal) lattice

Membrane Properties

For `MembraneProperties`, properties that are resolved on the surface of cells, Morpheus uses a special lattices with polar coordinate system

- **circular** 1D lattice (for 2D models)
- **spherical** 2D lattice (for 3D models)

Boundary conditions

See description in FAQ

Diffusion

Method of lines

Partial differential equations (PDEs) are numerically approximated by separating the reaction and diffusion steps using the method of lines. This method discretizes the PDE into a system of coupled ODEs. These ODEs are then solved using the numerical methods mentioned above.

Diffusion equation

For diffusion, Morpheus uses the simple and general forward Euler scheme. The time step δt is automatically adjusted according to the CFL condition to guarantuee numerical stability.

Alignment of multidimensional lattices to 1D memory (due to use of valarrays) results in a high computational efficiency of the implementation of diffusion.

The unconditionally stable method of alternate direction implicit (ADI) is also implemented (for constant (Dirichlet) boundary conditions), but is currently not used.

Transport equation

Higher order derivatives to model transport equations in PDEs are currently being developed and are planned for a future release.

Motility

Cellular Potts model

Cell shape and motility is implemented according to the cellular Potts model (CPM)(Graner and Glazier, 1992).

In this model formalism, biological cells are represented as a domains of lattices sites $\boldsymbol{x}$ with identical index σ. In its simplest form (ignoring differential adhesion), the changes in configuration of cells on the lattice are governed by the Hamiltonian:

$H=\sum_{\sigma > 0} \lambda_V (v_\sigma - V_t)^2 + \sum_{\sigma > 0} \lambda_P (p_\sigma - P_t)$

where v_σ and p_σ are the actual volume and perimeter of the cell with index σ and V_t and P_t are the target volume and perimeter. Deviations from these target values increase the *free energy* H according to the scalars λ_V and λ_P.

Monte Carlo

The CPM is a Monte Carlo method in which the lattice is updated by randomly sampling lattice sites.

Within the CPM, a Monte Carlo step (MCS) is often taken as a discrete unit of time. A single Monte Carlo step is defined as the number of random sampled updates equal to the number of lattice sites, i.e. within one step, each lattice sites would have had chance to be updated.

Metropolis kinetics

In each update:

1. A lattice site $\boldsymbol{x}$ is chosen at random.

2. From the neighborhood N of $\boldsymbol{x}$, a second lattice site $\boldsymbol{x'}$ is chosen.

3. Then, the change in *free energy* ΔH is calculated *if* the state σ at $\boldsymbol{x'}$ ($\sigma_\boldsymbol{x'}$) would be copied to $\boldsymbol{x}$ according to the specific Hamiltonian H.

4. The proposed update is always accepted when $\Delta H<0$ and is accepted with a Boltzmann probability when $\Delta H>0$:

$P(\sigma_\boldsymbol{x'} \rightarrow \sigma_\boldsymbol{x})=\begin{cases} 1 &\mbox{if } \Delta H<0 \\ e^{{(\Delta H-Y)}/T} &\mbox{otherwise} \end{cases}$

where T (for 'temperature') modulates the probability of unfavorable updates to be accepted and

can be taken to represent local protrusions/retractions of the cell membrane. The parameter Y (for 'yield') is sometimes used to avoid oscillations with energy-neutral updates. This can be said to represent cytoskeletal resistance to membrane fluctuations.

Random number generators (RNG)

By default, Morpheus uses the Mersenne Twister 19937 pseudo-random number generator included in the CPP GNU compiler (TR1). Alternatively, the Boost RNG can be used (specified in CMake options).

Note: In multithreaded simulations, each thread gets its own RNG (the random seeds for which are based on the specified seed of the master thread RNG). Therefore, to reproduce simulation results, not only the random seed needs to be specified (`Time/RandomSeed`), but also the same number of threads (`Settings → Local→ threads`).

Model integration

To be documented.

Symbolic references and spatial mapping

In order to reference variables in mathematical terms in a multi-model environment one requires to identify the respective sub-model a variable is defined in and to perform a mapping between different (spatial) contexts, i.e. determine the corresponding CPM cell from a spatial position to access a cell property.

This ensemble of a variable definition and contextual meta-data we call a symbol, which can be referenced all over the multi-model environment. (Also allows to iteration over the elements od a symbol context).

In case of spatial ambiguity Morpheus provides a set of predefined `Reporters` to take care for the spatial mapping. For example, a PDE`Reporter` should be defined to compute the average concentration of a diffusible within the area occupied by a particular cell.

Example Autocrine Chemotaxis

<IMAGE>

h AutocrineChemotaxis.xml |h

```
extern>
http://imc.zih.tu-dresden.de/morpheus/examples/Multiscale/AutocrineChemotaxis.xml
```

Time scheduling

The time scheduler takes care (i) to evolve numerical schemes in time, (ii) to schedule and execute spatial data mappings, and (iii) to run event-based schemes, i.e. schemes which are run in certain time intervals. In order to combine different schemes in a single scheduler we exploit the method of fractional time steps. The basis is for our automatic scheduling is extensive knowledge of the data dependencies of the schemes and what kind of updated data they provide. This information is represented as a symbolic dependency graph based on the expressions, input and output symbols defined within the schemes.

We envisage the following criteria for the design of our scheduler implementation.

- Time stepping for all schemes shall be automatically adjusted, as far as possible, to the maximum permitted time step.
- In addition, time stepping can be overridden from in the MDL, allowing for user guided step sizes.
- The order of the specification of the schemes in the MDL shall not matter.
- The sequential order of updates within a temporal step shall be determined by the symbolic interdependency graph.
- Tightly coupled systems, often bearing circular dependencies, have to be defined in a **System** tag to be updated jointly.

A robust automatic scheduling methods necessarily needs reliable information about the internal limits of numerical schemes and its external dependencies. External dependencies are extracted as a dependency tree. Internal time step limits are reported by the schemes and can be adoptive in time. Solving Reaction-Diffusion Systems, for example, using a finite difference scheme and operator splitting allows us to schedule the diffusion solver depending on the CFL condition and running the reaction part independently.

Schematic Representation of the Time Scheduler

Initialisation:

1. Adjusting time steps:
    - Continuous Time Solvers: Ensure numerical stability and .
    - Mappers: Ensuring updated data is available as needed, but not computed more often than necessary. We let the time step size depend on the time steps of the downstream dependency tree (e.g a spatial Reporter shall run as often as it's output is needed by another scheme), and on stability criteria of the numerical schemes (e.g. CFL for Diffusion).
2. Sorting of schemes to be updated sequentially: The order is determined by the dependency tree. Circular dependencies cannot be sorted unambiguously and thus are rejected.
3. Initialisation of initial time state (e.g. mappings)

Main Loop: 3 Phase Time Stepper:

1. `Phase I:` Synchronous Time Evolution Schemes. The schemes are executed sequentially or in parallel, but the results are buffered as intermediates and only applied at the very end of that phase, such that any scheme exclusively depends on the previous time step solutions. Tightly coupled systems, defined in a **System** tag, are evolved by a system solver.
2. Time Step Progression
3. `Phase II:` Sequentially Updated Schemes (Mappers, Events), depending on data that has already been updated.
4. `Phase III:` Output generation in Analysis plug-ins. Can be performed in any order, just because there is nor interdependency.
5. Checkpointing of simulation state.

Plugins

Morpheus provides a plugin interface to extend the feature set. Plugins are written in C++ and requires recompilation from source. Therefore, this type of extensibility is only available when building from source (currently limited to developers and collaborators).

Morpheus has interfaces for different types of plugins:

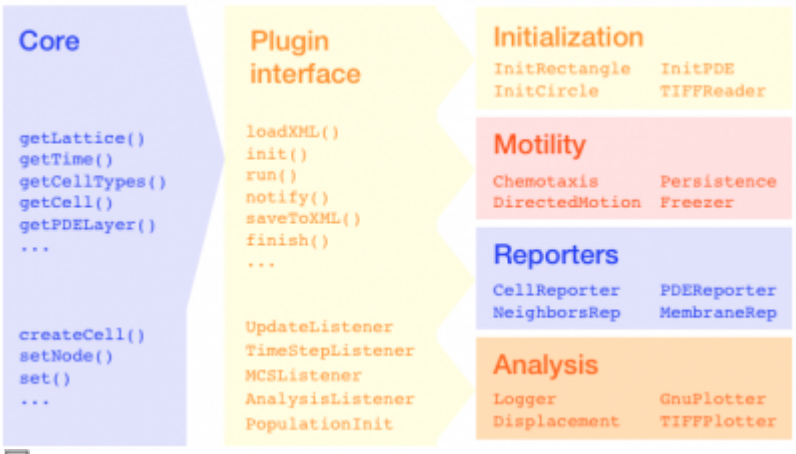
- **Initialization**: Plugins with methods to define an initial condition. Examples: `TIFFReader` and `InitPDEExpression`.
- **Reporters**: Plugins with customizable spatial mappings used in situations in which spatial mapping cannot be automated due to ambiguity. Examples are `NeighborsReporter` and `PDEReporter`.
- **Motility**: Plugins with methods for cell movement. Examples: `Chemotaxis` and `Persistence`.
- **Shape**: Plugins with methods to constrain cell shape. Examples: `VolumeConstraint` and `SurfaceConstraint`.
- **Analysis**: Plugins for data logging, plotting or processing. Examples: `Gnuplotter` and `HistogramLogger`.
- **Miscellaneous**: Set of plugins that do not fit in the categories above. Examples: `Proliferation` and `InsertMedium`.

Writing new plugins requires you to add three files:

- Header (MyPlugin.h): declares variables and functions
- Implementation (MyPlugin.cpp): implements the core functions
- XSD (MyPlugin.xsd): specifies the rules what constitutes valid XML input (used by GUI)

See an example for an `Analysis` plugin below.

Plugin interfaces



Plugin interface provides interfaces for various types of plugins.

Header file (MyPlugin.h)

```
<Analysis>
  <MyPlugin celltype="cells" symbol="s" interval="1.0"/>
<Analysis>
```

```
#include "core/interfaces.h"
#include "core/simulation.h"
#include "core/celltype.h"

class MyPlugin : public Analysis_Listener
{

public:
    DECLARE_PLUGIN("MyPlugin");
    NetworkLogger(){};
    virtual void loadFromXML(const XMLNode);
    virtual void notify(double time);
    virtual void init(double time);

private:
    string symbolstr, celltypestr;
    SymbolAccessor<double> symbol;
    shared_ptr<const CellType> celltype;
};
```

Inherit interface

Override functions

Symbol references

example header file

Implementation (MyPlugin.cpp)

```
 #include "myPlugin.h"
REGISTER_PLUGIN(MyPlugin);

void MyPlugin::loadFromXML(const XMLNode Node)
{
  Analysis_Listener::loadFromXML( Node );
  getXMLAttribute(Node,"celltype", celltypestr);
  getXMLAttribute(Node,"symbol", symbolstr);
}

void MyPlugin::init(double time)
{
   Analysis_Listener::init(time);
   celltype = CPM::findCellType(celltypestr);
   symbol   = SIM::findSymbol<double>(symbolstr, celltype);
}

void MyPlugin::notify(double time)
{
   Analysis_Listener::notify(time);
   for(uint c=0; c < celltype->getCellIDs().size(); c++){
      double value = CPM::getCell( cells[c] ).get( symbol );
      // do something
   }
   // write to file
}
```

Read parameters

Initialize references

Use symbols

example implementation file

XML Schema (MyPlugin.xsd)

```
<Analysis>
  <MyPlugin interval="1.0" celltype="cells" symbol="s"/>
<Analysis>
```

```
<xs:schema>
   <xs:complexType name="MyPlugin">
      <xs:annotation>
         <xs:documentation>Text shown in GUI</xs:documentation>
      </xs:annotation>
      <xs:attribute name="interval" type="cpmDouble"          use="required"/>
      <xs:attribute name="celltype" type="cpmCellTypeRef"     use="optional"/>
      <xs:attribute name="symbol"   type="cpmDoubleSymbolRef" use="required"/>
   </xs:complexType>
</xs:schema>
```

Documentation

Attributes

example XML schema file