Robin Boundary Conditions for Poisson Solvers

Brian T.N. Gunney

Introduction

The Poisson equation solvers in SAMRAI support a general boundary condition known as the Robin boundary condition. A Robin boundary condition is one 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 $g = \frac{\gamma}{\alpha+\beta}$.

The Robin boundary condition is a generalization of the Dirichlet boundary condition ($a=1$) and the Neuman boundary condition ($a=0$). In the most general case, $a \in [0,1]$.

The method to set the coefficients a and g are abstracted in the interface class `RobinBcCoefStrategy`, an implementation of which is required by the scalar Poisson solver classes for computing boundary condition information. This approach allows the flexibility needed to make the solvers robust but requires you to inherit and implement an abstract base class. If you are not interested in this much flexibility, please consider using one of the provided implementations, such as

- `SimpleCellRobinBcCoefs`
- `LocationIndexRobinBcCoefs`
- `GhostCellRobinBcCoefs`

which are described below in section “Provided Implementations”.

Note that `RobinBcCoefStrategy` is intended for communicating Robin boundary conditions to the scalar Poisson solver classes. It is not necessarily intended for setting boundary conditions outside this context. The SAMRAI library provides other methods for general boundary condition setting.

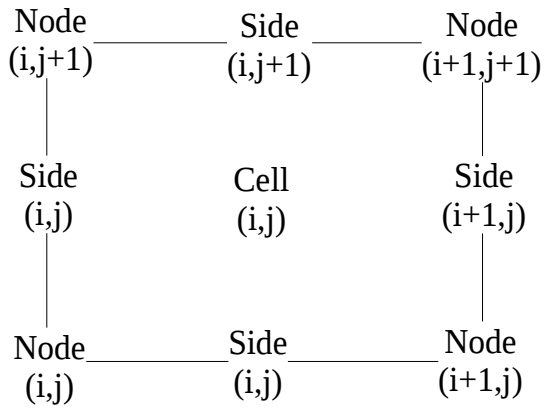
This document covers how to implement `RobinBcCoefStrategy` to set the coefficients (once you've determined what they should be) and how to use some library-provided implementations of `RobinBcCoefStrategy`. After presenting some examples on how to implement and use the strategy class for cell-centered data, we describe some library-provided implementations which may be used without having to write code to inherit the strategy class.

Code Setup

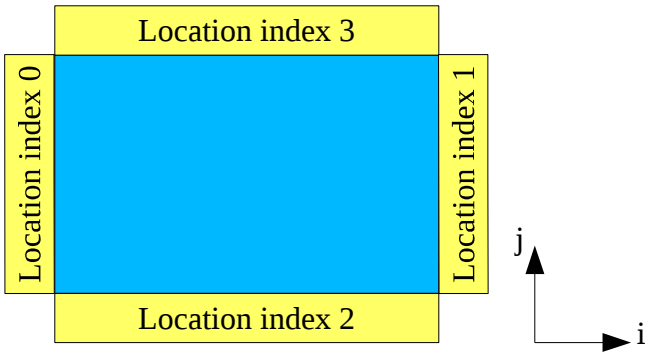
In this section, we describe how to set up a concrete class implementing `RobinBcCoefStrategy`. We first review the relationship between the indices of a cell and the indices of the nodes, edges and sides of the cell, as this is important to determining the discrete locations at which to set the boundary coefficients. Then we describe the virtual methods to be implemented and give example implementations.

The Robin equations are applied at discrete locations on the patch boundaries rather than at the ghost cell centers, even though boundary boxes are defined by a box of ghost cells. The discrete locations

correspond to the alignment of grid data and each has specific indices. To implement `RobinBcCoefStrategy`, it is important to understand the relationship between the indices of variously aligned data and the indices of the cells defining a boundary box. In 2D they are described by this figure of a cell `(i,j)` and its neighboring nodes and sides:



Observe that the side or node index in the direction of decreasing index is always the same as the cell index. This simple rule also applies in 3D. The side of the ghost cell that touches the patch boundary depends on the location index of the boundary, as illustrated in this figure,



In 3D, location indices 4 and 5 refer to the minimum and maximum k sides, respectively. From these rules, it should be clear that for a boundary (ghost) cell `(i,j,k)` in a boundary box having a given location index, the side touching the patch boundary has the following indices:

Location Index	***Side Indices***
0	`(i+1,j,k)`
1	`(i,j,k)`
2	`(i,j+1,k)`
3	`(i,j,k)`
4	`(i,j,k+1)`
5	`(i,j,k)`

The indices of nodes on the patch boundary follows the same picture, but note that while the number of sides on the patch boundary equals the number of cells in the boundary box, the number of nodes will be greater. This information should be sufficient to determine the indices of the sides or nodes on any boundary specified by a boundary box.

Two pure virtual functions to be implemented are:

1. `setBcCoefs`: provide the discrete (side- or node-based) values for a and g
2. `numberOfExtensionsFillable`: state the maximum number of discrete points past the corner of the patch where the coefficients can be provided. (This is a limit used to truncate the boundary boxes defined on the mesh. The implementation is not always required to fill all the extensions specified here.)

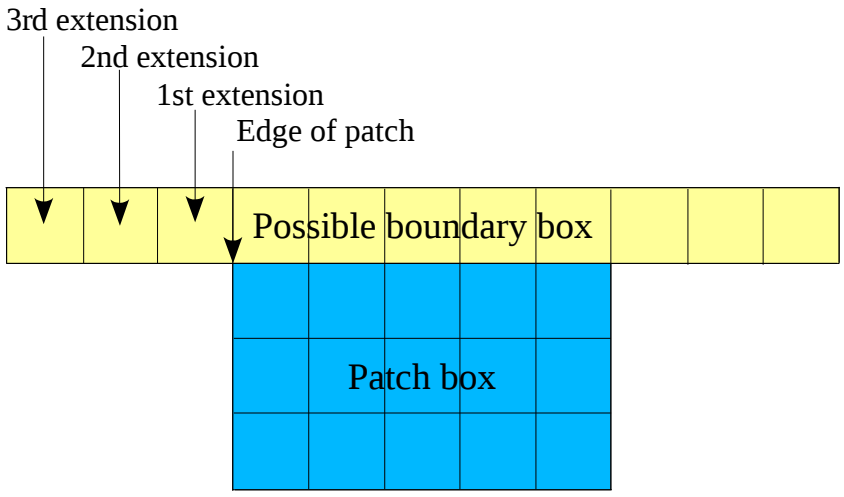
We will discuss each of these functions in this section.

To provide an implementation of `RobinBcCoefStrategy`, one must compute discrete values of a and g for a boundary box and write them to arrays. This is done in the implementation of

```
void RobinBcCoefStrategy::setBcCoefs(
   const boost::shared_ptr<pdat::ArrayData<double> > &acoef_data,
   cosnt boost::shared_ptr<pdat::ArrayData<double> > &bcoef_data
   const boost::shared_ptr<pdat::ArrayData<double> > &gcoef_data,
   const boost::shared_ptr<hier::Variable> &variable,
   const hier::Patch &patch,
   const hier::BoundaryBox &boundary_box,
   const double fill_time=0 ) const;
```

This method is given a patch, a boundary box, the fill time, the variable for which the coefficients are intended and two arrays in which to store a and g. This method should use `variable` to determine the alignment of the discrete points at which the Robin formula are applied. These points are determined by projecting onto the boundary the alignment of `variable`. Thus for cell variables, the formula is applied on sides and for node variables, the formula is applied on nodes. The parameter `variable` may also be used to determine, in a multi-variable implementation, which variable to set the coefficients for. (In practice, the usage of the implementation may be limited enough to allow you to assume the alignment and variable without performing any check.) The alignment and the boundary box thus determine all locations and indices where the coefficients are required. The arrays `acoef_data` and `gcoef_data`, in which to store the coefficients, are allocated exactly for these. Naturally, the coefficients are only requested for boundary boxes of codimension one, as that is the only boundary type associated with a boundary normal vector. If either `acoef_data` or `gcoef_data` is a null pointer, then the calling function is not interested in it at that point, so you can ignore it.

The implementation of `RobinBcCoefStrategy` must be able to state how many cells past the edge or corner of a patch it is able to provide the coefficients. These cells are illustrated in this figure:



The method

```
hier::IntVector numberOfExtensionsFillable() const;
```
is provided for this purpose. This function should return a vector indicating the number of cells in each spatial direction for which the implementation can provide the boundary coefficients. The number zero indicates that the implementation cannot provide any coefficients beyond the edge or corner of the patch. Generally, implementations that require data already stored on the grid may have this type of limitation, depending on the availability of grid data. An example is the library-provided `SimpleCellRobinBcCoefs` class. For a specific boundary, the number of cells in the direction away from the boundary is irrelevant--the arrays of coefficients are of unit width in that direction. The solver uses the number of extensions fillable to determine if a boundary box needs to be "trimmed" down before requesting the coefficients for it. Regardless this number, the boundary box argument never extends further than would ordinary boundary boxes given by the patch geometry. The inability to fill all the extensions that a solver needs may limit certain solver capabilities. For example, filling extensions is used in linear refinement of cell-centered data, which maybe used in prolongation operations.

Uniform Coefficients Example:

To illustrate a simple case, this example sets the coefficients in the case where the the boundary condition is uniform. All boundaries have Dirichlet boundary condition of value 0.5 except for that on the maximum i side, which has zero gradient. The implementation determines which side the boundary box is on and sets the coefficients accordingly. This simple implementation simply fills in the coefficient arrays with constants, so it has no limit on the number of extensions it can fill.

```
void UserClass::setBcCoefs(
   const boost::shared_ptr<pdat::ArrayData<double> > &acoef_data,
   const boost::shared_ptr<pdat::ArrayData<double> > &bcoef_data,
   const boost::shared_ptr<pdat::ArrayData<double> > &gcoef_data,
   const boost::shared_ptr<hier_Variable> &variable,
   const hier::Patch &patch,
   const hier::BoundaryBox &boundary_box,
   const double fill_time=0 ) const
{
   double a, g;
   /*
     Determine from the available information
     (i.e., the patch, boundary box and any internal data)
     the value of a and g for this boundary box.  The
     parameter variable is ignored, because we want the
     coefficients to be independent of the variable.
     For this example, let us set the boundary value
     to 0.5 on the minimum i side and the slope to
     zero everywhere else.
   */
   if ( boundary_box.getLocationIndex() == 1 ) {
      a = 0.0;
      b = 1.0
      g = 0.0;
   } else {
      a = 1.0;
      b = 0.0;
      g = 0.5;
   }
   if ( acoef_data ) {
      acoef_data->fill(a);
```

```
   }
   if ( gcoef_data ) {
      gcoef_data->fill(g);
   }
   return;
}


hier::IntVector UserClass::numberOfExtensionsFillable()
   const
{
   /*
     Return a really big number.  We have no limits.
   */
   return hier::IntVector(d_dim, 1000); // d_dim is the object's
dimension.
}
```

In this simple example, we disregarded the argument `variable`, because the coefficients are independent of what variable is being set. If this implementation is set up for both a cell-centered quantity and a node-centered quantity, we can use the argument `variable` to determine which is being set. This implementation is similar to the slightly more general library-provided implementation `LocationIndexRobinBcCoefs` class described below.

Exact Boundary Condition Example:

In this example, we implement a boundary condition on two variables which are named “u” and “v”. For each, there is an exact Dirichlet boundary condition we wish to use.

We must loop through the index space of the boundary surfaces and set each coefficient individually, because the coefficient g varies in space. This is the most general form for g. In the following code-setup example, we have internally two functors of type `Fcn`, `d_u_functor` and `d_v_functor`, which compute the boundary values for the two variables as a function of spatial coordinates. The code sets up for Dirichlet boundary conditions by setting $a=1$ $b=0$ and g to the value of the functor. Thus g is a function of space, requiring us to determine the spatial coordinates of the side centers where the Robin formula is applied. As in the above example, this implementation can fill an unlimited number of extensions past the patch corner.

```
void PoissonSineSolution::setBcCoefs (
   const boost::shared_ptr<pdat::ArrayData<double> > &acoef_data ,
   const boost::shared_ptr<pdat::ArrayData<double> > &bcoef_data ,
   const boost::shared_ptr<pdat::ArrayData<double> > &gcoef_data ,
   const boost::shared_ptr<hier::Variable> &variable ,
   const hier_Patch &patch ,
   const hier_BoundaryBox &bdry_box ,
   const double fill_time ) const
{
   if ( bdry_box.getBoundaryType() != 1 ) {
      // Must be a face boundary.
      TBOX_ERROR("Bad boundary type in\n"
                     << "PoissonSineSolution::setBcCoefs \n");
   }
```

```
// a=1, b=0 for Dirichlet boundary condition.
if ( acoef_data ) {
   acoef_data->fill( 1.0 , 0 );
   bcoef_data->fill( 0.0 , 0 );
}

if (gcoef_data) {

   Fcn &functor( variable->getName() == "u" ?
     d_u_functor : d_v_functor );

   const int location_index = bdry_box.getLocationIndex();

   /*
      Get geometry information needed to compute coordinates
      of side centers.
   */
   boost::shared_ptr<geom::CartesianPatchGeometry> patch_geom
      = patch.getPatchGeometry();
   hier_BoxX patch_box(patch.getBox());
   const double *xlo = patch_geom->getXLower();
   const double *xup = patch_geom->getXUpper();
   const double *dx = patch_geom->getDx();

   if ( patch.getDim().getValue() == 2 ) {
      hier::BoxIterator boxit(gcoef_data->getBox());
      int i, j;
      double x, y;
      switch (location_index) {
      /*
        For each case, we loop through the data box,
        compute the coordinates corresponding to the
        side center and set g to the exact value of
        the function there.
      */
      case 0:
         // min i edge
         x = xlo[0];
         if(gcoef_data) for ( ; boxit; boxit++ ) {
            j = (*boxit)[1];
            y = xlo[1] + dx[1]*(j-patch_box.lower()[1]+0.5);
            (*gcoef_data)(*boxit,0) = functor(x,y);
         }
      break;
      case 1:
         // max i edge
         x = xup[0];
         if(gcoef_data) for ( ; boxit; boxit++ ) {
            j = (*boxit)[1];
            y = xlo[1] + dx[1]*(j-patch_box.lower()[1]+0.5);
            (*gcoef_data)(*boxit,0) = functor(x,y);
         }
      break;
      case 2:
         // min j edge
```

```
            y = xlo[1];
            if(gcoef_data) for ( ; boxit; boxit++ ) {
               i = (*boxit)[0];
               x = xlo[0] + dx[0]*(i-patch_box.lower()[0]+0.5);
               (*gcoef_data)(*boxit,0) = functor(x,y);
            }
         break;
         case 3:
            // max j edge
            y = xup[1];
            if(gcoef_data) for ( ; boxit; boxit++ ) {
               i = (*boxit)[0];
               x = xlo[0] + dx[0]*(i-patch_box.lower()[0]+0.5);
               (*gcoef_data)(*boxit,0) = functor(x,y);
            }
         break;
         default:
            TBOX_ERROR("Invalid location index in\n"
                       << "PoissonSineSolution::setBcCoefs");
         }
      }

    } else if ( patch.getDim().getValue() == 3 ) {
       // Similar to 2D...
    }

   return;
}


hier::IntVector PoissonSineSolution::numberOfExtensionsFillable()
   const
{
   /*
     Return a really big number.  We have no limits.
   */
   return hier::IntVector(d_dim, 1000); // d_dim is the object's
dimension.
}
```

Provided Implementations:

The Robin boundary coefficients interface described above is simply a flexible mean to describe the boundary condition. To avoid having to implement the interface for some common boundary conditions, the SAMRAI library provides several implementations. These are `SimpleCellRobinBcCoefs`, `GhostCellRobinBcCoefs` and `LocationIndexRobinBcCoefs`. To help you determine if one of these may be suitable for your application, we briefly describe the intended purpose of each. Please refer to the source code documentation for more details.

The `SimpleCellRobinBcCoefs` implementation allows users to specify boundary conditions using an interface identical to older Poisson solvers in the SAMRAI library. This implementation is helps codes that use the old Poisson solvers to switch to the current solvers. Since the old Poisson solvers are cell-centered, this implementation assumes that it is only used for cell-centered data. This implementation

provides a method, `setBoundaries()`, which is compatible with the identically-named method in the old Poisson solvers. Underneath, it computes the Robin boundary coefficients according to the information specified by the call to `setBoundaries()`.

The `LocationIndexRobinBcCoefs` 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.

The `GhostCellRobinBcCoefs` implementation is for specific cases where the discrete function is fixed at the center of the first ghost cell. The user sets the ghost cell values at a specific cell-centered patch data index. This implementation sets the boundary coefficients to values that are equivalent to setting the same value at the same ghost-cell-centered locations.

Usage in Solvers:

After implementing your boundary coefficient class or deciding to use a library-provided implementation, you can use an object of that class in a solver. The exact detail on how to do that depends on the solver's interface for specifying the boundary condition object and on the setup steps required by the boundary coefficient class. The following is an example of using the `LocationIndexRobinBcCoefs` class with the cell-centered Poisson FAC solver (`CellPoissonFACSolver`) in the library.

```
// Create the solver.
solv::CellPoissonFACSolver poisson_solver;
/*
  Create and set up the boundary coefficient implenentation.
  The exact setup steps depends on the implementation being used.
  This example uses the LocationIndexRobinBcCoefs class to set
  Dirichlet and Neumann boundary conditions.
  Set up Dirichlet boundary values of 1.0 and 0.0 at the minimum
  and maximum x boundaries and zero slope at the minimum and
  maximum y boundaries.  Note that the first argument of
  setBoundaryValue and setBoundarySlope is the location index.
*/
LocationIndexRobinBcCoefs bccoef( tbox::Dimension(2), “bccoef” );
bccoef.setBoundaryValue(0,1.0);
bccoef.setBoundaryValue(1,0.0);
bccoef.setBoundarySlope(2,0.0);
bccoef.setBoundarySlope(3,0.0);
// Tell the solver to use the boundary coefficient object we
created.
poisson_solver.setBcObject(&bccoef);
/* ... Set up the solver ... */
// Solve
poisson_solver.solveSystem(...);
```

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.