Introduction to the Propag-5 code

Mark Potse

July 17, 2020
## Contents

Preface .......................................................... 3

1 Purpose ......................................................... 4
   1.1 General model formulation ............................. 4
   1.2 Bidomain reaction-diffusion model .................. 6
   1.3 Monodomain reaction-diffusion model .............. 6
   1.4 Forward model ......................................... 7
   1.5 Lead-field computation ............................... 7

2 Approach ....................................................... 9
   2.1 Spatial discretization .................................. 9
   2.2 Matrix storage ......................................... 10
   2.3 Temporal discretization .............................. 11
   2.4 Linear systems ........................................ 11
   2.5 Domain decomposition ............................... 12
   2.6 Parallelization, distributed mesh, communication .. 12
   2.7 Parameters ............................................ 13
   2.8 Input of a model description ....................... 13
   2.9 Node-specific parameters ............................ 13
   2.10 Output ................................................ 14

3 Concepts and data structures ............................... 15
   3.1 Glossary of concepts .................................. 15
   3.2 Data structures ....................................... 17

4 Control flow .................................................. 19
   4.1 Monodomain reaction-diffusion model .............. 19
   4.2 Bidomain reaction-diffusion model .................. 20
   4.3 Forward model ....................................... 20
   4.4 Lead-field computation .............................. 20
5 Code overview

5.1 Documentation .......................................................... 21
5.2 Dependencies ............................................................ 21
5.3 Installation ............................................................... 21
5.4 Modular organization ...................................................... 22
5.5 Housekeeping code ......................................................... 22
  5.5.1 The propag module .................................................. 22
  5.5.2 The types module ................................................... 22
  5.5.3 The memory module ................................................ 22
  5.5.4 The errors module .................................................. 24
  5.5.5 The logging module ................................................. 24
  5.5.6 The comm module ................................................. 24
5.6 File input/output .......................................................... 24
  5.6.1 The reading module ................................................. 24
  5.6.2 The forward module .............................................. 24
  5.6.3 The saving module ................................................ 24
5.7 The distributed mesh ..................................................... 25
  5.7.1 The distmesh module .............................................. 25
  5.7.2 The bootstrap module ............................................. 25
  5.7.3 The partition module ............................................. 25
  5.7.4 The repart module ................................................. 25
  5.7.5 The meshcomm module ................................... 25
  5.7.6 The anatomy module .............................................. 26
  5.7.7 The octree module ................................................. 26
  5.7.8 The ordering module ............................................. 26
5.8 The membrane models ...................................................... 26
  5.8.1 The ion module ................................................... 26
  5.8.2 The membrane modules ........................................... 26
5.9 Application modules ........................................................ 27
5.10 Numerical modules ........................................................ 27
5.11 Handling of parameters .................................................. 28
5.12 The main function ........................................................ 28
5.13 Experimental/dysfunctional modules ...................................... 29

6 Developments ............................................................... 30

6.1 Work to do ................................................................. 30
Preface

The documentation of the Propag-5 program is divided over 3 documents:

- *Introduction to Propag-5* (*propag-intro.pdf*) introduces the purpose and most important characteristics.

- *Propag-5 Manual* (*manual.pdf*) is the user manual; it describes the user interface and the usual workflows for readers who will not necessarily modify and compile the code themselves.

- *Introduction to the Propag-5 code* (this document) describes the algorithmic approach and code organization. This document is targeted at users/contributors who do wish to modify the code, or want to have a deeper understanding of it.

In addition, almost all of the about 30 modules that make up the Propag code are documented in cweb, providing excellent documentation of themselves.

**notation conventions** Throughout this document we use boldface type to indicate vectors, \( \mathbf{x} \) to indicate position, and \( t \) to indicate time. We occasionally write \( \partial_t \mathbf{X} \) for \( \frac{\partial \mathbf{X}}{\partial t} \) and such.
Chapter 1

Purpose

Propag-5 [7,10] is a program for the integration of reaction-diffusion equations, with much emphasis on large-scale simulation of action potentials and ionic current in the heart. It can operate in 5 major modes:

monodomain reaction-diffusion model: the most often-used method to simulate the macroscopic electrophysiology of the heart

bidomain reaction-diffusion model: a more accurate model that takes extracellular current flow into account

forward model: to compute extracellular potentials from the results of a reaction-diffusion model of the heart. The term forward model is traditionally used to refer to electrocardiogram (ECG) simulation but we use it also when referring to a simulation of local electrograms.

combined monodomain and forward model: to integrate the reaction-diffusion problem efficiently with a monodomain model, and still compute $\phi_e$ at the same time and at the same high spatial resolution

lead-field computation: is similar to a forward problem but with a very sparse right-hand side

1.1 General model formulation

Propag is based on the bidomain model of cardiac electrophysiology. The bidomain model is a homogeneous approximation of the electrically active cardiac tissue, which in reality consists of a network of interconnected muscle cells embedded in an extracellular matrix and other structures such as fibroblasts and capillaries. The bidomain model considers two co-located spaces: the intracellular domain, consisting of the interior of the cells and the gap junctions that connect them, and the extracellular domain, consisting of everything else.

The two domains of the bidomain model are characterized by conductivity tensors $G_i$ and $G_e$, respectively. A transmembrane current with surface density $I_m$ flows between the two domains and interacts with the potential field in each. We refer to these fields as the
intracellular potentials \( \phi_i \) and the extracellular potential \( \phi_e \). Their interaction with the transmembrane current can be expressed in terms of a double reaction-diffusion equation:

\[
\nabla \cdot (G_i(x) \nabla \phi_i(x, t)) = \beta(x)I_m(x, t) \\
\nabla \cdot (G_e(x) \nabla \phi_e(x, t)) = -\beta(x)I_m(x, t)
\]

(1.1)

(1.2)

where \( \beta \) is the membrane surface-to-volume ratio, i.e. the amount of membrane found in a given volume of tissue. The transmembrane current density \( I_m \) consists of a capacitive part, an ionic part \( I_{\text{ion}} \) generated by the cell membrane, and an imposed stimulation current density \( I_s \):

\[
I_m(x, t) = C_m \frac{\partial V_m(x, t)}{\partial t} + I_{\text{ion}}(V_m, y(x, t)) + I_s(x, t)
\]

(1.3)

where \( C_m \) is the membrane capacitance per unit area and \( y \) is a set of 20 to 50 variables that describe the state of the membrane at position \( x \).

Locally, the ionic current \( I_{\text{ion}}(V_m, y(x, t)) \) is given by a sum of individual currents \( I_i \) computed by expressions of the form

\[
I_i(t) = g_i \prod_j y_{ij}(t) [V_m(t) - E_i]
\]

(1.4)

where \( g_i \) is the maximum conductivity of the current, \( y_{ij} \) a small number of membrane state variables that are called “gating variables,” and \( E_i \) the “reversal potential” of the current, which depends on the ionic species that the current involves and their concentrations inside and outside the cell. The gating variables are governed by ordinary differential equations of the form

\[
\frac{\partial y}{\partial t} = \alpha(V_m)[1 - y] - \beta(V_m)y.
\]

(1.5)

In a large ensemble of channels, the variable \( y \) represents the fraction of gates that are open, and \( \alpha \) and \( \beta \) give the transition rates between the open and closed populations. These transition rates depend, usually in a nonlinear way, on \( V_m \) and are determined experimentally [4]. The concept of an open gate relates to the conformational state of the channel molecule in which the channel allows ions to pass. Not all membrane state variables are gating variables; there are also ion concentrations and sometimes other quantities. However, generally all state variables are governed by first-order differential equations, so we can write

\[
\frac{\partial y}{\partial t} = F(V_m, y)
\]

(1.6)

where \( F \) is a nonlinear vector-valued function of the current state. The pair of functions \( I_{\text{ion}} \) and \( F \) constitutes a membrane model. Four membrane models have been implemented in Propag and many more have been published [1,16].

For each of the two domains we need a boundary condition to ensure zero flow normal to each interface with a nonconductive part. For example,

\[
G_i \nabla \phi_i \cdot \partial \Omega_A = 0
\]

(1.7)

on the boundary \( \Omega_A \) of the active tissue and

\[
G_e \nabla \phi_e \cdot \partial \Omega_T = 0
\]

(1.8)

on a torso boundary \( \Omega_T \).
In summary, the bidomain model reads

\[
\begin{align*}
\nabla \cdot (G_l(x) \nabla \phi_l(x, t)) &= \beta(x) I_m(x, t) \\
\nabla \cdot (G_e(x) \nabla \phi_e(x, t)) &= -\beta(x) I_m(x, t) \\
I_m(x, t) &= C_m \partial_t V_m(x, t) + I_{ion}(V_m, y) + I_s(x, t) \\
\partial_t y &= F(V_m, y) \\
G_l \nabla \phi_l \cdot \partial \Omega_L &= 0 \\
G_e \nabla \phi_e \cdot \partial \Omega_T &= 0
\end{align*}
\]

(1.9)

This system does not have a unique solution; there is an arbitrary (i.e. physically meaningless) offset potential. Some authors therefore add a “grounding” equation to require \( \phi_e = 0 \) at some specific location.

### 1.2 Bidomain reaction-diffusion model

Using an operator splitting approach [17] one can write the bidomain equations in terms of \( \phi_e \) and \( V_m \) as an R-D equation

\[
\frac{\partial V_m}{\partial t} = \frac{1}{\beta C_m} \left\{ \nabla \cdot \left( G_l \nabla (V_m + \phi_e) \right) - \beta (I_{ion} + I_s) \right\}
\]

(1.10)

which is used to integrate \( V_m \) over time, and a linear system

\[
\nabla \cdot ((G_l + G_e) \nabla \phi_e) = -\nabla \cdot (G_l \nabla V_m)
\]

(1.11)

from which \( \phi_e \) can be solved. Together with a membrane model \( (F, G) \) this is named a bidomain reaction-diffusion model. Running a bidomain reaction-diffusion model means integrating this system over one or more heart beats. Propag does this in a loop consisting of 1) integrating the membrane model for each node, 2) evaluating the right-hand side of equation 1.10, 3) solving \( V_m \) from equation 1.10, 4) solving \( \phi_e \) from equation 1.11, 5) optionally outputting data.

In summary, a practical bidomain reaction-diffusion model reads

\[
\begin{align*}
\partial_t V_m(x, t) &= (\beta C_m)^{-1} \left\{ \nabla \cdot \left( G_l \nabla (V_m(x, t) + \phi_e(x, t)) \right) - \beta (I_{ion}(V_m, y) + I_s) \right\} \\
\nabla \cdot ((G_l(x) + G_e(x)) \nabla \phi_e(x, t)) &= -\nabla \cdot (G_l(x) \nabla V_m(x, t)) \\
\partial_t y &= F(V_m, y) \\
(G_l(x) + G_e(x)) \nabla \phi_e(x, t) \cdot \partial \Omega_T &= 0
\end{align*}
\]

(1.12)

### 1.3 Monodomain reaction-diffusion model

The monodomain reaction-diffusion model is a simplification of the bidomain reaction-diffusion model that can be derived by assuming that \( G_l \) is proportional to \( G_e: G_l = \xi G_e \) where \( \xi \) is a scalar. This leads to a linear relation between \( \phi_e \) and \( V_m \), allowing the system to be written as a single reaction-diffusion equation. The system (1.12) then reduces to

\[
\begin{align*}
\partial_t V_m(x, t) &= (\beta C_m)^{-1} \left\{ \nabla \cdot (G_e(x) \nabla V_m(x, t)) - \beta (I_{ion}(V_m, y) + I_s) \right\} \\
\partial_t y &= F(V_m, y)
\end{align*}
\]

(1.13)

Running a monodomain reaction-diffusion model means integrating this equation over one or more heart beats, i.e. 0.5 to 10 seconds. Typical time steps are in the order of 10 ms, so
there are in the order of $10^5$ to $10^6$ time steps in a simulation. Spatial step sizes need to be in the order of 100 to 200 µm, while a healthy human heart has about 200 ml of active tissue. This leads to models with in the order of $20 \cdot 10^6$ nodes.

In a monodomain reaction-diffusion model, the integration of the membrane state and the computation of $I_{ion}(V_m, t)$ at each model node is by far the most expensive part of the simulation.

The steps in a monodomain or bidomain integration are grouped in laps. At the end of a lap output is performed as configured, tactics for the next lap are considered, and a line is written to the logfile to report progress.

Propag recognizes two phases in the integration, phase 1 corresponding to the depolarization of the tissue and phase 2 to the repolarization. This is a simple scheme for adaptive time stepping. During phase 2 the time step can often be 2 to 5 times larger than during phase 1.

1.4 Forward model

By forward model we mean a computation of $\phi_e$ from given cardiac sources, by solving equation 1.11. The sources may be $V_m(x, t)$ itself, a current density field $I(x, t) = \nabla \cdot (G_i(x)\nabla V_m(x, t))$, or a current dipole field $J(x, t) = G_i(x)\nabla V_m(x, t)$. There are several reasons to run separate forward models:

- Bidomain reaction-diffusion models require a high spatial resolution. It is very expensive to simulate a whole body at this resolution, and Propag does not support inhomogeneous meshes (yet). Such inhomogeneous meshes could also be hard to solve. However, a separate computation with a lower-resolution body mesh can be done.

- Monodomain simulations are run on the myocardium alone, because they are meaningless in passive tissue. But $\phi_e$ can be computed from monodomain $V_m$ in a separate computation, using either a mesh of the heart or a lower-resolution mesh of the whole body.

- One may wish to experiment with different torso conductivities for the same cardiac simulation.

A forward model is typically run on a torso model with in the order of $10^7$ nodes, and for 500 to $10^4$ samples. Since it deals only with a set of static problems it has no notion of time.

1.5 Lead-field computation

A lead field is a Green’s function that is computed once for each ECG lead (typically 12 to 300). It is then used to compute the ECG at each time step of the reaction-diffusion model (typically 500 to several thousands) and, as long as the conductivity parameters are not changed, can be re-used for many simulations. This is an efficient method because the ECG computation with a lead field is cheap and highly scalable, in contrast to the solution of equation 1.11.

An ECG lead is a time-dependent potential difference between two points in the domain. When a field of transmembrane potentials $V_m(x, t)$ is given, an ECG lead potential $V(t)$ can be computed as

$$V(t) = \int \nabla Z(x) \cdot G_i(x)\nabla V_m(x, t) \, dx$$
where the integration is over the myocardium and \( Z(x) \) is the lead field [9] of the specific lead. The lead field is defined as the potential field created by a unit current applied at the electrode locations [3]:

\[
\nabla \cdot (G(x) \nabla Z(x)) = \begin{cases} 
-1 & \text{at the positive electrode,} \\
1 & \text{at the negative electrode, and} \\
0 & \text{elsewhere,} 
\end{cases}
\]

where \( G = G_i + G_e \) is the bulk conductivity \((G_i = 0 \text{ outside the myocardium})\) [11]. This is the same as solving equation 1.11, but for \( Z \) instead of \( \phi_e \). Lead fields can also be computed for configurations of more than two electrodes, as long as the sum of their currents is zero.

Lead fields are not only useful for ECG simulation but also for education. A visualization of a lead field illustrates how a given ECG lead “views” the heart (i.e. in which direction it is maximally sensitive to cardiac dipole sources, in different locations of the heart).

Propag runs a lead-field computation in a loop over all fields to be computed, similar to the samples in a forward model.
Chapter 2

Approach

2.1 Spatial discretization

Propag uses a semi-structured finite difference (FD) mesh. By “structured” we mean that the mesh elements all have the same size and shape; in case of Propag they are hexahedral elements. By “semi-structured” we mean that Propag does not store data for mesh elements that are irrelevant because they represent air or, in case of a monodomain model, represent non-myocardial tissue. A semi-structured mesh takes 5 times less space than a structured mesh for the whole heart. In case of an isolated pair of atria the difference is even 10 times, because the atria have a very thin wall.

The internal representation of the mesh in Propag (the distributed mesh, introduced in section 2.6) is very similar to an unstructured mesh. There is only information about the relevant nodes. For each of them there is a list of neighbour nodes, just like in an unstructured mesh. The only reminder of a structured mesh is that there is storage space for at most 18 neighbours (exactly the number we need for a node that does not lie at a boundary) and that instead of the spatial coordinates of a node we store only its vertex number; the (discrete) coordinates can be computed from the vertex number when needed.

For the computation of the differential operators $\nabla \cdot G_e \nabla$ etc. Propag uses expressions that assume constant conductivity tensors on the hexahedral mesh elements and take all heterogeneity into account [14]. From Propag’s perspective a tissue boundary is just another heterogeneity, so there is no special code to implement boundary conditions. The domain boundary is implemented by assuming $G_e = G_i = 0$ on the virtual elements outside the domain. The differential operators use values from 18 neighbouring nodes, of which 12 are zero in case of an isotropic medium.

This choice for the expression of differential operators, and the assumption of constant conductivity on elements, led to a dualism of mesh elements and mesh nodes that is quite pervasive in Propag. Conductivities are defined on elements, while active properties such as potentials and membrane state variables are defined on the nodes of the mesh. Gradients of active properties are defined on elements by averaging gradients along the 4 appropriate edges of the element. The model anatomy that Propag reads from file is given in elements. This governs the conductivities of the elements. The properties of the nodes are determined by the element properties using a method that assigns the same properties to all 8 corners of an element if that element has a higher priority than its neighbours. Myocardial elements have higher priority than non-myocardial elements, and conductive elements have higher priority than non-conductive elements. The amount of membrane represented by a node (a
CHAPTER 2. APPROACH

Figure 2.1 Finite-difference grid illustrated in 2 dimensions. The gray area represents one of the 3 media in one of the 2 domains, e.g. active tissue in the intracellular domain. Filled circles represent grid nodes that are internal to the medium, open circles nodes that are external. Grid squares correspond to “elements”; only elements that are completely inside the medium have nonzero conductivity. Dashed squares indicate the areas represented by three example nodes: one inside the medium, which has the standard \( \beta \) value; one on a boundary, with \( \beta/2 \); and one on a corner, with \( \beta/4 \). Internal nodes cannot occur as pseudopods, without a neighboring internal element.

scaling factor for \( \beta \) depends on the type of all 8 elements surrounding it (figure 2.1). This prevents an inconsistency between the current strength and the conductivity at myocardium boundaries. The priorities are determined by 3 “medium types:” void (lowest priority), passive tissue, and active tissue (highest priority).

We use the following terminology to describe the different mesh entities unambiguously:

- A voxel is each of the hexahedral elements that make up the cube of the simulation domain.
- A cube is a voxel with a non-void type.
- A vertex is each of the corners of the voxels.
- A node is a vertex with a non-void type.

In general, Propag only stores data for cubes and nodes, but it reads its anatomy in terms of voxels and outputs simulation results also on a full vertex or voxel mesh, using a blank value for all non-node or non-cube positions.

2.2 Matrix storage

After discretization the differential operators of the form \( \nabla \cdot (G \nabla \cdot ) \) become \( N_n \times N_n \) matrices where \( N_n \) is the number of nodes in the mesh. As discussed in section 2.1 these operators consider 18 neighbours, so there are up to 19 nonzero elements per matrix row.

Our storage scheme allocates storage for exactly 19 elements per matrix row. It consists of an array \( C \) for the column numbers and an array \( A \) for the (possibly) nonzero values. All non-stored values are zero by design. The stored values can also be zero, for example if the corresponding node does not exist and \( G \) is zero on the voxels that lie inbetween. The 19 nodes of the difference stencil have fixed positions in \( C \) and \( A \). If the corresponding node does not exist, the value in \( A \) is zero and the value in \( C \) is \(-1\). Our vectors have an extra
zero element at position $-1$ so that we can safely index them with this special value when a matrix-vector product is computed. Thus we don’t need to place exceptions in the loop.

The arrays $C$ and $A$ can be thought of as having dimensions $N_n \times 19$ but for performance are coded as flat arrays.

The conductivity tensor fields $G(x)$ are symmetric so they have 6 elements per cube. They are stored as (conceptually) a $N_c \times 6$ array where $N_c$ is the number of cubes.

Similarly, gradient operators $G\nabla$ for cubes would be $N_c \times 3 \times 8$ arrays, but we currently don’t store them as such a product is easy enough to code where needed, and then relies only on the 6 elements per cube of $G$.

There is a storage format for gradient operators on nodes that uses $N_n \times 3 \times 7$ elements storage. It is only allocated when needed and might be removed because gradients on cubes fit better in the methods we use.

### 2.3 Temporal discretization

The membrane models in Propag are always integrated with explicit methods, in most cases using an exponential method known in the field as the Rush-Larsen method [12]. Integration of potentials is usually done with a Forward Euler method, but a Backward Euler method is also available. When this is used we speak of the whole model integration as an implicit-explicit (IMEX) method. IMEX integration is particularly interesting when the spatial stepsize is small, so that the temporal stepsize for an explicit method would be prohibitively small. In most cases, however, the poorer scalability of an IMEX approach makes the explicit method more attractive.

The discrete version of the monodomain reaction-diffusion equation (1.13) for node $i$ at time step $t$ reads

\[
\begin{cases}
V_{m_i}^{t+1} = (\beta C_m)^{-1} \left\{ \sum_{j=0}^{18} A_{ij} V_{m_i}^t C_{ij} - \beta (I_{\text{ion}}(V_{m_i}^t, y_i^t) + I_s) \right\} \\
y_{i}^{t+1} = F(V_{m_i}^t, y_i^t)
\end{cases}
\] (2.1)

where the matrices $A$ and $C$ represent the diffusion operator $\nabla \cdot G' \nabla$, as in section 2.2.

### 2.4 Linear systems

To solve equation 1.11 Propag has its own implementation of a modified BiCGStab solver [10]. In addition Propag interfaces to the PETSc library from which it can use alternative solvers. The preconditioner used for equation 1.11 is typically BoomerAMG (from PETSc/Hypre); several other preconditioners can be accessed through a unified interface.

For backward Euler integration we typically use BiCGStab with a Jacobi preconditioner.

We do not adapt the problem matrices to make the problem fully determined. The Krylov subspace solvers that we use will happily converge to one of the possible solutions. “Grounding” in Propag means that we subtract the potential at a given node from the solution of the problem. This is only done for the user’s convenience. However, the BiCGStab solver has provisions to prevent large nullspace components, as these could reduce the accuracy of the search directions [10].
2.5 Domain decomposition

Propag interfaces to ParMetis for domain decomposition. The call to ParMetis is performed in parallel mode, to allow loading of meshes that could not be fit on a single system node. For this purpose we use a “bootstrap mesh” based on a trivial decomposition. In the bootstrap phase Propag loads all voxels from the anatomy files, uses the substance definitions and mapping parameters to decide which voxels become cubes, creates the surrounding nodes, and performs the domain decomposition and distribution of the final mesh.

2.6 Parallelization, distributed mesh, communication

The code has an mpi parallelization layer in addition to the OpenMP parallelization that it inherited from Propag-4. All mpi functions that we call are wrapped in functions in the comm module (section 5.5.6), which allows the code to be compiled without mpi support (but severely handicapped of course).

Each process handles a part (subdomain) of the mesh. A process owns a domain of model cubes and has no information about neighbouring cubes. The processes do have information about 1 layer of nodes outside their domain (the “halo” of shared nodes), which they need to evaluate differential operators. Each shared node has one “owner” process, which is determined in an arbitrary way during mesh distribution. Matrices are stored incompletely: a process will have matrix rows for all nodes that it owns or shares, but columns only for the nodes it owns. The result of a matrix-vector product is therefore incomplete (it has partial values) and inconsistent (different processes have different values for the same node). Matrix-vector products are therefore followed by a sumup at owner operation, which makes the values at the owned nodes complete. In some cases this suffices because the copies at the other processes are not used in the next operation. Otherwise a copy owner to others operation follows to make the vector consistent. These operations are performed by a family of functions defined in the meshcomm module (section 5.7.5).

The sumup at owner functions (and similar ones such as maximum, and norm) and the copy owner to others functions perform inter-process communication based on the concept of communication traces [13]. Basically this means that processes learn at initialization time which data they will receive from each other during the computations, and that henceforth only the data themselves are transferred.

There is no global-to-local mapping for either cubes or nodes, but each process has a local-to-global mapping in terms of vertex numbers, from which the vertex coordinates can easily be computed. Vertex coordinates are needed in the mesh partitioning process and occasionally also to establish the identity of neighbouring cubes and nodes outside the halo of a mesh part. Efficient lookup of nodes or cubes in such cases is done with octree structures, which are implemented in the octree module (section 5.7.7).

The distributed mesh and the models that define and create it is discussed in more detail in section 5.7.

Within each process there is an OpenMP parallelization at the level of loops. We use a static distribution scheme to ensure that threads access the nearest possible memory on systems with non-uniform memory access (NUMA).

The number of system nodes and the balance between mpi and OpenMP parallelization can be chosen at will, as long as it provides enough memory. However, the BoomerAMG preconditioner seems to have limited support for thread-level parallelism, so that it must always be used with a very low number of threads.
2.7 Parameters

Propag can be entirely configured at runtime. The parameters can be given on the command line and in a set of configuration files (referred to by command-line parameters). The parameter scanners are produced by a program named PRM that was developed at the Institute of Biomedical Engineering of the Université de Montréal, where Propag versions 1 through 4 were developed. In addition there are configuration files in dedicated formats to govern stimulation sites, output sampling, and lead-field computation, which would be inconvenient to express in terms of a simple “parameter equals value” scheme. The user manual shows examples of these.

2.8 Input of a model description

The model anatomy is read from 4 files, which give the type of each model voxel and three angles specifying the orientation of the conductivity tensor at each voxel. Heterogeneity in membrane model parameters is typically implemented with discrete types. A membrane model typically implements a small number of model variants. The user specifies a mapping from voxel types to membranes and variants. Another user-specified mapping gives the substance for each voxel; this determines the conductivity values and $\beta$.

Some membrane parameters can be specified on a per-node basis. This is done by giving Propag a file with a value for each vertex (so the dimensions of these data are 1 element larger than the anatomy in each spatial dimension).

Typical dimensions for model descriptions are in the order of 500 voxels in each dimension, both for cardiac meshes (at 200 $\mu$m resolution) and for torso meshes (at 1 mm resolution).

2.9 Node-specific parameters

The membrane models in Propag work with a small set of “cell types” or sub-models, each of which can have different parameter settings; this is entirely determined by the particular membrane model. However, sometimes it is necessary to give each node in the model its own value for a particular parameter. The membrane models can allow for this through a special mechanism. Since this causes extra complexity in the code and extra time it is implemented only for a small number of parameters. This mechanism is not active unless it is enabled with a parameter. For example, the option tnnp.Ko-param allows the TNNP model to use a node-specific parameter for the extracellular potassium concentration.

The parameter values come from files, much like in the geometry description, except that the latter is given on elements and the membrane parameters on nodes. The fname_param parameter (an array) tells Propag which files to read to obtain node-specific parameter values.

Parameters like tnnp.Ko-param can be set to an integer value $k$, which means that the file given by the $k$th element of fname_param will be associated with this parameter, for any node that uses the TNNP membrane.

The parameter values are given to the membrane model in the param element of a struct Membrane_cell_info that is given as input to the step function. The param element is a (short) dynamically allocated array of floating-point values. The order of the values in param does not necessarily correspond to the order of fname_param; it depends on which
node-specific parameters are enabled in the membrane model and on the hardcoded order in which the membrane model specifies them. The info function of the membrane model tells the main program from which files it wants to obtain values, and in which element of param they should be placed. With this information the main program also allocates param for each node.

The param arrays thus don’t have unused elements. They are allocated and filled in at initialization time. When this is done the input file buffers for the parameter files are discarded.

Parameter files can be used by different membrane models, at different positions in the param array, and even for multiple positions.

2.10 Output

A typical Propag run produces far too much data to save to file. Typically, variables are output once per millisecond of simulated time, i.e. once per 100 time steps. Spatially data are also subsampled, typically with a factor 1:5 in each spatial dimension. The output subsampling format can be specified, for each variable separately if desired, in a configuration file. It is possible to output specific regions or intervals at higher resolutions than the rest.

Small arrays are output in the IGB format on which most of our workflow is based. This is done with (collective) MPI I/O.

Large arrays are output in an intermediate HDF5 format, using one output file per process or for a small group of processes. This is done through the NetCDF API. A separate program called prmerge gathers the files and produces a single file in IGB format. On HPC systems this can be done in a separate job on a single node.
Chapter 3

Concepts and data structures

3.1 Glossary of concepts

Here we list some of the important concepts in Propag as well as a few words that we use with a very specific meaning. The following relate to mesh partitioning and parallel operation.

**part** a subset of the set of cubes or nodes (*entities*). Typically, one process handles one part.

**partition** We have also used the word “partition” to indicate either a *part* or a *partitioning*. The English language is rather sloppy about words like partition and division, which can mean both a part and the way something is divided into parts. We will therefore avoid the word “partition,” and use either part or partitioning.

**partitioning** When used as a direct object, this gerund means the collection of all partitions, and the specific way in which it is divided. Otherwise it means the action of dividing something into parts.

We represent a partitioning as an integer array with the process ranks as entries.

The partitioning of cubes induces a partitioning of nodes by including a node in part $p$ if and only if there is a cube in part $p$ that contains the node.

The partitioning of the cubes is *non-overlapping* in the sense that each cube is in exactly one part. This is not the case for the induced partitioning of the nodes. Handling this *overlap* is a complicated task.

**domain of a process** This is the union of all the cubes in the part.

**repartitioning** Given a distributed mesh, *repartitioning* (verb: to repartition) is the process of sending the node and cube data to the new target process for each entity as defined by the new partitioning. This involves a lot of index conversions creating a large bunch of complicated and error-prone code.

One possible approach to repartitioning would be to send the element and cubes along with their GIDs (globally unique ids) to the new owner process and afterwards assemble the connectivity matrix. However, implementing the global-to-local mapping can be demanding when the index input ids are highly scattered. The Propag code repartitions without referring to global-to-local maps. Rather than exchanging data and
computing connectivity in a second step we compute connectivity prior to exchanging
the data. This allows us to employ a local-to-local map (from the index range of the
current process to the target processes) which is simpler to implement.

Note that this approach is not suited to “incremental” repartitioning as used, e.g., for
dynamic load balancing.

**shared nodes** Since the partitioning of nodes is overlapping, the “same” node (when look-
ing from a global point of view) is stored on up to 8 processes (since a node has 8
surrounding cubes). Such a node is called a *shared node*, or a *boundary node*, since
it lies on an inter-process boundary. One of these processes is said to *own* the node.
This node is called the *master copy*, all other are *copies*.

**owner** of a shared node: each node has a unique owner process, which has a special role in
making data values on the node complete and consistent.

In the *distmesh* module we need to deal with two different kinds of *ownership*: On the
one hand, there are owner processes for the nodes in the current bootstrap mesh. On
the other hand we also have owner processes for the distribute mesh created during
repartitioning. Of course the owner processes for a single node will in general not
coincide. If we speak of the “current” owner of the node we refer to the owner according
to the distribution of the bootstrap mesh.

In the same way, nodes that are shared in the bootstrap mesh will in general not be
shared in the distributed mesh and vice versa.

**copies** of a shared node: storage for shared node on the processes that share but do not
own the node. There can be up to 7 copies of a node. Maybe we also call the one on
the owner process a copy too; then there are up to 8.

**process** each of the processes that *MPI* spawns for a program run. Typically, one process
handles one domain. Ideally, each process runs on its own *processor* or *cpu*. A process
may have multiple *threads*. If a *cpu* has multiple *cores*, we can either associate a
process with each core, or associate a *thread* with each core. The latter is presumably
more efficient, because it abolishes *MPI* communication between the *cores* of a single
*cpu*.

**core** The actual processing unit inside a multicore *processor* chip. Ideally, each *thread* or
each *process* is associated with a single core. *Cores* with *hyperthreading* capability can
handle more than one thread.

**processor** a word to avoid in our documentation, because a program deals with *processes*
and *threads*, and not with cores, processors, chips, boards, and cabinets, unless we are
doing something really machine-specific.

**cpu** another word to avoid because although it should refer to a core, it is often used to
indicate a chip.

**process interface** The intersection surface of two domains on different processes. For the
bootstrap mesh the interface is a plane. The nodes on this surface have multiple copies.

**halo** There are different names for this concept in the literature. Another popular name
for this concept *ghost elements* or *ghost layer* but this name is also used for a trick
to handle boundaries in finite-difference methods with uniform stencils. The halo
provides additional layers of cubes or nodes around the domain of a process to cache
data owned by other processes [5].
incomplete storage (of data values on shared nodes): each process has a partial value for a node, which need to be added up for all sharing processes to make it complete. The code documentation may confuse this occasionally with “inconsistent storage.”

inconsistent storage (of data values on shared nodes): the value of a quantity on a node is not the same on all processes that share the node. Typically the node’s owner has the correct value and it must be distributed to the others.

The following terms, introduced in section 2.1 are used to refer to mesh elements:

voxel each of the hexahedral elements that make up the cube of the simulation domain.
cube a voxel with a non-void type.
vertex each of the corners of the voxels.
node a vertex with a non-void type.

Other terminology:

membrane model A mathematical model of the electrophysiological behaviour of (a small part) of an electrically active cell (see the document Introduction to Propag-5), or our implementation thereof in one of the membrane modules.

3.2 Data structures

Main data structures are the distributed mesh, cube and node properties, the status arrays for voltages and membrane parameters, tensor fields, differential operators, ...

the distributed mesh Each process has information about its own part of the mesh and about the interface with the neighbouring parts. This information is stored in an instance of the DistMesh structure which is stored in the global variable mesh. It contains the mappings of node to vertex and cube to voxel numbers, ownership of nodes, types of cubes and nodes, the list of neighbours of each node (matrix C in section 2.2), information on the communication traces, and numbers of various entities such as the number of nodes and cubes in the part, nnodes and ncubes.

the bootstrap mesh A simpler mesh structure that is only used during initialization, to load the anatomy and create the final distributed mesh.

model status variables The transmembrane and extracellular potential fields are stored in arrays over nodes in the variables Vmem and Vex. We never compute or store intracellular potentials.

The values of the various membrane status variables are stored in a large array yyy, which is conceptually a table with nnodes rows. The number of columns equals the maximum of the number of variables used by all the membrane models that are used by the simulation.

state observations There is global storage for the latest depolarization time of each node (dtime), next stimulation time (atime), ionic current at the present time step (Iion).
Conductivity tensors are $n_{\text{cubes}} \times 6$ arrays that are allocated and computed locally when they are needed, for example for the computation of a differential operator. Conductivity tensors are computed by the \texttt{conductivities} function (\texttt{anatomy} module) from cube types, substance definitions, and fiber angles.

Fiber angles: The three angles that define fiber and sheet orientation are kept in global arrays \texttt{angle\_p}, \texttt{angle\_a}, and \texttt{angle\_g}, ranging over cubes.

Differential operators: The most often-used operators are the diffusion operators of the form $\nabla \cdot G \nabla$ where $G$ is a conductivity tensor field. These are computed once and stored in global variables by the main function. Diffusion operators are conceptually $n_{\text{nodes}} \times 19$ matrices, as the expressions that we use depend on 18 neighbor nodes. We used a fixed order to store information relating to these neighbors (see also section 2.2).

Gradient operators $G \nabla V$ where $V$ is a scalar field are used occasionally; these are computed and stored/freed near the point where they are used.

Problem matrices: The problem matrices for the elliptic bidomain problem equation 1.11 and the implicit Euler integration have the same sparsity pattern as the differential operators and are stored in the same way. They are computed when needed by the initializers of the \texttt{bidofex} and \texttt{euler} modules, respectively, and kept during the computation.

Program parameters: The parameters declared by the \texttt{prm} program are stored in global variables. In case of simple parameters the variable name starts with \texttt{prm\_}. In case of parameter structs there is no common prefix.

The substance mapping and membrane mapping parameters are so often used that we translate them to internal numerical formats at initialization; these values are stored in \texttt{tmap} and \texttt{cell\_info}. The latter also includes any node-specific parameter values that may have been imported.

MPI communicator: The MPI “communicator” is a central piece of information for MPI communication. It is kept in static storage by the \texttt{comm} module and completely hidden from the rest of the program.
Chapter 4

Control flow

Each run begins with a common initialization phase to read parameters and input files, partition the mesh, allocate storage, and compute storage matrices. This is treated in more detail in section 5.12.

4.1 Monodomain reaction-diffusion model

In its simplest form the control flow for a monodomain reaction-diffusion model is very simple (figure 4.1). Time stepping takes place in two loops: the inner over time steps, and the outer over “laps.” Output takes place once per lap. The duration of a lap is often 1 ms but it can be freely chosen.

However, usually a more efficient but more complex version is used in which communication of the diffusion current is partially overlapped with the integration of the membrane models. This is shown in figure 4.2.

```plaintext
while(simtime<Tend){
    for(step=0; step<Nsteps_per_lap; step++)
    {
        compute diffusion current Idif
        communicate Idif across partition boundaries
        integrate membrane models
        integrate Vm
        simtime += dt;
    }
    output
}
```

Figure 4.1 Control flow in a monodomain reaction-diffusion model (without overlap).
while(simtime<Tend){
    for(step=0; step<Nsteps_per_lap; step++){
        compute diffusion current \(Idif\)
        gather \(Idif\) on owner
        start copying \(Idif\) to others
        integrate membrane model on owned nodes
        block until copy of \(Idif\) is complete
        integrate membrane model on non-owned nodes
        integrate \(V_m\)
        simtime += dt;
    }
    output
}

Figure 4.2  Control flow in a monodomain reaction-diffusion model with overlap.

4.2  Bidomain reaction-diffusion model

4.3  Forward model

4.4  Lead-field computation
Chapter 5

Code overview

5.1 Documentation

The source code of Propag is written and documented using the \texttt{cweb}() system for structured documentation \cite{6,8}. The compilable C program code as well as the typeset document are created by filter programs from a common source file written in “\texttt{cweb}” format. The \texttt{cweb} system allows the author of the program to divide the code in tangible pieces, called “refinements,” and present these in an appropriate order for presentation to human readers. The program code is built up with a hierarchy of refinements; the program text that will be presented to the compiler is the top of this hierarchy.

Each of the modules in the Propag code corresponds to one \texttt{cweb} document and vice versa. Typically the \texttt{cweb} document generates the header, implementation, and the parameter description file for PRM if there is any.

5.2 Dependencies

Propag depends on the \texttt{mpi}, PETSc/Hypre, ParMetis, and HDF5/NetCDF libraries. Propag is coded in standard C (C98). OpenMP support in the compiler is optional; OpenMP parallelism improves performance but is not necessary.

The scanner for the output configurations is implemented in lex and yacc.

The parameter scanners and online documentation of the parameters are generated by the PRM program.

The membrane models on which Propag relies are implemented specifically for use with Propag (memory management, handling of errors and warnings) but can also be used elsewhere.

5.3 Installation

Propag is packaged with a few libraries and toolsets that are necessary to use it. The main tree contains the following directories:

\texttt{bin} scripts for compilation
CHAPTER 5. CODE OVERVIEW

format documentation formatting instructions for TeX/LaTeX
igb a library to read and write IGB file headers
make shared code for Makefiles
membranes the membrane models
paravis
 prm the PRM program and its library
propag5 the propag5 and prmerge5 programs
tools a set of tools that operate on IGB files
util utility functions shared by propag and the tools
VisIt a bit of code to read IGB files with the VisIt program

Detailed installation instructions are given in the README file at the root of the tree.
For propag itself we have a Makefile system where we recognize the host on which we compile
and select the appropriate host-specific include file. These files are versioned with the sources
so we can easily use them as a starting point for new systems that are similar.

5.4 Modular organization

The propag code is organized in modules, each of which corresponds to a single cweb doc-
ument. There is no strict layering of modules but there are lower-level modules (e.g. for
internal communication, logging) and higher-level modules. Figure 5.1 gives an overview of
Propag’s modules and their relationships.

5.5 Housekeeping code

There are several modules containing “housekeeping” code such as memory allocation (with
accounting), informative output (configurable), error messages, and mpi-based communica-
tion.

5.5.1 The propag module
The propag module contains the main function and a couple of things that don’t belong
anywhere else. The main function is treated in detail in section 5.12.

5.5.2 The types module
The types module is one of the smallest. It contains definitions for most of our numerical
data types. Notably it determines whether state variables will be single or double-precision.

5.5.3 The memory module
This module defines the memory allocation interface that should be used throughout Propag
instead of the standard calloc and malloc functions. Our memory allocation code checks
for allocation failure, initializes memory in the right way for numa systems, and keeps track
of all major allocations to estimate how much memory each process is using.
Figure 5.1  Overview of (most of) Propag’s modules. Lower-level modules are placed more towards the periphery. Green boxes indicate third-party libraries.
5.5.4 The errors module

The errors module defines the Error and Warning functions (actually macros). The Warning function has functionality to count warnings of the same kind and to suppress them when there are more than a given threshold.

5.5.5 The logging module

Propag writes extensive information to a journal file, using a unified line format that includes the current runtime and a line identifier that facilitates extraction of specific information. There is support in this code for a general mechanism by which the user can switch specific loglines on and off, although this is not completely implemented at this time. Instead, several modules have a verbose parameter to govern the level of output.

5.5.6 The comm module

The comm module describes the communication model used for Propag. This is a layer between the mpi functions and the rest of the program. Functions defined here serve for inter-process communication but also for (mesh-unaware) file I/O support on which the mesh-aware I/O functions in reading and saving are based. Functions defined here are also heavily used in the meshcomm, bootstrap, distmesh, and partition modules. The comm module also wraps ParMetis, simply because ParMetis is the only other program part that needs access to the mpi communicator.

5.6 File input/output

Three modules are concerned with input from and output to files. Currently most of our large files are in a format called IGB, which is suitable for 2D to 4D structured meshes. There are modules for input and output of these files. The output module uses extensive information about the (distributed) mesh in order to provide services such as subsampling.

5.6.1 The reading module

The reading module contains generic functions for reading structured data from IGB files. Most of the code uses the functionality provided here to read into vectors defined over either the cubes or the nodes of the mesh.

5.6.2 The forward module

The forward module handles input of source data for forward models (section 1.4).

5.6.3 The saving module

File output is a complex topic in Propag due to the flexibility in choice of variables and spatiotemporal subsampling that it provides. The saving module deals with most of this. The parser for output configurations is defined in separate lex and yacc source files.
5.7 The distributed mesh

Several modules deal with the preparation of the distributed mesh structure and interprocess exchange of information defined on this mesh. In order to allow a fully parallel workflow, we use an initial “bootstrap” mesh, partitioned in a simple way, to initially load the geometry data and call the partitioner in parallel. Subsequently the definitive distributed mesh can be created. Finally there is a module to create differential operators dedicated to this mesh.

5.7.1 The distmesh module

The distmesh module defines the DistMesh struct and the code that creates it. The DistMesh struct contains all information that a process needs about the distributed mesh, including the communication traces. It relies on the repart module to handle the distributed aspects of the mesh.

There is currently only one instance of DistMesh, which is accessed through the global variable mesh.

5.7.2 The bootstrap module

The bootstrap module creates a simple partitioning of the complete voxel mesh, then uses it to load and process the voxels in parallel. Based on the substance definitions and mapping parameter it decides which voxels become cubes and which vertices become nodes.

5.7.3 The partition module

The partition module implements the partmesh function, which performs the mesh partitioning during bootstrapping. It uses parMetis.

5.7.4 The repart module

The repart module contains the code that sends the distributed mesh data from the process that owns it during bootstrapping to the processes that will perform computations on it (owner or not).

5.7.5 The meshcomm module

The meshcomm module provides functionality for the exchange of the data associated to the nodes and cubes between different processes. In particular it provides functionality to exchange data associated to the nodes in the overlap between processes. The module bases on the message passing primitives defined in comm and the communication traces defined as part of the DistMesh structure.

The various NodeVector and CubeVector types are defined here together with the functions that create them. These types are extensively used throughout the other modules. There is functionality for exchange between the owner and the other users of shared nodes, which is the basis of the most common inter-process communication in Propag. The module further contains functions for collective operations such as sum, max, norm, and dot product on NodeVector and CubeVector types.
5.7.6 The anatomy module
The anatomy module deals with the properties of the mesh nodes and cubes. It intervenes both before and after the mesh itself has been created. Initially it handles the substance mapping parameters and so determines which voxels will become cubes and which will be ignored. Then after the mesh has been created and distributed it will compute the conductivity tensor fields and the differential operators.

5.7.7 The octree module
The octree module provides functionality for the creation and exploration of octree data structures. We use these to find the IDs of neighbour nodes or cubes during mesh distribution and when preparing parallel output.

5.7.8 The ordering module
The ordering module implements the reorder_DistMesh function, which puts the nodes (locally to the process) in a specific order with the aim to optimize performance.

5.8 The membrane models

5.8.1 The ion module
The ion module is Propag’s interface to the membrane models. It handles the mapping from node types to models. Based on this it calls the initializer functions of the models that are used, and later, at each time step the integration function of each model. It also keeps track of the depolarization status of all nodes, for phase selection.

The ion_step function is the interface for integration. It contains a loop over all nodes in the part and calls the appropriate stepper function for each node. This loop can partially overlap with the communication of the diffusion current, but not entirely, because the diffusion current must be passed to the membrane models (so they can account for it if they keep track of ion concentrations).

The ion_save_status function collects the membrane model variables that are requested for output.

The ion_query.polarization.stats function determines whether all cells depolarized (so Propag can switch to phase 2).

5.8.2 The membrane modules
The membrane models themselves live in the directory ../membranes. They are meant to be usable in other main programs as well. The functions in the membrane model all deal with one node at a time; they are called from a loop in the ion module.

The file membranes/membrane.web defines a unified interface for all membrane models. The interface includes registration of models, allowing the program to obtain information such as the number of status variables that need to be allocated. There is of course a function to integrate a model over one time step, and a function that can extract internal variables from the membrane model for unusual data output purposes. This interface allows almost any model variable to be output when the user requests it.

ion.web computation of ionic currents (calls the membrane models)
Each membrane model has a stepper function (called by `ion_step`) which integrates the status variables and computes the ionic current. This function received the following inputs:

- **Idif** diffusion current, can be used to keep track of ion concentrations
- **Istim** stimulation current, can be used to keep track of ion concentrations
- **Idt** time step for integration
- **sintime** the current time in the simulation, needed to set the `dtime` output and to implement “freezing.”

The models return the total ionic current `Iia` and set the activation time `dtime` when they decide that the node depolarizes. For this decision a membrane model can use its status variables, which generally is more reliable than a decision based on the information that is available outside the membrane model.

Some models support “freezing,” which means that they don’t integrate their status variables but return an ionic current normally. This feature can be used to initiate arrhythmia by creating a line of temporarily frozen tissue and stimulating just next to it.

### 5.9 Application modules

These modules govern major steps in the computations. The level of delegation from the main module has been increasing over time. The management of monodomain reaction-diffusion models is defined in the main function; for bidomain reaction-diffusion models it resides partly in the `bidofex` module. The more recent `leadfields` module has a function that runs the computation of a set of lead fields all by itself.

- **diffusion.web** diffusion current
- **euler.web** integrate reaction-diffusion equation
- **bidofex.web** bidomain code (integrator and static solvers)
- **leadfields.web** create and use lead fields

The distribution of work between the main function, the `diffusion` module, the `euler` module, and the `bidofex` module is currently a bit strange. It grew historically and we’re thinking about a better way to organize this.

### 5.10 Numerical modules

A couple of modules implement linear-systems algebra and interfacing to external libraries.

- **petsc.web** PETSc interface
- **preco.web** preconditioner interface
- **solver.web** solver interface
- **spblas.web** Sparse Parallel BLAS functionality
- **bicgstab.web** dedicated linear system solver
5.11 Handling of parameters

Scanner code for the command-line and configuration file parameters is generated by the PRM utility. The definitions of the parameters that PRM uses are written in files with extension .prm. There is a main parameter file propag.prm in which parameters are defined that govern global behaviour, as well as module-specific parameters. In most cases the parameters for a module are packed in a parameter structure, which is defined in a separate file, e.g. somemodule.prs, that is being imported with a preprocessor include directive.

A little complexity follows from our desire to document these parameter structs in cweb format. This requires the parameter definitions to be acceptable C code in the eyes of cweb. We made a small adaptation of the PRM language to achieve this. Files in this adapted language are output by cweb as e.g. somemodule.prr, which is then transformed into somemodule.prs by the C preprocessor. The prs file is in PRM format and can be processed normally.

5.12 The main function

After some common initialization, the main function decides in which mode Propag is running, allocates the major data structures that will be needed, and calls the appropriate initializers. Finally it runs a reaction-diffusion model or forward model, or calls one of the handlers in bidofex or leadfields.

The initialization phase is a puzzle of priorities. Among other things we try to do all verifications that are possible without actually loading the anatomy first, before bailing out if we are in “verify mode.” Currently the order is as follows.

1. initialization of variables used for timing
2. disable stdout buffering
3. initialize the comm module so we can talk to ourselves
4. get the program parameters
5. open the logfile (the name is determined by a parameter) and redirect stdout to it
6. show the parameter settings in the logfile
7. initialize the memory module (memory tracking and reporting)
8. call comm info for information on the cartesian decomposition
9. additional checks and interpretation of the program parameters
10. interpret the cell type mappings (so we can interpret the anatomy data)
11. get the model dimensions (not the data themselves which may be huge)
12. check correctness of the saving configuration (syntax, limits)
13. if running a lead-field computation, read the configuration and check it
14. stop if running in verify mode
15. create the bootstrap mesh
16. partition the mesh
17. create the distributed mesh structure and distribute it
18. free bootstrap data structures
19. show how the mesh is distributed
20. initialize the saving module (checks the saving configuration again, now verifying that there are nodes at all vertices where output is requested)
21. load the fiber orientations
22. compute and show vertex statistics (for debugging simulations)
23. initialize variables and allocate memory
24. set up the model anatomy: compute diffusion operators, volume fractions, and stimulation times for each node
25. optionally save the node types that we created from the cube types
26. set up ECG computation with lead fields, if configured
27. switch to phase 1 (integration parameters for the depolarization phase)
28. initialize everything that depends on $\Delta t$, such as membrane models
29. initialize the bidofex module if we’re doing a bidomain simulation
30. open forward-model input if we’re running a forward model

After that the work starts: Propag runs either a lead-field computation, a forward model, or a propagation model.

Finally it writes a last memory report to the logfile, closes files, and exits.

### 5.13 Experimental/dysfunctional modules

Finally there are some experimental modules and things that are currently not used. The checkpointing code, inherited from propag4, is currently not used or maintained because on current machines our checkpoints are too slow compared to the computational performance. A simple domain partitioner and a domain-decomposition preconditioner (OpenMP only) are also inherited and not used.

- **domains.web** experimental domain partitioner (currently not used)
- **ddpreco.web** domain-decomposition ILU preconditioner for bidofex (rarely used)
- **status.web** status dumps (checkpoints) and reloading – currently dysfunctional
Chapter 6

Developments

A possible successor to Propag is under development in the EXACARD project. It will use a radically different parallelization method based on tasks. This is expected to be more efficient on large and heterogeneous computing systems. At least initially, this new code will have reduced functionality, e.g. fewer membrane models and no bidomain code. It may also break backward compatibility in other areas.

6.1 Work to do

Many things can be improved in the Propag code itself, and may be forwarded to its successors. Here is an incomplete list.

- have all Warnings output by the master process, because the current method screws up monitoring with tail(1). They could be assembled periodically, e.g. at the end of a lap.
- output in “lean” format directly from prmerge (not through igb2iga), for floating-point data such as status variables. The IGA format cannot handle this, and is not expected to be suitable for it either because the compression depends only works with discrete data values.
- Use a dedicated data format instead of NetCDF for communication between Propag and Prmerge. This has several advantages: 1) It means one less library to depend on, which is really helpful at HPC centers. 2) We can be sure to write and read data sequentially where possible. 3) We will know the file pointer position, so we can generate better advise to the file system (e.g. through the Lustre API). 4) We can choose to do asynchronous I/O, or control the buffer size. 5) We can take control over the data communication between the processes in a write group and see if we can be more efficient. For example shared memory (within a node) turned out to be faster than MPI Gather operations in Prmerge.
- Handle (partially) unstructured meshes.
- Further parallelize the gepetto and voxelize codes, to allow much larger meshes
- Further parallelize the post-processing tools, and use less Matlab
- Further unify the makefiles in CEMPACK (propag5/ and igb/ still have their own settings)
Bibliography


