Running and visualizing the IBAMR tutorials

1 Navier Stokes Ex 1

Open a terminal and log in to one of the clot servers.

```
ssh username@clot128a.math.utah.edu
```

where `username` is your math username. You will be asked to type your password. Create and navigate to a folder that you will use to build the IBAMR examples. Clone the github repository into that directory

```
mkdir ibamr_tutorials && cd ibamr_tutorials
git clone https://github.com/abarret/ibamr_tutorial.git
```

At this point, you will have the IBAMR examples in a directory named `ibamr_tutorial`. It contains the examples and CMake build system files. Next, we create a build directory and run `CMake` to generate the Makefiles that can be used to compile the examples. This searches the provided directory for the IBAMR header files and compiled libraries.

```
mkdir build && cd build
cmake ../ibamr_tutorial -DIBAMR_ROOT=/u/ma/barrett/ibamr/ibamr/linux-opt \
  -DCMAKE_CXX_COMPILER=/u/ma/barrett/ibamr/openmpi/4.1.3/bin/mpicxx
```

Note that this uses the same compiler flags that were used to build IBAMR, in this case those are `-O3 -march=native`. Now we can build the examples.

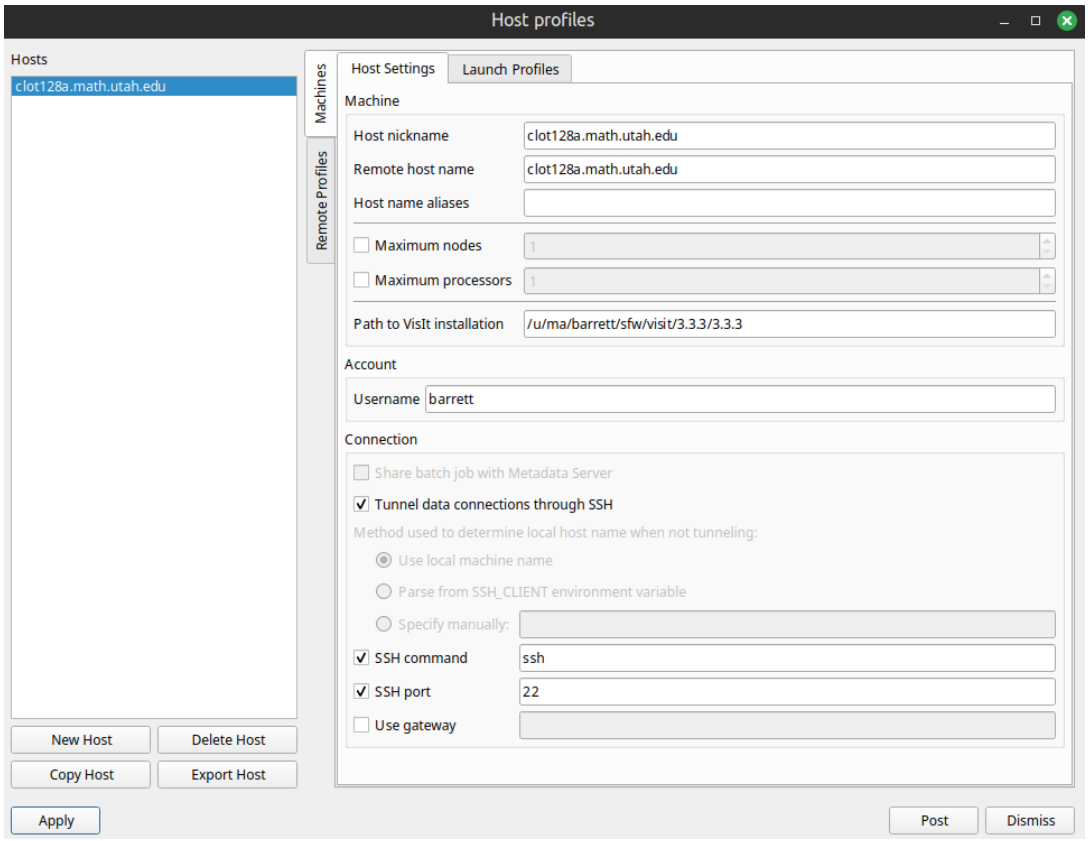
```
make -j4 examples
```

The argument `-j4` will parallelize the build and compile at most 4 files at the same time. Once `CMake` is run correctly, the Makefiles will automatically track changes in depedencies, and automatically determine which files need to be recompiled when running `make`. Now you can navigate into one of the examples and run it.

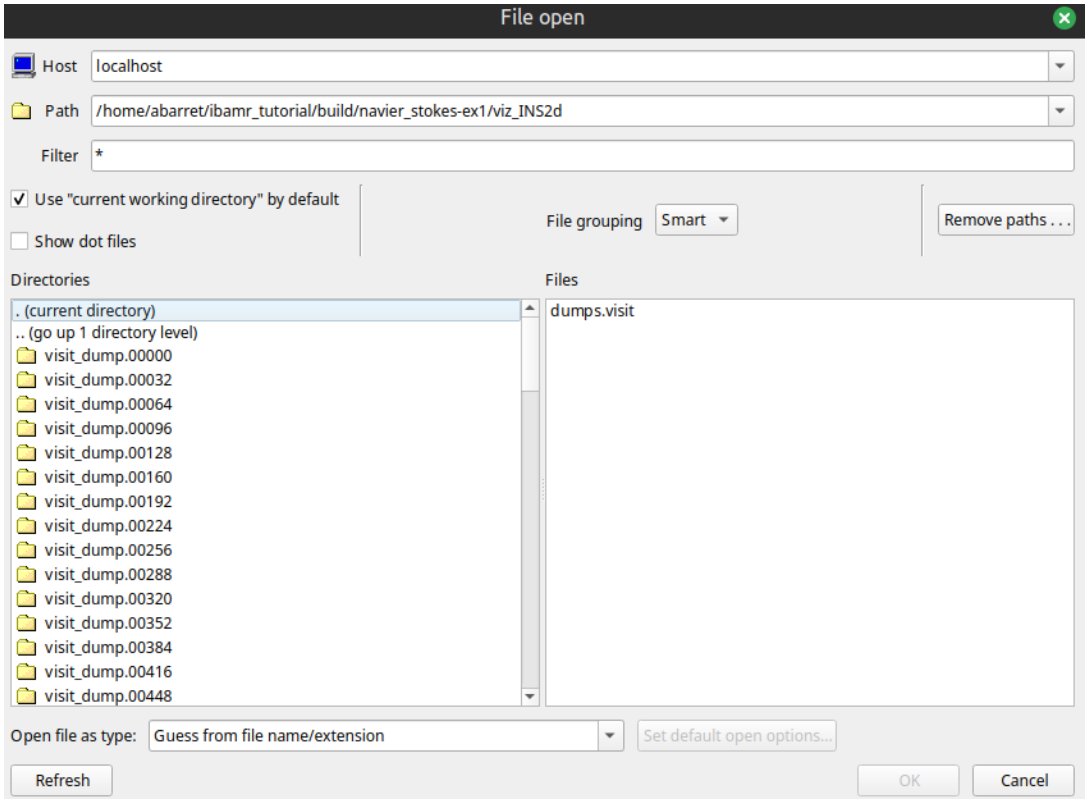
```
cd navier_stokes-ex1
./main2d input2d
```

This example is a simulation of the Kelvin-Helmholtz instability, in which the initial conditions consist of a sharp shear layer with a small perturbation. While this is running, we will open VisIt to visualize the results. At this time, you must use VisIt version 3.3 or 3.4. We will set up VisIt can be run in client-server mode to visualize results remotely. Alternatively, you can download the visualization files to analyze them locally.

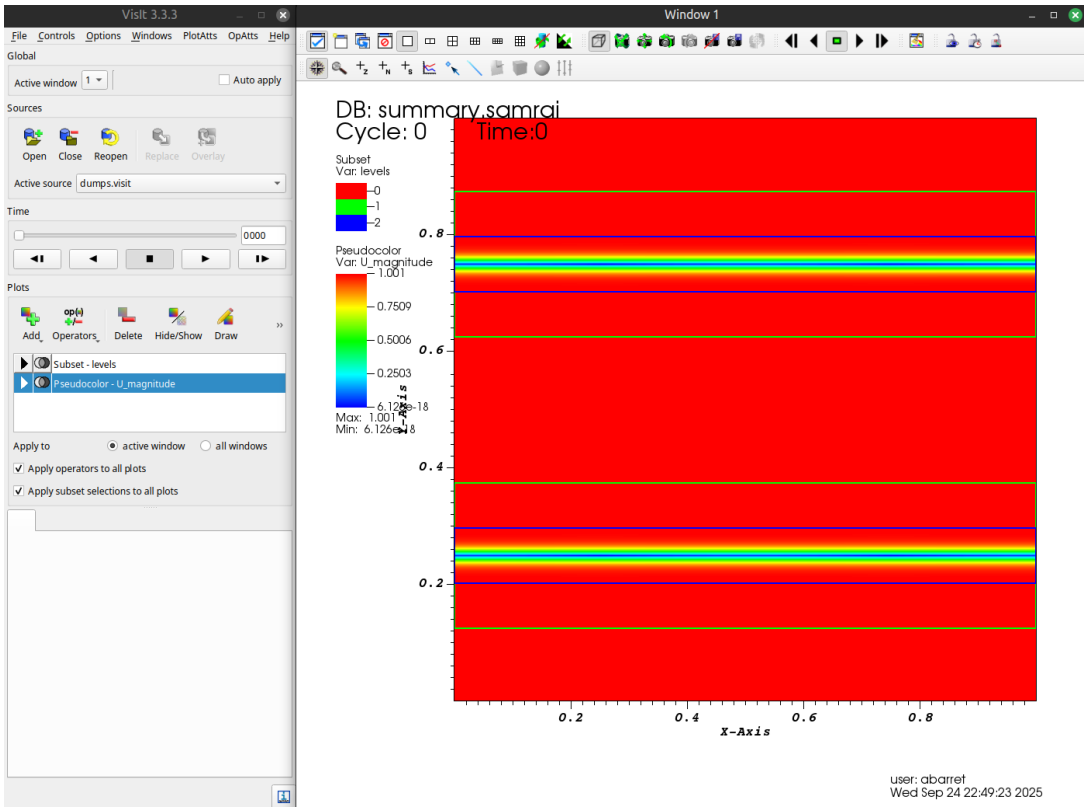
With VisIt open in the `Options` menu, click the `Host profiles` button. Create a new host and set it as in the following image. If you are using a different minor version than 3.3, you should use the appropriate VisIt release. Minor versions for the client and the server must match. The path to the VisIt installation should be the path to the executable, without the `bin/visit` extension.



Click apply and dismiss. Now we can connect to the clot server. Click `Open` and change the host to `clot128a.math.utah.edu`. You will be asked for your password. Now navigate to the build directory and open the `dumps.visit` file



By default, if running with multiple levels, a levels outline will be created by default. Now you can add various plots through the `Add` button. In particular, we can add a color plot of the velocity magnitude by selecting `Add`, then `Pseudocolor`, then `U_magnitude`.



Make sure to click `Draw` after adding any plot that you want to visualize. Double click any of the plots in the list to look at different plotting options, setting, e.g. color bars, color tables, line widths, and more. You can animate over time by pressing to play button.

2 IB Ex 1

As before, we want to navigate to the build directory of the IBAMR tutorials.

```
cd ${HOME}/ibamr_tutorial/build
```

For this example, we want to build `IB-ex1`, which is the classic IB simulation of an elastic rubberband.

```
make IB-ex1 && cd IB-ex1
```

Note that if `make` determines that no depedencies have changed, then the tool may correctly not compile anything. In this directory, in addition to the compiled binary and input file, there are several `.spring` and `.vertex` files. These files set the initial configuration and elasticity of the immersed structure. There is also a MATLAB file `generate_curve2d.m` that will generate these input files. Looking at the `curve2d_64` files, they begin with

```
cat curve2d_64.vertex
304
6.7857142857142860e-01 5.0000000000000000e-01
6.7853328871213048e-01 5.0723341542964395e-01
6.7841888542630380e-01 5.1446374098708458e-01
6.7822826758319077e-01 5.2168788812000810e-01
6.7796151660833393e-01 5.2890277091531634e-01
6.7761874644879327e-01 5.3610530741732487e-01
6.7720010352447235e-01 5.4329242094427166e-01
6.7670576666557192e-01 5.5046104140257135e-01
6.7613594703620039e-01 5.5760810659825688e-01
6.7549088804417134e-01 5.6473056354504547e-01
6.7477086523702756e-01 5.7182536976847198e-01
```

```
cat curve2d_64.spring
304
  0      1 1.9353241079974475e+02 0.0000000000000000e+00
  1      2 1.9353241079974475e+02 0.0000000000000000e+00
  2      3 1.9353241079974475e+02 0.0000000000000000e+00
  3      4 1.9353241079974475e+02 0.0000000000000000e+00
  4      5 1.9353241079974475e+02 0.0000000000000000e+00
  5      6 1.9353241079974475e+02 0.0000000000000000e+00
  6      7 1.9353241079974475e+02 0.0000000000000000e+00
  7      8 1.9353241079974475e+02 0.0000000000000000e+00
  8      9 1.9353241079974475e+02 0.0000000000000000e+00
  9     10 1.9353241079974475e+02 0.0000000000000000e+00
 10     11 1.9353241079974475e+02 0.0000000000000000e+00
```

The `.vertex` file prints the number of vertices, followed by the initial position of each vertex. The `.spring` file prints the number of springs, followed by the Lagrangian index of each vertex that defines a spring. The two constants that follow

are the spring constant and the resting length. The Lagrangian index is set by the order in which vertices are printed in the vertex file.

If you open the input file, you will notice the database titled `IBStandardInitializer` as follows

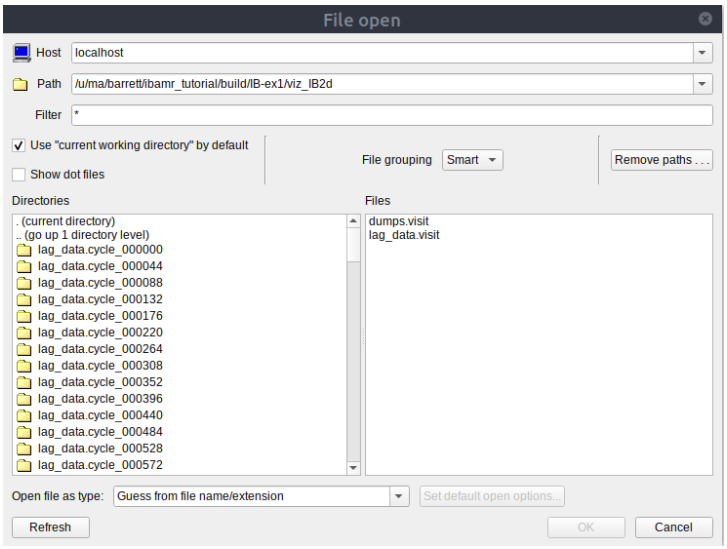
```
IBStandardInitializer {
   max_levels      = MAX_LEVELS
   structure_names = "curve2d_64"

   beta  = 0.35
   alpha = 0.25^2/beta

   A = PI*alpha*beta  // area of ellipse
   R = sqrt(A/PI)     // radius of disc with equivalent area as the ellipse
   perim = 2*PI*R     // perimeter of the equivalent disc

   dx = L/NFINEST
   dx_64 = L/64
   num_node_circum = (dx_64/dx)*ceil(perim/(dx_64/3)/4)*4
   ds = 2.0*PI*R/num_node_circum

   curve2d_64 {
      level_number = MAX_LEVELS - 1
      uniform_spring_stiffness = K/ds
   }
}
```

The structure is named `curve2d_64`, which points the program to open the files with that base name to initialize the structure. The structure lives on the level number `MAX_LEVELS - 1`, which is the finest level number. The input file also specified a uniform spring stiffness of `K/ds`, which overrides the value in the `.spring` file.

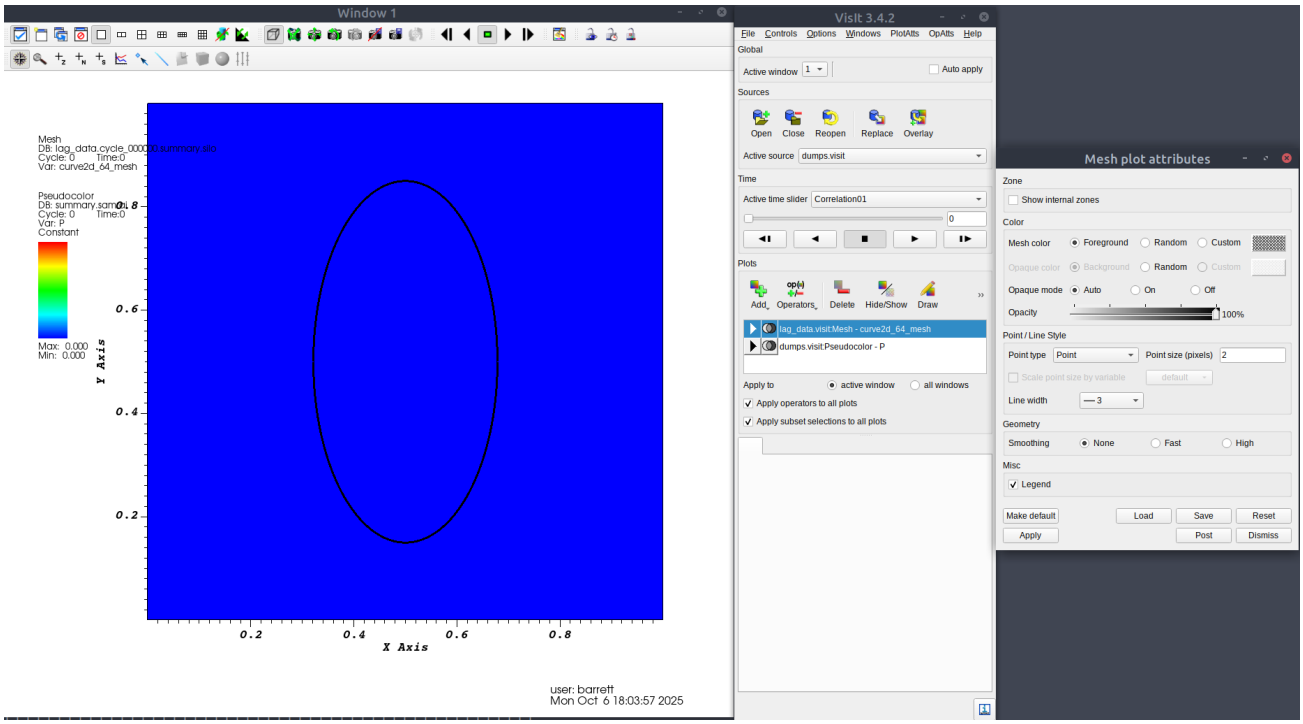
We can run this example in the same way that we ran the Navier-Stokes example

```
./main2d input2d
```

As before, we can open visit, click `Open`, navigate to the example directory, and open the `viz_IB2d` folder. In addition to the `dumps.visit` file, which points to the Eulerian data, it also contains a `lag_data.visit` file, which points to the Lagrangian data.



Open the Lagrangian file, and add a plot of the mesh by selecting `Add`, the `Mesh`, the `curve2d_64_mesh`. Make sure to press `Draw` to make the plot appear in the window. To increase the width of the line, double click the `curve2d_64_mesh` option to bring up the `Mesh plot attributes` window. You can increase the line width, then apply the changes. Also open the `dumps.visit` file, and add a pressure plot via the `Pseudocolor` option. VisIt will ask if you want to create a database correlation. These are used to correlate two separate databases and use a single time slider to move advance both databases simultaneously. If opt not to create one, you can create one later through the `Controls` and `Database correlations` tab.



As before, you can step through time by pressing the play button.

Now we will add adaptive mesh refinement to this simulation. In the build directory, open the input file, and change `MAX_LEVELS` from 1 to 2. This will add an additional level of refinement. Note the current refinement ratio is set to 4, so this additional level will have an equivalent uniform grid size of $N = 256$ points in each direction.

```
10 // grid spacing parameters
11 MAX_LEVELS = 2                                 // maximum number of levels in locally refined grid
12 REF_RATIO  = 4                                 // refinement ratio between levels
13 N = 64                                         // actual    number of grid cells on coarsest grid level
14 NFINEST = (REF_RATIO^(MAX_LEVELS - 1))*N       // effective number of grid cells on finest   grid level
15 DX_FINEST = L/NFINEST
```

Before we run the simulation, we should make sure that the structure is refined as well. To do this, we need to point the `IBStandardInitializer` database towards a different structure file, `curve2d_256`. Note that all the constants computed in the database will automatically adjust for a change in grid spacing.

```
 96 IBStandardInitializer {
 97    max_levels      = MAX_LEVELS
 98    structure_names = "curve2d_256"
 99
100    beta  = 0.35
101    alpha = 0.25^2/beta
102
103    A = PI*alpha*beta  // area of ellipse
104    R = sqrt(A/PI)     // radius of disc with equivalent area as the ellipse
105    perim = 2*PI*R     // perimeter of the equivalent disc
106
107    dx = L/NFINEST
108    dx_64 = L/64
109    num_node_circum = (dx_64/dx)*ceil(perim/(dx_64/3)/4)*4
110    ds = 2.0*PI*R/num_node_circum
111
112    curve2d_256 {
113       level_number = MAX_LEVELS - 1
114       uniform_spring_stiffness = K/ds
115    }
116 }
```

With the input file modified, we can rerun the example and open the visualization files as described above.

2.1 IBAMR Major Concepts

IBAMR borrows heavily from SAMRAI concepts, and in a lot of ways, is a application that lives on top of SAMRAI's data structures and algorithms.

The `PatchHierarchy<NDIM>` object manages all Eulerian data and the layout of data on the adaptive grid. It is composed of sequences of `PatchLevel<NDIM>` objects that are manage data for a specific grid spacing. Each `PatchLevel<NDIM>` object contains a sequence of `Patch<NDIM>` objects, each of which consists of a logically rectangular box of grid cells. Data on a patch is stored according to a patch data index, which is represented by an `int`. Associated patch data can be accessed from

the patch by the member function `Patch<NDIM>::getPatchData()`. Note that patch data can correspond to cell, side, face, edge, or node centered data, and must be cast to the appropriate type before use. Typically, patch data indices are retrieved using the `VariableDatabase<NDIM>` object, which will map pairs of `Variable` and `VariableContext` to patch data indices. The usual policy of IBAMR is for `Variable` objects to correspond to specific state data, while `VariableContext` objects correspond to time points. All the `HierarchyIntegrator` objects store state data with a current, new, and scratch context. For example, the following code will retrieve the current velocity patch index from an `INSStaggeredHierarchyIntegrator` object

```
auto var_db = VariableDatabase<NDIM>::getDatabase();
Pointer<SideVariable<NDIM, double>> u_var = ins_integrator->getVelocityVariable();
const int u_idx = var_db->mapVariableAndContextToIndex(u_var, ins_integrator->getCurrentContext());
```

With that patch index, we can compute various quantities on the `PatchHierarchy<NDIM>`. For example, we can compute the cell centered velocity magnitude by looping across all patches, interpolating velocities to cell centers, and computing the magnitude of the vector. The following code will do that.

```
Pointer<CellVariable<NDIM, double>> u_mag_var = new CellVariable<NDIM, double>("U_MAG");
auto var_db = VariableDatabase<NDIM>::getDatabase();
const int u_mag_idx = var_db->registerVariableAndContext(u_mag_var, var_db->getContext("CTX"));
for (int ln = 0; ln <= patch_hierarchy->getFinestLevelNumber(); ++ln)
{
  Pointer<PatchLevel<NDIM>> level = patch_hierarchy->getPatchLevel(ln);
  level->allocatePatchData(u_mag_idx);
  for (PatchLevel<NDIM>::Iterator p(level); p; p++)
  {
    Pointer<Patch<NDIM>> patch = level->getPatch(p());
    Pointer<CellData<NDIM, double>> u_mag_data = patch->getPatchData(u_mag_idx);
    Pointer<SideData<NDIM, double>> u_data = patch->getPatchData(u_idx);
    for (CellIterator<NDIM> ci(patch->getBox()); ci; ci++)
    {
      const CellIndex<NDIM>& idx = ci();
      double u_mag_sq = 0.0;
      for (int axis = 0; axis < NDIM; ++axis)
      {
        SideIndex<NDIM> up(idx, axis, 1), low(idx, axis, 0);
        double u = 0.5*((*u_data)(up) + (*u_data)(low));
        u_mag_sq += u * u;
      }
      (*u_mag_data)(idx) = std::sqrt(u_mag_sq);
    }
  }
}
```

Note that we must allocate the patch data for `u_mag_idx` before we can use it. Patch data stored with the `getCurrentContext()` context is always allocated. The scratch and new context data may or may not be allocated. Efficient routines to compute various quantities can be found in the `HierarchyMathOps` or `HierarchyDataOpsReal` classes.

The major algorithms used in IBAMR are the `HierarchyIntegrator` objects, which know how to integrate certain equations on a `PatchHierarchy`. The main base integrators are

- `AdvDiffHierarchyIntegrator` which can integrate advection diffusion reaction equations. It can be broken down further into two subclasses
  - `AdvDiffSemiImplicitHierarchyIntegrator` uses a semi-implicit, method of lines approach to descritize equations.
  - `AdvDiffPredictorCorrectorHierarchyIntegrator` is an older integrator that does not see much use. I don't know what this does...
- `INSHierarchyIntegrator` which can integrate various forms of the Navier-Stokes equations. It has several subclasses
  - `INSStaggeredHierarchyIntegrator` is the standard Navier-Stokes integrator which uses a staggered discretization of the momentum equation, storing velocities as `SideVariable<NDIM, double>` type and pressure as a `CellVariable<NDIM, double>` type. This is the standard integrator that should be used in most applications.

- `INSCollocatedHierarchyIntegrator` discretizes the momentum equation with a collocated discretization, storing velocities and pressures as `CellVariable<NDIM, double>` types.
- `INSVCStaggeredHierarchyIntegrator` discretizes the Navier-Stokes equations with spatially varying viscosity and density terms. Note that this utilizes the `AdvDiffHierarchyIntegrator` class to advect the viscosity and density. This class further breaks down into non-conservative and conservative discretizations.

- `IBHierarchyIntegrator` which can integrator the immersed boundary equations. Currently, the only concrete implementation is the `IBExplicitHierarchyIntegrator` class, which treats the immersed boundary explicitly in time.

To complement the integrators and provide a method to customize the equations, each integrator can handle specialized source terms, forcing functions, or strategy classes. Below covers some of the typicaly use cases for each of these integrator classes.

2.1.1 AdvDiffHierarchyIntegrator

Advected variables can be registered with the `AdvDiffHierarchyIntegrator` object to be integrated in the order in which they are registered. The advection diffusion integrators provide frameworks to set convective discretizations, variable diffusion coefficients, and source terms. For example, in `navier_stokes-ex5`, a temperature proxy variable used in the Boussinesq approximation is registered in the source code `example.cpp`:

```
147         // Setup the advected and diffused quantity.
148         Pointer<CellVariable<NDIM, double> > T_var = new CellVariable<NDIM, double>("T");
149         adv_diff_integrator->registerTransportedQuantity(T_var);
150         adv_diff_integrator->setDiffusionCoefficient(T_var, input_db->getDouble("KAPPA"));
151         adv_diff_integrator->setInitialConditions(
152             T_var,
153             new muParserCartGridFunction(
154                 "T_init", app_initializer->getComponentDatabase("TemperatureInitialConditions"), grid_geometry));
155         RobinBcCoefStrategy<NDIM>* T_bc_coef = nullptr;
156         if (!periodic_domain)
157         {
158             T_bc_coef = new muParserRobinBcCoefs(
159                 "T_bc_coef", app_initializer->getComponentDatabase("TemperatureBcCoefs"), grid_geometry);
160             adv_diff_integrator->setPhysicalBcCoef(T_var, T_bc_coef);
161         }
162         adv_diff_integrator->setAdvectionVelocity(T_var, time_integrator->getAdvectionVelocityVariable());
```

Convective Operators: The convective discretization is typically set in the input file. For example, in `navier_stokes-ex5`, the input file has the following set of options

```
 22 ADV_DIFF_SOLVER_TYPE        = "SEMI_IMPLICIT"   // the advection-diffusion solver to use (PREDICTOR_CORRECTOR or SEMI_IMPLICIT)
 23 ADV_DIFF_NUM_CYCLES         = 2                 // number of cycles of fixed-point iteration
 24 ADV_DIFF_CONVECTIVE_TS_TYPE = "MIDPOINT_RULE"   // convective time stepping type
 25 ADV_DIFF_CONVECTIVE_OP_TYPE = "PPM"             // convective differencing discretization type
 26 ADV_DIFF_CONVECTIVE_FORM    = "ADVECTIVE"       // how to compute the convective terms
 27 NORMALIZE_PRESSURE          = TRUE              // whether to explicitly force the pressure to have mean zero
```

```
186 AdvDiffSemiImplicitHierarchyIntegrator {
187    start_time                    = START_TIME
188    end_time                      = END_TIME
189    grow_dt                       = GROW_DT
190    num_cycles                    = ADV_DIFF_NUM_CYCLES
191    convective_time_stepping_type = ADV_DIFF_CONVECTIVE_TS_TYPE
192    convective_op_type            = ADV_DIFF_CONVECTIVE_OP_TYPE
193    convective_difference_form    = ADV_DIFF_CONVECTIVE_FORM
194    cfl                           = CFL_MAX
195    dt_max                        = DT_MAX
196    tag_buffer                    = TAG_BUFFER
197    enable_logging                = ENABLE_LOGGING
198 }
```

In this case, the input file sets the convective operator to use the `PPM` operator, discretized in `ADVECTIVE_FORM`, using a `MIDPOINT_RULE` to discretize in time.

IBAMR has several default convective operators to choose from:

- `PPM` uses a piecewise parabolic method,
- `CUI` uses a cubic upwinded interpolant method,
- `WAVE_PROP` uses a wave propagation method,
- `CENTERED` uses centered differences.

The convective terms can be integrated using `MIDPOINT_RULE`, `TRAPEZOIDAL_RULE`, or `FORWARD_EULER`. Note that `TRAPEZOIDAL_RULE` is an explicit RK2 method. Each convective operator can discretize the equations in either `ADVECTIVE` form: $\mathbf{u} \cdot \nabla Q$ or `CONSERVATIVE` form: $\nabla \cdot (\mathbf{u}Q)$.

Diffusion Coefficients: Uniform diffusion coefficients can be set by the call `setDiffusionCoefficient` in the `AdvDiffHierarchyIn` object. If no diffusion coefficient is used, or the diffusion coefficient is *exactly* equal to 0.0, no diffusion solver will be used. Otherwise, a Krylov solver will be used to solve the resulting implicit system, depending on the time stepping type specified.

Variable diffusion coefficients can be set by registering a variable diffusion coefficient with the `AdvDiffHierarchyIntegrator` object via the call `registerDiffusionCoefficientVariable`. Note that the variable registered must be of type `SideVariable<NDIM,` Then, a function that specifies the value of the diffusion coefficient should be registered, followed by setting the diffusion variable of the appropriate diffused quantities. For example, assuming `DiffusionCoefficient` is a class that extends `CartGridFunction`, the following code will set a variable coefficient for the advected quantity `Q_var`.

```
Pointer<SideVariable<NDIM, double>> D_var = new SideVariable<NDIM, double>("D");
adv_diff_integrator->registerDiffusionCoefficientVariable(D_var);
Pointer<CartGridFunction> D_fcn = new DiffusionCoefficient();
adv_diff_integrator->setDiffusionCoefficientFunction(D_var, D_fcn);
adv_diff_integrator->setDiffusionCoefficientVariable(Q_var, D_var);
```

Source Terms: Source terms in the advection diffusion integrator can be set in a similar method. A `CellVariable<NDIM, double>` object must be registered with the integrator. Then, the function that specifies the source must be registered and the source term must be set for the advected variable. If `SourceFunction` is a class that extends `CartGridFunction`, the following code will set a source term for the advected quantity `Q_var`.

```
Pointer<CellVariable<NDIM, double>> R_var = new CellVariable<NDIM, double>("R");
adv_diff_integrator->registerSourceTerm(R_var);
Pointer<CartGridFunction> R_fcn = new SourceFunction();
adv_diff_integrator->setSourceTermFunction(R_var, R_fcn);
adv_diff_integrator->setSourceTerm(Q_var, R_var);
```

2.1.2 INSStaggeredHierarchyIntegrator

The `INSHierarchyIntegrator` class and derived classes generally know how to solve the Navier-Stokes equations with different data centerings or variable coefficients. Our focus here is on the `INSStaggeredHierarchyIntegrator` class, although the other classes function similarly. The `INSStaggeredHierarchyIntegrator` class discretizes the constant coefficient Navier-Stokes equations on a staggerd grid, with the velocity stored as type `SideVariable<NDIM, double>` and pressure as type `CellVariable<NDIM, double>`. These integrator classes allow for a substantial amount of customization.

Forcing Functions: The computation of the body force must be done in an extension of the `CartGridFunction` interface. Note the force must be evaluated in accordance with the grid spacing used for the velocity. For example, if the integrator is of type `INSStaggeredHierarchyIntegrator`, the data index provided to the forcing function corresponds to type `SideData<NDIM, double>`. The forcing function can be registered with the `INSHierarchyIntegrator` with the call `registerBodyForceFunction`.

Viscoelasticity: Viscoelastic fluids that follow a Maxwell type model can be added via the `CFINSForcing` class. It requires a derived class of type `CFStrategy` be written that can compute the right hand side of the upper convective derivative and the transformation from the advected tensor to the stress tensor. Classes for the Oldroyd-B, Giesekus, and Rolie-Poly models already exist in IBAMR, and can be specified by the `fluid_model` parameter in the input database.

The class `CFINSForcing` is a specific implementation of a `CartGridFunction` whose application of `setDataOnPatchHierarchy` computes the divergence of the extra stress tensor. Therefore, to add viscoelasticity to a fluid solver, one needs to register the `CFINSForcing` object with the `INSHierarchyIntegrator` via the `registerBodyForceFunction` function.

Boundary Conditions: Boundary conditions can be registered with the Navier-Stokes integrator. Typically, boundary conditions will be read from the input file by the class `muParserRobinBcCoef`, which can parse strings into mathematical functions. The input file for these conditions specify the a, b, and g coefficients on each box face, for each spatial dimension. Boundary conditions take the form of $a\mathbf{u} + b\mathbb{T} \cdot \mathbf{n} = \mathbf{g}$, in which $\mathbb{T} = -p\mathbb{I} + \frac{\mu}{2}\left(\nabla\mathbf{u} + \nabla\mathbf{u}^{\mathrm{T}}\right)$ is the Newtonian stress and μ is the fluid viscosity. Current implementations require that at any physical location, the values of a and b satisfy that either $a = 1$ or $b = 1$ and that $a + b = 1$.

In all cases, periodic boundaries take prescendence over physical boundary conditions. Periodic conditions are specified in the `CartesianGeometry` database in the input file.

A typical implementation would look like this

```
vector<RobinBcCoefStrategy<NDIM>*> u_bc_coefs(NDIM, nullptr);
for (unsigned int d = 0; d < NDIM; ++d)
  {
```

```
  const std::string bc_coefs_name = "u_bc_coefs_" + std::to_string(d);
  const std::string bc_coefs_db_name = "VelocityBcCoefs_" + std::to_string(d);
  u_bc_coefs[d] = new muParserRobinBcCoefs(
    bc_coefs_name, app_initializer->getComponentDatabase(bc_coefs_db_name), grid_geometry);
  }
time_integrator->registerPhysicalBoundaryConditions(u_bc_coefs);
```

and the corresponding input file would look like

```
VelocityBcCoefs_0 {
   acoef_function_0 = "1.0"
   acoef_function_1 = "1.0"
   acoef_function_2 = "1.0"
   acoef_function_3 = "1.0"

   bcoef_function_0 = "0.0"
   bcoef_function_1 = "0.0"
   bcoef_function_2 = "0.0"
   bcoef_function_3 = "0.0"

   gcoef_function_0 = "0.0"
   gcoef_function_1 = "0.0"
   gcoef_function_2 = "0.0"
   gcoef_function_3 = "0.0"
}
```

with another database for `VelocityBcCoefs_1`. This boundary condition sets up no slip for the horizontal component of the velocity.

2.1.3 IBHierarchyIntegrator

IBStrategy:

IBMethod:

IBFEMethod:

3 Navier Stokes Ex 5

The example `navier_stokes-ex5` models stratified layer of fluids marked by temperature differences. The Boussinesq approximation is used to compute the force of gravity on the system. The temperature of the fluid is modeled by an advection-diffusion equation

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = D \nabla^2 T, \tag{1}$$

and an extra force is added to the momentum equation

$$\mathbf{F} = \gamma \mathbf{g}(T - T_0), \tag{2}$$

in which T_0 is the baseline temperature and $\mathbf{g}$ is the force of gravity, here assumed to orient in the negative y direction.

As before, we can navigate to the build directory and build the example

```
cd ibamr_tutorials/build && make -j4 examples
cd navier_stokes-ex5
./main2d input2d
```

If we look at the source code, we can see the `BoussinesqForcing` source and header files as well as the main source `example.cpp`. The main source code contains very similar code to that of `navier-stokes_ex1`, with the addition of the `AdvDiffHierarchyIntegrator` class that is registered with the fluid solver. This new integrator will integrate the temperature in equation (1).

```
104         Pointer<AdvDiffHierarchyIntegrator> adv_diff_integrator;
105         const string adv_diff_solver_type =
106             main_db->getStringWithDefault("adv_diff_solver_type", "PREDICTOR_CORRECTOR");
107         if (adv_diff_solver_type == "PREDICTOR_CORRECTOR")
108         {
109             Pointer<AdvectorExplicitPredictorPatchOps> predictor = new AdvectorExplicitPredictorPatchOps(
110                 "AdvectorExplicitPredictorPatchOps",
111                 app_initializer->getComponentDatabase("AdvectorExplicitPredictorPatchOps"));
112             adv_diff_integrator = new AdvDiffPredictorCorrectorHierarchyIntegrator(
113                 "AdvDiffPredictorCorrectorHierarchyIntegrator",
114                 app_initializer->getComponentDatabase("AdvDiffPredictorCorrectorHierarchyIntegrator"),
115                 predictor);
116         }
117         else if (adv_diff_solver_type == "SEMI_IMPLICIT")
118         {
119             adv_diff_integrator = new AdvDiffSemiImplicitHierarchyIntegrator(
120                 "AdvDiffSemiImplicitHierarchyIntegrator",
121                 app_initializer->getComponentDatabase("AdvDiffSemiImplicitHierarchyIntegrator"));
122         }
123         else
124         {
125             TBOX_ERROR("Unsupported solver type: " << adv_diff_solver_type << "\n"
126                                                    << "Valid options are: PREDICTOR_CORRECTOR, SEMI_IMPLICIT");
127         }
128         time_integrator->registerAdvDiffHierarchyIntegrator(adv_diff_integrator);
```

As noted earlier, the variable denoting the temperature T is created and registered with the advection diffusion integrator.

```
147         // Setup the advected and diffused quantity.
148         Pointer<CellVariable<NDIM, double> > T_var = new CellVariable<NDIM, double>("T");
149         adv_diff_integrator->registerTransportedQuantity(T_var);
150         adv_diff_integrator->setDiffusionCoefficient(T_var, input_db->getDouble("KAPPA"));
151         adv_diff_integrator->setInitialConditions(
152             T_var,
153             new muParserCartGridFunction(
154                 "T_init", app_initializer->getComponentDatabase("TemperatureInitialConditions"), grid_geometry));
155         RobinBcCoefStrategy<NDIM>* T_bc_coef = nullptr;
156         if (!periodic_domain)
157         {
158             T_bc_coef = new muParserRobinBcCoefs(
159                 "T_bc_coef", app_initializer->getComponentDatabase("TemperatureBcCoefs"), grid_geometry);
160             adv_diff_integrator->setPhysicalBcCoef(T_var, T_bc_coef);
161         }
162         adv_diff_integrator->setAdvectionVelocity(T_var, time_integrator->getAdvectionVelocityVariable());
```

Finally, a `BoussinesqForcing` object is registered with the integrator

```
164         // Set up the fluid solver.
165         time_integrator->registerBodyForceFunction(
166             new BoussinesqForcing(T_var, adv_diff_integrator, input_db->getDouble("GAMMA")));
```

As noted in the section on forces for the Navier-Stokes integrators, the class `BoussinesqForcing` is an extension of a `CartGridFunction` that can evaluate equation (2) on the patch hierarchy. On creation, it takes the variable representing the temperature, the advection diffusion integrator that manages the data, and the parameter γ, as seen in the header file

```
31 class BoussinesqForcing : public CartGridFunction
32 {
33 public:
34     /*!
35      * \brief Class constructor.
36      */
37     BoussinesqForcing(Pointer<Variable<NDIM> > T_var,
38                       Pointer<AdvDiffHierarchyIntegrator> adv_diff_hier_integrator,
39                       int gamma);
```

The corresponding source file `BoussinesqForcing.cpp` contains two important functions, `setDataOnPatchHierarchy` and `setDataOnPatch`.

In order to evaluate the force in equation (2) on cell sides, we must interpolate the cell centered data to the side of a cell, then multiply by $-\gamma$. To do the interpolation, we require one layer of ghost cells to be filled in. The function `setDataOnPatchHierarchy` allocates scratch data for `T_var` (if necessary), then fills in ghost cells using the class `HierarchyGhostCellInterpolation`, which contains general routines to make filling ghost cells easy. Then the function loops over levels and calls `setDataOnPatchLevel`, which by default will loop over patches and call `setDataOnPatch`.

The actual computation of the force is done in `setDataOnPatch`. First, the cell data corresponding to the scratch context is retrieved from the patch. Then, we loop over all cell sides whose normal points in the y direction. We interpolate the cell data to the cell side using simple averaging, and evaluate the force.

```
127     Pointer<CellData<NDIM, double> > T_scratch_data =
128         patch->getPatchData(d_T_var, d_adv_diff_hier_integrator->getScratchContext());
129     const Box<NDIM>& patch_box = patch->getBox();
130     const int axis = NDIM - 1;
131     for (Box<NDIM>::Iterator it(SideGeometry<NDIM>::toSideBox(patch_box, axis)); it; it++)
132     {
133         SideIndex<NDIM> s_i(it(), axis, 0);
134         (*F_data)(s_i) = -d_gamma * 0.5 * ((*T_scratch_data)(s_i.toCell(1)) + (*T_scratch_data)(s_i.toCell(0)));
135     }
136     return;
```

3.1 Adding a Simple Reaction System

We can add a simple reaction system to this model by adding two new advected variables R and Q. Let's say the reaction system will be

$$\frac{\partial Q}{\partial t} + \mathbf{u} \cdot \nabla Q = D_Q \nabla^2 Q - \kappa Q(1 - R), \tag{3}$$

$$\frac{\partial R}{\partial t} + \mathbf{u} \cdot \nabla R = D_R \nabla^2 R + \kappa Q(1 - R). \tag{4}$$

To accomplish this, we need to add two advected quantities, Q and R, and write two source functions `QSourceFunction` and `RSourceFunction`. To start, we can checkout the branch that adds a skeleton for these functions.

```
cd ibamr_tutorial/ibamr_tutorial
git pull && git checkout add_source
```

In the source directory `navier_stokes-ex5`, there should now be two additional files `QSourceFunction.h` and `QSourceFunction.cpp`. To complete the `QSourceFunction` class, we need to be able to evaluate the source term in equation (3). To evaluate this term, we need the values of κ, Q, and R. Therefore, we will need to store the patch data corresponding to Q and R and have a `double` that corresponds to the value of κ. Recall that the patch data index can be retrieved with the variable and context pair. Therefore, we pass in the `CellVariable` corresponding to the state of Q and R, as well as the `AdvDiffHierarchyIntegrator` object with which we can retrieve the variable context. These will be stored as member variables of the class. We also need to update the constructor to ensure these objects are avaiable upon construction of the object

```
31 class QSourceFunction : public CartGridFunction
32 {
33 public:
34     /*!
35      * \brief Class constructor.
36      */
37     QSourceFunction(SAMRAI::tbox::Pointer<SAMRAI::pdat::CellVariable<NDIM, double> > Q_var,
38                     SAMRAI::tbox::Pointer<SAMRAI::pdat::CellVariable<NDIM, double> > R_var,
39                     SAMRAI::tbox::Pointer<IBAMR::AdvDiffHierarchyIntegrator> adv_diff_hier_integrator,
40                     double kappa);
```

```
70 private:
71     Pointer<SAMRAI::pdat::CellVariable<NDIM, double> > d_Q_var, d_R_var;
72     Pointer<AdvDiffHierarchyIntegrator> d_adv_diff_hier_integrator;
73     double d_kappa = std::numeric_limits<double>::quiet_NaN();
74 };
```

In the implementation file, we need to update the constructor to set the appropriate member variables. We also need to update the `setDataOnPatch` function to correctly evaluate the source function. Note we do not need ghost cell information to evaluate this function. Therefore, there is no ghost filling in the function `setDataOnPatchHierarchy`, and that function is not implemented. By default, the base class loops over patch levels and calls `setDataOnPatchLevel`.

```
QSourceFunction::QSourceFunction(Pointer<CellVariable<NDIM, double> > Q_var,
                                 Pointer<CellVariable<NDIM, double> > R_var,
                                 Pointer<AdvDiffHierarchyIntegrator> adv_diff_hier_integrator,
                                 const double kappa)
    : d_Q_var(Q_var), d_R_var(R_var), d_adv_diff_hier_integrator(adv_diff_hier_integrator), d_kappa(kappa)
{
    // intentionally blank
    return;
} // QSourceFunction
```

```
void
QSourceFunction::setDataOnPatch(const int data_idx,
                                Pointer<Variable<NDIM> > /*var*/,
                                Pointer<Patch<NDIM> > patch,
                                const double /*data_time*/,
                                const bool initial_time,
                                Pointer<PatchLevel<NDIM> > /*patch_level*/)
{
    if (initial_time) return;
    Pointer<CellData<NDIM, double> > return_data = patch->getPatchData(data_idx);
    Pointer<CellData<NDIM, double> > Q_data =
        patch->getPatchData(d_Q_var, d_adv_diff_hier_integrator->getCurrentContext());
    Pointer<CellData<NDIM, double> > R_data =
        patch->getPatchData(d_R_var, d_adv_diff_hier_integrator->getCurrentContext());

    for (CellIterator<NDIM> ci(patch->getBox()); ci; ci++)
    {
        const CellIndex<NDIM>& idx = ci();
        (*return_data)(idx) = -1.0 * d_kappa * (*Q_data)(idx) * (1.0 - (*R_data)(idx));
    }
    return;
} // setDataOnPatch
```

We need to do the same for the class `RSourceFunction`. Because the function for R is the negative of the function for Q, we can simply copy the Q files.

```
cp QSourceFunction.cpp RSourceFunction.cpp
cp QSourceFunction.h RSourceFunction.h
```

You will need to make sure the names inside the source and header files are changed to their appropriate class, as well as fixing the sign of the operator. We also need to update the `CMakeLists.txt` to ensure that the files are compiled.

```
14 IBAMR_ADD_EXAMPLE(
15   TARGET_NAME
16     "navier_stokes-ex5"
17   OUTPUT_DIRECTORY
18     "${CMAKE_BINARY_DIR}/navier_stokes-ex5"
19   OUTPUT_NAME
20     main2d
21   EXAMPLE_GROUP
22     examples
23   SOURCES
24     QSourceFunction.cpp RSourceFunction.cpp BoussinesqForcing.cpp example.cpp
25   LINK_TARGETS
26     IBAMR::IBAMR2d
27   INPUT_FILES
28     input2d
29   )
```

Finally, we need to update the main driver to create and register two cell centered variables with the advection diffusion integrator. First, we add the header files to the main routine.

```
37 // Set up application namespace declarations
38 #include <ibamr/app_namespaces.h>
39
40 // Application objects
41 #include "BoussinesqForcing.h"
42 #include "QSourceFunction.h"
43 #include "RSourceFunction.h"
```

Next, we create cell variables corresponding to Q and R. We need to register them with the advection-diffusion solver as well as set coefficients, set advection velocities, and specify boundary and initial conditions. By default, initial conditions are set to 0 and boundary conditions default to linear extrapolation from interior values.

Finally, we need to create source variables that correspond to the sources for Q and R. We register these with the advection-diffusion integrator, as well as register functions that set the source terms. As shown below, when we create the source functions, the rate constant is being read from the input file.

```
183         Pointer<CellVariable<NDIM, double> > Q_var = new CellVariable<NDIM, double>("Q");
184         Pointer<CellVariable<NDIM, double> > R_var = new CellVariable<NDIM, double>("R");
185         adv_diff_integrator->registerTransportedQuantity(Q_var);
186         adv_diff_integrator->registerTransportedQuantity(R_var);
187         adv_diff_integrator->setDiffusionCoefficient(Q_var, input_db->getDouble("Q_DIFF_COEF"));
188         adv_diff_integrator->setDiffusionCoefficient(R_var, input_db->getDouble("R_DIFF_COEF"));
189         adv_diff_integrator->setAdvectionVelocity(Q_var, time_integrator->getAdvectionVelocityVariable());
190         adv_diff_integrator->setAdvectionVelocity(R_var, time_integrator->getAdvectionVelocityVariable());
191
192         Pointer<CellVariable<NDIM, double> > Q_src_var = new CellVariable<NDIM, double>("Q_SRC");
193         Pointer<CellVariable<NDIM, double> > R_src_var = new CellVariable<NDIM, double>("R_SRC");
194         adv_diff_integrator->registerSourceTerm(Q_src_var);
195         adv_diff_integrator->registerSourceTerm(R_src_var);
196         adv_diff_integrator->setSourceTermFunction(
197             Q_src_var, new QSourceFunction(Q_var, R_var, adv_diff_integrator, input_db->getDouble("RATE")));
198         adv_diff_integrator->setSourceTermFunction(
199             R_src_var, new RSourceFunction(Q_var, R_var, adv_diff_integrator, input_db->getDouble("RATE")));
200         adv_diff_integrator->setSourceTerm(Q_var, Q_src_var);
201         adv_diff_integrator->setSourceTerm(R_var, R_src_var);
202         adv_diff_integrator->setInitialConditions(
203             Q_var,
204             new muParserCartGridFunction(
205                 "Q_init", app_initializer->getComponentDatabase("QInitialConditions"), grid_geometry));
```

With this, we can compile and run the new example as before. Make sure you edit the input file to specify the rate parameter `RATE` and the diffusion coefficients `Q_DIFF_COEF` and `R_DIFF_COEF`.

```
cd ibamr_tutorials/build && make -j4 examples
cd navier_stokes-ex5
./main2d input2d
```