Arbor

Aug 21, 2019

Getting Stared:

CHAPTER 1

What is Arbor?

Arbor is a high-performance library for computational neuroscience simulations.

The development team is from from high-performance computing (HPC) centers:

- Swiss National Supercomputing Center (CSCS), Jülich and BSC in work package 7.5.4 of the HBP.
- Aim to prepare neuroscience users for new HPC architectures;

Arbor is designed from the ground up for **many core** architectures:

- Written in C++11 and CUDA;
- Distributed parallelism using MPI;
- Multithreading with TBB and C++11 threads;
- **Open source** and **open development**;
- Sound development practices: **unit testing**, **continuous Integration**, and **validation**.

CHAPTER 2

Features

We are actively developing Arbor, improving performance and adding features. Some key features include:

- Optimized back end for CUDA
- Optimized vector back ends for Intel (KNL, AVX, AVX2) and Arm (ARMv8-A NEON) intrinsics.
- Asynchronous spike exchange that overlaps compute and communication.
- Efficient sampling of voltage and current on all back ends.
- Efficient implementation of all features on GPU.
- Reporting of memory and energy consumption (when available on platform).
- An API for addition of new cell types, e.g. LIF and Poisson spike generators.
- Validation tests against numeric/analytic models and NEURON.

CHAPTER 3

Citing Arbor

Specific versions of Arbor can be cited via Zenodo:

- v0.2:
- v0.1:

The following BibTeX can be used to cite Arbor:

```
@INPROCEEDINGS{
    paper:arbor2019,
    author={N. A. {Akar} and B. {Cumming} and V. {Karakasis} and A. {Küsters} and W.
↪{Klijn} and A. {Peyser} and S. {Yates}},
    booktitle={2019 27th Euromicro International Conference on Parallel, Distributed␣
↪and Network-Based Processing (PDP)},
    title={{Arbor --- A Morphologically-Detailed Neural Network Simulation Library␣
↪for Contemporary High-Performance Computing Architectures}},
    year={2019}, month={feb}, volume={}, number={},
    pages={274--282},
    doi={10.1109/EMPDP.2019.8671560},
    ISSN={2377-5750}}
```

Alternative citation formats for the paper can be downloaded here, and a preprint is available at arXiv.

3.1 Installing Arbor

Arbor is installed by obtaining the source code and compiling it on the target system.

This guide starts with an overview of the building process, and the various options available to customize the build. The guide then covers installation and running on *HPC clusters*, followed by a *troubleshooting guide* for common build problems.

3.1.1 Requirements

Minimum Requirements

The non distributed (i.e. no MPI) version of Arbor can be compiled on Linux or OS X systems with very few tools.

Table 1: Required Tools

Tool	Notes
Git	To check out the code, minimum version 2.0.
CMake	To set up the build, minimum version 3.9
compiler	A C++14 compiler. See *compilers*.

Compilers

Arbor requires a C++ compiler that fully supports C++14. We recommend using GCC or Clang, for which Arbor has been tested and optimised.

Table 2: Supported Compilers

Compiler	Min version	Notes
GCC	6.1.0	
Clang	4.0	Needs GCC 6 or later for standard library.
Apple Clang	9	Apple LLVM version 9.0.0 (clang-900.0.39.2)

Note: The `CC` and `CXX` environment variables specify which compiler executable CMake should use. If these are not set, CMake will attempt to automatically choose a compiler, which may be too old to compile Arbor. For example, the default compiler chosen below by CMake was GCC 4.8.5 at `/usr/bin/c++`, so the `CC` and `CXX` variables were used to specify GCC 6.1.0 before calling `cmake`.

```
# on this system CMake chooses the following compiler by default
$ c++ --version
c++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-16)

# check which version of GCC is available
$ g++ --version
g++ (GCC) 6.1.0
Copyright (C) 2015 Free Software Foundation, Inc.

# set environment variables for compilers
$ export CC=`which gcc`; export CXX=`which g++`;

# launch CMake
# the compiler version and path is given in the CMake output
$ cmake ..
-- The C compiler identification is GNU 6.1.0
-- The CXX compiler identification is GNU 6.1.0
-- Check for working C compiler: /cm/local/apps/gcc/6.1.0/bin/gcc
-- Check for working C compiler: /cm/local/apps/gcc/6.1.0/bin/gcc -- works
...
```

Note: Is is commonly assumed that to get the best performance one should use a vendor-specific compiler (e.g. the Intel, Cray or IBM compilers). These compilers are often better at auto-vectorizing loops, however for everything else GCC and Clang nearly always generate more efficient code.

The main computational loops in Arbor are generated from NMODL. The generated code is explicitly vectorised, obviating the need for vendor compilers, and we can take advantage of their benefits of GCC and Clang: faster compilation times; fewer compiler bugs; and better support for C++ standards.

Note: The IBM XL C++ compiler and Intel C++ compiler are not supported, owing to unresolved compiler issues. We strongly recommend building with GCC or Clang instead on PowerPC and Intel platforms.

Optional Requirements

GPU Support

Arbor has full support for NVIDIA GPUs, for which the NVIDIA CUDA toolkit version 9 is required.

Distributed

Arbor uses MPI to run on HPC cluster systems. Arbor has been tested on MVAPICH2, OpenMPI, Cray MPI, and IBM MPI. More information on building with MPI is in the *HPC cluster section*.

Python

Arbor has a Python frontend, for which Python 3.6 is required. In order to use MPI in combination with the python frontend the mpi4py Python package is recommended.

Documentation

To build a local copy of the html documentation that you are reading now, you will need to install Sphinx.

3.1.2 Getting the Code

The easiest way to acquire the latest version of Arbor is to check the code out from the Github repository:

```
git clone https://github.com/arbor-sim/arbor.git --recurse-submodules
```

We recommend using a recursive checkout, because Arbor uses Git submodules for some of its library dependencies. The CMake configuration attempts to detect if a required submodule is available, and will print a helpful warning or error message if not, but it is up to the user to ensure that all required submodules are downloaded.

The Git submodules can be updated, or initialized in a project that didn't use a recursive checkout:

```
git submodule update --init --recursive
```

You can also point your browser to Arbor's Github page and download a zip file. If you use the zip file, then don't forget to run Git submodule update manually.

3.1.3 Building and Installing Arbor

Once the Arbor code has been checked out, first run CMake to configure the build, then run make.

Below is a simple workflow for: **1)** getting the source; **2)** configuring the build; **3)** building; **4)** running tests; **5)** install.

For more detailed build configuration options, see the *quick start* guide.

```
# 1) Clone.
git clone https://github.com/arbor-sim/arbor.git --recurse-submodules
cd arbor

# Make a path for building
mkdir build
cd build

# 2) Use CMake to configure the build.
# By default Arbor builds in release mode, i.e. with optimizations on.
# Release mode should be used for installing and benchmarking Arbor.
cmake ..

# 3.1) Build Arbor library.
make -j 4
# 3.2) Build Arbor unit tests.
make -j 4 tests

# 4) Run tests.
./bin/unit

# 5) Install (by default, to /usr/local).
make install
```

This will build Arbor in release mode with the *default C++ compiler*.

Quick Start: Examples

Below are some example of CMake configurations for Arbor. For more detail on individual CMake parameters and flags, follow links to the more detailed descriptions below.

Debug mode with assertions enabled.

If you encounter problems building or running Arbor, compile with these options for testing and debugging.

```
cmake -DARB_WITH_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=debug
```

Release mode (compiler optimizations enabled) with the default compiler, optimized for the local system architecture.

```
cmake -DARB_ARCH=native
```

Release mode with Clang.

```
export CC=`which clang`
export CXX=`which clang++`
cmake
```

Release mode for the Haswell architecture and explicit vectorization of kernels.

```
cmake -DARB_VECTORIZE=ON -DARB_ARCH=haswell
```

Release mode with explicit vectorization, targeting the Broadwell architecture, with support for P100 GPUs, and building with GCC 6.

```
export CC=gcc-6
export CXX=g++-6
cmake -DARB_VECTORIZE=ON -DARB_ARCH=broadwell -DARB_WITH_GPU=ON
```

Release mode with explicit vectorization, optimized for the local system architecture and install in `/opt/arbor`

```
cmake -DARB_VECTORIZE=ON -DARB_ARCH=native -DCMAKE_INSTALL_PREFIX=/opt/arbor
```

Build Target

By default, Arbor is built in release mode, which should be used when installing or benchmarking Arbor. To compile in debug mode (which in practical terms means with `-g -O0` flags), use the `CMAKE_BUILD_TYPE` CMake parameter.

```
cmake -DCMAKE_BUILD_TYPE={debug,release}
```

Architecture

By default, Arbor is built to target whichever architecture is the compiler default, which often involves a sacrifice of performance for binary portability. The target architecture can be explicitly set with the `ARB_ARCH` configuration option. This will be used to direct the compiler to use the corresponding instruction sets and to optimize for that architecture.

When building and installing on the same machine, a good choice for many environments is to set `ARB_ARCH` to `native`:

```
cmake -DARB_ARCH=native
```

When deploying on a different machine (cross-compiling) specify the specific architecture of the target machine. The valid values correspond to those given to the `-mcpu` or `-march` options for GCC and Clang; the build system will translate these names to corresponding values for other supported compilers.

Specific recent x86-family Intel CPU architectures include `broadwell`, `skylake` and `knl`. Complete lists of architecture names can be found in the compiler documentation: for example GCC x86 options, PowerPC options, and ARM options.

```
# Intel architectures
cmake -DARB_ARCH=broadwell         # broadwell with avx2
cmake -DARB_ARCH=skylake-avx512    # skylake with avx512 (Xeon server)
cmake -DARB_ARCH=knl               # Xeon Phi KNL

# ARM Arm8a
cmake -DARB_ARCH=armv8-a

# IBM Power8
cmake -DARB_ARCH=power8
```

Vectorization

Explicit vectorization of computational kernels can be enabled in Arbor by setting the `ARB_VECTORIZE` CMake flag. This option is typically used in conjunction with the `ARB_ARCH` option to specify the target architecture: without SIMD support in Arbor for the architecture, enabling `ARB_VECTORIZE` will lead to a compilation error.

```
cmake -DARB_VECTORIZE=ON -DARB_ARCH=native
```

With this flag set, the library will use architecture-specific vectorization intrinsics to implement these kernels. Arbor currently has vectorization support for x86 architectures with AVX, AVX2 or AVX512 ISA extensions, and for ARM architectures with support for AArch64 NEON intrinsics (first available on ARMv8-A).

GPU Backend

Arbor supports NVIDIA GPUs using CUDA. The CUDA back end is enabled by setting the CMake `ARB_WITH_GPU` option.

```
cmake -DARB_WITH_GPU=ON
```

By default `ARB_WITH_GPU=OFF`. When the option is turned on, Arbor is built for all supported GPUs and the available GPU will be used at runtime.

Depending on the configuration of the system where Arbor is being built, the C++ compiler may not be able to find the `cuda.h` header. The easiest workaround is to add the path to the include directory containing the header to the `CPATH` environment variable before configuring and building Arbor, for example:

```
export CPATH="/opt/cuda/include:$CPATH"
cmake -DARB_WITH_GPU=ON
```

Note: Arbor supports and has been tested on the Kepler (K20 & K80), Pascal (P100) and Volta (V100) GPUs

Python Frontend

Arbor can be used with a python frontend which is enabled by toggling the CMake `ARB_WITH_PYTHON` option:

```
cmake -ARB_WITH_PYTHON=ON
```

By default `ARB_WITH_PYTHON=OFF`. When this option is turned on, a python module called *`arbor`* is built.

The Arbor Python wrapper has optional support for the `mpi4py` Python module for MPI. CMake will attempt to automatically detect `mpi4py` if configured with both `-ARB_WITH_PYTHON=ON` and MPI `-DARB_WITH_MPI=ON`.

If CMake fails to find `mpi4py` when it should, the easiest workaround is to add the path to the include directory for `mpi4py` to the `CPATH` environment variable before configuring and building Arbor:

```
# search for path tp python's site-package mpi4py
for p in `python3 -c 'import sys; print("\n".join(sys.path))'`; do echo ===== $p; ls
↪$p | grep mpi4py; done

===== /path/to/python3/site-packages
mpi4py

# set CPATH and run cmake
export CPATH="/path/to/python3/site-packages/mpi4py/include/:$CPATH"

cmake -ARB_WITH_PYTHON=ON -DARB_WITH_MPI=ON
```

Installation

Arbor can be installed with `make install` after configuration. The installation comprises:

- The static libraries `libarbor.a` and `libarborenv.a`.
- Public header files.
- The `lmorpho` l-system morphology generation utility
- The `modcc` NMODL compiler if built.
- The python module if built.
- The HTML documentation if built.

The default install path (`/usr/local`) can be overridden with the `CMAKE_INSTALL_PREFIX` configuration option.

Provided that Sphinx is available, HTML documentation for Arbor can be built with `make html`. Note that documentation is not built by default — if built, it too will be included in the installation.

Note that the `modcc` compiler will not be built by default if the `ARB_MODCC` configuration setting is used to specify a different executable for `modcc`. While `modcc` can be used to translate user-supplied NMODL mechanism descriptions into C++ and CUDA code for use with Arbor, this generated code currently relies upon private headers that are not installed.

3.1.4 HPC Clusters

HPC clusters offer their own unique challenges when compiling and running software, so we cover some common issues in this section. If you have problems on your target system that are not covered here, please make an issue on the Arbor Github issues page. We will do our best to help you directly, and update this guide to help other users.

MPI

Arbor uses MPI for distributed systems. By default it is built without MPI support, which can enabled by setting the `ARB_WITH_MPI` configuration flag. An example of building a 'release' (optimized) version of Arbor with MPI is:

```
# set the compiler wrappers
export CC=`which mpicc`
export CXX=`which mpicxx`
```

(continues on next page)

(continued from previous page)

```
# configure with mpi
cmake -DARB_WITH_MPI=ON

# run MPI-specific unit tests on 2 MPI ranks
mpirun -n 2 ./bin/unit-mpi
```

The example above sets the `CC` and `CXX` environment variables to use compiler wrappers provided by the MPI implementation. While the configuration process will attempt to find MPI libraries and build options automatically, we recommend using the supplied MPI compiler wrappers in preference.

Note: MPI distributions provide **compiler wrappers** for compiling MPI applications.

In the example above the compiler wrappers for C and C++ called `mpicc` and `mpicxx` respectively. The name of the compiler wrapper is dependent on the MPI distribution.

The wrapper forwards the compilation to a compiler, like GCC, and you have to ensure that this compiler is able to compile Arbor. For wrappers that call GCC or Clang compilers, pass the `--version` flag to the wrapper. For example, on a Cray system, where the C++ wrapper is called `CC`:

```
$ CC --version
g++ (GCC) 6.2.0 20160822 (Cray Inc.)
```

Cray Systems

The compiler used by the MPI wrappers is set using a "programming environment" module. The first thing to do is change this module, which by default is set to the Cray programming environment, to a compiler that can compile Arbor. For example, to use the GCC compilers, select the GNU programming environment:

```
module swap PrgEnv-cray PrgEnv-gnu
```

The version of GCC can then be set by choosing an appropriate gcc module. In the example below we use `module avail` to see which versions of GCC are available, then choose GCC 7.1.0

```
$ module avail gcc      # see all available gcc versions

------------------------- /opt/modulefiles ---------------------------
gcc/4.9.3    gcc/6.1.0    gcc/7.1.0    gcc/5.3.0(default)    gcc/6.2.0

$ module swap gcc/7.1.0 # swap gcc 5.3.0 for 7.1.0

$ CC --version          # test that the wrapper uses gcc 7.1.0
g++ (GCC) 7.1.0 20170502 (Cray Inc.)

# set compiler wrappers
$ export CC=`which cc`
$ export CXX=`which CC`
```

Note that the C and C++ compiler wrappers are called `cc` and `CC` respectively on Cray systems.

CMake detects that it is being run in the Cray programming environment, which makes our lives a little bit more difficult (CMake sometimes tries a bit too hard to help). To get CMake to correctly link our code, we need to set the `CRAYPE_LINK_TYPE` environment variable to `dynamic`.

```
export CRAYPE_LINK_TYPE=dynamic
```

Putting it all together, a typical workflow to build Arbor on a Cray system is:

```
export CRAYPE_LINK_TYPE=dynamic
module swap PrgEnv-cray PrgEnv-gnu
moudle swap gcc/7.1.0
export CC=`which cc`; export CXX=`which CC`;
cmake -DARB_WITH_MPI=ON    # MPI support
```

Note: If `CRAYPE_LINK_TYPE` isn't set, there will be warnings like the following when linking:

```
warning: Using 'dlopen' in statically linked applications requires at runtime
         the shared libraries from the glibc version used for linking
```

Often the library or executable will work, however if a different glibc is loaded, Arbor will crash at runtime with obscure errors that are very difficult to debug.

3.1.5 Troubleshooting

Cross Compiling NMODL

Care must be taken when Arbor is compiled on a system with a different architecture to the target system where Arbor will run. This occurs quite frequently on HPC systems, for example when building on a login/service node that has a different architecture to the compute nodes.

Note: If building Arbor on a laptop or desktop system, i.e. on the same computer that you will run Arbor on, cross compilation is not an issue.

Note: The `ARB_ARCH` setting is not applied to the building of `modcc`. On systems where the build node and compute node have different architectures within the same family, this may mean that separate compilation of `modcc` is not necessary.

Warning: `Illegal instruction` errors are a sure sign that Arbor is running on a system that does not support the architecture it was compiled for.

When cross compiling, we have to take care that the *modcc* compiler, which is used to convert NMODL to C++/CUDA code, is able to run on the compilation node.

By default, building Arbor will build the `modcc` executable from source, and then use that to build the built-in mechanisms specified in NMODL. This behaviour can be overridden with the `ARB_MODCC` configuration option, for example:

```
cmake -DARB_MODCC=path-to-local-modcc
```

Here we will use the example of compiling for Intel KNL on a Cray system, which has Intel Sandy Bridge CPUs on login nodes that don't support the AVX512 instructions used by KNL.

```
#
#   Step 1: Build modcc.
#

module swap PrgEnv-cray PrgEnv-gnu
# Important: use GNU compilers directly, not the compiler wrappers,
# which generate code for KNL, not the login nodes.
export CC=`which gcc`; export CXX=`which g++`;
export CRAYPE_LINK_TYPE=dynamic

# make a path for the modcc build
mkdir build_modcc
cd build_modcc

# configure and make modcc
cmake ..
make -j modcc

#
#   Step 2: Build Arbor.
#

cd ..
mkdir build; cd build;
# use the compiler wrappers to build Arbor
export CC=`which cc`; export CXX=`which CC`;
cmake .. -DCMAKE_BUILD_TYPE=release           \
         -DARB_WITH_MPI=ON                    \
         -DARB_ARCH=knl                       \
         -DARB_VECTORIZE=ON                   \
         -DARB_MODCC=../build_modcc/bin/modcc
```

Note: Cross compilation issues can occur when there are minor differences between login and compute nodes, e.g. when the login node has Intel Haswell, and the compute nodes have Intel Broadwell.

Other systems, such as IBM BGQ, have very different architectures for login and compute nodes.

If the *modcc* compiler was not compiled for the login node, illegal instruction errors will occur when building, e.g.

```
$ make
...
[ 40%] modcc generating: /users/bcumming/arbor_knl/mechanisms/multicore/pas_cpu.hpp
/bin/sh: line 1: 12735 Illegal instruction     (core dumped) /users/bcumming/arbor_
↪knl/build_modcc/modcc/modcc -t cpu -s\ avx512 -o /users/bcumming/arbor_knl/
↪mechanisms/multicore/pas /users/bcumming/arbor_knl/mechanisms/mod/pas.mod
mechanisms/CMakeFiles/build_all_mods.dir/build.make:69: recipe for target '../
↪mechanisms/multicore/pas_cpu.hpp' failed
```

If you have errors when running the tests or a miniapp, then either the wrong `ARB_ARCH` target architecture was selected; or you might have forgot to launch on the compute node. e.g.:

```
$ ./bin/unit
Illegal instruction (core dumped)
```

On the Cray KNL system, `srun` is used to launch (it might be `mpirun` or similar on your system):

```
$ srun -n1 -c1 ./bin/unit
[==========] Running 609 tests from 108 test cases.
[----------] Global test environment set-up.
[----------] 15 tests from algorithms
[ RUN      ] algorithms.parallel_sort
[       OK ] algorithms.parallel_sort (15 ms)
[ RUN      ] algorithms.sum
[       OK ] algorithms.sum (0 ms)
...
```

Debugging

Sometimes things go wrong: tests fail, simulations give strange results, segmentation faults occur and exceptions are thrown.

A good first step when things to wrong is to turn on additional assertions that can catch errors. These are turned off by default (because they slow things down a lot), and have to be turned on by setting the `ARB_WITH_ASSERTIONS` CMake option:

```
cmake -DARB_WITH_ASSERTIONS=ON
```

Note: These assertions are in the form of `arb_assert` macros inside the code, for example:

```
void decrement_min_remaining() {
    arb_assert(min_remaining_steps_>0);
    if (!--min_remaining_steps_) {
        compute_min_remaining();
    }
}
```

A failing `arb_assert` indicates that an error inside the Arbor library, caused either by a logic error in Arbor, or incorrectly checked user input.

If this occurs, it is highly recommended that you attach the output to the bug report you send to the Arbor developers!

CMake Git Submodule Warnings

When running CMake, warnings like the following indicate that the Git submodules need to be *updated*.

```
The Git submodule for rtdtheme is not available.
To check out all submodules use the following commands:
    git submodule init
    git submodule update
Or download submodules recursively when checking out:
    git clone --recurse-submodules https://github.com/arbor-sim/arbor.git
```

3.2 Overview

Arbor's design aims to enable scalability through abstraction.

To achieve this, Arbor makes a distinction between the **description** of a model, and the **execution** of a model: a *recipe* describes a model, and a *simulation* is an executable instantiation of a model.

To be able to simulate a model, three basic steps need to be considered:

1. Describe the model by defining a recipe;
2. Define the computational resources available to execute the model;
3. Initiate and execute a simulation of the recipe on the chosen hardware resources.

Recipes represent a set of neuron constructions and connections with *mechanisms* specifying ion channel and synapse dynamics in a cell-oriented manner. This has the advantage that cell data can be initiated in parallel.

A cell represents the smallest unit of computation and forms the smallest unit of work distributed across processes. Arbor has built-in support for different *cell types*, which can be extended by adding new cell types to the C++ cell group interface.

Simulations manage the instantiation of the model and the scheduling of spike exchange as well as the integration for each cell group. A cell group represents a collection of cells of the same type computed together on the GPU or CPU. The partitioning into cell groups is provided by *Domain Decomposition* which describes the distribution of the model over the locally available computational resources.

In order to visualise the result of detected spikes a spike recorder can be used and to analyse Arbor's performance a meter manager is available.

3.3 Concepts

This section describes some of the core concepts of Arbor.

3.3.1 Cells

The basic unit of abstraction in an Arbor model is a cell. A cell represents the smallest model that can be simulated. Cells interact with each other via spike exchange and gap junctions. Cells can be of various types, admitting different representations and implementations. Arbor currently supports specialized leaky integrate and fire cells and cells representing artificial spike sources in addition to multi-compartment neurons.

Table 3: Identifiers used to uniquely refer to cells and objects like synapses on cells.

Identifier	Type	Description
gid	integral	The unique global identifier of a cell.
index	integral	The index of an item in a cell-local collection. For example the 7th synapse on a cell.
cell_member	tuple (*gid*, *index*)	The global identification of a cell-local item with *index* into a cell-local collection on the cell identified by *gid*. For example, the 7th synapse on cell 42.

Each cell has a global identifier *gid*, which is used to refer to cells in *recipes*. To describe or refer to cell-to-cell interactions, the following object types need to be enumerated:

1. **Sources**

2. **Targets**
3. **Gap Junction Sites**

Cells interact with other cells via *connections* or *gap junctions*. Connections are formed from **sources** to **targets**. Gap junctions are formed between two **gap junction sites**.

A cell can have multiple sources, targets and gap junction site objects. Each object has a local `index` relative to other objects of the same type on that cell. A unique (`gid`, `index`) pair defned by a `cell_member` can be used to uniquely identify objects on a cell in a global model.

Cell Kinds

Table 4: The types of cell supported by Arbor

Cell Kind	Description
cable	Cell with morphology described by branching 1D cable segments.
lif	Leaky-integrate and fire neuron.
spiking	Proxy cell that generates spikes from a user-supplied time sequence.
benchmark	Proxy cell used for benchmarking (developer use only).

1. **Cable Cells**

   Cable cells are morphologically-detailed cells represented as branching linear 1D segments. They can be coupled to other cell types via the following mechanisms:

   1. Spike exchange over a **connection** with fixed latency. Cable cells can *receive* spikes from any kind of cell, and can be a *source* of spikes cells that have target sites (i.e. *cable* and *lif* cells).
   2. Direct electrical coupling between two cable cells via **gap junctions**.

   Key concepts:

   - **Morphology**: The morphology of a cable cell is composed of a branching tree of one-dimensional line segments. Strictly speaking, Arbor represents a morphology is an *acyclic directed graph*, with the soma at the root.
   - **Detectors**: Spike detectors generate spikes when the voltage at location on the cell passes a threshold. Dectectors act as **sources** of *connections*.
   - **Synapses**: Synapases act as **targets** of *connections*. A synapse is described by a synapse type (with associated parameters) and location on a cell.
   - **Gap Junction Sites**: These refer to the sites of *gap junctions*. They are declared by specifying a location on a branch of the cell.

2. **LIF Cells**

   A single compartment leaky integrate and fire neuron with one **source** and one **target**. LIF cells does not support adding additional **sources** or **targets** or gap junctions.

3. **Spiking Cells**

   Spike source from values inserted via a *schedule description*. It is a point neuron with one built-in **source** and no **targets**. It does not support adding additional **sources** or **targets**. It does not support gap junctions.

4. **Benchmark Cells**

   Proxy cell used for benchmarking, and used by developers to benchmark the spike exchange and event delivery infrastructure.

3.3.2 Connections

Connections implement chemical synapses between **source** and **target** cells and are characterized by having a transmission delay.

Connections in Arbor are defined in two steps:

1. Create **Source** and **Target** on two cells: a source defined on one cell, and a target defined on another.
2. Declare the connection in the *recipe*: with a source and target idenfied using `cell_member`, a connection delay and a connection weight.

3.3.3 Gap Junctions

Gap junctions represent electrical synapses where transmission between cells is bidirectional and direct. They are modeled as a conductance between two **gap junction sites** on two cells.

Similarly to *Connections*, Gap Junctions in Arbor are defined in two steps:

1. A **gap junction site** is created on each of the two cells. These locations need to be declared on the *cell*.
2. Gap Junction instantiation in the *recipe*: The **gap junction sites** are indexed using `cell_member` because a single cell may have more than one gap junction site. A gap junction is instantiated by providing two **gap junction sites'** and a conductance in μS.

Note: Only cable cells support gap junctions as of now.

3.4 Hardware

Local resources are locally available computational resources, specifically the number of hardware threads and the number of GPUs.

An *allocation* enumerates the computational resources to be used for a simulation, typically a subset of the resources available on a physical hardware node.

Note: New users can find using contexts a little verbose. The design is very deliberate, to allow fine-grained control over which computational resources an Arbor simulation should use. As a result Arbor is much easier to integrate into workflows that run multiple applications or libraries on the same node, because Arbor has a direct API for using on node resources (threads and GPU) and distributed resources (MPI) that have been partitioned between applications/libraries.

3.4.1 Execution Context

An *execution context* contains the local thread pool, and optionally the GPU state and MPI communicator, if available. Users of the library configure contexts, which are passed to Arbor methods and types.

See *Hardware Management* for documentation of the Python interface and *Hardware Management* for the C++ interface for managing hardware resources.

3.5 Recipes

An Arbor *recipe* is a description of a model. The recipe is queried during the model building phase to provide information about cells in the model, such as:

- the number of cells in the model;
- the type of a cell;
- a description of a cell, e.g. with soma, synapses, detectors, stimuli;
- the number of spike targets;
- the number of spike sources;
- the number of gap junction sites;
- incoming network connections from other cells terminating on a cell;
- gap junction connections on a cell.

3.5.1 Why Recipes?

The interface and design of Arbor recipes was motivated by the following aims:

- Building a simulation from a recipe description must be possible in a distributed system efficiently with minimal communication.
- To minimise the amount of memory used in model building, to make it possible to build and run simulations in one run.

Recipe descriptions are cell-oriented, in order that the building phase can be efficiently distributed and that the model can be built independently of any runtime execution environment.

During model building, the recipe is queried first by a load balancer, then later when building the low-level cell groups and communication network. The cell-centered recipe interface, whereby cell and network properties are specified "per-cell", facilitates this.

The steps of building a simulation from a recipe are:

1. Load balancing

First, the cells are partitioned over MPI ranks, and each rank parses the cells assigned to it to build a cost model. The ranks then coordinate to redistribute cells over MPI ranks so that each rank has a balanced workload. Finally, each rank groups its local cells into `cell_group` s that balance the work over threads (and GPU accelerators if available).

2. Model building

The model building phase takes the cells assigned to the local rank, and builds the local cell groups and the part of the communication network by querying the recipe for more information about the cells assigned to it.

3.5.2 General Best Practices

Think of the cells

When formulating a model, think cell-first, and try to formulate the model and the associated workflow from a cell-centered perspective. If this isn't possible, please contact the developers, because we would like to develop tools that help make this simpler.

Be lazy

A recipe does not have to contain a complete description of the model in memory. Precompute as little as possible, and use lazy evaluation to generate information only when requested. This has multiple benefits, including:

- thread safety;
- minimising the memory footprint of the recipe.

Be reproducible

Arbor is designed to give reproducible results when the same model is run on a different number of MPI ranks or threads, or on different hardware (e.g. GPUs). This only holds when a recipe provides a reproducible model description, which can be a challenge when a description uses random numbers, e.g. to pick incoming connections to a cell from a random subset of a cell population. To get a reproducible model, use the cell *gid* (or a hash based on the *gid*) to seed random number generators, including those for `event_generator` s.

3.5.3 Mechanisms

The description of multi-compartment cells also includes the specification of ion channel and synapse dynamics. In the recipe, these specifications are called *mechanisms*. Implementations of mechanisms are either hand-coded or a translator (modcc) is used to compile a subset of NEURON's mechanism specification language NMODL.

Examples Common examples are the *passive/ leaky integrate-and-fire* model, the *Hodgkin-Huxley* mechanism, the *(double-) exponential synapse* model, or the *Natrium current* model for an axon.

Detailed documentation for Python recipes can be found in *Recipes*. C++ *Recipes* are documented and best practices are shown as well.

3.6 Domain Decomposition

A *domain decomposition* describes the distribution of the model over the available computational resources. The description partitions the cells in the model as follows:

- group the cells into cell groups of the same kind of cell;
- assign each cell group to either a CPU core or GPU on a specific MPI rank.

The number of cells in each cell group depends on different factors, including the type of the cell, and whether the cell group will run on a CPU core or the GPU. The domain decomposition is solely responsible for describing the distribution of cells across cell groups and domains.

3.6.1 Load Balancers

A *load balancer* generates the domain decomposition using the model recipe and a description of the available computational resources on which the model will run described by an execution context. Currently Arbor provides one load balancer and more will be added over time.

Arbor's Python interface of domain decomposition and load balancers is documented in *Domain Decomposition* and the C++ interface in *Domain Decomposition*.

3.7 Simulations

A simulation is the executable form of a model and is used to interact with and monitor the model state. In the simulation the neuron model is initiated and the spike exchange and the integration for each cell group are scheduled.

3.7.1 From recipe to simulation

To build a simulation the following are needed:

- A recipe that describes the cells and connections in the model.
- A context used to execute the simulation.

The workflow to build a simulation is to first generate a domain decomposition that describes the distribution of the model over the local and distributed hardware resources (see *Domain Decomposition*), then build the simulation from the recipe, the domain decomposition and the execution context. Optionally experimental inputs that can change between model runs, such as external spike trains, can be injected.

The recipe describes the model, the domain decomposition describes how the cells in the model are assigned to hardware resources and the context is used to execute the simulation.

3.7.2 Simulation execution and interaction

Simulations provide an interface for executing and interacting with the model:

- The simulation is executed/ *run* by advancing the model state from the current simulation time to another with maximum time step size.
- The model state can be *reset* to its initial state before the simulation was started.
- *Sampling* of the simulation state can be performed during execution with samplers and probes (e.g. compartment voltage and current) and spike output with the total number of spikes generated since either construction or reset.

The documentation for Arbor's Python simulation interface can be found in *Simulations*. See *Simulations* for documentation of the C++ simulation API.

3.8 Overview

The Python frontend for Arbor is an interface that the vast majority of users will use to interact with Arbor. This section covers how to use the frontend with examples and detailed descriptions of features.

3.8.1 Prerequisites

Once Arbor has been built and installed (see the *installation guide*), the location of the installed module needs to be set in the `PYTHONPATH` environment variable. For example:

```
export PYTHONPATH="/usr/lib/python3.7/site-packages/:$PYTHONPATH"
```

With this setup, Arbor's python module *arbor* can be imported with python3 via

```
>>> import arbor
```

3.8.2 Simulation steps

The workflow for defining and running a model defined in *Simulations* can be performed in Python as follows:

1. Describe the neuron model by defining an *arbor.recipe*;
2. Describe the computational resources to use for simulation using *arbor.proc_allocation* and *arbor.context*;
3. Partition the model over the hardware resources using *arbor.partition_load_balance*;
4. Run the model by initiating then running the *arbor.simulation*.

These details are described and examples are given in the next sections *Common Types*, *Recipes*, *Domain Decomposition*, *Simulations*, and *Metering*.

Note: Detailed information on Arbor's python features can also be obtained with Python's `help` function, e.g.

```
>>> help(arbor.proc_allocation)
Help on class proc_allocation in module arbor:

class proc_allocation(pybind11_builtins.pybind11_object)
|  Enumerates the computational resources on a node to be used for simulation.
|...
```

3.9 Common Types

3.9.1 Cell Identifiers and Indexes

The types defined below are used as identifiers for cells and members of cell-local collections.

class arbor.**cell_member**

cell_member(*gid*, *index*)

Construct a cell member with parameters *gid* and *index* for global identification of a cell-local item.

Items of type *cell_member* must:

- be associated with a unique cell, identified by the member *gid*;
- identify an item within a cell-local collection by the member *index*.

An example is uniquely identifying a synapse in the model. Each synapse has a post-synaptic cell (with *gid*), and an *index* into the set of synapses on the post-synaptic cell.

Lexographically ordered by *gid*, then *index*.

gid
The global identifier of the cell.

index
The cell-local index of the item. Local indices for items within a particular cell-local collection should be zero-based and numbered contiguously.

An example of a cell member construction reads as follows:

```
import arbor

# construct
cmem = arbor.cell_member(0, 0)

# set gid and index
cmem.gid = 1
cmem.index = 42
```

class arbor.**cell_kind**
Enumeration used to identify the cell kind, used by the model to group equal kinds in the same cell group.

cable
A cell with morphology described by branching 1D cable segments.

lif
A leaky-integrate and fire neuron.

spike_source
A proxy cell that generates spikes from a spike sequence provided by the user.

benchmark
A proxy cell used for benchmarking.

An example for setting the cell kind reads as follows:

```
import arbor

kind = arbor.cell_kind.cable
```

3.10 Recipes

The *recipe* class documentation is below.

A recipe describes neuron models in a cell-oriented manner and supplies methods to provide cell information. Details on why Arbor uses recipes and general best practices can be found in *Recipes*.

class arbor.**recipe**
Describe a model by describing the cells and network, without any information about how the model is to be represented or executed.

All recipes derive from this abstract base class.

Recipes provide a cell-centric interface for describing a model. This means that model properties, such as connections, are queried using the global identifier (*arbor.cell_member.gid*) of a cell. In the description below, the term `gid` is used as shorthand for the cell with global identifier.

Required Member Functions

The following member functions (besides a constructor) must be implemented by every recipe:

num_cells()
The number of cells in the model.

cell_kind(*gid*)
The cell kind of the cell with global identifier *arbor.cell_member.gid* (return type: *arbor.cell_kind*).

cell_description(*gid*)
A high level decription of the cell with global identifier *arbor.cell_member.gid*, for example the morphology, synapses and ion channels required to build a multi-compartment neuron. The type used to describe a cell depends on the kind of the cell. The interface for querying the kind and description of a cell are seperate to allow the cell type to be provided without building a full cell description, which can be very expensive.

Optional Member Functions

connections_on(*gid*)
Returns a list of all the **incoming** connections to *arbor.cell_member.gid*. Each connection should have post-synaptic target `connection.dest.gid` that matches the argument *arbor.cell_member.gid*, and a valid synapse id `connection.dest.index` on *arbor.cell_member.gid*. See *connection*.

By default returns an empty list.

gap_junctions_on(*gid*)
Returns a list of all the gap junctions connected to *arbor.cell_member.gid*. Each gap junction `gj` should have one of the two gap junction sites `gj.local.gid` or `gj.peer.gid` matching the argument *arbor.cell_member.gid*, and the corresponding synapse id `gj.local.index` or `gj.peer.index` should be valid on *arbor.cell_member.gid*. See *gap_junction_connection*.

By default returns an empty list.

event_generators(*gid*)
A list of all the *event_generator* s that are attached to *arbor.cell_member.gid*.

By default returns an empty list.

num_sources(*gid*)
The number of spike sources on *arbor.cell_member.gid*.

By default returns 0.

num_targets(*gid*)
The number of post-synaptic sites on *arbor.cell_member.gid*, which corresponds to the number of synapses.

By default returns 0.

num_gap_junction_sites(*gid*)
Returns the number of gap junction sites on *arbor.cell_member.gid*.

By default returns 0.

class arbor.**connection**
Describes a connection between two cells: Defined by source and destination end points (that is pre-synaptic and post-synaptic respectively), a connection weight and a delay time.

connection(*source*, *destination*, *weight*, *delay*)
Construct a connection between the *source* and the *dest* with a *weight* and *delay*.

source
The source end point of the connection (type: *arbor.cell_member*).

dest
The destination end point of the connection (type: *arbor.cell_member*).

weight
The weight delivered to the target synapse. The weight is dimensionless, and its interpretation is specific to the type of the synapse target. For example, the expsyn synapse interprets it as a conductance with units μS (micro-Siemens).

delay
The delay time of the connection [ms]. Must be positive.

An example of a connection reads as follows:

```
import arbor

# construct a connection between cells (0,0) and (1,0) with weight 0.01 and delay␣
↪of 10 ms.
src  = arbor.cell_member(0,0)
dest = arbor.cell_member(1,0)
w    = 0.01
d    = 10
con  = arbor.connection(src, dest, w, d)
```

class arbor.**gap_junction_connection**
Describes a gap junction between two gap junction sites. Gap junction sites are represented by *arbor.cell_member*.

local
The gap junction site: one half of the gap junction connection.

peer
The gap junction site: other half of the gap junction connection.

ggap
The gap junction conductance [μS].

3.10.1 Event Generator and Schedules

class arbor.**event_generator**

event_generator(*target*, *weight*, *schedule*)
Construct an event generator for a *target* synapse with *weight* of the events to deliver based on a schedule (i.e., *arbor.regular_schedule*, *arbor.explicit_schedule*, *arbor.poisson_schedule*).

target
The target synapse of type *arbor.cell_member*.

weight
The weight of events to deliver.

class arbor.**regular_schedule**
Describes a regular schedule with multiples of *dt* within the interval [*tstart*, *tstop*).

regular_schedule(*tstart*, *dt*, *tstop*)
Construct a regular schedule as list of times from *tstart* to *tstop* in *dt* time steps.

By default returns a schedule with *tstart* = *tstop* = None and *dt* = 0 ms.

tstart
The delivery time of the first event in the sequence [ms]. Must be non-negative or None.

dt
The interval between time points [ms]. Must be non-negative.

tstop
No events delivered after this time [ms]. Must be non-negative or None.

events (*t0*, *t1*)
Returns a view of monotonically increasing time values in the half-open interval [t0, t1).

class arbor.**explicit_schedule**
Describes an explicit schedule at a predetermined (sorted) sequence of *times*.

explicit_schedule (*times*)
Construct an explicit schedule.

By default returns a schedule with an empty list of times.

times
The list of non-negative times [ms].

events (*t0*, *t1*)
Returns a view of monotonically increasing time values in the half-open interval [t0, t1).

class arbor.**poisson_schedule**
Describes a schedule according to a Poisson process.

poisson_schedule (*tstart*, *freq*, *seed*)
Construct a Poisson schedule.

By default returns a schedule with events starting from *tstart* = 0 ms, with an expected frequency *freq* = 10 Hz and *seed* = 0.

tstart
The delivery time of the first event in the sequence [ms].

freq
The expected frequency [Hz].

seed
The seed for the random number generator.

events (*t0*, *t1*)
Returns a view of monotonically increasing time values in the half-open interval [t0, t1).

An example of an event generator reads as follows:

```
import arbor

# define a Poisson schedule with start time 1 ms, expected frequency of 5 Hz,
# and the target cell's gid as seed
target = arbor.cell_member(0,0)
seed   = target.gid
tstart = 1
freq   = 5
sched  = arbor.poisson_schedule(tstart, freq, seed)

# construct an event generator with this schedule on target cell and weight 0.1
```

(continues on next page)

(continued from previous page)

```
w      = 0.1
gen    = arbor.event_generator(target, w, sched)
```

3.10.2 Cells

class arbor.**cable_cell**
See *Python Cable Cells*.

class arbor.**lif_cell**
A benchmarking cell (leaky integrate-and-fire), used by Arbor developers to test communication performance, with neuronal parameters:

tau_m
Membrane potential decaying constant [ms].

V_th
Firing threshold [mV].

C_m
Membrane capacitance [pF].

E_L
Resting potential [mV].

V_m
Initial value of the Membrane potential [mV].

t_ref
Refractory period [ms].

V_reset
Reset potential [mV].

class arbor.**spike_source_cell**
A spike source cell, that generates a user-defined sequence of spikes that act as inputs for other cells in the network.

spike_source_cell(*schedule*)
Construct a spike source cell that generates spikes

- at regular intervals (using an *arbor.regular_schedule*)
- at a sequence of user-defined times (using an *arbor.explicit_schedule*)
- at times defined by a Poisson sequence (using an *arbor.poisson_schedule*)

Parameters **schedule** – User-defined sequence of time points (choose from *arbor.regular_schedule*, *arbor.explicit_schedule*, or *arbor.poisson_schedule*).

class arbor.**benchmark_cell**
A benchmarking cell, used by Arbor developers to test communication performance.

benchmark_cell(*schedule*, *realtime_ratio*)
A benchmark cell generates spikes at a user-defined sequence of time points:

- at regular intervals (using an *arbor.regular_schedule*)
- at a sequence of user-defined times (using an *arbor.explicit_schedule*)

- at times defined by a Poisson sequence (using an *arbor.poisson_schedule*)

and the time taken to integrate a cell can be tuned by setting the parameter `realtime_ratio`.

Parameters

- **schedule** – User-defined sequence of time points (choose from *arbor.regular_schedule*, *arbor.explicit_schedule*, or *arbor.poisson_schedule*).
- **realtime_ratio** – Time taken to integrate a cell, for example if `realtime_ratio` = 2, a cell will take 2 seconds of CPU time to simulate 1 second.

Below is an example of a recipe construction of a ring network of multi-compartmental cells. Because the interface for specifying cable morphology cells is under construction, the temporary helpers in cell_parameters and make_cable_cell for building cells are used.

```
import sys
import arbor

class ring_recipe (arbor.recipe):

    def __init__(self, n=4):
        # The base C++ class constructor must be called first, to ensure that
        # all memory in the C++ class is initialized correctly.
        arbor.recipe.__init__(self)
        self.ncells = n
        self.params = arbor.cell_parameters()

    # The num_cells method that returns the total number of cells in the model
    # must be implemented.
    def num_cells(self):
        return self.ncells

    # The cell_description method returns a cell
    def cell_description(self, gid):
        return arbor.make_cable_cell(gid, self.params)

    def num_targets(self, gid):
        return 1

    def num_sources(self, gid):
        return 1

    # The kind method returns the type of cell with gid.
    # Note: this must agree with the type returned by cell_description.
    def cell_kind(self, gid):
        return arbor.cell_kind.cable

    # Make a ring network
    def connections_on(self, gid):
        src = (gid-1)%self.ncells
        w = 0.01
        d = 10
        return [arbor.connection(arbor.cell_member(src,0), arbor.cell_member(gid,0),␣
↪w, d)]

    # Attach a generator to the first cell in the ring.
    def event_generators(self, gid):
```

(continues on next page)

(continued from previous page)

```
        if gid==0:
            sched = arbor.explicit_schedule([1])
            return [arbor.event_generator(arbor.cell_member(0,0), 0.1, sched)]
        return []
```

3.11 Python Cable Cells

The interface for specifying cell morphologies with the distribution of ion channels and synapses is a key part of the user interface. Arbor will have an advanced and user-friendly interface for this, which is currently under construction.

To allow users to experiment will multi-compartment cells, we provide some helpers for generating cells with random morphologies, which are documented here.

Warning: These features will be deprecated once the morphology interface has been implemented.

make_cable_cell (*seed*, *params*)

Construct a branching `cable_cell` with a random morphology (via parameter `seed`) and synapse end points locations described by parameter `params`.

The soma has an area of 500 μm^2, a bulk resistivity of 100 Ω·cm, and the ion channel and synapse dynamics are described by a Hodgkin-Huxley (HH) mechanism. The default parameters of HH mechanisms are:

- Na-conductance 0.12 Scm^2,
- K-conductance 0.036 Scm^2,
- passive conductance 0.0003 Scm^2, and
- passive potential -54.3 mV

Each cable has a diameter of 1 μm, a bulk resistivity of 100 Ω·cm, and the ion channel and synapse dynamics are described by a passive/ leaky integrate-and-fire model with parameters:

- passive conductance 0.001 Scm^2, and
- resting potential -65 mV

Further, a spike detector is added at the soma with threshold 10 mV, and a synapse is added to the mid point of the first dendrite with an exponential synapse model:

- time decaying constant 2 ms
- resting potential 0 mV

Additional synapses are added based on the number of randomly generated *cell_parameters.synapses* on the cell.

Parameters

- **seed** – The seed is an integral value used to seed the random number generator, for which the *arbor.cell_member.gid* of the cell is a good default.
- **params** – By default set to *cell_parameters()*.

class cell_parameters

Parameters used to generate random cell morphologies. Where parameters must be given as ranges, the first value is at the soma, and the last value is used on the last level. Values at levels in between are found by linear interpolation.

depth
The maximum depth of the branch structure (i.e., maximum number of levels in the cell (not including the soma)).

lengths
The length of the branch [μm], given as a range `[l1, l2]`.

synapses
The number of randomly generated synapses on the cell.

branch_probs
The probability of a branch occuring, given as a range `[p1, p2]`.

compartments
The compartment count on a branch, given as a range `[n1, n2]`.

3.12 Hardware Management

Arbor provides two ways for working with hardware resources:

- *Prescribe* the hardware resources and their contexts for use in Arbor simulations.
- *Query* available hardware resources (e.g. the number of available GPUs), and initializing MPI.

3.12.1 Available Resources

Helper functions for checking cmake or environment variables, as well as configuring and checking MPI are the following:

`arbor.`**`config`**`()`
Returns a dictionary to check which options the Arbor library was configured with at compile time:

- `ARB_MPI_ENABLED`
- `ARB_WITH_MPI4PY`
- `ARB_GPU_ENABLED`
- `ARB_VERSION`

```
import arbor
arbor.config()

{'mpi': True, 'mpi4py': True, 'gpu': False, 'version': '0.2.1-dev'}
```

`arbor.`**`mpi_init`**`()`
Initialize MPI with `MPI_THREAD_SINGLE`, as required by Arbor.

`arbor.`**`mpi_is_initialized`**`()`
Check if MPI is initialized.

`class` `arbor.`**`mpi_comm`**

mpi_comm()
By default sets MPI_COMM_WORLD as communicator.

mpi_comm(*object*)
Converts a Python object to an MPI Communicator.

arbor.**mpi_finalize**()
Finalize MPI by calling `MPI_Finalize`.

arbor.**mpi_is_finalized**()
Check if MPI is finalized.

3.12.2 Prescribed Resources

The Python wrapper provides an API for:

- prescribing which hardware resources are to be used by a simulation using *proc_allocation*.
- opaque handles to hardware resources used by simulations called *context*.

class arbor.**proc_allocation**
Enumerates the computational resources on a node to be used for a simulation, specifically the number of threads and identifier of a GPU if available.

proc_allocation()
By default selects one thread and no GPU.

proc_allocation(*threads*, *gpu_id*)
Constructor that sets the number of *threads* and the id *gpu_id* of the available GPU.

threads
The number of CPU threads available, 1 by default.

gpu_id
The identifier of the GPU to use. Must be `None`, or a non-negative integer.

The *gpu_id* corresponds to the `int device` parameter used by CUDA API calls to identify gpu devices. Set to `None` to indicate that no GPU device is to be used. See `cudaSetDevice` and `cudaDeviceGetAttribute` provided by the CUDA API.

has_gpu()
Indicates whether a GPU is selected (i.e., whether *gpu_id* is `None`).

Here are some examples of how to create a *proc_allocation*.

```
import arbor

# default: one thread and no GPU selected
alloc1 = arbor.proc_allocation()

# 8 threads and no GPU
alloc2 = arbor.proc_allocation(8, None)

# reduce alloc2 to 4 threads and use the first available GPU
alloc2.threads = 4
alloc2.gpu_id  = 0
```

class arbor.**context**
An opaque handle for the hardware resources used in a simulation. A *context* contains a thread pool, and optionally the GPU state and MPI communicator. Users of the library do not directly use the functionality provided

by *context*, instead they configure contexts, which are passed to Arbor interfaces for domain decomposition and simulation.

context()
Construct a local context with one thread, no GPU, no MPI.

context(*alloc*)
Create a local context, with no distributed/MPI, that uses the local resources described by *proc_allocation*.

alloc
The computational resources, one thread and no GPU by default.

context(*alloc*, *mpi*)
Create a distributed context, that uses the local resources described by *proc_allocation*, and uses the MPI communicator for distributed calculation.

alloc
The computational resources, one thread and no GPU by default.

mpi
The MPI communicator (see *mpi_comm*). mpi must be `None`, or an MPI communicator.

context(*threads*, *gpu_id*)
Create a context that uses a set number of *threads* and the GPU with id *gpu_id*.

threads
The number of threads available locally for execution, 1 by default.

gpu_id
The identifier of the GPU to use, `None` by default. Must be `None`, or a non-negative integer.

context(*threads*, *gpu_id*, *mpi*)
Create a context that uses a set number of *threads* and gpu identifier *gpu_id* and MPI communicator *mpi* for distributed calculation.

threads
The number of threads available locally for execution, 1 by default.

gpu_id
The identifier of the GPU to use, `None` by default. Must be `None`, or a non-negative integer.

mpi
The MPI communicator (see *mpi_comm*). mpi must be `None`, or an MPI communicator.

Contexts can be queried for information about which features a context has enabled, whether it has a GPU, how many threads are in its thread pool.

has_gpu
Query whether the context has a GPU.

has_mpi
Query whether the context uses MPI for distributed communication.

threads
Query the number of threads in the context's thread pool.

ranks
Query the number of distributed domains. If the context has an MPI communicator, return is equivalent to `MPI_Comm_size`. If the communicator has no MPI, returns 1.

rank
: The numeric id of the local domain. If the context has an MPI communicator, return is equivalent to `MPI_Comm_rank`. If the communicator has no MPI, returns 0.

Here are some simple examples of how to create a *context*:

```
import arbor
import mpi4py.MPI as mpi

# Construct a context that uses 1 thread and no GPU or MPI.
context = arbor.context()

# Construct a context that:
#   * uses 8 threads in its thread pool;
#   * does not use a GPU, reguardless of whether one is available
#   * does not use MPI.
alloc   = arbor.proc_allocation(8, None)
context = arbor.context(alloc)

# Construct a context that uses:
#   * 4 threads and the first GPU;
#   * MPI_COMM_WORLD for distributed computation.
alloc   = arbor.proc_allocation(4, 0)
comm    = arbor.mpi_comm(mpi.COMM_WORLD)
context = arbor.context(alloc, comm)
```

3.13 Domain Decomposition

The Python API for partitioning a model over distributed and local hardware is described here.

3.13.1 Load Balancers

Load balancing generates a *domain_decomposition* given an *arbor.recipe* and a description of the hardware on which the model will run. Currently Arbor provides one load balancer, *partition_load_balance()*, and more will be added over time.

If the model is distributed with MPI, the partitioning algorithm for cells is distributed with MPI communication. The returned *domain_decomposition* describes the cell groups on the local MPI rank.

`arbor.`**`partition_load_balance`**(*recipe*, *context*, *hints*)
: Construct a *domain_decomposition* that distributes the cells in the model described by an *arbor.recipe* over the distributed and local hardware resources described by an *arbor.context*.

  The algorithm counts the number of each cell type in the global model, then partitions the cells of each type equally over the available nodes. If a GPU is available, and if the cell type can be run on the GPU, the cells on each node are put into one large group to maximise the amount of fine grained parallelism in the cell group. Otherwise, cells are grouped into small groups that fit in cache, and can be distributed over the available cores. Optionally, provide a dictionary of *partition_hint* s for certain cell kinds, by default this dictionary is empty.

Note: The partitioning assumes that all cells of the same kind have equal computational cost, hence it may not produce a balanced partition for models with cells that have a large variance in computational costs.

---

class arbor.**partition_hint**
Provide a hint on how the cell groups should be partitioned.

partition_hint(*cpu_group_size*, *gpu_group_size*, *prefer_gpu*)
Construct a partition hint with arguments *cpu_group_size* and *gpu_group_size*, and whether to *prefer_gpu*.

By default returns a partition hint with *cpu_group_size* = `1`, i.e., each cell is put in its own group, *gpu_group_size* = `max`, i.e., all cells are put in one group, and *prefer_gpu* = `True`, i.e., GPU usage is preferred.

cpu_group_size
The size of the cell group assigned to CPU. Must be positive, else set to default value.

gpu_group_size
The size of the cell group assigned to GPU. Must be positive, else set to default value.

prefer_gpu
Whether GPU usage is preferred.

max_size
Get the maximum size of cell groups.

An example of a partition load balance with hints reads as follows:

```
import arbor

# Get a communication context (with 4 threads, no GPU)
context = arbor.context(threads=4, gpu_id=None)

# Initialise a recipe of user defined type my_recipe with 100 cells.
n_cells = 100
recipe = my_recipe(n_cells)

# The hints perfer the multicore backend, so the decomposition is expected
# to never have cell groups on the GPU, regardless of whether a GPU is
# available or not.
cable_hint                  = arb.partition_hint()
cable_hint.prefer_gpu       = False
cable_hint.cpu_group_size   = 3
spike_hint                  = arb.partition_hint()
spike_hint.prefer_gpu       = False
spike_hint.cpu_group_size   = 4
hints = dict([(arb.cell_kind.cable, cable_hint), (arb.cell_kind.spike_source, spike_
↪hint)])

decomp = arb.partition_load_balance(recipe, context, hints)
```

3.13.2 Decomposition

As defined in *Domain Decomposition* a domain decomposition is a description of the distribution of the model over the available computational resources. Therefore, the following data structures are used to describe domain decompositions.

class arbor.**backend**
Enumeration used to indicate which hardware backend to execute a cell group on.

multicore
Use multicore backend.

gpu
Use GPU backend.

Note: Setting the GPU back end is only meaningful if the cell group type supports the GPU backend.

---

class arbor.**domain_decomposition**
Describes a domain decomposition and is soley responsible for describing the distribution of cells across cell groups and domains. It holds cell group descriptions (*groups*) for cells assigned to the local domain, and a helper function (*gid_domain()*) used to look up which domain a cell has been assigned to. The *domain_decomposition* object also has meta-data about the number of cells in the global model, and the number of domains over which the model is destributed.

Note: The domain decomposition represents a division of **all** of the cells in the model into non-overlapping sets, with one set of cells assigned to each domain.

---

gid_domain(*gid*)
A function for querying the domain id that a cell is assigned to (using global identifier *arbor.cell_member.gid*).

num_domains
The number of domains that the model is distributed over.

domain_id
The index of the local domain. Always 0 for non-distributed models, and corresponds to the MPI rank for distributed runs.

num_local_cells
The total number of cells in the local domain.

num_global_cells
The total number of cells in the global model (sum of *num_local_cells* over all domains).

groups
The descriptions of the cell groups on the local domain. See *group_description*.

class arbor.**group_description**
Return the indexes of a set of cells of the same kind that are grouped together in a cell group in an *arbor.simulation*.

group_description(*kind*, *gids*, *backend*)
Construct a group description with parameters *kind*, *gids* and *backend*.

kind
The kind of cell in the group.

gids
The list of gids of the cells in the cell group.

backend
The hardware *backend* on which the cell group will run.

3.14 Simulations

3.14.1 From recipe to simulation

To build a simulation the following concepts are needed:

- an *arbor.recipe* that describes the cells and connections in the model;
- an *arbor.context* used to execute the simulation.

The workflow to build a simulation is to first generate an *arbor.domain_decomposition* based on the *arbor.recipe* and *arbor.context* describing the distribution of the model over the local and distributed hardware resources (see *Domain Decomposition*). Then, the simulation is build using this *arbor.domain_decomposition*.

```
import arbor

# Get a communication context (with 4 threads, no GPU)
context = arbor.context(threads=4, gpu_id=None)

# Initialise a recipe of user defined type my_recipe with 100 cells.
n_cells = 100
recipe = my_recipe(n_cells)

# Get a description of the partition of the model over the cores.
decomp = arbor.partition_load_balance(recipe, context)

# Instatitate the simulation.
sim = arbor.simulation(recipe, decomp, context)

# Run the simulation for 2000 ms with time stepping of 0.025 ms
tSim = 2000
dt = 0.025
sim.run(tSim, dt)
```

class arbor.**simulation**

The executable form of a model. A simulation is constructed from a recipe, and then used to update and monitor the model state.

Simulations take the following inputs:

The **constructor** takes

- an *arbor.recipe* that describes the model;
- an *arbor.domain_decomposition* that describes how the cells in the model are assigned to hardware resources;
- an *arbor.context* which is used to execute the simulation.

Simulations provide an interface for executing and interacting with the model:

- **Advance the model state** from one time to another and reset the model state to its original state before simulation was started.
- Sample the simulation state during the execution (e.g. compartment voltage and current) and generate spike output by using an **I/O interface**.

Constructor:

simulation (*recipe*, *domain_decomposition*, *context*)
Initialize the model described by an *arbor.recipe*, with cells and network distributed according to *arbor.domain_decomposition*, and computational resources described by *arbor.context*.

Updating Model State:

reset()
Reset the state of the simulation to its initial state.

run (*tfinal*, *dt*)
Run the simulation from current simulation time to `tfinal`, with maximum time step size `dt`.

Parameters

- **tfinal** – The final simulation time [ms].
- **dt** – The time step size [ms].

set_binning_policy (*policy*, *bin_interval*)
Set the binning `policy` for event delivery, and the binning time interval `bin_interval` if applicable [ms].

Parameters

- **policy** – The binning policy of type *binning*.
- **bin_interval** – The binning time interval [ms].

Types:

class binning
Enumeration for event time binning policy.

none
No binning policy.

regular
Round time down to multiple of binning interval.

following
Round times down to previous event if within binning interval.

3.14.2 Recording spikes

In order to analyze the simulation output spikes can be recorded.

Types:

class arbor.**spike**

spike()
Construct a spike.

source
The spike source (type: *arbor.cell_member*).

time
The spike time [ms].

class arbor.**spike_recorder**

spike_recorder()
Initialize the spike recorder.

spikes
The recorded spikes (type: *spike*).

I/O interface:

arbor.**attach_spike_recorder**(*sim*)
Attach a spike recorder to an arbor *simulation* sim. The recorder that is returned will record all spikes generated after it has been attached (spikes generated before attaching are not recorded).

```
import arbor

# Instatitate the simulation.
sim = arbor.simulation(recipe, decomp, context)

# Build the spike recorder
recorder = arbor.attach_spike_recorder(sim)

# Run the simulation for 2000 ms with time stepping of 0.025 ms
tSim = 2000
dt = 0.025
sim.run(tSim, dt)

# Print the spikes and according spike time
for s in recorder.spikes:
    print(s)
```

```
>>> <arbor.spike: source (0,0), time 2.15168 ms>
>>> <arbor.spike: source (1,0), time 14.5235 ms>
>>> <arbor.spike: source (2,0), time 26.9051 ms>
>>> <arbor.spike: source (3,0), time 39.4083 ms>
>>> <arbor.spike: source (4,0), time 51.9081 ms>
>>> <arbor.spike: source (5,0), time 64.2902 ms>
>>> <arbor.spike: source (6,0), time 76.7706 ms>
>>> <arbor.spike: source (7,0), time 89.1529 ms>
>>> <arbor.spike: source (8,0), time 101.641 ms>
>>> <arbor.spike: source (9,0), time 114.125 ms>
```

3.15 Metering

Arbor's python module *arbor* has a *meter_manager* for measuring time (and if applicable memory) consumptions of regions of interest in the python code.

Users manually instrument the regions to measure. This allows the user to only measure the parts of the python code that are of interest. Once a region of code is marked for the *meter_manager*, the application will track the total time (and memory) spent in this region.

3.15.1 Marking Metering Regions

First the *meter_manager* needs to be initiated, then the metering started and checkpoints set, wherever the *meter_manager* should report the meters. The measurement starts from the *meter_manager.start()* to the first *meter_manager.checkpoint()* and then in between checkpoints. Checkpoints are defined by a string describing the process to be measured.

class arbor.**meter_manager**

meter_manager()
Construct the meter manager.

start(*context*)
Start the metering using the chosen execution *arbor.context*. Records a time stamp, that marks the start of the first checkpoint timing region.

checkpoint(*name*, *context*)
Create a new checkpoint `name` using the chosen execution *arbor.context*. Records the time since the last checkpoint (or the call to start if no previous checkpoints exist), and restarts the timer for the next checkpoint.

checkpoint_names()
Returns a list of all metering checkpoint names.

times()
Returns a list of all metering times.

For instance, the following python code will record and summarize the total time (and memory) spent:

```
import arbor

context = arbor.context(threads=8, gpu_id=None)
meter_manager = arbor.meter_manager()
meter_manager.start(context)

n_cells = 100
recipe = my_recipe(n_cells)

meter_manager.checkpoint('recipe-create', context)

decomp = arbor.partition_load_balance(recipe, context)

meter_manager.checkpoint('load-balance', context)

sim = arbor.simulation(recipe, decomp, context)

meter_manager.checkpoint('simulation-init', context)

tSim = 2000
dt = 0.025
sim.run(tSim, dt)

meter_manager.checkpoint('simulation-run', context)
```

3.15.2 Metering Output

At any point a summary of the timing regions can be obtained by the *meter_report*.

class arbor.**meter_report**

meter_report(*meter_manager*, *context*)

Summarises the performance meter results, used to print a report to screen or file. If a distributed context is used, the report will contain a summary of results from all MPI ranks.

Take the example output from above:

```
print(arbor.meter_report(meter_manager, context))
```

```
>>> ---- meters ----------------------------------------------------------------
 ↪---------
>>> meter                          time(s)      memory(MB)
>>> ----------------------------------------------------------------------------
 ↪---------
>>> recipe-create                   0.000           0.001
>>> load-balance                    0.000           0.009
>>> simulation-init                 0.026           3.604
>>> simulation-run                  4.171           0.021
>>> meter-total                     4.198           3.634
```

3.16 Overview

The C++ API for is the main interface through which application developers will access Arbor, though it is designed to be usable for power users to implement models.

Arbor makes a distinction between the **description** of a model, and the **execution** of a model.

A *arb::recipe* describes a model, and a *arb::simulation* is an executable instantiation of a model.

3.17 Common Types

3.17.1 Cell Identifiers and Indexes

These types, defined in `common_types.hpp`, are used as identifiers for cells and members of cell-local collections.

Note: Arbor uses `std::unit32_t` for *cell_gid_type*, *cell_size_type*, *cell_lid_type*, and *cell_local_size_type* at the time of writing, however this could change, e.g. to handle models that cell gid that don't fit into a 32 bit unsigned integer. It is thus recommended that these type aliases be used whenever identifying or counting cells and cell members.

---

type cell_gid_type
: An integer type used for identifying cells globally.

type cell_size_type
: An unsigned integer for sizes of collections of cells. Unsigned type for counting *cell_gid_type*.

type cell_lid_type
: For indexes into cell-local data. Local indices for items within a particular cell-local collection should be zero-based and numbered contiguously.

type cell_local_size_type
: An unsigned integer for for counts of cell-local data.

class cell_member_type
: For global identification of an item of cell local data. Items of *cell_member_type* must:

  - be associated with a unique cell, identified by the member *gid*;
  - identify an item within a cell-local collection by the member *index*.

An example is uniquely identifying a synapse in the model. Each synapse has a post-synaptic cell (*gid*), and an index (*index*) into the set of synapses on the post-synaptic cell.

Lexicographically ordered by *gid*, then *index*.

cell_gid_type **gid**
Global identifier of the cell containing/associated with the item.

cell_lid_type **index**
The index of the item in a cell-local collection.

enum class cell_kind
Enumeration used to identify the cell type/kind, used by the model to group equal kinds in the same cell group.

enumerator cable
A cell with morphology described by branching 1D cable segments.

enumerator lif
Leaky-integrate and fire neuron.

enumerator spike_source
Proxy cell that generates spikes from a spike sequence provided by the user.

enumerator benchmark
Proxy cell used for benchmarking.

3.17.2 Probes

using probe_tag = int
Extra contextual information associated with a probe.

class probe_info
Probes are specified in the recipe objects that are used to initialize a model; the specification of the item or value that is subjected to a probe will be specific to a particular cell type.

cell_member_type **id**
Cell gid, index of probe.

probe_tag **tag**
Opaque key, returned in sample record.

util::*any* **address**
Cell-type specific location info, specific to cell kind of `id.gid`.

3.17.3 Utility Wrappers and Containers

template<typename **T**>
class optional
A wrapper around a contained value of type *T*, that may or may not be set. A faithful copy of the C++17 `std::optional` type. See the online C++ standard documentation https://en.cppreference.com/w/cpp/utility/optional for more information.

class any
A container for a single value of any type that is copy constructable. Used in the Arbor API where a type of a value passed to or from the API is decided at run time.

A faithful copy of the C++17 `std::any` type. See the online C++ standard documentation https://en.cppreference.com/w/cpp/utility/any for more information.

The `arb::util` namespace also implementations of the `any_cast`, `make_any` and `bad_any_cast` helper functions and types from C++17.

class unique_any
Equivalent to *util::any*, except that:

- it can store any type that is move constructable;
- it is move only, that is it can't be copied.

3.18 Hardware Management

Arbor provides two library APIs for working with hardware resources:

- The core *libarbor* is used to *describe* the hardware resources and their contexts for use in Arbor simulations.
- The *libarborenv* provides an API for querying available hardware resources (e.g. the number of available GPUs), and initializing MPI.

3.18.1 libarborenv

The *libarborenv* API for querying and managing hardware resources is in the `arbenv` namespace. This functionality is kept in a separate library to enforce separation of concerns, so that users have full control over how hardware resources are selected, either using the functions and types in *libarborenv*, or writing their own code for managing MPI, GPUs, and thread counts.

arb::util::*optional*<int> **get_env_num_threads**()
Tests whether the number of threads to use has been set in an environment variable. First checks `ARB_NUM_THREADS`, and if that is not set checks `OMP_NUM_THREADS`.

Return value:

- **no value**: the `optional` return value contains no value if the no thread count was specified by an environment variable.
- **has value**: the number of threads set by the environment variable.

Throws:

- throws `std::runtime_error` if environment variable set with invalid number of threads.

```
#include <arborenv/concurrency.hpp>

if (auto nt = arbenv::get_env_num_threads()) {
   std::cout << "requested " << nt.value() << "threads \n";
}
else {
   std::cout << "no environment variable set\n";
}
```

int **thread_concurrency**()
Attempts to detect the number of available CPU cores. Returns 1 if unable to detect the number of cores.

```
#include <arborenv/concurrency.hpp>

// Set num_threads to value from environment variable if set,
// otherwise set it to the available number of cores.
int num_threads = 0;
```

(continues on next page)

(continued from previous page)

```
if (auto nt = arbenv::get_env_num_threads()) {
   num_threads = nt.value();
}
else {
   num_threads = arbenv::thread_concurrency();
}
```

int **default_gpu**()
Returns the integer identifier of the first available GPU, if a GPU is available

Return value:

- **non-negative value**: if a GPU is available, the index of the selected GPU is returned. The index will be in the range `[0, num_gpus)` where `num_gpus` is the number of GPUs detected using the `cudaGetDeviceCount` CUDA API call.
- **-1**: if no GPU available, or if Arbor was built without GPU support.

```
#include <arborenv/gpu_env.hpp>

if (arbenv::default_gpu()>-1) {}
   std::cout << "a GPU is available\n";
}
```

int **find_private_gpu**(MPI_Comm *comm*)
A helper function that assigns a unique GPU to every MPI rank.

Note: Arbor allows at most one GPU per MPI rank, and furthermore requires that an MPI rank has exclusive access to a GPU, i.e. two MPI ranks can not share a GPU. This function assigns a unique GPU to each rank when more than one rank has access to the same GPU(s). An example use case is on systems with "fat" nodes with multiple GPUs per node, in which case Arbor should be run with multiple MPI ranks per node. Uniquely assigning GPUs is quite difficult, and this function provides what we feel is a robust implementation.

All MPI ranks in the MPI communicator *comm* should call to avoid a deadlock.

Return value:

- **non-negative integer**: the identifier of the GPU assigned to this rank.
- **-1**: no GPU was available for this MPI rank.

Throws:

- `std::runtime_error`: if there was an error in the CUDA runtime on the local or remote MPI ranks, i.e. if one rank throws, all ranks will throw.

class with_mpi
The *with_mpi* type is a simple RAII scoped guard for MPI initialization and finalization. On creation *with_mpi* will call `MPI_Init_thread` to initialize MPI with the minimum level thread support required by Arbor, that is `MPI_THREAD_SERIALIZED`. When it goes out of scope it will automatically call `MPI_Finalize`.

with_mpi(int *&argcp*, char **&argvp*, bool *fatal_errors* = true)
The constructor takes the `argc` and `argv` arguments passed to main of the calling application, and an additional flag *fatal_errors* that toggles whether errors in MPI API calls should return error codes or terminate.

Warning: Handling exceptions is difficult in MPI applications, and it is the users responsibility to do so.

The *with_mpi* scope guard attempts to facilitate error reporting of uncaught exceptions, particularly in the case where one rank throws an exception, while the other ranks continue executing. In this case there would be a deadlock if the rank with the exception attempts to call `MPI_Finalize` and other ranks are waiting in other MPI calls. If this happens inside a try-catch block, the deadlock stops the exception from being handled. For this reason the destructor of *with_mpi* only calls `MPI_Finalize` if there are no uncaught exceptions. This isn't perfect because the other MPI ranks still deadlock, however it gives the exception handling code to report the error for debugging.

An example workflow that uses the MPI scope guard. Note that this code will print the exception error message in the case where only one MPI rank threw an exception, though it would either then deadlock or exit with an error code that one or more MPI ranks exited without calling `MPI_Finalize`.

```
#include <exception>
#include <iostream>

#include <arborenv/with_mpi.hpp>

int main(int argc, char** argv) {
    try {
        // Constructing guard will initialize MPI with a
        // call to MPI_Init_thread()
        arbenv::with_mpi guard(argc, argv, false);

        // Do some work with MPI here

        // When leaving this scope, the destructor of guard will
        // call MPI_Finalize()
    }
    catch (std::exception& e) {
        std::cerr << "error: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

3.18.2 libarbor

The core Arbor library *libarbor* provides an API for:

- prescribing which hardware resources are to be used by a simulation using *arb::proc_allocation*.
- opaque handles to hardware resources used by simulations called *arb::context*.

class proc_allocation
Enumerates the computational resources on a node to be used for simulation, specifically the number of threads and identifier of a GPU if available.

Note: Each MPI rank in a distributed simulation uses a *proc_allocation* to describe the subset of resources on its node that it will use.

```
#include <arbor/context.hpp>

// default: 1 thread and no GPU selected
arb::proc_allocation resources;

// 8 threads and no GPU
arb::proc_allocation resources(8, -1);

// 4 threads and the first available GPU
arb::proc_allocation resources(8, 0);

// Construct with
auto num_threads = arbenv::thread_concurrency();
auto gpu_id = arbenv::default_gpu();
arb::proc_allocation resources(num_threads, gpu_id);
```

proc_allocation() = default
: By default selects one thread and no GPU.

proc_allocation(unsigned *threads*, int *gpu_id*)
: Constructor that sets the number of `threads` and the id *gpu_id* of the available GPU.

unsigned **num_threads**
: The number of CPU threads available.

int **gpu_id**
: The identifier of the GPU to use. The gpu id corresponds to the `int device` parameter used by CUDA API calls to identify gpu devices. Set to -1 to indicate that no GPU device is to be used. See `cudaSetDevice` and `cudaDeviceGetAttribute` provided by the CUDA API.

bool **has_gpu**() **const**
: Indicates whether a GPU is selected (i.e. whether *gpu_id* is `-1`).

class context
: An opaque handle for the hardware resources used in a simulation. A *context* contains a thread pool, and optionally the GPU state and MPI communicator. Users of the library do not directly use the functionality provided by *context*, instead they create contexts, which are passed to Arbor interfaces for domain decomposition and simulation.

Arbor contexts are created by calling *make_context()*, which returns an initialized context. There are two versions of *make_context()*, for creating contexts with and without distributed computation with MPI respectively.

context **make_context**(*proc_allocation alloc* = *proc_allocation*())
: Create a local *context*, with no distributed/MPI, that uses local resources described by *alloc*. By default it will create a context with one thread and no GPU.

context **make_context**(*proc_allocation alloc*, MPI_Comm *comm*)
: Create a distributed *context*. A context that uses the local resources described by *alloc*, and uses the MPI communicator `comm` for distributed calculation.

Contexts can be queried for information about which features a context has enabled, whether it has a GPU, how many threads are in its thread pool, using helper functions.

bool **has_gpu**(**const** *context*&)
: Query whether the context has a GPU.

unsigned **num_threads**(**const** *context*&)
: Query the number of threads in a context's thread pool.

bool **has_mpi**(**const** *context*&)
: Query whether the context uses MPI for distributed communication.

unsigned **num_ranks**(**const** *context*&)
: Query the number of distributed ranks. If the context has an MPI communicator, return is equivalent to `MPI_Comm_size`. If the communicator has no MPI, returns 1.

unsigned **rank**(**const** *context*&)
: Query the rank of the calling rank. If the context has an MPI communicator, return is equivalent to `MPI_Comm_rank`. If the communicator has no MPI, returns 0.

Here are some simple examples of how to create a *arb::context* using *make_context()*.

```
#include <arbor/context.hpp>

// Construct a context that uses 1 thread and no GPU or MPI.
auto context = arb::make_context();

// Construct a context that:
//  * uses 8 threads in its thread pool;
//  * does not use a GPU, regardless of whether one is available;
//  * does not use MPI.
arb::proc_allocation resources(8, -1);
auto context = arb::make_context(resources);

// Construct one that uses:
//  * 4 threads and the first GPU;
//  * MPI_COMM_WORLD for distributed computation.
arb::proc_allocation resources(4, 0);
auto mpi_context = arb::make_context(resources, MPI_COMM_WORLD)
```

Here is a more complicated example of creating a *context* on a system where support for GPU and MPI support are conditional.

```
#include <arbor/context.hpp>
#include <arbor/version.hpp>   // for ARB_MPI_ENABLED

#include <arborenv/concurrency.hpp>
#include <arborenv/gpu_env.hpp>

int main(int argc, char** argv) {
    try {
        arb::proc_allocation resources;

        // try to detect how many threads can be run on this system
        resources.num_threads = arbenv::thread_concurrency();

        // override thread count if the user set ARB_NUM_THREADS
        if (auto nt = arbenv::get_env_num_threads()) {
            resources.num_threads = nt;
        }

#ifdef ARB_WITH_MPI
        // initialize MPI
        arbenv::with_mpi guard(argc, argv, false);

        // assign a unique gpu to this rank if available
        resources.gpu_id = arbenv::find_private_gpu(MPI_COMM_WORLD);

        // create a distributed context
        auto context = arb::make_context(resources, MPI_COMM_WORLD);
```

(continues on next page)

(continued from previous page)

```
        root = arb::rank(context) == 0;
#else
        resources.gpu_id = arbenv::default_gpu();

        // create a local context
        auto context = arb::make_context(resources);
#endif

        // Print a banner with information about hardware configuration
        std::cout << "gpu:      " << (has_gpu(context)? "yes": "no") << "\n";
        std::cout << "threads:  " << num_threads(context) << "\n";
        std::cout << "mpi:      " << (has_mpi(context)? "yes": "no") << "\n";
        std::cout << "ranks:    " << num_ranks(context) << "\n" << std::endl;

        // run some simulations!
    }
    catch (std::exception& e) {
        std::cerr << "exception caught in ring miniapp: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

3.19 Recipes

The *arb::recipe* class documentation is below.

3.19.1 C++ Best Practices

Here we collect rules of thumb to keep in mind when making recipes in C++.

Stay thread safe

The load balancing and model construction are multithreaded, that is multiple threads query the recipe simultaneously. Hence calls to a recipe member should not have side effects, and should use lazy evaluation when possible (see Be lazy).

3.19.2 Class Documentation

class recipe

A description of a model, describing the cells and network, without any information about how the model is to be represented or executed.

All recipes derive from this abstract base class, defined in `src/recipe.hpp`.

Recipes provide a cell-centric interface for describing a model. This means that model properties, such as connections, are queried using the global identifier (*gid*) of a cell. In the description below, the term *gid* is used as shorthand for "the cell with global identifier *gid*".

Warning: All member functions must be **thread safe**, because the recipe is used by the multithreaded model building stage. In practice, this means that multiple threads should be able to call member functions of a recipe simultaneously. Model building is multithreaded to reduce model building times, so recipe implementations should avoid using locks and mutexes to introduce thread safety. See *recipe best practices* for more information.

Required Member Functions

The following member functions must be implemented by every recipe:

virtual *cell_size_type* **num_cells** () **const** = 0
The number of cells in the model.

virtual *cell_kind* **get_cell_kind** (*cell_gid_type gid*) **const** = 0
The kind of *gid* (see `arb::cell_kind`).

virtual util::*unique_any* **get_cell_description** (*cell_gid_type gid*) **const** = 0
A description of the cell *gid*, for example the morphology, synapses and ion channels required to build a multi-compartment neuron.

The type used to describe a cell depends on the kind of the cell. The interface for querying the kind and description of a cell are separate to allow the cell type to be provided without building a full cell description, which can be very expensive.

Optional Member Functions

virtual std::vector<*cell_connection*> **connections_on** (*cell_gid_type gid*) **const**
Returns a list of all the **incoming** connections for *gid* . Each connection `con` should have post-synaptic target `con.dest.gid` that matches the argument `gid`, and a valid synapse id `con.dest.index` on *gid*. See `cell_connection`.

By default returns an empty list.

virtual std::vector<*gap_junction_connection*> **gap_junctions_on** (*cell_gid_type gid*) **const**
Returns a list of all the gap junctions connected to *gid*. Each gap junction `gj` should have one of the two gap junction sites `gj.local.gid` or `gj.peer.gid` matching the argument `gid`, and the corresponding synapse id `gj.local.index` or `gj.peer.index` should be valid on *gid*. See `gap_junction_connection`.

By default returns an empty list.

virtual std::vector<event_generator> **event_generators** (*cell_gid_type gid*) **const**
Returns a list of all the event generators that are attached to *gid*.

By default returns an empty list.

virtual *cell_size_type* **num_sources** (*cell_gid_type gid*) **const**
Returns the number of spike sources on *gid*. This corresponds to the number of spike detectors on a multi-compartment cell. Typically there is one detector at the soma of the cell, however it is possible to attache multiple detectors at arbitrary locations.

By default returns 0.

virtual *cell_size_type* **num_targets** (*cell_gid_type gid*) **const**
The number of post-synaptic sites on *gid*, which corresponds to the number of synapses.

By default returns 0.

virtual *cell_size_type* **num_probes** (*cell_gid_type gid*) **const**
The number of probes attached to the cell.

By default returns 0.

virtual *cell_size_type* **num_gap_junction_sites**(*cell_gid_type gid*) **const**
Returns the number of gap junction sites on *gid*.

By default returns 0.

virtual *probe_info* **get_probe**(*cell_member_type*) **const**
Intended for use by cell group implementations to set up sampling data structures ahead of time and for putting in place any structures or information in the concrete cell implementations to allow monitoring.

By default throws `std::logic_error`. If *num_probes()* returns a non-zero value, this must also be overridden.

virtual util::*any* **get_global_properties**(*cell_kind*) **const**
Global property type will be specific to given cell kind.

By default returns an empty container.

class cell_connection
Describes a connection between two cells: a pre-synaptic source and a post-synaptic destination. The source is typically a threshold detector on a cell or a spike source. The destination is a synapse on the post-synaptic cell.

using cell_connection_endpoint = *cell_member_type*
Connection end-points are represented by pairs (cell index, source/target index on cell).

cell_connection_endpoint **source**
Source end point.

cell_connection_endpoint **dest**
Destination end point.

float **weight**
The weight delivered to the target synapse. The weight is dimensionless, and its interpretation is specific to the synapse type of the target. For example, the *expsyn* synapse interprets it as a conductance with units μS (micro-Siemens).

float **delay**
Delay of the connection (milliseconds).

class gap_junction_connection
Describes a gap junction between two gap junction sites. Gap junction sites are represented by :cpp:type:cell_member_type.

cell_member_type **local**
gap junction site: one half of the gap junction connection.

cell_member_type **peer**
gap junction site: other half of the gap junction connection.

float **ggap**
gap junction conductance in μS.

3.20 Domain Decomposition

The C++ API for partitioning a model over distributed and local hardware is described here.

3.20.1 Load Balancers

Load balancing generates a *domain_decomposition* given an *arb::recipe* and a description of the hardware on which the model will run. Currently Arbor provides one load balancer, *partition_load_balance()*, and

more will be added over time.

If the model is distributed with MPI, the partitioning algorithm for cells is distributed with MPI communication. The returned *domain_decomposition* describes the cell groups on the local MPI rank.

Note: The *domain_decomposition* type is independent of any load balancing algorithm, so users can define a domain decomposition directly, instead of generating it with a load balancer. This is useful for cases where the provided load balancers are inadequate, or when the user has specific insight into running their model on the target computer.

Important: When users supply their own *domain_decomposition*, if they have **Gap Junction connections**, they have to be careful to place all cells that are connected via gap junctions in the same group. Example: `A -gj- B -gj- C` and `D -gj- E`. Cells A, B and C need to be in a single group; and cells D and E need to be in a single group. They may all be placed in the same group but not necessarily. Be mindful that smaller cell groups perform better on multi-core systems and try not to overcrowd cell groups if not needed. Arbor provided load balancers such as *partition_load_balance()* guarantee that this rule is obeyed.

---

domain_decomposition **partition_load_balance**(**const** *recipe* &*rec*, **const** arb::*context* &*ctx*)

Construct a *domain_decomposition* that distributes the cells in the model described by *rec* over the distributed and local hardware resources described by *ctx*.

The algorithm counts the number of each cell type in the global model, then partitions the cells of each type equally over the available nodes. If a GPU is available, and if the cell type can be run on the GPU, the cells on each node are put one large group to maximise the amount of fine grained parallelism in the cell group. Otherwise, cells are grouped into small groups that fit in cache, and can be distributed over the available cores.

Note: The partitioning assumes that all cells of the same kind have equal computational cost, hence it may not produce a balanced partition for models with cells that have a large variance in computational costs.

3.20.2 Decomposition

Documentation for the data structures used to describe domain decompositions.

enum class backend_kind

Used to indicate which hardware backend to use for running a `cell_group`.

enumerator multicore

Use multicore backend.

enumerator gpu

Use GPU back end.

Note: Setting the GPU back end is only meaningful if the `cell_group` type supports the GPU backend.

---

class domain_decomposition

Describes a domain decomposition and is solely responsible for describing the distribution of cells across cell groups and domains. It holds cell group descriptions (*groups*) for cells assigned to the local domain, and a helper function (*gid_domain*) used to look up which domain a cell has been assigned to. The *domain_decomposition* object also has meta-data about the number of cells in the global model, and the number of domains over which the model is distributed.

Note: The domain decomposition represents a division **all** of the cells in the model into non-overlapping sets, with one set of cells assigned to each domain. A domain decomposition is generated either by a load balancer or is directly specified by a user, and it is a requirement that the decomposition is correct:

- Every cell in the model appears once in one and only one cell *groups* on one and only one local *domain_decomposition* object.
- *num_local_cells* is the sum of the number of cells in each of the *groups*.
- The sum of *num_local_cells* over all domains matches *num_global_cells*.

std::function<int (*cell_gid_type*) > **gid_domain**
: A function for querying the domain id that a cell assigned to (using global identifier `gid`). It must be a pure function, that is it has no side effects, and hence is thread safe.

int **num_domains**
: Number of domains that the model is distributed over.

int **domain_id**
: The index of the local domain. Always 0 for non-distributed models, and corresponds to the MPI rank for distributed runs.

cell_size_type **num_local_cells**
: Total number of cells in the local domain.

cell_size_type **num_global_cells**
: Total number of cells in the global model (sum of *num_local_cells* over all domains).

std::vector<*group_description*> **groups**
: Descriptions of the cell groups on the local domain. See *group_description*.

class group_description
: The indexes of a set of cells of the same kind that are group together in a cell group in a *arb::simulation*.

group_description (*cell_kind* *k*, std::vector<*cell_gid_type*> *g*, *backend_kind* *b*)
: Constructor.

const *cell_kind* **kind**
: The kind of cell in the group.

const std::vector<*cell_gid_type*> **gids**
: The gids of the cells in the cell group.

const *backend_kind* **backend**
: The back end on which the cell group is to run.

3.21 Simulations

3.21.1 From recipe to simulation

To build a simulation the following concepts are needed:

- An *arb::recipe* that describes the cells and connections in the model.
- An *arb::context* used to execute the simulation.

The workflow to build a simulation is to first generate a *arb::domain_decomposition* that describes the distribution of the model over the local and distributed hardware resources (see *Domain Decomposition*), then build the simulation.

```
#include <arbor/context.hpp>
#include <arbor/domain_decomposition.hpp>
#include <arbor/simulation.hpp>

// Get a communication context
arb::context context = make_context();

// Make a recipe of user defined type my_recipe.
my_recipe recipe;

// Get a description of the partition the model over the cores
// (and gpu if available) on node.
arb::domain_decomposition decomp = arb::partition_load_balance(recipe, context);

// Instantiate the simulation.
arb::simulation sim(recipe, decomp, context);
```

3.21.2 Class Documentation

class simulation

The executable form of a model. A simulation is constructed from a recipe, and then used to update and monitor model state.

Simulations take the following inputs:

- The **constructor** takes:
  - an *arb::recipe* that describes the model;
  - an *arb::domain_decomposition* that describes how the cells in the model are assigned to hardware resources;
  - an *arb::context* which is used to execute the simulation.
- **Experimental inputs** that can change between model runs, such as external spike trains.

Simulations provide an interface for executing and interacting with the model:

- **Advance model state** from one time to another and reset model state to its original state before simulation was started.
- **I/O** interface for sampling simulation state during execution (e.g. compartment voltage and current) and spike output.

Types:

using spike_export_function = std::function<void (**const** std::vector<spike>&) >

User-supplied callback function used as a sink for spikes generated during a simulation. See *set_local_spike_callback()* and *set_global_spike_callback()*.

Constructor:

simulation (**const** *recipe* &*rec*, **const** *domain_decomposition* &*decomp*, **const** *context* &*ctx*)

Experimental inputs:

void **inject_events**(**const** pse_vector &*events*)
Add events directly to targets. Must be called before calling *run()*, and must contain events that are to be delivered at or after the current simulation time.

Updating Model State:

void **reset**()
Reset the state of the simulation to its initial state.

time_type **run**(time_type *tfinal*, time_type *dt*)
Run the simulation from current simulation time to *tfinal*, with maximum time step size *dt*.

void **set_binning_policy**(binning_kind *policy*, time_type *bin_interval*)
Set event binning policy on all our groups.

I/O:

sampler_association_handle **add_sampler**(cell_member_predicate *probe_ids*, schedule *sched*, sampler_function *f*, sampling_policy *policy* = sampling_policy::lax)
Note: sampler functions may be invoked from a different thread than that which called *run()*.

(see the *Sampling API* documentation.)

void **remove_sampler**(sampler_association_handle)
Remove a sampler. (see the *Sampling API* documentation.)

void **remove_all_samplers**()
Remove all samplers from probes. (see the *Sampling API* documentation.)

std::size_t **num_spikes**() **const**
The total number of spikes generated since either construction or the last call to *reset()*.

void **set_global_spike_callback**(*spike_export_function export_callback*)
Register a callback that will periodically be passed a vector with all of the spikes generated over all domains (the global spike vector) since the last call. Will be called on the MPI rank/domain with id 0.

void **set_local_spike_callback**(*spike_export_function export_callback*)
Register a callback that will periodically be passed a vector with all of the spikes generated on the local domain (the local spike vector) since the last call. Will be called on each MPI rank/domain with a copy of the local spikes.

3.22 Cable cells

Warning: The interface for building and modifying cable cell objects will be thoroughly revised in the near future. The documentation here is primarily a place holder.

Cable cells, which use the `cell_kind cable`, represent morphologically-detailed neurons as 1-d trees, with electrical and biophysical properties mapped onto those trees.

A single cell is represented by an object of type `cable_cell`. Properties shared by all cable cells, as returned by the recipe `get_global_properties` method, are described by an object of type *cable_cell_global_properties*.

3.22.1 The `cable_cell` object

Cable cells are built up from a series of `segment` objects, which themselves describe an unbranched component of the cell morphology. These segments are added via the methods:

soma_segment *`cable_cell`**`::add_soma`**(double *radius*)
: Add the soma to the cable cell with the given radius. There can be only one per cell.

  The soma segment has index 0, and must be added before any cable segments.

cable_segment *`cable_cell`**`::add_cable`**(cell_lid_type *index*, Args&&... *args*)
: Add a unbranched section of the cell morphology, with its proximal end attached to the segment given by *`index`*. The following arguments are forwarded to the `cable_segment` constructor.

Segment indices are exactly the order in which they have been added to a cell, counting from zero (for the soma). Both `soma_segment` and `cable_segment` are derived from the abstract base class `segment`.

Todo: Describe cable_segment constructor arguments, unless we get to the replace cell building/morphology implementation first.

Each segment will inherit the electrical properties of the cell, unless otherwise overriden (see below).

3.22.2 Cell dynamics

Each segment in a cell may have attached to it one or more density *mechanisms*, which describe biophysical processes. These are processes that are distributed in space, but whose behaviour is defined purely by the state of the cell and the process at any given point.

Cells may also have *point* mechanisms, which are added directly to the `cable_cell` object.

A third type of mechanism, which describes ionic reversal potential behaviour, can be specified for cells or the whole model via cell parameter settings, described below.

Mechanisms are described by a `mechanism_desc` object. These specify the name of the mechanism (used to find the mechanism in the mechanism catalogue) and parameter values for the mechanism that apply within a segment. A `mechanism_desc` is effectively a wrapper around a name and a dictionary of parameter/value settings.

Mechanism descriptions can be constructed implicitly from the mechanism name, and mechanism parameter values then set with the `set` method. Relevant `mechanism_desc` methods:

`mechanism_desc``::`**`mechanism_desc`**(std::string *name*)
: Construct a mechanism description for the mechanism named *name*.

mechanism_desc &`mechanism_desc`**`::set`**(**`const`** std::string &*key*, double *value*)
: Sets the parameter associated with *`key`* in the description. Returns a reference to the mechanism description, so that calls to *`set`* can be chained in a single expression.

Density mechanisms are associated with a cable cell object with:

void `segment`**`::add_mechanism`**(mechanism_desc *mech*)

Point mechanisms, which are associated with connection end points on a cable cell, are attached to a cell with:

void `cable_cell`**`::add_synapse`**(segment_location *loc*, mechanism_desc *mech*)

where `segment_location` is a simple struct holding a segment index and a relative position (from 0, proximal, to 1, distal) along that segment:

`segment_location``::`**`segment_location`**(cell_lid_type *index*, double *position*)

3.22.3 Electrical properities and ion values

On each cell segment, electrical and ion properties can be specified by the `parameters` field, of type *`cable_cell_local_parameter_set`*.

The *`cable_cell_local_parameter_set`* has the following members, where an empty optional value or missing map key indicates that the corresponding value should be taken from the cell or global parameter set.

class cable_cell_local_parameter_set

std::unordered_map<std::string, cable_cell_ion_data> **ion_data**

The keys of this map are names of ions, whose parameters will be locally overriden. The struct `cable_cell_ion_data` has three fields: `init_int_concentration`, `init_ext_concentration`, and `init_reversal_potential`.

Internal and external concentrations are given in millimolars, i.e. mol/m^3. Reversal potential is given in millivolts.

util::optional<double> **init_membrane_potential**

Initial membrane potential in millivolts.

util::optional<double> **temperature_K**

Local temperature in Kelvin.

util::optional<double> **axial_resistivity**

Local resistivity of the intracellular medium, in ohm-centimetres.

util::optional<double> **membrane_capacitance**

Local areal capacitance of the cell membrane, in Farads per square metre.

Default parameters for a cell are given by the `default_parameters` field in the `cable_cell` object. This is a value of type *`cable_cell_parameter_set`*, which extends *`cable_cell_local_parameter_set`* by adding an additional field describing reversal potential computation:

class cable_cell_parameter_set : **public** *cable_cell_local_parameter_set*

std::unordered_map<std::string, mechanism_desc> **reversal_potential_method**

Maps the name of an ion to a 'reversal potential' mechanism that describes how it should be computed. When no mechanism is provided for an ionic reversal potential, the reversal potential will be kept at its initial value.

Default parameters for all cells are supplied in the *`cable_cell_global_properties`* struct.

3.22.4 Global properties

class cable_cell_global_properties

const mechanism_catalogue ***catalogue**

All mechanism names refer to mechanism instances in this mechanism catalogue. By default, this is set to point to *global_default_catalogue()*, the catalogue that contains all mechanisms bundled with Arbor.

double **membrane_voltage_limit_mV**

If non-zero, check to see if the membrane voltage ever exceeds this value in magnitude during the course of a simulation. If so, throw an exception and abort the simulation.

bool **coalesce_synapses**

When synapse dynamics are sufficiently simple, the states of synapses within the same discretized element can be combined for better performance. This is true by default.

std::unordered_map<std::string, int> **ion_species**

Every ion species used by cable cells in the simulation must have an entry in this map, which takes an ion name to its charge, expressed as a multiple of the elementary charge. By default, it is set to include sodium "na" with charge 1, calcium "ca" with charge 2, and potassium "k" with charge 1.

cable_cell_parameter_set **default_parameters**

The default electrical and physical properties associated with each cable cell, unless overridden locally. In the global properties, *every optional field must be given a value*, and every ion must have its default values set in `default_parameters.ion_data`.

add_ion(**const** std::string &*ion_name*, int *charge*, double *init_iconc*, double *init_econc*, double *init_revpot*)

Convenience function for adding a new ion to the global `ion_species` table, and setting up its default values in the *ion_data* table.

add_ion(**const** std::string &*ion_name*, int *charge*, double *init_iconc*, double *init_econc*, mechanism_desc *revpot_mechanism*)

As above, but set the initial reversal potential to zero, and use the given mechanism for reversal potential calculation.

For convenience, `neuron_parameter_defaults` is a predefined `cable_cell_local_parameter_set` value that holds values that correspond to NEURON defaults. To use these values, assign them to the `default_parameters` field of the global properties object returned in the recipe.

3.22.5 Reversal potential dynamics

If no reversal potential mechanism is specified for an ion species, the initial reversal potential values are maintained for the course of a simulation. Otherwise, a provided mechanism does the work, but it subject to some strict restrictions. A reversal potential mechanism described in NMODL:

- May not maintain any STATE variables.
- Can only write to the "eX" value associated with an ion.
- Can not given as a POINT mechanism.

Essentially, reversal potential mechanisms must be pure functions of cellular and ionic state.

If a reversal potential mechanism writes to multiple ions, then if the mechanism is given for one of the ions in the global or per-cell parameters, it must be given for all of them.

Arbor's default catalogue includes a "nernst" reversal potential, which is parameterized over a single ion, and so can be assigned to e.g. calcium in the global parameters via

```
cable_cell_global_properties gprop;
// ...
gprop.default_parameters.reversal_potential_method["ca"] = "nernst/ca";
```

This mechanism has global scalar parameters for the gas constant R and Faraday constant F, corresponding to the exact values given by the 2019 redifinition of the SI base units. These values can be changed in a derived mechanism in order to use, for example, older values of these physical constants.

```
mechanism_catalogue mycat(global_default_catalogue());
mycat.derive("nernst1998", "nernst", {{"R", 8.314472}, {"F", 96485.3415}});

gprop.catalogue = &mycat;
gprop.default_parameters.reversal_potential_method["ca"] = "nernst1998/ca";
```

3.23 Library Reference

Low level library reference material goes here, e.g. *range* library documentation.

3.24 SIMD Classes

The purpose of the SIMD classes is to abstract and consolidate the use of compiler intrinsics for the manipulation of architecture-specific vector (SIMD) values.

The implementation is rather loosely based on the data-parallel vector types proposal P0214R6 for the C++ Parallelism TS 2.

Unless otherwise specified, all classes, namespaces and top-level functions described below are all within the top-level *arb::simd* namespace.

Example usage

The following code performs an element-wise vector product, storing only non-zero values in the resultant array.

```
#include <simd/simd.hpp>
using namespace arb::simd;

void product_nonzero(int n, const double* a, const double* b, double* result) {
    constexpr int N = simd_abi::native_width<double>::value;
    using simd = simd<double, N>;
    using mask = simd::simd_mask;

    int i = 0;
    for (; i+N<=n; i+=N) {
        auto vp = simd(a+i)*simd(b+i);
        where(vp!=0, vp).copy_to(result+i);
    }

    int tail = n-i;
    auto m = mask::unpack((1<<tail)-1);

    auto vp = simd(a+i, m)*simd(b+i, m);
    where(m && vp!=0, vp).copy_to(c+i);
}
```

3.24.1 Classes

Three user-facing template classes are provided:

1. `simd<V, N, I = simd_abi::default_abi>`

   N-wide vector type of values of type *V*, using architecture-specific implementation *I*. The implementation parameter is itself a template, acting as a type-map, with `I<V, N>::type` being the concrete implementation class (see below) for *N*-wide vectors of type *V* for this architecture.

   The implementation `simd_abi::generic` provides a `std::array`-backed implementation for arbitrary *V* and *N*, while `simd_abi::native` maps to the native architecture implementation for *V* and *N*, if one is available for the target architecture.

   `simd_abi::default_abi` will use `simd_abi::native` if available, or else fall back to the generic implementation.

2. `simd_mask<V, N, I = simd_api::default_abi>`

   The result of performing a lane-wise comparison/test operation on a `simd<V, N, I>` vector value. `simd_mask` objects support logical operations and are used as arguments to `where` expressions.

   `simd_mask<V, N, I>` is a type alias for `simd<V, N, I>::simd_mask`.

3. `where_expression<simd<V, N, I>>`

   The result of a `where` expression, used for masked assignment.

There is, in addition, a templated class `detail::indirect_expression` that holds the result of an *indirect(...)* expression. These arise in gather and scatter operations, and are detailed below.

Implementation typemaps live in the `simd_abi` namespace, while concrete implementation classes live in `detail`. A particular specialization for an architecture, for example 4-wide double on AVX, then requires:

- A concrete implementation class, e.g. `detail::avx_double4`.
- A specialization of its ABI map, so that `simd_abi::avx<double, 4>::type` is an alias for `detail::avx_double4`.
- A specialization of the native ABI map, so that `simd_abi::native<double, 4>::type` is an alias for `simd_abi::avx<double, 4>::type`.

The maximum natively supported width for a scalar type *V* is recorded in `simd_abi::native_width<V>::value`.

Indirect expressions

An expression of the form `indirect(p, k)` or `indirect(p, k, constraint)` describes a sequence of memory locations based at the pointer *p* with offsets given by the `simd` variable *k*. A constraint of type `index_constraint` can be provided, which promises certain guarantees on the index values in *k*:

Constraint	Guarantee
`index_constraint::none`	No restrictions.
`index_constraint::independent`	No indices are repeated, i.e. $k_i = k_j$ implies $i = j$.
`index_constraint::contiguous`	Indices are sequential, i.e. $k_i = k_0 + i$.
`index_constraint::constant`	Indices are all equal, i.e. $k_i = k_j$ for all i and j.

Class `simd`

The class `simd<V, N, I>` is an alias for `detail::simd_impl<I<V, N>::type>`; the class `detail::simd_impl<C>` provides the public interface and arithmetic operators for a concrete implementation class *C*.

In the following:

- *S* stands for the class `simd<V, N, I>`.
- *s* is a SIMD value of type *S*.
- *m* is a mask value of type `S::simd_mask`.
- *t*, *u* and *v* are const objects of type *S*.
- *w* is a SIMD value of type `simd<W, N, J>`.
- *i* is an index of type `int`.
- *j* is a const object of type `simd<U, N, J>` where *U* is an integral type.
- *x* is a value of type *V*.
- *p* is a pointer to *V*.
- *c* is a const pointer to *V* or a length *N* array of *V*.

Here and below, the value in lane *i* of a SIMD vector or mask *v* is denoted by v_i

Type aliases and constexpr members

Name	Type	Description
`S::scalar_type`	*V*	The type of one lane of the SIMD type.
`S::simd_mask`	`simd_mask<V, N, I>`	The `simd_mask` specialization resulting from comparisons of *S* SIMD values.
`S::width`	`unsigned`	The SIMD width *N*.

Constructors

Expression	Description
`S(x)`	A SIMD value *v* with v_i equal to *x* for $i = 0 \ldots N\text{-}1$.
`S(t)`	A copy of the SIMD value *t*.
`S(c)`	A SIMD value *v* with v_i equal to `c[i]` for $i = 0 \ldots N\text{-}1$.
`S(w)`	A copy or value-cast of the SIMD value *w* of a different type but same width.
`S(indirect(p, j))`	A SIMD value *v* with v_i equal to `p[j[i]]` for $i = 0 \ldots N\text{-}1$.
`S(c, m)`	A SIMD value *v* with v_i equal to `c[i]` for *i* where m_i is true.

Member functions

Expression	Type	Description
`t.copy_to(p)`	`void`	Set `p[i]` to t_i for $i = 0 \ldots N$-1.
`t.copy_to(indirect(p, j))`	`void`	Set `p[j[i]]` to t_i for $i = 0 \ldots N$-1.
`s.copy_from(c)`	`void`	Set s_i to `c[i]` for $i = 0 \ldots N$-1.
`s.copy_from(indirect(c, j))`	`void`	Set s_i to `c[j[i]]` for $i = 0 \ldots N$-1.
`s.sum()`	`V`	Sum of s_i for $i = 0 \ldots N$-1.

Expressions

Expression	Type	Description
`t+u`	`S`	Lane-wise sum.
`t-u`	`S`	Lane-wise difference.
`t*u`	`S`	Lane-wise product.
`t/u`	`S`	Lane-wise quotient.
`fma(t, u, v)`	`S`	Lane-wise FMA $t * u + v$.
`s<t`	`S::simd_mask`	Lane-wise less-than comparison.
`s<=t`	`S::simd_mask`	Lane-wise less-than-or-equals comparison.
`s>t`	`S::simd_mask`	Lane-wise greater-than comparison.
`s>=t`	`S::simd_mask`	Lane-wise greater-than-or-equals comparison.
`s==t`	`S::simd_mask`	Lane-wise equality test.
`s!=t`	`S::simd_mask`	Lane-wise inequality test.
`s=t`	`S&`	Lane-wise assignment.
`s+=t`	`S&`	Equivalent to `s=s+t`.
`s-=t`	`S&`	Equivalent to `s=s-t`.
`s*=t`	`S&`	Equivalent to `s=s*t`.
`s/=t`	`S&`	Equivalent to `s=s/t`.
`s=x`	`S&`	Equivalent to `s=S(x)`.
`indirect(p, j)=t`	`decltype(indirect(p, j))&`	Equivalent to `t.copy_to(indirect(p, j))`.
`indirect(p, j)+=t`	`decltype(indirect(p, j))&`	Compound indirect assignment: `p[j[i]]+=t[i]` for $i = 0 \ldots N$-1.
`indirect(p, j)-=t`	`decltype(indirect(p, j))&`	Compound indirect assignment: `p[j[i]]-=t[i]` for $i = 0 \ldots N$-1.
`t[i]`	`V`	Value t_i
`s[i]=x`	`S::reference`	Set value s_i to x.

The (non-const) index operator `operator[]` returns a proxy object of type `S::reference`, which writes the corresponding lane in the SIMD value on assignment, and has an implicit conversion to `scalar_type`.

Class `simd_mask`

`simd_mask<V, N, I>` is an alias for `simd<V, N, I>::simd_mask`, which in turn will be an alias for a class `detail::simd_mask_impl<D>`, where D is a concrete implementation class for the SIMD mask repre-

sentation. `simd_mask_impl<D>` inherits from, and is implemented in terms of, `detail::simd_impl<D>`, but note that the concrete implementation class *D* may or may not be the same as the concrete implementation class `I<V, N>::type` used by `simd<V, N, I>`.

Mask values are read and written as `bool` values of 0 or 1, which may differ from the internal representation in each lane of the SIMD implementation.

In the following:

- *M* stands for the class `simd_mask<V, N, I>`.
- *m* and *q* are const objects of type `simd_mask<V, N, I>`.
- *u* is an object of type `simd_mask<V, N, I>`.
- *b* is a boolean value.
- *q* is a pointer to `bool`.
- *y* is a const pointer to `bool` or a length *N* array of `bool`.
- *i* is of type `int`.
- *k* is of type `unsigned long long`.

Constructors

Expression	Description
`M(b)`	A SIMD mask *u* with u_i equal to *b* for $i = 0 \ldots N\text{-}1$.
`M(m)`	A copy of the SIMD mask *m*.
`M(y)`	A SIMD value *u* with u_i equal to `y[i]` for $i = 0 \ldots N\text{-}1$.

Note that `simd_mask` does not (currently) offer a masked pointer/array constructor.

Member functions

Expression	Type	Description
`m.copy_to(q)`	`void`	Write the boolean value m_i to `q[i]` for $i = 0 \ldots N\text{-}1$.
`u.copy_from(y)`	`void`	Set u_i to the boolean value `y[i]` for $i = 0 \ldots N\text{-}1$.

Expressions

Expression	Type	Description
`!m`	`M`	Lane-wise negation.
`m&&q`	`M`	Lane-wise logical and.
`m||q`	`M`	Lane-wise logical or.
`m==q`	`M`	Lane-wise equality (equivalent to `m!=!q`).
`m!=q`	`M`	Lane-wise logical xor.
`m=q`	`M&`	Lane-wise assignment.
`m[i]`	`bool`	Boolean value m_i.
`m[i]=b`	`M::reference`	Set m_i to boolean value *b*.

Static member functions

Expression	Type	Description
`M::unpack(k)`	`M`	Mask with value m_i equal to the ith bit of k.

Class `where_expression`

`where_expression<S>` represents a masked subset of the lanes of a SIMD value of type `S`, used for conditional assignment, masked scatter, and masked gather. It is a type alias for `S::where_expression`, and is the result of an expression of the form `where(mask, simdvalue)`.

In the following:

- *W* stands for the class `where_expression<simd<V, N, I>>`.
- *s* is a reference to a SIMD value of type `simd<V, N, I>&`.
- *t* is a SIMD value of type `simd<V, N, I>`.
- *m* is a mask of type `simd<V, N, I>::simd_mask`.
- *j* is a const object of type `simd<U, N, J>` where *U* is an integral type.
- *x* is a scalar of type *V*.
- *p* is a pointer to *V*.
- *c* is a const pointer to *V* or a length *N* array of *V*.

Expression	Type	Description
`where(m, s)`	`W`	A proxy for masked-assignment operations.
`where(m, s)=t`	`void`	Set s_i to t_i for i where m_i is true.
`where(m, s)=x`	`void`	Set s_i to x for i where m_i is true.
`where(m, s).copy_to(p)`	`void`	Set `p[i]` to s_i for i where m_i is true.
`where(m, s).copy_to(indirect(p, j))`	`void`	Set `p[j[i]]` to s_i for i where m_i is true.
`where(m, s).copy_from(c)`	`void`	Set s_i to `c[i]` for i where m_i is true.
`where(m, s).copy_from(indirect(c, j))`	`void`	Set s_i to `c[j[i]]` for i where m_i is true.

3.24.2 Top-level functions

Lane-wise mathematical operations *abs(x)*, *min(x, y)* and *max(x, y)* are offered for all SIMD value types, while the transcendental functions are only usable for SIMD floating point types.

Vectorized implementations of some of the transcendental functions are provided: refer to the *vector transcendental functions documentation* for details.

In the following:

- *I* and *J* are SIMD implementations.
- *A* is a SIMD class `simd<K, N, I>` for some scalar type *K*.

- *S* is a SIMD class `simd<V, N, I>` for a floating point type *V*.
- *L* is a scalar type implicitly convertible from *K*.
- *a* and *b* are values of type *A*.
- *s* and *t* are values of type *S*.
- *r* is a value of type `std::array<K, N>`.

Expression	Type	Description
`abs(a)`	*A*	Lane-wise absolute value of *a*.
`min(a, b)`	*A*	Lane-wise minimum of *a* and *b*.
`max(a, b)`	*A*	Lane-wise maximum of *a* and *b*.
`sin(s)`	*S*	Lane-wise sine of *s*.
`cos(s)`	*S*	Lane-wise cosine of *s*.
`log(s)`	*S*	Lane-wise natural logarithm of *s*.
`exp(s)`	*S*	Lane-wise exponential of *s*.
`expm1(s)`	*S*	Lane-wise $x \mapsto e^x - 1$.
`exprelr(s)`	*S*	Lane-wise $x \mapsto x/(e^x - 1)$.
`pow(s, t)`	*S*	Lane-wise raise *s* to the power of *t*.
`simd_cast<std::array<L, N>>(a)`	`std::array<L, N>`	Lane-wise cast of values in *a* to scalar type *L* in `std::array<L, N>`.
`simd_cast<simd<L, N, J>>(a)`	`simd<L, N, J>`	Lane-wise cast of values in *a* to scalar type *L* in `simd<L, N, J>`.
`simd_cast<simd<L, N, J>>(r)`	`simd<L, N, J>`	Lane-wise cast of values in the `std::array<K, N>` value *r* to scalar type *L* in `simd<L, N, J>`.

3.24.3 Implementation requirements

Each specific architecture is represented by a templated class *I*, with `I<V, N>::type` being the concrete implementation for an *N*-wide SIMD value with `scalar_type` *V*.

A concrete implementation class *C* inherits from `detail::implbase<C>`, which provides (via CRTP) generic implementations of most of the SIMD functionality. The base class `implbase<C>` in turn relies upon `detail::simd_traits<C>` to look up the SIMD width, and associated types.

All the required SIMD operations are given by static member functions of *C*.

Some arguments to static member functions use a tag class (`detail::tag`) parameterized on a concrete implementation class for dispatch purposes.

Minimal implementation

In the following, let *C* be the concrete implementation class for a *N*-wide vector of scalar_type *V*, with low-level representation `archvec`.

The specialization of `detail::simd_traits<C>` then exposes these types and values, and also provides the concrete implementation class *M* for masks associated with *C*:

```
template <>
struct simd_traits<C> {
    static constexpr unsigned width = N;
    using scalar_type = V;
    using vector_type = archvec;
```

(continues on next page)

(continued from previous page)

```
    using mask_impl = M;
};
```

The mask implementation class *M* may or may not be the same as *C*. For example, `detail::avx_double4` provides both the arithmetic operations and mask operations for an AVX 4 × double SIMD vector, while the mask implementation for `detail::avx512_double8` is `detail::avx512_mask8`.

The concrete implementation class must provide at minimum implementations of `copy_to` and `copy_from` (see the section below for semantics):

```
struct C: implbase<C> {
    static void copy_to(const arch_vector&, V*);
    static arch_vector copy_from(const V*);
};
```

If the implementation is also acting as a mask implementation, it must also provide `mask_copy_to`, `mask_copy_from`, `mask_element` and `mask_set_element`:

```
struct C: implbase<C> {
    static void copy_to(const arch_vector&, V*);
    static arch_vector copy_from(const V*);

    static void mask_copy_to(const arch_vector& v, bool* w);
    static arch_vector mask_copy_from(const bool* y);
    static bool mask_element(const arch_vector& v, int i);
    static void mask_set_element(arch_vector& v, int i, bool x);
};
```

The `simd_detial::generic<T, N>` provides an example of a minimal implementation based on an `arch_vector` type of `std::array<T, N>`.

Concrete implementation API

In the following, *C* represents the concrete implementation class for a SIMD class of width *N* and value type *V*.

- *u*, *v*, and *w* are values of type `C::vector_type`.
- *r* is a reference of type `C::vector_type&`.
- *x* is a value of type `C::scalar_type`.
- *c* is a const pointer of type `const C::scalar_type*`.
- *p* is a pointer of type `C::scalar_type*`.
- *j* is a SIMD index representation of type `J::vector_type` for an integral concrete implementation class *J*.
- *d* is a SIMD representation of type `D::vector_type` for a (different) concrete implementation class *D*.
- *b* is a `bool` value.
- *q* is a pointer to `bool`.
- *y* is a const pointer to `bool`.
- *i* is an unsigned (index) value.
- *k* is an unsigned long long value.
- *m* is a mask representation of type `C::mask_type`.

- z is an `index_constraint` value.

Types and constants

Name	Type	Description
`C::vector_type`	`simd_traits<C>::vector_type`	Underlying SIMD representation type.
`C::scalar_type`	`simd_traits<C>::scalar_type`	Should be convertible to/from V.
`C::mask_impl`	`simd_traits<C>::mask_impl`	Concrete implementation class for mask SIMD type.
`C::mask_type`	`C::mask_impl::vector_type`	Underlying SIMD representation for masks.
`C::width`	`unsigned`	The SIMD width N.

Initialization, load, store

Expression	Type	Description
`C::cast_from(tag<D>, d)`	`C::vector_type`	Return a vector with values $v_i = d_i$, where `D::scalar_type` is implicitly convertible to `C::scalar_type`.
`C::broadcast(x)`	`C::vector_type`	Fill representation with scalar x.
`C::copy_to(v, p)`	`void`	Store values v_i to $p+i$. p may be unaligned.
`C::copy_to_masked(v, p, m)`	`void`	Store values v_i to $p+i$ wherever m_i is true. p may be unaligned.
`C::copy_from(c)`	`C::vector_type`	Return a vector with values v_i loaded from $c+i$. c may be unaligned.
`C::copy_from_masked(c, m)`	`C::vector_type`	Return a vector with values v_i loaded from $c+i$ wherever m_i is true. c may be unaligned.
`C::copy_from_masked(u, c, m)`	`void`	Return a vector with values v_i loaded from $c+i$ wherever m_i is true, or equal to u_i otherwise. c may be unaligned.

Lane access

Expression	Type	Description
`C::element(v, i)`	`C::scalar_type`	Value in ith lane of v.
`C::set_element(r, i, x)`	`void`	Set value in lane i of r to x.

Gather and scatter

The offsets for gather and scatter operations are given by a vector type `J::vector_type` for some possibly different concrete implementation class J, and the static methods implementing gather and scatter are templated on this class.

Implementations can provide optimized versions for specific index classes J; this process would be simplified with more support for casts between SIMD types and their concrete implementations, functionality which is not yet provided.

The first argument to these functions is a dummy argument of type J, used only to disambiguate overloads.

Expression	Type	Description
`C::gather(tag<J>(), p, j)`	`C::vector_type`	Vector v with values v_i = `p[j[i]]`.
`C::gather(tag<J>(), u, p, j, m)`	`C::vector_type`	Vector v with values $v_i = m_i$? `p[j[i]]` : u_i.
`C::scatter(tag<J>(), u, p, j)`	`void`	Write values u_i to `p[j[i]]`.
`C::scatter(tag<J>(), u, p, j, m)`	`void`	Write values u_i to `p[j[i]]` for lanes i where m_i is true.
`C::compound_indexed_add(tag<J>(), u, p, j, z)`	`void`	Update values `p[j[i]] += u[i]` for lanes i, subject to constraint z.

Casting

Implementations can provide optimized versions of lane-wise value casting from other specific implementation classes. The first argument is a dummy argument of type J, used only to disambiguate overloads.

Expression	Type	Description
`C::cast_from(tag<J>(), d)`	`C::vector_type`	Returns vector v with values $v_i = d_i$, cast from `D::scalar_type` to `C::scalar_type`.

Arithmetic operations

Expression	Type	Description
`C::negate(v)`	`C::vector_type`	Lane-wise unary minus.
`C::mul(u, v)`	`C::vector_type`	Lane-wise multiplication.
`C::add(u, v)`	`C::vector_type`	Lane-wise addition.
`C::sub(u, v)`	`C::vector_type`	Lane-wise subtraction.
`C::div(u, v)`	`C::vector_type`	Lane-wise division.
`C::fma(u, v, w)`	`C::vector_type`	Lane-wise fused multiply-add (u*v+w).
`C::reduce_add(u)`	`C::scalar_type`	(Horizontal) sum of values u_i in each lane.

Comparison and blends

Expression	Type	Description
`C::cmp_eq(u, v)`	`C::mask_type`	Lane-wise $u = v$.
`C::cmp_neq(u, v)`	`C::mask_type`	Lane-wise $u \neq v$.
`C::cmp_gt(u, v)`	`C::mask_type`	Lane-wise $u > v$.
`C::cmp_geq(u, v)`	`C::mask_type`	Lane-wise $u \geq v$.
`C::cmp_lt(u, v)`	`C::mask_type`	Lane-wise $u < v$.
`C::cmp_leq(u, v)`	`C::mask_type`	Lane-wise $u \leq v$.
`C::ifelse(m, u, v)`	`C::vector_type`	Vector w with values $w_i = m_i \ ? \ u_i : v_i$.

Mathematical function support.

With the exception of `abs`, `min` and `max`, these are only required for floating point vector implementations.

Expression	Type	Description
`C::abs(v)`	`C::vector_type`	Lane-wise absolute value.
`C::min(u, v)`	`C::vector_type`	Lane-wise minimum.
`C::max(u, v)`	`C::vector_type`	Lane-wise maximum.
`C::sin(v)`	`C::vector_type`	Lane-wise sine.
`C::cos(v)`	`C::vector_type`	Lane-wise cosine.
`C::log(v)`	`C::vector_type`	Lane-wise natural logarithm.
`C::exp(v)`	`C::vector_type`	Lane-wise exponential.
`C::expm1(v)`	`C::vector_type`	Lane-wise $x \mapsto e^x - 1$.
`C::exprelr(v)`	`C::vector_type`	Lane-wise $x \mapsto x/(e^x - 1)$.
`C::pow(u, v)`	`C::vector_type`	Lane-wise u raised to the power of v.

Mask value support

Mask operations are only required if C constitutes the implementation of a SIMD mask class.

Expression	Type	Description
`C::mask_broadcast(b)`	`C::vector_type`	Fill mask representation with bool b.
`C::mask_element(v, i)`	`bool`	Mask value v_i.
`C::mask_set_element(u, i, b)`	`void`	Set mask value u_i to b.
`C::mask_copy_to(w, q)`	`void`	Write bool values to memory (unaligned).
`C::mask_copy_from(y)`	`C::vector_type`	Load bool values from memory (unaligned).
`C::mask_unpack(k)`	`C::vector_type`	Return vector v with boolean value v_i equal to the ith bit of k.

Logical operations

Logical operations are only required if *C* constitutes the implementation of a SIMD mask class.

Expression	Type	Description
`C::logical_not(u)`	`C::vector_type`	Lane-wise negation.
`C::logical_and(u, v)`	`C::vector_type`	Lane-wise logical and.
`C::logical_or(u, v)`	`C::vector_type`	Lane-wise logical or.
`C::select(m, v, w)`	`C::vector_type`	Lane-wise *m*? *v*: *w*.

3.24.4 Missing functionality

There is no support yet for the following features, although some of these will need to be provided in order to improve the efficiency of SIMD versions of our generated mechanisms.

- Contraint-based dispatch for indirect operations other than += and −=.
- Vectorizable implementations of trigonometric functions.

3.24.5 Implementation of vector transcendental functions

When building with the Intel C++ compiler, transcendental functions on SIMD values in `simd<double, 8, detail::avx512>` wrap calls to the Intel scalar vector mathematics library (SVML).

Outside of this case, the functions *exp*, *log*, *expm1* and *exprelr* use explicit approximations as detailed below. The algortihms follow those used in the Cephes library, with some accommodations.

Exponentials

$\exp(x)$

The exponential is computed as

$$e^x = 2^n e^g,$$

with $|g| 0.5$ and n an integer. The power of two is computed via direct manipulation of the exponent bits of the floating point representation, while e^g is approximated by a rational polynomial.

n and g are computed by:

$$n = \left\lfloor \frac{x}{\log 2} + 0.5 \right\rfloor$$
$$g = x - n \log 2$$

where the subtraction in the calculation of g is performed in two stages, to limit cancellation error:

$$g \leftarrow \mathrm{fl}(x - nc_1)$$
$$g \leftarrow \mathrm{fl}(g - nc_2)$$

where $c_1 + c_2 = \log 2$, c_1 comprising the first 32 bits of the mantissa. (In principle c_1 might contain more bits of the logarithm, but this particular decomposition matches that used in the Cephes library.) This decomposition gives $|g| \leq \frac{1}{2}\log 2 \approx 0.347$.

The rational approximation for e^g is of the form

$$e^g \approx \frac{R(g)}{R(-g)}$$

where $R(g)$ is a polynomial of order 6. The coefficients are again those used by Cephes, and probably are derived via a Remez algorithm. $R(g)$ is decomposed into even and odd terms

$$R(g) = Q(x^2) + xP(x^2)$$

so that the ratio can be calculated by:

$$e^g \approx 1 + \frac{2gP(g^2)}{Q(g^2) - gP(g^2)}.$$

Randomized testing indicates the approximation is accurate to 2 ulp.

$\mathrm{expm1}(x)$

A similar decomposition of $x = g + n\log 2$ is performed so that $g0.5$, with the exception that n is always taken to be zero for $|x|0.5$, i.e.

$$n = \begin{cases} 0 & \text{if } |x|0.5, \\ \left\lfloor \frac{x}{\log 2} + 0.5 \right\rfloor & \text{otherwise.} \end{cases}$$

$\mathrm{expm1}(x)$ is then computed as

$$e^x - 1 = 2^n(e^g - 1) + (2^n - 1).$$

and the same rational polynomial is used to approximate $e^g - 1$,

$$e^g - 1 \approx \frac{2gP(g^2)}{Q(g^2) - gP(g^2)}.$$

The scaling by step for $n0$ is in practice calculated as

$$e^x - 1 = 2(2^{n-1}(e^g - 1) + (2^{n-1} - 0.5)).$$

in order to avoid overflow at the upper end of the range.

The order 6 rational polynomial approximation for small x is insufficiently accurate to maintain 1 ulp accuracy; randomized testing indicates a maximum error of up to 3 ulp.

$\mathrm{exprelr}(x)$

The function is defined as

$$\mathrm{exprelr}(x) = x/(e^x - 1),$$

and is the reciprocal of the relative exponential function,

$$\begin{aligned} \mathrm{exprel}(x) &= {}_1F_1(1; 2; x) \\ &= \frac{e^x - 1}{x}. \end{aligned}$$

This is computed in terms of expm1 by:

$$\operatorname{exprelr}(x) := \begin{cases} 1 & \text{if } \mathrm{fl}(1+x) = 1, \\ x/\operatorname{expm1}(x) & \text{otherwise.} \end{cases}$$

With the approximation for $\operatorname{expm1}$ used above, randomized testing demonstrates a maximum error on the order of 4 ulp.

Logarithms

The natural logarithm is computed as

$$\log x = \log u + nlog2$$

where n is an integer and u is in the interval $[\frac{1}{2}\sqrt{2}, \sqrt{2}]$. The logarithm of u is then approximated by the rational polynomial used in the Cephes implementation,

$$\log u \approx R(u-1)$$
$$R(z) = z - \frac{z^2}{2} + z^3 \frac{P(z)}{Q(z)},$$

where P and Q are polynomials of degree 5, with Q monic.

Cancellation error is minimized by computing the sum for $\log x$ as:

$$s \leftarrow \mathrm{fl}(z^3 P(z)/Q(z))$$
$$s \leftarrow \mathrm{fl}(s + nc_4)$$
$$s \leftarrow \mathrm{fl}(s - 0.5z^2)$$
$$s \leftarrow \mathrm{fl}(s + z)$$
$$s \leftarrow \mathrm{fl}(s + nc_3)$$

where $z = u - 1$ and $c_3 + c_4 = \log 2$, c_3 comprising the first 9 bits of the mantissa.

3.25 Profiler

The Arbor library has a built-in profiler for fine-grained timings of regions of interest in the code. The time stepping code in `arb::simulation` has been instrumented, so by enabling profiler support at compile time, users of the library can generate profile reports from calls to `arb::simulation::run()`.

3.25.1 Compilation

There are some non-trivial overheads associated with using the profiler, so it is not enabled by default. The profiler can be enabled at compile time, by setting the CMake flag `ARB_WITH_PROFILING`. For example to compile a debug build with profiling turned on:

```
cmake .. -DARB_WITH_PROFILING=ON
```

3.25.2 Instrumenting Code

Developers manually instrument the regions to profile. This allows the developer to only profile the parts of the code that are of interest, and choose the appropriate granularity for profiling different regions.

Once a region of code is marked for the profiler, each thread in the application will track the total time spent in the region, and how many times the region is executed on that thread.

Marking Regions

To instrument a region, use `PE` (profiler enter) and `PL` (profiler leave) macros to mark the beginning and end of a region. For example, the following will record the call count and total time spent in the `foo()` and `bar()` function calls:

```
while (t<tfinal) {
    PE(foo);
    foo();
    PL();

    PE(bar);
    bar();
    PL();

    t += dt;
}
```

It is not permitted to nest regions if a profiling region is entered while recording. For example, a `std::runtime_error` would be thrown if the call to `foo()` in the above example attempted to enter a region:

```
void foo() {
    PE(open); // error: already in the "foo" profiling region in the main time loop
    foo_open();
    PL();

    write();
}
```

Whenever a profiler region is entered with a call to `PE`, the time stamp is recorded, and on the corresponding call to `PL` another time stamp is recorded, and the difference accumulated. If a region includes time executing other tasks, for example when calling `arb::threading::parallel_for`, the time spent executing the other tasks will be included, which will give meaningless timings.

Organising Regions

The profiler allows the user to build a hierarchy of regions by grouping related regions together.

For example, network simulations have two main regions of code to profile: those associated with *communication* and *updating cell state*. These regions each have further subdivisions. We would like to break these regions down further, e.g. break the *communication* time into time spent performing *spike exchange* and *event binning*.

The subdivision of profiling regions is encoded in the region names. For example, `PE(communication_exchange)` indicates that we are profiling the `exchange` sub-region of the top level `communication` region.

Below is an example of using sub-regions:

```
#include <profiling/profiler.hpp>

using namespace arb;

spike_list global_spikes;
int num_cells = 100;

void communicate() {
    PE(communication_sortspikes);
```

(continues on next page)

(continued from previous page)

```
    auto local_spikes = get_local_spikes();
    sort(local_spikes);
    PL();

    PE(communication_exchange);
    global_spikes = exchange_spikes(local_spikes);
    PL();
}

void update_cell(int i) {
    PE(update_setup);
    setup_events(i);
    PL();

    PE(update_advance_state);
    update_cell_states(i);
    PL();

    PE(update_advance_current);
    update_cell_current(i);
    PL();
}

void run(double tfinal, double dt) {
    double t = 0;
    while (t<tfinal) {
        communicate();
        parallel_for(0, num_cells, update_cell);
        t += dt;
    }

    // print profiler results
    std::cout << util::profiler_summary() << "\n";
}
```

The `communication` region, is broken into two sub regions: `exchange` and `sortspikes`. Likewise, `update` is broken into `advance` and `setup`, with `advance` further broken into `state` and `current`.

Using the information encoded in the region names, the profiler can build a hierarchical report that shows accumulated time spent in each region and its children:

```
_p_ REGION                    CALLS      THREAD        WALL       %
_p_ root                          -       4.705       2.353   100.0
_p_   update                      -       4.200       2.100    89.3
_p_     advance                   -       4.100       2.050    87.1
_p_       state                1000       2.800       1.400    59.5
_p_       current              1000       1.300       0.650    27.6
_p_     setup                  1000       0.100       0.050     2.1
_p_   communication               -       0.505       0.253    10.7
_p_     exchange                 10       0.500       0.250    10.6
_p_     sortspikes               10       0.005       0.003     0.1
```

For _p_ more information on interpreting the profiler's output see *Running the Profiler* and *Profiler Output*.

3.25.3 Running the Profiler

The profiler does not need to be started or stopped by the user. It needs to be initialized before entering any profiling region. It is initialized using the information provided by the simulation's thread pool. At any point a summary of profiler region counts and times can be obtained, and the profiler regions can be reset.

```
#include <profiling/profiler.hpp>

using namespace arb;

void main() {
    execution_context context;

    // Initialize the profiler with thread information from the execution context
    profile::profiler_initialize(context.thread_pool);

    PE(init);
    // ...
    PL();

    PE(simulate);
    // ...
    PL();

    // Print a summary of the profiler to stdout
    std::cout << profile::profiler_summary() << "\n";

    // Clear the profiler state, which can then be used to record
    // profile information for a different part of the code.
    profile::profiler_clear();
}
```

After a call to `util::profiler_clear`, all counters and timers are set to zero. This could be used, for example, to generate separate profiler reports for model building and model execution phases.

Profiler Output

The profiler keeps accumulated call count and time values for each region in each thread. The `util::profile` type, defined in `src/profiling/profiler.hpp` holds a summary of the accumulated recorders. Calling `util::profiler_summary()` will generate a profile summary, which can be printed using the `operator<<` for `std::ostream`.

```
// get a profile summary
util::profile report = util::profiler_summary();

// print a summary of the profiler to stdout
std::cout << report << "\n";
```

Take the example output above:

```
_p_ REGION                    CALLS      THREAD        WALL       %
_p_ root                          -       5.379       1.345   100.0
_p_   advance                     -       5.368       1.342    99.8
_p_     integrate                 -       5.367       1.342    99.8
_p_       current             26046       3.208       0.802    59.6
_p_       state               26046       1.200       0.300    22.3
```

(continues on next page)

(continued from previous page)

```
_p_       matrix                  -       0.808       0.202    15.0
_p_         solve             26046       0.511       0.128     9.5
_p_         build             26046       0.298       0.074     5.5
_p_       events              78138       0.123       0.031     2.3
_p_       ionupdate           26046       0.013       0.003     0.2
_p_       samples             26046       0.007       0.002     0.1
_p_       threshold           26046       0.005       0.001     0.1
_p_   communication               -       0.012       0.003     0.2
_p_     enqueue                   -       0.011       0.003     0.2
_p_       sort                   88       0.011       0.003     0.2
```

For each region there are four values reported:

Value	Definition
CALLS	The number of times each region was profiled, summed over all threads. Only the call count for the leaf regions is presented.
THREAD	The total accumulated time (in seconds) spent in the region, summed over all threads.
WALL	The thread time divided by the number of threads.
%	The proportion of the total thread time spent in the region

3.26 Sampling API

The new API replaces the flexible but irreducibly inefficient scheme where the next sample time for a sampling was determined by the return value of the sampler callback.

3.26.1 Definitions

probe A location or component of a cell that is available for monitoring.

sample A record of data corresponding to the value at a specific *probe* at a specific time.

sampler A function or function object that receives a sequence of *sample* records.

schedule A function or function object that, given a time interval, returns a list of sample times within that interval.

3.26.2 Probes

Probes are specified in the recipe objects that are used to initialize a simulation; the specification of the item or value that is subjected to a probe will be specific to a particular cell type.

```
using probe_tag = int;

struct probe_info {
    cell_member_type id;    // cell gid, index of probe
    probe_tag tag;          // opaque key, returned in sample record
    any address;            // cell-type specific location info
};

probe_info recipe::get_probe(cell_member_type probe_id);
```

The `id` field in the `probe_info` struct will be the same value as the `probe_id` used in the `get_probe` call.

The `get_probe()` method is intended for use by cell group implementations to set up sampling data structures ahead of time and for putting in place any structures or information in the concrete cell implementations to allow monitoring.

The `tag` field has no semantics for the engine. It is provided merely as a way of passing additional metadata about a probe to any sampler that polls it, with a view to samplers that handle multiple probes, possibly with different value types.

Probe addresses are now decoupled from the cell descriptions themselves — this allows a recipe implementation to construct probes independently of the cells themselves. It is the responsibility of a cell group implementation to parse the probe address objects wrapped in the `any address` field.

3.26.3 Samplers and sample records

Data collected from probes (according to a schedule described below) will be passed to a sampler function or function object:

```
using sampler_function =
    std::function<void (cell_member_type, probe_tag, size_t, const sample_record*)>;
```

where the parameters are respectively the probe id, the tag, the number of samples and a pointer to the sequence of sample records.

The `probe_tag` is the key given in the `probe_info` returned by the recipe.

One `sample_record` struct contains one sample of the probe data at a given simulation time point:

```
struct sample_record {
    time_type time;     // simulation time of sample
    any_ptr data;       // sample data
};
```

The `data` field points to the sample data, wrapped in `any_ptr` for type-checked access. The exact representation will depend on the nature of the object that is being probed, but it should depend only on the cell type and probe address.

The data pointed to by `data`, and the sample records themselves, are only guaranteed to be valid for the duration of the call to the sampler function. A simple sampler implementation for `double` data might be:

```
using sample_data = std::map<cell_member_type, std::vector<std::pair<double, double>>>
↪;

struct scalar_sampler {
    sample_data& samples;

    explicit scalar_sample(sample_data& samples): samples(samples) {}

    void operator()(cell_member_type id, probe_tag, size_t n, const sample_record*␣
↪records) {
        for (size_t i=0; i<n; ++i) {
            const auto& rec = records[i];

            const double* data = any_cast<const double*>(rec.data);
            assert(data);
            samples[id].emplace_back(rec.time, *data);
        }
    }
};
```

The use of `any_ptr` allows type-checked access to the sample data, which may differ in type from probe to probe.

3.26.4 Model and cell group interface

Polling rates, policies and sampler functions are set through the `simulation` interface, after construction from a recipe.

```
using sampler_association_handle = std::size_t;
using cell_member_predicate = std::function<bool (cell_member_type)>;

sampler_association_handle simulation::add_sampler(
    cell_member_predicate probe_ids,
    schedule sched,
    sampler_function fn,
    sampling_policy policy = sampling_policy::lax);

void simulation::remove_sampler(sampler_association_handle);

void simulation::remove_all_samplers();
```

Multiple samplers can then be associated with the same probe locations. The handle returned is only used for managing the lifetime of the association. The `cell_member_predicate` parameter defines the set of probe ids in terms of a membership test.

Two helper functions are provided for making `cell_member_predicate` objects:

```
// Match all probe ids.
cell_member_predicate all_probes = [](cell_member_type pid) { return true; };

// Match just one probe id.
cell_member_predicate one_probe(cell_member_type pid) {
    return [pid](cell_member_type x) { return pid==x; };
}
```

The `sampling_policy` policy is used to modify sampling behaviour: by default, the `lax` policy is to perform a best-effort sampling that minimizes sampling overhead and which will not change the numerical behaviour of the simulation. Other policies may be implemented in the future, e.g. `interpolated` or `exact`.

The simulation object will pass on the sampler setting request to the cell group that owns the given probe id. The `cell_group` interface will be correspondingly extended:

```
void cell_group::add_sampler(sampler_association_handle h, cell_member_predicate␣
↪probe_ids, sample_schedule sched, sampler_function fn, sampling_policy policy);

void cell_group::remove_sampler(sampler_association_handle);

void cell_group::remove_all_samplers();
```

Cell groups will invoke the corresponding sampler function directly, and may aggregate multiple samples with the same probe id in one call to the sampler. Calls to the sampler are synchronous, in the sense that processing of the cell group state does not proceed while the sampler function is being executed, but the times of the samples given to the sampler will typically precede the time corresponding to the current state of the cell group. It should be expected that this difference in time should be no greater the the duration of the integration period (i.e. `mindelay/2`).

If a cell group does not support a given `sampling_policy`, it should raise an exception. All cell groups should support the `lax` policy, if they support probes at all.

3.26.5 Schedules

Schedules represent a non-negative, monotonically increasing sequence of time points, and are used to specify the sampling schedule in any given association of a sampler function to a set of probes.

A `schedule` object has two methods:

```
void schedule::reset();

time_event_span events(time_type t0, time_type t1)
```

A `time_event_span` is a `std::pair` of pointers *const time_type**, representing a view into an internally maintained collection of generated time values.

The `events(t0, t1)` method returns a view of monotonically increasing time values in the half-open interval `[t0, t1)`. Successive calls to `events` — without an intervening call to `reset()` — must request strictly subsequent intervals.

The data represented by the returned `time_event_span` view is valid for the lifetime of the `schedule` object, and is invalidated by any subsequent call to `reset()` or `events()`.

The `reset()` method resets the state such that events can be retrieved from again from time zero. A schedule that is reset must then produce the same sequence of time points, that is, it must exhibit repeatable and deterministic behaviour.

The `schedule` object itself uses type-erasure to wrap any schedule implementation class, which can be any copy–constructable class that provides the methods `reset()` and `events(t0, t1)` above. Three schedule implementations are provided by the engine:

```
// Schedule at integer multiples of dt:
schedule regular_schedule(time_type dt);

// Schedule at a predetermined (sorted) sequence of times:
template <typename Seq>
schedule explicit_schedule(const Seq& seq);

// Schedule according to Poisson process with lambda = 1/mean_dt
template <typename RandomNumberEngine>
schedule poisson_schedule(time_type mean_dt, const RandomNumberEngine& rng);
```

The `schedule` class and its implementations are found in `schedule.hpp`.

3.26.6 Helper classes for probe/sampler management

The `simulation` and `mc_cell_group` classes use classes defined in `scheduler_map.hpp` to simplify the management of sampler–probe associations and probe metdata.

`sampler_association_map` wraps an `unordered_map` between sampler association handles and tuples (*schedule*, *sampler*, *probe set*), with thread-safe accessors.

`probe_association_map<Handle>` is a type alias for an unordered map between probe ids and tuples (*probe handle*, *probe tag*), where the *probe handle* is a cell-group specific accessor that allows efficient polling.

3.26.7 Batched sampling in `mc_cell_group`

The `fvm_multicell` implementations for CPU and GPU simulation of multi-compartment cable neurons perform sampling in a batched manner: when their integration is initialized, they take a sequence of `sample_event` objects

which are used to populate an implementation-specific `multi_event_stream` that describes for each cell the sample times and what to sample over the integration interval.

When an integration step for a cell covers a sample event on that cell, the sample is satisfied with the value from the cell state at the beginning of the time step, after any postsynaptic spike events have been delivered.

It is the responsibility of the `mc_cell_group::advance()` method to create the sample events from the entries of its `sampler_association_map`, and to dispatch the sampled values to the sampler callbacks after the integration is complete. Given an association tuple (*schedule*, *sampler*, *probe set*) where the *schedule* has (non-zero) *n* sample times in the current integration interval, the `mc_cell_group` will call the *sampler* callback once for probe in *probe set*, with *n* sample values.

3.27 Distributed Context

To support running on systems from laptops and workstations to large distributed HPC clusters, Arbor uses *distributed contexts* to:

- Describe the distributed computer system that a simulation is to be distributed over and run on.
- Perform collective operations over the distributed system, such as gather and synchronization.
- Query information about the distributed system, such as the number of distributed processes and the index/rank of the calling process.

The global context used to run a simulation is determined at run time, not at compile time. This means that if Arbor is compiled with support for MPI enabled, then at run time the user can choose between using a non-distributed (local) context, or an distributed MPI context.

An execution context is created by a user before building and running a simulation. This context is then used to perform domain decomposition and initialize the simulation (see *Simulations* for more about the simulation building workflow). In the example below, a context that uses MPI is used to run a distributed simulation:

The public API does not directly expose *arb::distributed_context* or any of its implementations. By default *arb::context* uses only local "on-node" resources. To use an MPI communicator for distributed communication, it can be initialised with the communicator:

```
arb::proc_allocation resources;
my_recipe recipe;

// Create a context that uses the local resources enumerated in resources,
// and that uses the standard MPI communicator MPI_COMM_WORLD for
// distributed communication.
arb::context context = arb::make_context(resources, MPI_COMM_WORLD);

// Partition model over the distributed system.
arb::domain_decomposition decomp = arb::partition_load_balance(recipe, context);

// Instantiate the simulation over the distributed system.
arb::simulation sim(recipe, decomp, context);

// Run the simulation for 100ms over the distributed system.
sim.run(100, 0.01);
```

In the back end *arb::distributed_context* defines the interface for distributed contexts, for which two implementations are provided: *arb::local_context* and *arb::mpi_context*. Distributed contexts are wrapped in shared pointers:

using distributed_context_handle = std::shared_ptr<distributed_context>

A distributed context can then be generated using helper functions *arb::make_local_context()* and *arb::make_mpi_context()*:

```
// Create a context that uses only local resources (is non-distributed).
auto dist_ctx  arb::make_local_context();

// Create an MPI context that uses the standard MPI_COMM_WORLD communicator.
auto dist_ctx = arb::make_mpi_context(MPI_COMM_WORLD);
```

3.27.1 Class Documentation

class distributed_context

Defines the interface used by Arbor to query and perform collective operations on distributed systems.

Uses value-semantic type erasure. The main benefit of this approach is that classes that implement the interface can use duck typing instead of deriving from *distributed_context*.

Constructor:

distributed_context()
: Default constructor initializes the context as a *local_context*.

distributed_context(*distributed_context* &&*other*)
: Move constructor.

distributed_context &**operator=**(*distributed_context* &&*other*)
: Copy from rvalue.

template<typename **Impl**>
distributed_context(*Impl* &&*impl*)
: Initialize with an implementation that satisfies the interface.

Interface:

int **id**() **const**
: Each distributed process has a unique integer identifier, where the identifiers are numbered contiguously in the half open range [0, size). (for example `MPI_Rank`).

int **size**() **const**
: The number of distributed processes (for example `MPI_Size`).

void **barrier**() **const**
: A synchronization barrier where all distributed processes wait until every process has reached the barrier (for example `MPI_Barrier`).

std::string **name**() **const**
: The name of the context implementation. For example, if using MPI returns `"MPI"`.

std::vector<std::string> **gather**(std::string *value*, int *root*) **const**
: Overload for gathering a string from each domain into a vector of strings on domain *root*.

T **min**(T *value*) **const**
: Reduction operation over all processes.

  The type `T` is one of `float`, `double`, `int`, `std::uint32_t`, `std::uint64_t`.

T **max**(T *value*) **const**
: Reduction operation over all processes.

  The type `T` is one of `float`, `double`, `int`, `std::uint32_t`, `std::uint64_t`.

T **sum**(T *value*) **const**
Reduction operation over all processes.

The type `T` is one of `float`, `double`, `int`, `std::uint32_t`, `std::uint64_t`.

std::vector<T> **gather**(T *value*, int *root*) **const**
Gather operation. Returns a vector with one entry for each process.

The type `T` is one of `float`, `double`, `int`, `std::uint32_t`, `std::uint64_t`, `std::string`.

class local_context
Implements the *arb::distributed_context* interface for non-distributed computation.

This is the default *arb::distributed_context*, and should be used when running on laptop or workstation systems with one NUMA domain.

Note: *arb::local_context* provides the simplest possible distributed context, with only one process, and where all reduction operations are the identity operator.

Constructor:

local_context()
Default constructor.

distributed_context_handle **make_local_context**()
Convenience function that returns a handle to a local context.

class mpi_context
Implements the *arb::distributed_context* interface for distributed computation using the MPI message passing library.

Constructor:

mpi_context(MPI_Comm *comm*)
Create a context that will uses the MPI communicator *comm*.

distributed_context_handle **make_mpi_context**(MPI_Comm *comm*)
Convenience function that returns a handle to a *arb::mpi_context* that uses the MPI communicator comm.

Note: This is a developer feature for benchmarking, and is not useful for scientific use cases.

3.28 Dry-run Mode

Dry-run mode is used to mimic the performance of running an MPI distributed simulation without having access to an HPC cluster or even MPI support. It is verifiable against an MPI run with the same parameters. In dry-run mode, we describe the model on a single domain and translate it to however many domains we want to mimic. This allows us to know the exact behavior of the entire system by only running the simulation on a single node. To support dry-run mode we use the following classes:

class dry_run_context
Implements the *arb::distributed_context* interface for a fake distributed simulation.

unsigned **num_ranks_**
Number of domains we are mimicking.

unsigned **num_cells_per_tile_**
Number of cells assigned to each domain.

Constructor:

dry_run_context_impl (unsigned *num_ranks*, unsigned *num_cells_per_tile*)
Creates the dry run context and sets up the information needed to fake communication between domains.

Interface:

int **id**() **const**
Always 0. We are only performing the simulation on the local domain which will be root.

int **size**() **const**
Equal to *num_ranks_*.

std::string **name**() **const**
Returns "dry_run".

std::vector<std::string> **gather** (std::string *value*, int *root*) **const**
Duplicates the vector of strings from local domain, *num_ranks_* times. Returns the concatenated vector.

gathered_vector<arb::spike> **gather_spikes** (**const** std::vector<arb::spike> &*local_spikes*) **const**
The vector of *local_spikes* represents the spikes obtained from running a simulation of *num_cells_per_tile_* on the local domain. The returned vector should contain the spikes obtained from all domains in the dry-run. The spikes from the non-simulated domains are obtained by copying *local_spikes* and modifying the gids of each spike to refer to the corresponding gids on each domain. The obtained vectors of spikes from each domain are concatenated along with the original *local_spikes* and returned.

distributed_context_handle **make_dry_run_context** (unsigned *num_ranks*, unsigned *num_cells_per_tile*)
Convenience function that returns a handle to a *dry_run_context*.

class tile : **public** *recipe*

Note: While this class inherits from *arb::recipe*, it breaks one of its implicit rules: it allows connection from gids greater than the total number of cells in a recipe, ncells.

---

arb::tile describes the model on a single domain containing num_cells = num_cells_per_tile cells, which is to be duplicated over *num_ranks()* domains in dry-run mode. It contains information about *num_ranks()* which is provided by the following function:

cell_size_type **num_tiles**() **const**

Most of the overloaded functions in *arb::tile* describe a recipe on the local domain, as if it was the only domain in the simulation, except for the following two functions that accept gid arguments in the half open interval [0, num_cells*num_tiles):

std::vector<*cell_connection*> **connections_on** (*cell_gid_type gid*) **const**

std::vector<event_generator> **event_generators** (*cell_gid_type gid*) **const**

class symmetric_recipe : **public** *recipe*
A symmetric_recipe mimics having a model containing num_tiles() instances of *arb::tile* in a simulation of one tile per domain.

std::unique_ptr<*tile*> **tiled_recipe_**
symmetric_recipe owns a unique pointer to a *arb::tile*, and uses *tiled_recipe_* to query information about the tiles on the local and mimicked domains.

Most functions in *symmetric_recipe* only need to call the underlying functions of *tiled_recipe_* for the corresponding gid in the simulated domain. This is done with a simple modulo operation. For example:

```
cell_kind get_cell_kind(cell_gid_type i) const override {
    return tiled_recipe_->get_cell_kind(i % tiled_recipe_->num_cells());
}
```

The exception is again the following 2 functions:

std::vector<*cell_connection*> **connections_on**(*cell_gid_type i*) **const**

Calls

```
tiled_recipe_.connections_on(i % tiled_recipe_->num_cells())
```

But the obtained connections have to be translated to refer to the correct gids corresponding to the correct domain.

std::vector<event_generator> **event_generators**(*cell_gid_type i*) **const**

Calls

```
tiled_recipe_.event_generators(i)
```

Calls on the domain gid without the modulo operation, because the function has a knowledge of the entire network.

Python Module Index

a

Index

A

B

C

D

E

F

G

S

T

V

W