Evaluating a Laplacian operator on a patch

The purpose of this document is to briefly describe three different ways in which one may manipulate data on a SAMRAI patch. We illustrate how to evaluate a Laplacian operator in `FORTRAN`, `C`, and `C++`. We compute Δu using the standard 5-point Laplacian in 2D, where u is a cell-centered quantity. To make things slightly interesting, we compute the operator in two steps as $\text{laplaceu} = \Delta u = \nabla \cdot (\nabla u) = \text{divergence}(\text{gradu})$. Thus, "gradu" is side-centered and "laplaceu" is cell-centered.

We will evaluate Δu on each patch on a given patch level. We ignore the issue of setting ghost cell values for *u*. Thus, we simply evaluate Δu by iterating over the "local" patches. For each patch, we obtain the necessary quantities and perform our computations in serial:

```
   PatchLevel& level = . . .;

    for (PatchLevel::Iterator ip(level.begin()); ip!=level.end(); ++ip)
{

      const Pointer<Patch> patch = *ip;

      Box pbox = patch->getBox();

      Pointer<CartesianPatchGeometry> geom = patch->getPatchGeometry();
      double* dx = geom->getDx();

      Pointer< CellData<double> > u = patch->getPatchData(...);
      Pointer< SideData<double> > gradu = patch->getPatchData(...);
      Pointer< CellData<double> > laplaceu = patch->getPatchData(...);

      /*
       * Code to evaluate Laplacian goes here…
       */

   }
```

The operations above are the same for each of methods that we use to evaluate Δu . The `PatchLevel::Iterator` is used so that we only access `Patch` objects owned by the local process. The `Box` object `pbox` represents the region of index space covered by the patch. We assume that we are operating on a mesh defined by a Cartesian coordinate system. The `CartesianPatchGeometry` object provides us with the mesh spacing `dx`, and an array of doubles representing the mesh increments. Finally, we access each of the data quantities we will use, `u`, `gradu`, and `laplaceu`.

The first option to compute Δu is calling a FORTRAN subroutine. The code snippets that follow are placed within the loop over patches above. Since FORTRAN automatically handles multi-dimensional dynamically allocated arrays, we dimension the array associated with each data pointer based on the box over which the data is defined and the centering of the data on the mesh. We begin by getting the upper and lower indices of the patch box, and the number of ghost cells for *u*. Then, we call the FORTRAN routine `evallaplace` passing the array dimensions and pointers to the start of the data arrays:

```
      Index ilo = pbox.getLower();
      Index ihi = pbox.getUpper();

      IntVector ngc = u->getGhostCellWidth();

      evallaplace_(ilo(0), ihi(0),
                   ilo(1), ihi(1),
                   ngc(0), ngc(1),
                   dx,
                   u->getPointer(),
                   gradu->getPointer(0),
                   gradu->getPointer(1),
                   laplaceu->getPointer());
```

The FORTRAN routine `evallaplace` is implemented as follows:

```
   subroutine evallaplace(
  &  ilo0, ihi0, ilo1, ihi1, ngc0, ngc1,
  &  dx, u,
  &  grad0, grad1, laplace)

   integer
  &  ilo0, ilo1, ihi0, ihi1, ngc0, ngc1
   double precision dx(0:1)
   double precision
  &  u(ilo0-ngc0:ihi0+ngc0, ilo1-ngc1:ihi1+ngc1),
  &  grad0(ilo0:ihi0+1, ilo1:ihi1),
  &  grad1(ilo0:ihi0  , ilo1:ihi1+1),
  &  laplaceu(ilo0:ihi0, ilo1:ihi1)

   integer i0,i1

c  compute the gradient of u in the x-direction
   do i1 = ilo1, ihi1
      do i0 = ilo0, ihi0+1
         grad0(i0,i1) = (u(i0,i1) – u(i0-1,i1))/dx(0)
      enddo
   enddo

c  compute the gradient of u in the y-direction
   do i1 = ilo1, ihi1+1
      do i0 = ilo0, ihi0
         grad1(i0,i1) = (u(i0,i1) – u(i0,i1-1))/dx(1)
      enddo
   enddo

c  compute the Laplacian of u using the gradients
   do i1 = ilo1, ihi1
      do i0 = ilo0, ihi0
         laplaceu(i0,i1) = (grad0(i0+1,i1) – grad0(i0,i1))/dx(0)
  &                      + (grad1(i0,i1+1) – grad1(i0,i1))/dx(1)
      enddo
   enddo
   return
   end
```

In the FORTRAN routine, we compute the gradients of *u* in the x-direction and the y-direction, then we compute Δu using those gradients.

The second option to compute Δu is using C code. Since C has no support for dimensioning dynamically allocated arrays, we must compute the pointer offsets and apply them explicitly as we loop through the data. First, we obtain pointers to each of the data arrays and dimensions of the patch box used in the offset computations. Then, we compute the gradients of *u* in the x-direction and the y-direction, and Δu using those gradients. The numerical computations are identical to the FORTRAN operations above.

```
double* udat = u->getPointer();
double* grad0dat = gradu->getPointer(0);
double* grad1dat = gradu->getPointer(1);
double* laplaceudat = laplaceu->getPointer();

IntVector ngc = u->getGhostCellWidth();
int nc0 = pbox.numberCells(0);
int nc1 = pbox.numberCells(1);
int uoffset = nc0 + 2*ngc(0);
udat += uoffset*(ngc(1)-1);

/*
 * compute the gradient of u in the x-direction
 */
for (int j = 0; j < nc1; j++) {
   int uofflo = uoffset*(j+1);
   int uoffup = uofflo+1;
   int g0off = (nc0+1)*j;
   for (int i = 0; i < nc0+1; i++) {
      grad0dat[g0off+i] =
         (udat[uoffup+i] – udat[uofflo+i])/dx[0];
   }
}

/*
 * compute the gradient of u in the y-direction
 */
for (int j = 0; j < nc1+1; j++) {
   int uofflo = uoffset*j+1;
   int uoffup = uofflo+uoffset;
   int g1off = nc0*j;
   for (int i = 0; i < nc0; i++) {
      grad1dat[g1off+i] =
         (udat[uoffup+i] – udat[uofflo+i])/dx[1];
   }
}

/*
 * compute the Laplacian of u using the gradients
 */
for (int j = 0; j < nc1; j++) {
   int g0offlo = (nc0+1)*j;
   int g0offup = g0offlo+1;
   int g1offlo = nc0*j;
   int g1offup = g1offlo+nc0;
   int lapoff = nc0*j;
   for (int i = 0; i < nc0; i++) {
      laplaceudat[lapoff+i] = (grad0dat[g0offup+i]
                               - grad0dat[g0offlo+i]) /dx[0]
                             + (grad1dat[g1offup+i]
                               - grad1dat[g1offlo+i]) /dx[1];
   }
}
```

The third option for computing Δu is to use C++ code. We use iterator and indexing objects that SAMRAI provides for array-based data types to access individual array entries. These items eliminate the need to explicitly compute array offsets. In the following code, we compute gradients of u in the x-direction and the y-direction, then we compute Δu using those gradients. The numerical computations are identical to the FORTRAN and C operations shown above.

```
   /*
    * compute the gradient of u in the x-direction
    */
   Box gbox0(pbox);
   gbox0.growUpper(0, 1);

   for (CellData<double>::iterator ic=CellGeometry::begin(gbox0);
        ic!=CellGeometry::end(gbox0); ++ic) {
      CellIndex cell = *ic;
      CellIndex icm = cell;
      CellIndex icp = cell;
      icp(0) -= 1;

      (*grad)(SideIndex(cell, SideIndex::X, SideIndex::Lower)) =
         ((*u)(icp) - (*u)(icm)) / dx[0];
   }

   /*
    * compute the gradient of u in the y-direction
    */
   Box gbox1(pbox);
   gbox1.growUpper(1, 1);

   for (CellData<double>::iterator ic=CellGeometry::begin(gbox1);
        ic!=CellGeometry::end(gbox1); ++ic) {
      CellIndex cell = *ic;
      CellIndex icm = cell;
      CellIndex icp = cell;
      icp(1) -= 1;

      (*graduu)(SideIndex(cell, SideIndex::Y, SideIndex::Lower)) =
         ((*u)(icp) - (*u)(icm)) / dx[1];
   }

   /*
```

```
  * compute the Laplacian of u using the gradients
  */
 for (CellData<double>::iterator ic=CellGeometry::begin(gbox1);
      ic!=CellGeometry::end(gbox1); ++ic) {
    CellIndex cell = *ic;

    (*laplaceu)(cell) =
       ( (*gradu)(SideIndex(cell, SideIndex::X, SideIndex::Upper))
      -  (*gradu)(SideIndex(cell, SideIndex::X, SideIndex::Lower))
        ) / dx[0]
      +

       ( (*gradu)(SideIndex(cell, SideIndex::Y, SideIndex::Upper))
      -  (*gradu)(SideIndex(cell, SideIndex::Y, SideIndex::Lower))
         ) / dx[1]
  }
```

The iterators and indices make the code more readable and less error-prone than the C code shown above. While they are useful for rapidly developing prototype code, they should not be used for production code. They are inefficient since C++ compilers cannot optimize loops in which they are used.