Cell-centered Poisson Solvers

Brian Gunney

Introduction

SAMRAI provides the classes for solving Poisson's equation on a single level (using hypre) or a hierarchy (using the fast adaptive composite, or FAC, algorithm). These classes solve the general equation of the form

$$C(\boldsymbol{x})+\nabla\cdot D(\boldsymbol{x})\nabla u(\boldsymbol{x})=f(\boldsymbol{x})$$

for $u(\boldsymbol{x})$, where $C(\boldsymbol{x})$ is a scalar field, $D(\boldsymbol{x})$ is the diffusion coefficient and $f(\boldsymbol{x})$ is the source term. The The solver supports the Robin boundary condition, which is any that can be writen in the form

$$\alpha u+\beta\frac{\partial u}{\partial n}=\gamma$$

or (using two parameters)

$$a u+(1-a)\frac{\partial u}{\partial n}=g$$

where n is the coordinate in the direction of the outward normal on the boundary. Note that

$a=\frac{\alpha}{\alpha+\beta}$ and $g=\frac{\gamma}{\alpha+\beta}$.

$a u+(1-a)\frac{\partial u}{\partial n}=g$, where n is the coordinate along the outward normal. This is a generalization of the Dirichlet (a=1) and Neumann (a=0) boundary conditions.

The discretization is a standard central-difference, cell-centered finite-volume. This implies that $C(\boldsymbol{x})$, $f(\boldsymbol{x})$ and $u(\boldsymbol{x})$ are cell-centered quantities, and $D(\boldsymbol{x})$ is side-centered.

Simpler forms of the partial differential equation (PDE) can be solved, and different optimizations are made to take advantage of those cases. For example, $C(\boldsymbol{x})$ and $D(\boldsymbol{x})$ may be constants and $C(\boldsymbol{x})$ may be zero.

The class `solv_CellPoissonHypreSolverX` solves a single-level problem, and the class `solv_CellPoissonFACSolverX` solves the problem on a hierarchy. This document shows the basics of using these classes. It covers the possible settings for specifying the PDE and controlling the solver algorithm, calling the `solveSystem` methods to perform the solve, getting data on the solve, and how to set up an input file. We follow with some examples at the end. The solver classes documented here references the FAC preconditioner class (`solv_FACPreconditionerX`) and the Robin boundary condition class (`solv_RobinBcCoefStrategyX`), which are documented separately.

Providing the Robin Boundary Condition Coefficients

To set up code to use `solv_CellPoissonHypreSolverX` or `solv_CellPoissonFACSolverX`, you need decide whether to use the internal boundary condition implementation or provide an external one. For this document to be sufficiently general, we will assume an external implementation is used. To keep things simple, we choose the library-provided implementation `solv_LocationIndexRobinBcCoefsX`. In a later section, we will describe the choices for specifying the boundary condition coefficients. Provide a pointer to this implementation using the member function `solv_CellPoissonHypreSolverX::setBcObject()`. You may choose one of the general implementations in the library or implement your own.

The `solv_LocationIndexRobinBcCoefsX` implementation is appropriate for problems where the coefficients are determined completely by the location index of the boundary box. This covers, among others, the case of parallelpiped domains where each side of the parallelpiped is set to some uniform boundary condition. For each location index, one may specify uniform Dirichlet boundary values, uniform Neumann boundary values or uniform values of a and g.

Usage of the Single-level Hypre-Poisson Solver

After setting up your code as described above, solving the Poisson equation is simple.

1. First, initialize the solver object to an existing hierarchy using `solv_CellPoissonHypreSolverX::initializeSolverState()`. You can specify the hierarchy and level number (which defaults to level zero) to solve on.
2. Then use `solv_CellPoissonHypreSolverX::setBcObject()` to tell it about the Robin boundary condition object you have set up.
3. Make sure that ghost cells at the coarse-fine boundaries are initialized with the appropriate values. For example, refine the coarse grid values into the ghost cells using a refine schedule.
4. Then use `solv_CellPoissonHypreSolverX::setMatrixCoefficients()` to set up the matrix coefficients. You can specify the descriptor indices of $C(\boldsymbol{x})$ and $D(\boldsymbol{x})$.
5. Finally, call `solv_CellPoissonHypreSolverX::solveSystem()` to solve the system. You have to specify the descriptor indices of the cell-centered solution and the cell-centered source function. The cell-centered solution data will be modified.

Hypre-Poisson Solver Example

Here is an example following the steps outlined in the previous section, on using `solv_CellPoissonHypreSolverX`.

```
/*
  Initialize the solver.
*/
d_poisson_hypre.initializeSolverState( hierarchy, ln );

/*
  solv_LocationIndexRobinBcCoefsX is an implementation of the
  Robin boundary condition coefficient strategy class.  It allows
  one to specify a uniform boundary condition for each location
index.
  We set Dirichlet values of 0.0 and 10.0 on the min x and max x
sides
  and zero slope on the min y and max y sides.
*/
solv_LocationIndexRobinBcCoefX bc_coefs;
bc_coefs.setBoundaryValue(0, 0.0);
bc_coefs.setBoundaryValue(1, 10.0);
bc_coefs.setBoundarySlope(2, 0.0);
bc_coefs.setBoundarySlope(3, 0.0);
d_poisson_hypre.setBcObject( bc_object );

/*
  The solver can now set up the matrix.
  The solv_PoissonSpecifications object holds the values of C and D
  in Poisson's equation.  By default, C=0 and D=1, leading to
Laplace's
  equation.  These values can be changed through the public
```

```
interfaces
  of solv_PoissonSpecifications.
*/
solv_PoissonSpecifications spec("Laplace");
d_poisson_hypre.setMatrixCoefficients(spec);

/*
  Solve the system.
  solution is the patch data index of the solution u.
  souce is the patch data index of the source f.
*/
int solver_ret = d_poisson_hypre.solveSystem( solution,
                                              source );
```

Multi-level FAC-Poisson Solver Settings

The solv_CellPoissonFACSolverX class uses the following methods for setting the parameters C and D:

```
setDConstant(double value)
setDPatchDataId(int id)
setCConstant(double value)
setCPatchDataId(int id)
```

To set the diffusion coefficient $D(\boldsymbol{x})$, use `setDConstant(double value)` or setDPatchDataId (int id), depending on whether it is a constant or spatially varying and stored on the mesh. Similarly, use `setCConstant(double value)` or `setCPatchDataId(int id)`, to set $C(\boldsymbol{x})$.

The methods specifying the algorithm parameters, corresponding input parameter name and the default settings are:

Methods	***Input name***	***Default setting***
`setPresmoothingSweeps(int num_pre_sweeps)`	`num_pre_sweeps`	1
`setPostsmoothingSweeps(int num_post_sweeps)`	`num_post_sweeps`	1
`setMaxCycles(int max_cycles)`	`max_cycles`	10
`setResidualTolerance(double residual_tol)`	`residual_tol`	1.00E-006
`setCoarseFineDiscretization (const string &coarsefine_method)`	`coarse_fine_discretization`	“Ewing”
`setCoarsestLevelSolverChoice (const string &choice)`	`coarsest_level_solver_choice`	“hypre” or “redblack”
`setCoarsestLevelSolverTolerance (double tol)`	`coarsest_level_solver_tolerance`	1e-10 or 1e-8
`setCoarsestLevelSolverMaxIterations(int max_iterations)`	`coarsest_level_solver_max_iterations`	20 or 500
`setUseSMG(bool use_smg)`	`use_smg`	FALSE
`setProlongationMethod(const string &prolongation_method)`	`prolongation_method`	“CONSTANT_REFINE”

- The pre- and post-smoothing sweeps refer to the amount of smoothing used in the FAC cycle.
- The max cycles refer to the maximum number of FAC cycles to take.
- The coarse-fine discretization refers to how the PDE is discretized at the coarse-fine boundary. Other than one specific exception, the argument must be the name of a refinement operator is used (i.e., "`LINEAR_REFINE`", "`CONSTANT_REFINE`", etc.). The coarse-fine discretization results from using the specified refinement to get the fine-grid ghost cell, followed by a normal stencil applied across the fine patch boundary. This may seem reasonable, but it results in a discretization that is specified implicitly by the refinement operator. These discretizations may have unanticipated, though usually subtle, numerical behaviors. In the exceptional case, the string "`Ewing`" specifies the coarse-fine discretization of Ewing, Lazarov and Vassilevski. This discretization tends to give better accuracy at the coarse-fine boundaries and is the default.
- Methods beginning with "`setCoarsestLevelSolver...`" refer to the coarsest level solver. By default, hypre is used to solve the coarsest level if it is available, otherwise, red-black Gauss-Seidel iterations are used.
- The different default tolerance and max iterations for the coarsest level reflect the fact that the hypre solver converges much faster than the Gauss-Seidel algorithm.
- The usage of hypre's SMG (semicoarsening multigrid) algorithm is set by the function `setUseSMG`. This setting has effect only when hypre is chosen as the coarsest level solver. If SMG usage is false, hypre's PFMG (parallel semicoarsening multigrid) algorithm is used.
- The prolongation method should be the name of a refine operator, such as "`CONSTANT_REFINE`" or "`LINEAR_REFINE`". Be aware that linear refinement (or any refinement that involves the creation and filling of temporary levels) requires an `solv_RobinBcCoefStrategyX` implementation that can fill non-hierarchy data). The default implementation used (`solv_SimpleCellRobinBcCoefsX`) does not satisfy this requirement.

Multi-level FAC-Poisson Usage

Once the solver object is set up using the above methods, the method

```
bool solveSystem( int solution,
                  int rhs,
                  tbox_Pointer<hier_PatchHierarchyX> hierarchy,
                  int coarse_ln=-1,
                  int fine_ln=-1 )
```

performs the solve on the hierarchy and level range specified in the arguments. The integers `solution` and `rhs` are patch data indices for the solution and the right hand side, respectively. The solution data must have a ghost cell width of at least one. Since the solver is for a scalar equation, only the first depth is used. The solver returns true if convergence to the specified level is reached.

The above function call is the simplest way to perform a solve. It initialize the solver state, which is dependent on the hierarchy configuration, level range, boundary condition types, etc. After performing the solve, it deallocates the solver state. When performing multiple solves with different right-hand-side values, efficiency is improved if you take steps to preserve the solver state between solves. This is done with two functions:

```
void initializeSolverState( const int solution,
                            const int rhs,
                            tbox_Pointer<hier_PatchHierarchyX>
hierarchy
                            const int coarse_level=-1,
                            const int fine_level=-1 )
void deallocateSolverState()
```

In between these two function calls, you can perform any number of solves using

```
bool solveSystem(const int solution,
                 const int rhs)
```

The solution and rhs used in the solve may be different from those used to initialize the solver state and different each time the caller is solved. The solver state initializer uses the patch data indices to set up

matching temporary memory.

Note that it is an error to mix up the two `solveSystem` methods, as one expects an uninitialized state and one expects an initialized state.

After a solve, the number of FAC iterations, the residual norm and the convergence factors can be retrieved by the functions

```
int getNumberIterations() const
void getConvergenceFactors(double &avg_factor,
                           double &final_factor) const
double getResidualNorm() const
```

The convergence factor is the factor by which the residual is reduced by one FAC iteration. The average factor is that which, when applied the number of iterations used gives the same overall reduction, while the final factor is that of the last iteration taken. The residual norm is the RMS norm of the final residual.

Multi-level FAC-Poisson Examples

```
int ln;
for ( ln=0; ln<=hierarchy->getFinestLevelNumber(); ++ln ) {
  /*
    Fill in the initial guess and Dirichlet boundary condition
    data.  For this example, we want u=0 on all boundaries.
    The easiest way to do this is to just write 0 everywhere,
    simultaneous setting the boundary values and initial guess.
  */
  Pointer<PatchLevel> level = hierarchy->getPatchLevel(ln);
  PatchLevel::Iterator ip(level);
  for ( ; ip; ip++ ) {
    Pointer<Patch> patch = level->getPatch(*ip);
    Pointer<CellData<double> > data = patch->getPatchData
(comp_soln_id);
    data->fill(0.0);
  }
}
solver.setBoundaries( "Dirichlet" );

/*
  Set up solver object.
  The problem specification is set using the
  PoissonSpecifications object then passed to the solver
  for setting the coefficients.
*/
solver.setCConstant(0.0);
solver.setDConstant(1.0);
solver.initializeSolverState( comp_soln_id,
                    rhs_id,
                    hierarchy,
                    0,
                    hierarchy->getFinestLevelNumber() );

/*
  Solve the system.
```

```
*/
pout << "solving..." << endl;
bool solver_ret;
solver_ret = solver.solveSystem( comp_soln_id ,
                         rhs_id );

/*
  Present data on the solve.
*/
double avg_factor, final_factor;
solver.getConvergenceFactors(&avg_factor, &final_factor);
if ( solver_ret ) pout << " converged\n";
else pout << " NOT converged\n";
pout << "   iterations: " << solver.getNumberIterations() << "\n"
     << "   residual: " << solver.getResidualNorm() << "\n"
     << "   average convergence: " << avg_factor << "\n"
     << "   final convergence: " << final_factor << "\n"
     << flush;

/*
  Deallocate state.
*/
solver.deallocateSolverState();
```

Boundary Condition Options

As mentioned above, the Robin boundary conditions are supported. These boundary conditions are specified by two coefficients, a and g. Technically, one provides an implementation of `solv_RobinBcCoefStrategyX` through which the solvers obtain the coefficients. But in most cases, one of the library-provided implementations would suffice. Once you have an implementation, use the method setBcObject(), described above, to specifiy it.

For the se solvers, there is one additional choice, which is aimed at providing compatibility to codes that were written for SAMRAI's older Poisson solvers: using the internal boundary condition implementation. The internal implementation is the library-provided `solv_SimpleCellRobinBcCoefsX` class, whose primary interface, the `setBoundaries()` method, is duplicated in the solver classes.

The boundary condition is specified by calling

```
setBoundaries( const string &boundary_type,
           const int fluxes=-1,
           const int flags =-1,
           int *bdry_types=NULL );
```

The boundary conditions specified as the string argument "boundary_type." The boundary type argument can be `"Dirichlet"`, `"Neumann"`, or `"Mixed"`.

If using Dirichlet boundary conditions, then before the solver is called, the storage for the unknown must have a layer of ghost cells at least one cell wide that includes the Dirichlet boundary values.

If using Neumann boundary conditions, then before the solver is called, the outerface boundary flux data must be set for the Neumann conditions. The fluxes argument gives the patch data index of this flux data.

The mixed boundary type is for a mixture of Dirichlet and Neumann boundary conditions at the physical domain boundary. The `fluxes` argument gives the patch data index of the outerface data that specifies the flux data for the Neumann conditions. The `flags` array is an outerface data array of integer flags that specifies whether Dirichlet (`flag == zero`) or Neumann (`flag == one`) conditions are to be used at

a particular cell boundary face. Note that the flag data must be set before the solver state is initialized. The `bdry_types` argument can be used if the boundary conditions are mixed but one or more of the faces of the physical boundary are entirely either Dirichlet or Neumann boundaries. The `bdry_types` argument should be an array of 2*NDIM integers, specifying the boundary conditions on each side of the physical domain. It should be ordered {x_lo, x_hi, y_lo, y_hi, z_lo, z_hi}, with the values for each face being 0 for Dirichlet conditions, 1 for Neumann conditions, and 2 for mixed boundary conditions. The `bdry_type` argument is never required, but if used it can sometimes make the solver more efficient.

As mentioned above, the default boundary condition implementation does not allow linear refinement in the prolongation of error in its most general usage. If you chose to provide an external implementation of `solv_RobinBcCoefStategyX`, the method `setBcObject(const solv_RobinBcCoefStrategyX *bc_object)` is used to set it. Some relatively easy to use implementations are available in the library. For example, problems where the coefficients are strictly a function of the boundary box's location index (see `hier_BoundaryBoxX::getLocationIndex()`), is supported by the class `solv_LocationIndexRobinBcCoefsX`.

Acknowledgements:

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence-Livermore National Laboratory under contract No. W-7405-Eng-48. Document UCRL-TM-202156.