Manual for Immersed Boundary Projection Method (IBPM)

Clancy Rowley

Revision: ; Last changed by on May 4, 2021.

1 Overview

The code described in this manual uses an immersed boundary method to solve the two-dimensional incompressible Navier-Stokes equations around complex geometries, using the projection method described in [4]. In particular, this code implements the "fast method" described in Section 3.3 of [2], using the multi-domain approach for far-field boundary conditions, as described in section 4 of [2].

1.1 How this manual is organized

This package provides a command-line tool (`ibpm`), as well as a library (`libibpm.a`) that can be used for writing customized programs, such as post-processing utilities, or computational wrappers around the main solver.

Section 2 describes how to compile the library and executables. Section 3 describes what the code does, giving the highlights of the numerical method. Section 4 describes the simplest way to use the code, via the command-line tool. Section 5 describes the high-level classes provided by the library, geared towards users of the library. More detailed documentation of the library (e.g., for developers making additions to the library) can also be generated (see Section 2 below).

2 Installation

2.1 System requirements

To compile and run the code, you will need the following:

- A C++ compiler
- The FFTW library, version 3, available free from `http://www.fftw.org/`

We assume you are building on a Unix-based system, such as Mac OS X or Linux, and have standard build tools such as GNU Make.

The package includes a suite of automated tests, and if you would like to run these, you will need the Google C++ Testing Framework, version 1.3.0, available free from `http://code.google.com/p/googletest/`.

Detailed documentation for the library can also be generated, in html and/or LaTeX formats. To generate the documentation you will need the Doxygen tool, available at `http://www.stack.nl/~dimitri/doxygen/`.

2.2 Building the library and executables

The default configuration is to compile the code with the GCC compiler, with the FFTW library in a default location known to the compiler (e.g. `/usr/local/lib`). If this is acceptable, then to build the library and executables, all you need to do is type `make` from the root `ibpm` directory. The library and command-line tools will be generated in the `build` directory.

In order to customize the build process for your system, make a copy of the file `config/make.inc.gcc` and modify it as needed (this is included in the main Makefiles, and the format is pretty self-explanatory). The new configuration file should be named `config/make.inc`. Then type `make` from the root `ibpm` directory, as before.

To build and run the automated tests, type `make test`.

2.3 Building the documentation

To build the detailed documentation, type `make doc`. The default configuration is to generate both html and LaTeX documentation, but this can be changed by modifying `doc/Doxyfile`, for instance by changing the line `GENERATE_LATEX = YES` to `GENERATE_LATEX = NO`. Once the documentation has been built, the html documentation (usually the most useful) can be found in `doc/html/index.html`, and the LaTeX documentation can be found in `doc/latex/refman.tex`.

3 What the code does

For details of the numerical method this code solves, see [2]. Here, we give only a brief overview.

Immersed boundary method The Navier-Stokes equations are solved in two dimensions, using a streamfunction-vorticity formulation, and a finite-volume method. Thus, fluxes are defined on cell edges, and the scalar vorticity and streamfunction are defined at cell nodes. Pressure is not used in this formulation.

Let q denote the (vector-valued) velocity flux, ψ the streamfunction, and ω the vorticity. The no-slip boundary condition at the surface of an object is imposed by delta-function forces at the boundary locations, and f is a vector of these force values. The equations to be solved are given as (22) in [2]:

$$\frac{d\omega}{dt} + C^T E^T \tilde{f} = \nu L\omega + C^T(q \times \omega), \qquad \omega\big|_\partial = bc_\omega \tag{1}$$

$$L\psi = -\omega, \qquad \psi\big|_\partial = bc_\psi \tag{2}$$

$$EC\psi = u_B. \tag{3}$$

The first equation is the momentum equation, the second is a Poisson equation for the streamfunction, and the third equation is a constraint representing the no-slip condition, so that velocities at the boundary points match prescribed velocities u_B. The discrete operators in the above equation are described in the table below, where $\nu = 1/Re$ is one over the Reynolds number. (Note that in [2], γ denotes the circulation about one cell, while in this code we work with vorticity ω. The two are related by $\gamma = \delta^2\omega$, where δ is the grid spacing.)

Operator	Maps	Definition
C	$\psi \mapsto q$	curl of a scalar
C^T	$q \mapsto \omega$	curl of a vector in 2d
S	$\omega \mapsto \hat{\omega}$	discrete sin transform
$L = -C^T C$	$\omega \mapsto \omega$	Laplacian, analogous to $\triangle u = \nabla(\nabla \cdot u) - \nabla \times \nabla \times u$
Λ	$\hat{\omega} \mapsto \hat{\omega}$	eigenvalues of Laplacian
E	$q \mapsto u_B$	restriction of fluxes everywhere to velocities at boundary
E^T	$f \mapsto q$	regularization of forces at boundary points to fluxes everywhere
D	$q \mapsto \varphi$	divergence of a vector field

Here, u_B is a vector of velocities at boundary points. An important aspect of the method is that the curl operator C is chosen such that its range is in the nullspace of a discrete divergence operator D, such that $DC = 0$, and the continuity equation is satisfied for any streamfunction ψ. The discrete Laplacian L is diagonalized by the discrete sin transform S, and its eigenvalues are known analytically, so the Poisson equation (2) can be solved efficiently. To retrieve the fluxes q from the streamfunction ψ, we need to add in a (prescribed) potential flow solution q_{pot}, so we have

$$q = C\psi + q_{\text{pot}}. \tag{4}$$

We consider farfield boundary conditions, for which bc_ω in (1) and bc_ψ in (2) are both zero.

Time discretization Equation (1–3) are stepped forward in time using a projection method, as follows. For instance, discretizing the linear terms of (1) using Crank-Nicolson (trapezoidal rule), and the nonlinear terms using explicit Euler, one obtains

$$\left(1 - \frac{\nu\Delta t}{2}L\right)\omega^{n+1} + \Delta t C^T E^T f = \left(1 + \frac{\nu\Delta t}{2}L\right)\omega^n + \Delta t C^T(q^n \times \omega^n) \tag{5}$$

$$-ECL^{-1}\omega^{n+1} = u_B^{n+1} \tag{6}$$

These equations are of the form

$$\begin{bmatrix} \mathcal{A} & \mathcal{B} \\ \mathcal{C} & 0 \end{bmatrix} \begin{bmatrix} \omega^{n+1} \\ f \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{7}$$

where

$$\mathcal{A} = 1 - \frac{\nu \Delta t}{2} L \tag{8}$$

$$\mathcal{B} = \Delta t C^T E^T \tag{9}$$

$$\mathcal{C} = -ECL^{-1} \tag{10}$$

$$a = \left(1 + \frac{\nu \Delta t}{2} L\right) \omega^n + \Delta t C^T (q^n \times \omega^n) \tag{11}$$

$$b = u_B. \tag{12}$$

Projection method We solve the constrained equation (7) using the following algorithm:

$$\begin{aligned} \mathcal{A}\omega^* &= a, \qquad \omega^*\big|_\partial \\ \mathcal{C}\mathcal{A}^{-1}\mathcal{B} f &= \mathcal{C}\omega^* - b \\ \omega^{n+1} &= \omega^* - \mathcal{A}^{-1}\mathcal{B} f \end{aligned} \tag{13}$$

Since the matrix $\mathcal{A}$ is easily invertible using a sin transform, one may solve the first equation easily (applying boundary conditions on ω^*). The second equation is rather small (#forces $\times$ #forces), and furthermore the matrix $\mathcal{M} = \mathcal{C}\mathcal{A}^{-1}\mathcal{B}$ is symmetric, since $\mathcal{A}$ and L have the same eigenvectors. When the boundary conditions are fixed (stationary bodies), then the matrix $\mathcal{M}$ is constant in time, and so may be LU decomposed (e.g. using a Cholesky decomposition) once beforehand, and solved rapidly at each timestep. When the boundary conditions vary in time, then E changes at each step, and a direct solve is not as efficient as an iterative solve, such as a conjugate-gradient method (also for symmetric matrices).

In [2], the algorithm for the timestepper (5) is coupled with the algorithm (13) for solving equation (7), but in the design of the present code, these two algorithms are decoupled, so that different timesteppers may be interchanged easily.

3.1 Multi-domain method

For large domains, it is not practical to use uniform grid spacing, so the method in [2] uses a multi-domain approach in which several nested uniform grids are used. The Poisson equations may still be solved efficiently in this case, also using a number of sin transforms, as outlined here. In the code, the only place the methods in this section are used is in the elliptic solvers (Poisson or Helmholtz equations—i.e., inverting the operators L and $\mathcal{A}$ from the previous section).

One defines a scalar field u on a nested set of grids $u^1, \ldots, u^{N_g}$, where u^1 is the finest grid and u^{N_g} is the coarsest. One then defines a sequence of *coarsening* operators P^k that move points from the full grid u to an individual coarse grid u^k, by suitably averaging. These coarsening operators are defined recursively, by starting with the finest grid and averaging values to each coarser grid sequentially.

In order to solve the system

$$Lu = f, \qquad u\big|_\partial = bc_u, \tag{14}$$

one first computes a sequence of coarsified forcing terms $f^1, \ldots, f^{N_g}$, for instance with $f^k = P^k f$. Then one considers the Laplacian L_1 on the finest (uniform) grid, and solves a sequence of Poisson problems

$$\begin{aligned} L_1 u^{N_g} &= 2^{N_g-1} f^{N_g}, & u^{N_g}\big|_\partial &= bc_u \\ L_1 u^{N_g-1} &= 2^{N_g-2} f^{N_g-1}, & u^{N_g-1}\big|_\partial &= bc(u^{N_g}) \\ &\vdots \\ L_1 u^1 &= f^1, & u^1\big|_\partial &= bc(u^2) \end{aligned} \tag{15}$$

where the operators $bc(u^{k+1})$ return the values on grid u^{k+1} that are on the boundary of the next finer grid k. The scaling terms 2^k on the right-hand side compensate for the different grid spacings on the different domains. Since the single-grid Laplacian L_1 is easily inverted using a sin transform, the overall solution is efficient.

4 Using the command-line tools

This distribution provides a command-line tool (`ibpm`) for running the main solver, as well as a simple tool (`checkgeom`) for checking the syntax of geometry files, and verifying whether they are suitable for a given grid.

4.1 Overview of the main IBPM tool

The command-line tool `ibpm` reads in a geometry file and an initial flow field (or initializes a default flow field of potential flow around the body), and advances the flow forward in time, writing various output files. Parameters are specified as command-line arguments, and for a complete list of options with brief descriptions, run `ibpm -h`.

Grid Parameters related to the grid are specified as follows:

Flag	Description	Default
`-nx <int>`	number of grid cells in x-direction	200
`-ny <int>`	number of grid cells in y-direction	200
`-ngrid <int>`	number of grid levels for multi-domain scheme	1
`-length <real>`	length of finest domain in x-dir	4
`-xoffset <real>`	x-coordinate of left edge of finest domain	-2
`-yoffset <real>`	y-coordinate of bottom edge of finest domain	-2

The grid spacing is `length/dx`, and is always the same in x and y directions, so the height in the y-direction is the x-length times `ny/nx`. Note that the arguments `nx` and `ny` specify the number of *cells* in each direction, so the number of grid points, including boundary points, is $(\texttt{nx}+1) \times (\texttt{ny}+1)$, as shown in Figure 1.

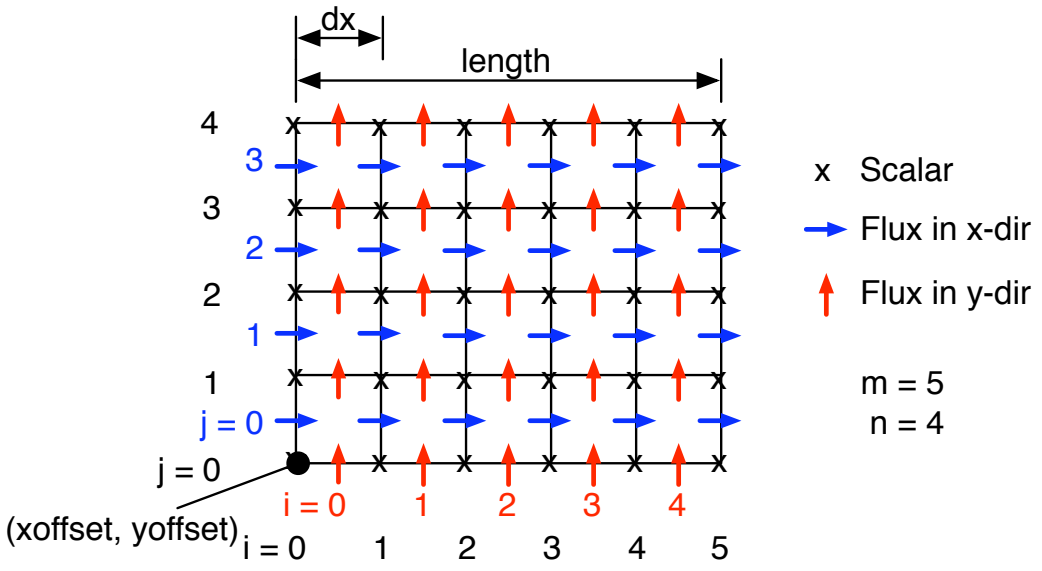

Figure 1: Layout of grid for the finite-volume method. In output files, all variables are given at nodes ($\times$).

Geometry The geometry of the body is read from a file, and the filename is specified as follows:

Flag	Description	Default
`-name <string>`	name of the run	`ibpm`
`-geom <string>`	filename for reading geometry	`<name>.geom`

For instance, if `-name foo` is specified, then the file `foo.geom` is read. If `-name foo -geom bar.geom` is specified, then the file `bar.geom` is read.

The format of the geometry file is discussed in Section 4.2. The run name specified by `-name` also determines the name of various output files (described below).

For instance, to run with a 200×300 grid, for $x \in [-1, 3]$ and $y \in [-3, 3]$ on the finest grid, reading the geometry from the file `myshape.geom`, execute

```
ibpm -nx 200 -ny 300 -length 4 -xoffset -1 -yoffset -3 -geom myshape.geom
```

Solver Parameters related to the flow solver are specified as follows:

Flag	Description	Default
`-Re <real>`	Reynolds number	100
`-dt <real>`	timestep	0.01
`-model <string>`	type of model (linear, nonlinear, adjoint, linearperiodic)	nonlinear
`-baseflow <string>`	base flow for linear/adjoint model	
`-scheme <string>`	timestepping scheme (euler, ab2, rk2, rk3)	rk2
`-ic <string>`	initial condition filename	
`-nsteps <int>`	number of timesteps to compute	250

The formulations of the linearized and adjoint Navier-Stokes equations are described in [1], and if either of these options is used, then a filename must be given, for a restart file from which to load the base flow for the linearization (e.g., `-model linear -baseflow steadystate.bin`). To restart the solver from a given restart file, use the `-ic` flag, as

```
ibpm -ic initial_condition.bin
```

Different timesteppers may be used, including explicit Euler (`euler`), 2nd-order Adams-Bashforth (`ab2`), and second- and third-order Runge-Kutta (`rk2`, `rk3`).

Output As the solution evolves in time, various output files can be written, and their output is specified as follows:

Flag	Description	Default
`-outdir <string>`	directory for saving output	`.`
`-tecplot <int>`	if > 0, write a Tecplot file every n timesteps	100
`-restart <int>`	if > 0, write a restart file every n timesteps	100
`-force <int>`	if > 0, write forces every n timesteps	1

If the directory for output files does not exist, it is created, and all output files are written to this directory. Forces are written to a single file, whose columns are:

```
Timestep        Time        Lift coef        Drag coef
```

Tecplot files are ASCII files readable by the Tecplot visualization software, and restart files are binary files that can be used by `ibpm` as initial conditions, base flows for linearization, etc.

The command-line options specified for the run are also written to a file `<name>.cmd`. Thus, any run can be repeated with

```
. <outdir>/<name>.cmd
```

Periodic flows Additional options are needed when running simulations linearized about a periodic base flow (`-model linearperiodic`):

Flag	Description	Default
`-period <int>`	period of periodic baseflow	1
`-periodstart <int>`	start time of periodic baseflow	0
`-pbaseflowname <string>`	name of periodic baseflow	
`-subbaseflow <0 or 1>`	Subtract baseflow from ic (1/0(true/false))	0
`-numdigfilename <string>`	number of digits for time representation in filename	`%05d`

4.2 Defining the geometry

Here, we describe the format of the geometry file. This file consists of a sequence of commands, all of the form

```
<command> [argument 1] [argument 2] ... [argument n]
```

Whitespace is ignored, and only one command can be given on each line. Comments can be included, using the comment character `#`.

A *geometry* consists of one or more *rigid bodies*. A rigid body is a collection of points, specified by commands, which may be given in any order. The following example illustrates the available commands:

```
name Name of this object
center x y  # location of the center of the object
point x y   # add a point at this location
point x y
point x y
line x1 y1 x2 y2 dx        # add a line, spacing dx
line_n x1 y1 x2 y2 npts    # add a line with npts points
circle xc yc radius dx     # add a circle, specifying spacing
circle_n xc yc radius npts # add a circle, specifying num points
raw naca0012.dat           # read a list of points from a file
motion fixed x y theta     # assign a particular motion
```

The command `line` constructs a line between points (x_1, y_1) and (x_2, y_2), with spacing `dx` between points. The command `line_n` constructs a similar line, but specifying the number of points instead of the spacing. The `circle` command behaves similarly, constructing a circle with the given center (x_c, y_c) and radius, where the spacing between points on the circle is approximately `dx`.

Note about spacing of boundary points The spacing between points on the boundary of a body should be smaller than the grid spacing, but not too small. If the spacing is too coarse, fluid will "leak" through the boundary, and if the spacing is too fine, the solver may not converge.

Motion The points that make up a rigid body may be translated together by specifying a `motion`. The simplest motion is just translation and rotation, specified by

```
motion fixed x y theta
```

which translates the body by (x, y), and rotates about the center of the body by the angle `theta` (expressed in radians). This may be used, for instance, to incline an airfoil at a desired angle of attack. The axis of rotation is defined using the `center` command to specify its x- and y-coordinates. One may also specify a pitch-plunge motion, as

```
motion pitchplunge amp1 freq1 amp2 freq2
```

which specifies a translation and rotation given by

$$\begin{aligned} x(t) &= 0 \\ y(t) &= A_2 \sin(2\pi f_2 t) \\ \theta(t) &= A_1 \sin(2\pi f_1 t) \end{aligned}$$

where A_1, f_1 are specified by `amp1`, `freq1`, etc.

Overall geometry The overall geometry is a collection of rigid bodies, for instance specified as follows:

```
name My Geometry
body Flat Plate
    line 0 0 1 0 0.1  # points on a line, spacing approximately 0.1
    center 0.25 0     # center at quarter chord
    motion fixed 0 0 0.3 # 0.3 radians angle of attack
end
body Large Circle
    circle 2 3 5 0.1 # Points on a circle
                     # default center is (2,3)
end
body Airfoil
    raw naca0012.in  # Read in a raw data file
end
```

The `name` command is optional, and whitespace is ignored.

4.3 Tool for checking geometries

The program `checkgeom` may be used to validate the syntax of geometry input files, and to plot the corresponding boundary points on a specified grid. This is useful both to check that the geometry looks generally as expected before running a large simulation, but more importantly, it can be used to check for "leaks": as stated previously, if the boundary points are too coarsely spaced, relative to the underlying uniform grid, then the fluid can penetrate the boundary.

The options for `checkgeom` are as follows:

`-h`	print a help message and exit	
`-nx <int>`	number of gridpoints in x-direction	200
`-ny <int>`	number of gridpoints in y-direction	200
`-ngrid <int>`	number of grid levels for multi-domain scheme	1
`-length <real>`	length of finest domain in x-dir	4
`-xoffset <real>`	x-coordinate of left edge of finest domain	-2
`-yoffset <real>`	y-coordinate of bottom edge of finest domain	-2
`-geom <string>`	filename for reading geometry	ibpm.geom
`-o <string>`	filename for writing Tecplot file	[none]

The default behavior is to read in a geometry file and exit normally if the file was successfully parsed. In this case, the grid parameters are irrelevant.

If the name of an output file is given with the `-o` flag, then as long as the file is successfully parsed, a Tecplot file is written, showing how the boundary points are regularized to the grid. This Tecplot file can then be used to check for "leaks."

5 Using the IBPM library

The algorithms used by the main `ibpm` code are made available in a library that can be used by other codes, for instance to write wrapper routines around the main timestepper, or to perform post-processing tasks. The library is written in an object-oriented style, with the various algorithms and data structures provided by different classes. Here, we give an overview of the various classes and their roles. For a detailed description of their interfaces, consult the Doxygen documentation, or the corresponding header files.

An overview of the classes and their interactions is shown in Figure 2.

5.1 Data structures

The main data structures consist of a `Grid` that specifies the grid characteristics, a `Geometry` that specifies the configuration of the bodies and the locations of their boundary points, and variables that take on values either on the grid (`Scalar`, `Flux`), or on the boundary points (`BoundaryVector`).

A `Grid` defines the number of grid points in x- and y-directions, as well as the physical dimensions, and the number of grids to be used in a multi-domain solver.

Two classes, `Scalar` and `Flux`, define variables that take on values on the grid. In particular, a `Scalar` variable (such as vorticity and streamfunction) takes on a single value at each node in Figure 1. A `Flux` is vector-valued: the x-component is defined as the flux through vertical edges in Figure 1 (blue arrows), while the y-component is defined as the flux through horizontal edges (red arrows).

A `Geometry` defines the configuration of the bodies and the locations of their boundary points. In particular, a `Geometry` contains a number of `RigidBody` objects, each with an associated `Motion` that prescribes the motion of that body in time.

A `BoundaryVector` defines a vector-valued function that takes on values at the boundary points defined by a particular `Geometry`. For instance, the x- and y-forces

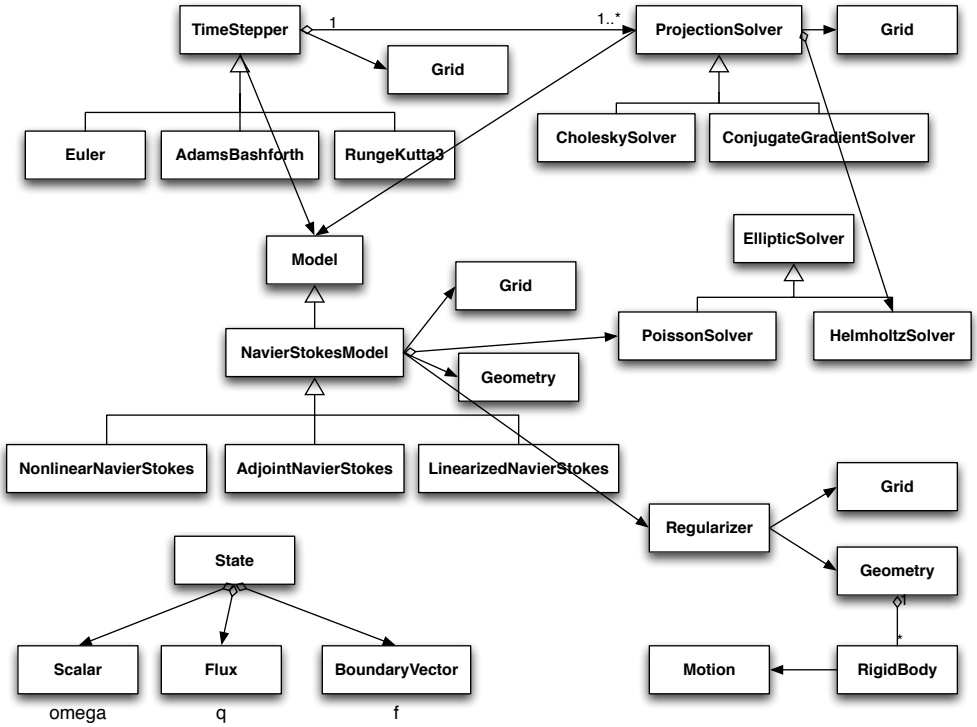

Figure 2: Overview of the various classes and their interactions.

at the boundary points are described by a `BoundaryVector`, as are the velocities at the boundary points, in the case of a moving body.

Finally, a `State` consists of the complete flow state at any time. In particular, a `State` has components

`Scalar omega`	vorticity at each node
`Flux q`	u- and v-velocity flux through cell edges
`BoundaryVector f`	forces at each of the boundary points
`int timestep`	current timestep
`double time`	current time

5.2 Mapping between grids and boundary points

Sometimes one needs to map functions defined on boundary points onto a grid, or vice-versa, restrict values from a grid onto boundary points. In Section 3, these operators were denoted E^T and E, respectively. In this library, the class that implements these operations is called `Regularizer`, and the sole purpose of the `Regularizer` is to compute these mappings, for a particular `Grid` and `Geometry`.

5.3 Models

Various different sets of equations may be solved by the library, and each is specified as a type of `Model`. Here, a `Model` specifies a set of equations of the form

$$\begin{aligned} \frac{d\omega}{dt} + Bf &= \alpha L\omega + N(x) \\ C\omega &= b, \end{aligned} \tag{16}$$

where ω is a `Scalar`, f and b are `BoundaryVectors`, x is a `State`, L is the Laplacian, and the scalar α and operators B, C, and N are specified by the `Model`.

A `NavierStokesModel` is a subclass of `Model` that defines some version of the Navier-Stokes equations, for a particular `Grid`, `Geometry`, Reynolds number, and base flow. Each `NavierStokesModel` contains a `PoissonSolver` (see Section 5.6), for computing the streamfunction ψ from the vorticity ω, using

$$\nabla^2\psi = -\omega,$$

from which the corresponding velocity fluxes q can then be computed as $q = \operatorname{curl}\psi + U_b$, where U_b is a base flow. The `NavierStokesModel` is responsible for all of this.

`NavierStokesModel` is an abstract base class, and may not be instantiated. Its subclasses may, however, be instantiated, and are described below. Note that, according to the formulation in [1], the only difference between the nonlinear, linearized, and adjoint models is in the nonlinear term.

`NonlinearNavierStokes` defines the nonlinear Navier-Stokes equations.

`LinearizedNavierStokes` defines the Navier-Stokes equations linearized about an equilibrium point (steady solution).

`AdjointNavierStokes` defines the adjoint of the Navier-Stokes equations linearized about an equilibrium point.

`LinearizedPeriodicNavierStokes` defines the Navier-Stokes equations linearized about a periodic orbit.

5.4 Timesteppers

Different types of timesteppers are available as well, for advancing the `State` forward in time. Each `Timestepper` is associated with a particular `Grid` and `Model`, and a specific timestep h.

In particular, a `Timestepper` advances governing equations in the form (16), where ω is a `Scalar`, f is a `BoundaryVector`, and x is a `State`. Here, L is the Laplacian, and the other operators are defined in the associated instance of `Model`.

The sole purpose of a `Timestepper` is to provide a method (`advance`) to march a `State` object forward to the next timestep. For each type of timestepper, the projection method (13) is used to solve for the next timestep. This projection method is handled by a separate class, `ProjectionSolver`, and each `Timestepper` contains one or more `ProjectionSolver` instances.

`Timestepper` is an abstract base class, so only its subclasses may be instantiated. These subclasses are described below.

The `Euler` timestepper uses the Crank-Nicolson scheme to discretize the linear terms, and explicit Euler for the nonlinear term. The resulting equations have the form

$$(1 - \frac{\alpha h}{2} L)\omega^{n+1} + hBf = (1 + \frac{\alpha h}{2} L)\omega^n + hN(x^n) \tag{17}$$

$$C\omega^{n+1} = b_{n+1} \tag{18}$$

where h is the timestep. Here, in the notation of (7),

$$\mathcal{A} = 1 - \frac{\alpha h}{2} L \tag{19}$$

$$\mathcal{B} = hB \tag{20}$$

$$\mathcal{C} = C \tag{21}$$

$$a = (1 + \frac{\alpha h}{2} L)\omega^n + hN(x^n) \tag{22}$$

$$b = b_{n+1}, \tag{23}$$

where quantities on the right-hand side are specified by the associated `Model`.

The `AdamsBashforth` timestepper uses Crank-Nicolson for linear terms, and second-order Adams Bashforth for the nonlinear terms. The resulting equations have the form

$$(1 - \frac{\alpha h}{2} L)\omega^{n+1} + hBf = (1 + \frac{\alpha h}{2} L)\omega^n + \frac{h}{2}\big(3N(x^n) - N(x^{n-1})\big) \tag{24}$$

$$C\omega^{n+1} = b_{n+1} \tag{25}$$

where h is the timestep. Here, in the notation of (7),

$$a = (1 + \frac{\alpha h}{2} L)\omega^n + \frac{h}{2}(3N(x^n) - N(x^{n-1}) \tag{26}$$

with other parameters the same as `Euler`. Because this is a multi-step scheme, an `AdamsBashforth` instance remembers the previous state that it computed. This previous state can also be specified, via the method `setPreviousState`.

The `RungeKutta2` timestepper uses Crank-Nicolson for the linear terms, and a second-order Runge-Kutta scheme for the nonlinear terms. In particular, this class uses the scheme given by Peyret, p. 148[3] (for Peyret's parameters $\alpha = 1$, $\beta = 1/2$):

$$(1 - \frac{\alpha h}{2}L)x_1 + hBf_1 = (1 + \frac{\alpha h}{2}L)x^n + hN(x^n) \tag{27}$$

$$Cx_1 = b_{n+1} \tag{28}$$

$$(1 - \frac{\alpha h}{2}L)x^{n+1} + hBf^{n+1} = (1 + \frac{\alpha h}{2}L)x^n + \frac{h}{2}\big(N(x^n) + N(x_1)\big) \tag{29}$$

$$Cx^{n+1} = b_{n+1}. \tag{30}$$

The `RungeKutta3` timestepper uses Crank-Nicolson for linear terms, and a 3rd-order Runge-Kutta scheme for nonlinear terms. In particular, this class uses the scheme given by Peyret, p. 149[3]:

$$Q_1 = hN(x^n) \tag{31}$$

$$(1 - \frac{\alpha h}{6}L)x_1 + \frac{h}{3}Bf_1 = (1 + \frac{\alpha h}{6}L)x^n + \frac{1}{3}Q_1 \tag{32}$$

$$Cx_1 = b_{n+1/3} \tag{33}$$

$$Q_2 = -\frac{5}{9}Q_1 + hN(x_1) \tag{34}$$

$$(1 - \frac{5\alpha h}{24}L)x_2 + \frac{5h}{12}Bf_2 = (1 + \frac{5\alpha h}{24}L)x_1 + \frac{15}{16}Q_2 \tag{35}$$

$$Cx_2 = b_{n+3/4} \tag{36}$$

$$Q_3 = -\frac{153}{128}Q_2 + hN(x_2) \tag{37}$$

$$(1 - \frac{\alpha h}{8}L)x^{n+1} + \frac{h}{4}Bf^{n+1} = (1 + \frac{\alpha h}{8}L)x_2 + \frac{8}{15}Q_3 \tag{38}$$

$$Cx^{n+1} = b_{n+1}. \tag{39}$$

5.5 Projection solver

The `ProjectionSolver` class implements the projection algorithm (13). In particular, it solves a system of the form

$$(1 - \frac{\alpha\beta}{2}L)x + \beta Bf = a \tag{40}$$

$$Cx = b, \tag{41}$$

where α and the operators B and C are determined by an associated `Model` instance, and the parameter β is specified to the `ProjectionSolver` instance.

`ProjectionSolver` is an abstract base class, as different strategies may be employed to solve the middle equation of (13), depending on whether the operator on

the left-hand side is constant, or time-dependent (i.e. when the body is moving, so the operators $\mathcal{B}, \mathcal{C}$ vary in time).

`CholeskySolver` solves the middle equation of (13) directly, using a Cholesky factorization, that is performed once and stored. The Cholesky factorization may be saved to a file and loaded in to save time at startup. This solver is much more efficient when the bodies are stationary, but if the bodies are moving, the factorization would need to be performed at each timestep, and in this case an iterative solver is faster.

`ConjugateGradientSolver` solves the middle equation of (13) iteratively, using a conjugate gradient method, which is iterated to a specified tolerance.

5.6 Elliptic solvers

Elliptic equations such as Poisson or Helmholtz equations need to be solved at several places, and these solutions are computed by an `EllipticSolver` instance. In particular, the `EllipticSolver` implements the multiple-domain solution described in Section 3. A related class, `EllipticSolver2d`, solves a corresponding equation on a single, uniform grid (using a sin transform), and each `EllipticSolver` instance contains a number of `EllipticSolver2d` instances, one for each level of grid.

`EllipticSolver` is an abstract base class, and only its subclasses may be instantiated. These subclasses are discussed below.

`PoissonSolver` solves a Poisson equation of the form

$$Lu = f$$

with zero Dirichlet boundary conditions on u, where L is the Laplacian, and f is given. The solver does some initial setup (computing eigenvalues of L at each grid level), and then can be used many times for different values of f.

`HelmholtzSolver` solves a Helmholtz equation of the form

$$(1 + \alpha L)u = f$$

with zero Dirichlet boundary conditions on u, where L is the Laplacian, α is specified once to the solver, and the solver can be used many times for different values of f.

References

[1] S. Ahuja and C. W. Rowley. Low-dimensional models for feedback stabilization of unstable steady states. AIAA Paper 2008-553, 46th AIAA Aerospace Sciences Meeting and Exhibit, January 2008.

[2] T. Colonius and K. Taira. A fast immersed boundary method using a nullspace approach and multi-domain far-field boundary conditions. *Comp. Meth. Appl. Mech. Eng.*, 197(25-28):2131–46, 2008.

[3] R. Peyret. *Spectral Methods for Incompressible Viscous Flow*, volume 148 of *Applied Mathematical Sciences*. Springer-Verlag, 2002.

[4] K. Taira and T. Colonius. The immersed boundary method: A projection approach. *J. Comput. Phys.*, 225(2):2118–2137, August 2007.