Chapter 1

oomph-lib's Block Preconditioning Framework

oomph-lib's block preconditioning framework provides an environment for the development of block preconditioners for the iterative solution of linear systems by Krylov subspace methods. The framework is based on the classification of the problem's unknowns (degrees of freedom; here abbreviated as dofs) into different "dof types" which, in a multi-physics context, typically represent different physical quantities. A key feature of the framework is that it allows existing block preconditioners (which were developed for a particular single-physics application, say) to be re-used, in a hierarchical fashion, in block preconditioners for related multi-physics problems. This means that existing Navier-Stokes and solid mechanics preconditioners can be used to create preconditioners for fluid-structure interaction problems, say.

Following a brief overview of the underlying ideas and their implementation in oomph-lib this tutorial discusses a sequence of increasingly complex block preconditioners that illustrate the framework's capabilities in the context of a (rather artificial) model problem. The final example illustrates a simple implementation of a block preconditioner for an FSI problem. We conclude with a few comments on the use of block preconditioners in parallel. Other tutorials discuss how the methodology is used in "real" preconditioners. See, for instance, the tutorials discussing

- oomph-lib's "general purpose" block preconditioners.
- The NavierStokesSchurComplementPreconditioner for Navier-Stokes problems
- The FSIPreconditioner for monolithically-discretised fluid-structure interaction problems.
- The preconditioner for large-displacement solid mechanics problems in which boundary displacements are prescribed.
- The previous preconditioner is mainly used as a subsidiary block preconditioner for the solution of fluid-structure interaction problems with (pseudo-)solid fluid mesh updates.

1.1 Theoretical background

In `oomph-lib`, all problems are solved by Newton's method, which requires the repeated solution of linear systems of the form

$$\mathbf{J}\ \delta\mathbf{x} = -\mathbf{r}$$

for the Newton correction $\delta\mathbf{x}$ where $\mathbf{J}$ is the Jacobian matrix and $\mathbf{r}$ is the vector of residuals. (Left) preconditioning represents a transformation of the original linear system to

$$\mathbf{P}^{-1}\mathbf{J}\ \delta\mathbf{x} = -\mathbf{P}^{-1}\mathbf{r},$$

introduced with the aim of accelerating the convergence of Krylov subspace solvers such as GMRES or CG. The application of the preconditioner requires the solution of

$$\mathbf{P}\mathbf{z} = \mathbf{y}$$

for $\mathbf{z}$ at each Krylov iteration.

Block preconditioners are based (at least formally) on a reordering of the linear system such that related unknowns (e.g. dofs representing the same physical quantity) are grouped together and enumerated consecutively.

For instance, in linear elasticity problems (discussed in another tutorial) where we compute the displacement field of an elastic body in response to an applied traction, the (discrete) unknowns can be sub-divided according to which component of the displacement vector they represent. Using this classification of the dofs, the re-ordered linear system for a two-dimensional problem then has the form

$$\begin{bmatrix} \mathbf{J}_{xx} & \mathbf{J}_{xy} \\ \mathbf{J}_{xy} & \mathbf{J}_{yy} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}_{\mathbf{x}} \\ \delta\mathbf{x}_{\mathbf{y}} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_{\mathbf{x}} \\ \mathbf{r}_{\mathbf{y}} \end{bmatrix}.$$

A simple (and, in fact, quite effective) block preconditioner for this linear system can be formed by retaining only the diagonal blocks of the system matrix, leading to the block diagonal preconditioner

$$\mathbf{P}_{diag} = \begin{bmatrix} \mathbf{J}_{xx} & \\ & \mathbf{J}_{yy} \end{bmatrix}.$$

The application of this preconditioner requires the solution of the linear system

$$\begin{bmatrix} \mathbf{J}_{xx} & \\ & \mathbf{J}_{yy} \end{bmatrix} \begin{bmatrix} \mathbf{z}_{\mathbf{x}} \\ \mathbf{z}_{\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{\mathbf{x}} \\ \mathbf{y}_{\mathbf{y}} \end{bmatrix},$$

which requires the (exact or approximate) solution of the two smaller linear systems $\mathbf{J}_{xx}\ \mathbf{z}_{\mathbf{x}} = \mathbf{y}_{\mathbf{x}}$ and $\mathbf{J}_{yy}\ \mathbf{z}_{\mathbf{y}} = \mathbf{y}_{\mathbf{y}}$.

1.2 Overview

The above example shows that the application of block preconditioners typically require several generic steps:

1. The classification of the dofs.
2. The application of subsidiary preconditioning operations such as the solution of (smaller) linear systems or the evaluation of matrix-vector products with some of the blocks that are extracted from the original linear system.

The following subsections describe how these tasks are performed within `oomph-lib's` block preconditioning framework.

1.2.1 The classification of dof types via block preconditionable elements

The classification of dofs is specified by the elements since they are the only objects within `oomph-lib's` data structure that "know" what role a specific dof plays in "their" equations. During the setup phase, the block preconditioner loops over "all elements" (specified via one or more `Meshes` – here simply used as containers for elements; see below for further details) to establish the "dof type" for each global unknown.

To achieve this, the class `GeneralisedElement` contains two broken virtual methods that must be re-implemented/overloaded to label each of the element's dofs with its type. These methods are:

- `GeneralisedElement::ndof_types()` must return the number of dof types associated with an element.
- `GeneralisedElement::get_dof_numbers_for_unknowns(...)` must return a list of pairs comprising a map from global equation number to dof type for all unknowns in the element.

These are already implemented for many elements. If not, the functions are easy to write. For instance, `oomph-lib's` `DIM`-dimensional linear elasticity elements from the `QLinearElasticityElement` family can be made block-preconditionable by using the following wrapper class:

```
//==start_of_mylinearelasticityelement==============================
/// Wrapper to make quadratic linear elasticity element block
/// preconditionable
//==================================================================
template<unsigned DIM>
class MyLinearElasticityElement : public virtual QLinearElasticityElement<DIM,3>
{

public:

 /// \short The number of "DOF types" that degrees of freedom in this element
 /// are sub-divided into: The displacement components
 unsigned ndof_types() const
  {
   return DIM;
  }

/// Create a list of pairs for all unknowns in this element,
/// so the first entry in each pair contains the global equation
/// number of the unknown, while the second one contains the number
/// of the "DOF type" that this unknown is associated with.
/// (Function can obviously only be called if the equation numbering
/// scheme has been set up.)
///
/// The dof type enumeration (in 3D) is as follows:
/// S_x = 0
/// S_y = 1
/// S_z = 2
///
 void get_dof_numbers_for_unknowns(
  std::list<std::pair<unsigned long,unsigned> >& dof_lookup_list) const
  {
```

```
   // number of nodes
   unsigned n_node = this->nnode();

   // temporary pair (used to store dof lookup prior to being added to list)
   std::pair<unsigned,unsigned> dof_lookup;

   // loop over the nodes
   for (unsigned j=0;j<n_node;j++)
    {
     //loop over displacement components
     for (unsigned i=0;i<DIM;i++)
      {
       // determine local eqn number
       int local_eqn_number = this->nodal_local_eqn(j,i);

       // ignore pinned values - far away degrees of freedom resulting
       // from hanging nodes can be ignored since these are be dealt
       // with by the element containing their master nodes
       if (local_eqn_number >= 0)
        {
         // store dof lookup in temporary pair: Global equation number
         // is the first entry in pair
         dof_lookup.first = this->eqn_number(local_eqn_number);

         // set dof numbers: Dof number is the second entry in pair
         dof_lookup.second = i;

         // add to list
         dof_lookup_list.push_front(dof_lookup);
        }
      }
    }
  }
};
```

Thus, in the two-dimensional `MyLinearElasticityElement<2>` we have two types of dofs, corresponding to the displacements in the x and y directions, respectively. They are enumerated as dof types 0 and 1, respectively.

1.2.2 dof types, blocks, compound blocks and meshes

In the block diagonal preconditioner for the two-dimensional linear elasticity problem, discussed above, we have dof types that correspond directly to the blocks in the (re-ordered) Jacobian matrix. However, as we will demonstrate below, it is also possible to combine the blocks associated with multiple dofs into a single (compound) block in which case the number of blocks is smaller than the number of dof types. The relationship between dof types, block types, the elemental dof type classification and meshes are as follows

- **Elemental dof type classification:** Each element classifies its own dof types in the function `get_dof_numbers_for_unknowns(...)`. In the case of the two-dimensional `MyLinearElasticityElement<2>` elements, the dof types are classified as `0` and `1`; for two-dimensional `QTaylorHoodElement<2>` Navier-Stokes elements, the dof types are classified as `0` and `1` for the x and y-velocities, and `2` for the pressure p; etc.

- **Role of meshes:** When classifying the degrees of freedom into dof types, the block preconditioning framework visits all elements that make contributions to the Jacobian matrix and associates the global equation number of each dof with the dof type specified by the element. The block preconditioning framework is given access to the elements via (possibly multiple) meshes (here simply interpreted as containers for elements), each of which is assumed to contain elements of a single type. The total number of dof types in the block preconditioner is the sum of the dof types of the elements in the meshes. For instance, in a 2D fluid-structure interaction problem we have two different element types, the solid elements (which contain the x and y solid displacements, u_x and u_y, respectively, assumed to be enumerated as dof types 0 and 1 by these elements) and the fluid elements (which contain the x- and y - fluid velocities, v_x and v_y , and the pressure, p, assumed to be enumerated as dof types 0, 1 and 2 by these elements). Assuming the mesh of solid elements is specified as mesh 0 and the mesh of fluid elements is mesh 1, the block preconditioner has a total of five dof types which represent, in order, u_x, u_y, v_x, v_y, p. Note that if certain degrees of freedom are classified by multiple elements, the most recent assignment of the dof type over-writes previous assignments. The order

in which meshes are specified therefore matters.
A corollary to this is that a block preconditioner does not need to "know" about elements that do not introduce any new unknowns. For instance, `FaceElements` that apply Neumann/flux boundary conditions operate on dofs that are already contained in (and therefore classified by) the elements in the "bulk" mesh. Conversely, if a `FaceElement` imposes a boundary condition via Lagrange-multipliers, the dofs that represent these Lagrange multipliers must be classified by the `FaceElements` since the "bulk elements" are not aware of them.
If `oomph-lib` is compiled with the `PARANOID` flag, an error is thrown if any of the global unknowns are not associated with a dof type.

- **Blocks:** The blocks are the sub-blocks of the system matrix (usually the Jacobian matrix from the Newton method) that the block preconditioner works with. By default, each block is associated with exactly one dof type. However, it is possible create "compound blocks" that are associated with more than one dof type. For example, in the Navier-Stokes LSC preconditioner (in 2D) we have three dof types (the x and y-velocities and the pressure), but the preconditioner works with just two block types (forming the velocity and pressure blocks). The setup of the block types is handled by the function `block_setup(...)` discussed below.

1.3 Simple preconditioner examples

We will now illustrate the capabilities of the block preconditioning framework by considering the system of N coupled PDEs

$$\left(\frac{\partial^2 u_i}{\partial x_j^2} + \beta \sum_{k=1}^{N} u_k\right) = f_i(x_j) \quad i = 1, ..., N \qquad (1)$$

for the N fields $u_i(x_j)$. If $\beta = 0$, the system represents N (uncoupled) Poisson equations, each with their own source function $f_i(x_j)$. If $\beta \neq 0$ the PDE for $u_i(x_j)$ is affected by all other fields via the Helmholtz-like second term on the left-hand-side.

The `MultiPoissonElements` discretise the equations with standard Galerkin-type finite elements in which each field is treated as its own dof type. If $N = 5$, the linear system to be solved in the course of the Newton method,

$$\mathbf{J}\,\delta\mathbf{x} = -\mathbf{r}, \qquad (2)$$

has a 5×5 block structure implying that, following a formal re-numbering of the unknowns, the matrix and the vectors can be written as

$$\mathbf{J} = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{23} & J_{24} & J_{25} \\ J_{31} & J_{32} & J_{33} & J_{34} & J_{35} \\ J_{41} & J_{42} & J_{43} & J_{44} & J_{45} \\ J_{51} & J_{52} & J_{53} & J_{54} & J_{55} \end{pmatrix}, \quad \delta\mathbf{x} = \begin{pmatrix} \delta x_1 \\ \delta x_2 \\ \delta x_3 \\ \delta x_4 \\ \delta x_5 \end{pmatrix} \text{ and } \mathbf{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{pmatrix}. \qquad (3)$$

We wish to solve this linear system by preconditioned Krylov subspace methods, using a block preconditioner $\mathbf{P}$ formed (formally) from the blocks of the system matrix J. As discussed above, the application of the preconditioner (typically once per iteration of the Krylov solver) then requires the solution of linear systems of the form $\mathbf{P}\mathbf{y} = \mathbf{z}$, for $\mathbf{y}$. The preconditioning operation can also be written as $\mathbf{y} = \mathbf{P}^{-1}\mathbf{z}$ where the operator $\mathbf{P}^{-1}$ represents the application of the preconditioner to a vector $\mathbf{z}$. Formally, the operator $\mathbf{P}^{-1}$ represents the inverse of the matrix $\mathbf{P}$ but its application may, of course, be performed
approximately by another `‘subsidiary'' preconditioner/inexact solver e.g. by performing a small number of multigrid cycles, say. (Note that we say `‘formally'' because the preconditioner does not actually have to be associated with a specific matrix – it simply has to act as a linear operator that `‘turns $\mathbf{z}$ into $\mathbf{y}$'').

A specific block preconditioner must be derived from the `BlockPreconditioner` base class and must implement two pure virtual member functions of the underlying `Preconditioner` class:

- `void Preconditioner::setup()`: This function is called once during the solution of a given linear system by any of `oomph-lib`'s Krylov subspace solvers. It typically extracts a certain number of blocks from the matrix $\mathbf{J}$, possibly manipulates its local copies of these blocks, and performs any preliminary computations required to allow the rapid subsequent application of $\mathbf{P}^{-1}$.
- `void Preconditioner::preconditioner_solve(` $\mathbf{z}$, $\mathbf{y}$`)`: This function applies $\mathbf{P}^{-1}$ to the input argument $\mathbf{z}$ and computes $\mathbf{y}$, typically using some data that has been pre-computed in the `setup()` function.

To allow a block preconditioner to classify all dofs, the preconditioner must be given access to all elements that contribute to the linear system to be solved. This is done by specifying pointers to these elements via one or more `Mesh` objects (which simply act as containers for the element pointers), using the functions `set_nmesh(...)` (which specifies how many meshes the preconditioner works with) and `set_mesh(...)`.

We will discuss the implementation of the required functions (and associated capabilities of the block preconditioning framework) in a number of increasingly complex block preconditioners for the solution of the 5×5 linear system defined by equations (2) and (3). We stress that the purpose of this exercise is not the development of particularly clever preconditioners but simply an excuse to demonstrate the use of the available `‘machinery’'. Specifically we will demonstrate how to:

- extract selected blocks from the system matrix (usually the Jacobian matrix assembled by the Newton solver).
- perform matrix vector products with selected off-diagonal blocks.
- solve linear systems associated with selected blocks, using either a direct solver and/or subsidiary preconditioners (inexact solvers), including cases where the subsidiary preconditioners are block preconditioners themselves.
- replace and modify selected blocks and how to make such modified blocks available to subsidiary block preconditioners.
- concatenate and coarsen blocks.

1.3.1 A block diagonal preconditioner

NEW FEATURES: How to extract matrix blocks and corresponding block vectors from their full-size counterparts

1.3.1.1 Theory

The simplest possible block preconditioner is a block-diagonal preconditioner, formed by retaining only the diagonal blocks of $\mathbf{J}$, so that

$$\mathbf{P} = \begin{pmatrix} J_{11} & & & & \\ & J_{22} & & & \\ & & J_{33} & & \\ & & & J_{44} & \\ & & & & J_{55} \end{pmatrix}. \qquad (4)$$

The application of this preconditioner (i.e. the solution of the linear system $\mathbf{P}\mathbf{y} = \mathbf{z}$ for $\mathbf{y}$) requires the solution of the five much smaller linear systems

$$\begin{aligned} \mathbf{J}_{11}\,\mathbf{y}_1 &= \mathbf{z}_1, \\ \mathbf{J}_{22}\,\mathbf{y}_2 &= \mathbf{z}_2, \\ \mathbf{J}_{33}\,\mathbf{y}_3 &= \mathbf{z}_3, \\ \mathbf{J}_{44}\,\mathbf{y}_4 &= \mathbf{z}_4, \\ \mathbf{J}_{55}\,\mathbf{y}_5 &= \mathbf{z}_5, \end{aligned} \qquad (5)$$

where we have assumed that the two vectors $\mathbf{y}$ and $\mathbf{z}$ are re-ordered into `‘block vectors’' in the same way as the vectors $\delta\mathbf{x}$ and $\mathbf{r}$ in "the original linear system" (2) are re-ordered into the `‘block vectors’' in (3).
The implementation of the preconditioning operations in (5) can naturally be subdivided into two distinct `setup()` and `preconditioner_solve(...)` phases. Assuming that the linear systems in (5) are solved exactly by a direct solver (an `‘exact preconditioner’') that can pre-compute and store the LU decomposition of the diagonal matrix blocks, the `setup()` phase involves the following operations [text in square brackets refers to their `oomph-lib` -specific implementation]:

- Set up any data structures/lookup tables that are required to extract matrix blocks from the original matrix $\mathbf{J}$ [by calling the `BlockPreconditioner::block_setup()` function].
- Extract the five diagonal blocks $\mathbf{J}_{ii}$ (for $i = 1, ..., 5$) [using the `BlockPreconditioner::get_block(...)` function].
- Compute and store the LU decomposition of the diagonal blocks to allow the rapid solution of the systems $\mathbf{J}_{ii}\ \mathbf{y}_i = \mathbf{z}_i$ (for $i = 1, ..., 5$) during the `preconditioner_solve(...)` phase by back-substitution. [This is done by calling the `setup(...)` function of the subsidiary preconditioner/inexact solver. Following this, the diagonal matrix blocks are longer required and can be deleted.]

Once the `setup()` phase has been completed, the solution of the linear system $\mathbf{P}\mathbf{y} = \mathbf{z}$ by the `preconditioner_solve(...)` function involves the following steps:

- Extract the five ``block vectors'' $\mathbf{z}_i$ (for $i = 1, ..., 5$) from the vector $\mathbf{z}$ [using the `BlockPreconditioner::get_block_vectors(...)` function].
- Solve the linear systems $\mathbf{J}_{ii}\ \mathbf{y}_i = \mathbf{z}_i$ for the vectors $\mathbf{y}_i$ (for $i = 1, ..., 5$) using the precomputed LU decomposition of the diagonal blocks $\mathbf{J}_{ii}$ (for $i = 1, ..., 5$) created during the `setup()` phase.
- Combine the five ``block vectors'' $\mathbf{y}_i$ (for $i = 1, ..., 5$) to the full-length vector $\mathbf{y}$ [using the `BlockPreconditioner::return_block_vectors(...)` function].

1.3.1.2 Implementation as a BlockPreconditioner

Here is a sample implementation of the diagonal block preconditioner as a class `Diagonal`, derived from the `BlockPreconditioner` base class. The class provides storage for the subsidiary preconditioners that solve the linear systems associated with the diagonal blocks, implements the `setup()` and `preconditioner_solve(...)` functions, and provides a helper function `clean_up_my_memory()` which does what it says. We also provide an access function which allows the user to specify the pointer to the `Mesh` that contains the `MultiPoissonElements` which classify the dofs.

```
//========================start_of_diagonal_class============================
/// \short Simple proof-of-concept block diagonal preconditioner for
/// demo purposes. There's a much better version in src/generic!
//===================================================================
 template<typename MATRIX>
 class Diagonal : public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor for Diagonal preconditioner
  Diagonal() : BlockPreconditioner<MATRIX>()
   {
    Multi_poisson_mesh_pt=0;
   } // end_of_constructor

  /// \short Destructor - delete the subsidiary preconditioners (solvers for
  /// linear systems involving diagonal block)
  ~Diagonal()
   {
    this->clean_up_my_memory();
   }

  /// clean up the memory
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  Diagonal(const Diagonal&)
   {
    BrokenCopy::broken_copy("Diagonal");
   }

  /// Broken assignment operator
  void operator=(const Diagonal&)
   {
    BrokenCopy::broken_assign("Diagonal");
   }

  /// \short Setup the preconditioner
  void setup();

  // Use the version in the Preconditioner base class for the alternative
  // setup function that takes a matrix pointer as an argument.
  using Preconditioner::setup;
```

```
  /// Apply preconditioner to r, i.e. return solution of P z = r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private :

  /// \short Vector of pointers to preconditioners/inexact solvers
  /// for each diagonal block
  Vector<Preconditioner*> Diagonal_block_preconditioner_pt;

  /// \short Mesh pointers with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.1.3 The setup() function

As mentioned above, a `Preconditioner's` `setup()` function is called at the beginning of the `Iterative↩ LinearSolver's solve(...)` function. In time-dependent and/or nonlinear problems many (different) linear systems have to be solved by the same linear solver (and the associated preconditioner) throughout the code execution. To avoid memory leaks it is therefore important to free up any memory that may have been allocated in any previous use of the preconditioner. The `setup()` function of all block preconditioner should therefore always start by freeing up such memory. This is best done by using a helper function that can also be called from the destructor.

```
 //=========================start_of_setup_for_simple=======================
 /// The setup function.
 //=========================================================================
 template<typename MATRIX>
 void Diagonal<MATRIX>::setup()
 {
  // clean the memory
  this->clean_up_my_memory();
```

Next we set the pointer to the preconditioner's one-and-only mesh, and call the `block_setup()` function to set up the internal data structures and lookup tables required to extract blocks from the system matrix.

```
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err « "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);

  // Set up the generic block lookup scheme
  this->block_setup();
```

We create five subsidiary preconditioners (all exact solvers – SuperLU in its incarnation as an "exact" preconditioner) for the solution of the linear systems involving the diagonal blocks:

```
  // Extract the number of blocks
  unsigned nblock_types = this->nblock_types();
  // Create the subsidiary preconditioners
  Diagonal_block_preconditioner_pt.resize(nblock_types);
  for (unsigned i=0;i<nblock_types;i++)
   {
    Diagonal_block_preconditioner_pt[i] = new SuperLUPreconditioner;
   }
```

Next we set up the subsidiary preconditioner by extracting the diagonal blocks from the system matrix and passing them to the subsidiary preconditioners.

Note that each preconditioner is expected to retain a copy of whatever data it needs to subsequently perform its `preconditioner_solve(...)` function. The deep copy of the block that is returned by the `get_block(...)` function can therefore be deleted (here simply go out of scope) once the subsidiary preconditioner has been set up. (In the specific case of the `SuperLUPreconditioner`, the `setup(...)` function computes and stores the LU decomposition of the matrix; the matrix itself is then no longer required).

```
  // Setup preconditioners
  for (unsigned i=0;i<nblock_types;i++)
   {
    // Get block -- this makes a deep copy of the relevant entries in the
    // full Jacobian (i.e. the matrix of the linear system we're
```

```
   // actually trying to solve); we can do with this copy whatever
   // we want...
   CRDoubleMatrix block;
   this->get_block(i,i,block);

   // Set up preconditioner (i.e. lu-decompose the block)
   Diagonal_block_preconditioner_pt[i]->setup(&block);

   // Done with this block now, so the diagonal block that we extracted
   // above can go out of scope. Its LU decomposition (which is the only
   // thing we need to apply the preconditioner in the preconditioner_solve(...)
   // function) is retained in the associated sub-preconditioner/(in)exact
   // solver(SuperLU).
  }
```

1.3.1.4 The preconditioner_solve() function

To apply the preconditioner to a given vector, r, we first extract the five block-vectors whose sizes (and permutations) match that of the diagonal matrix blocks, using the `get_block_vectors(...)` function.

```
//=============================================================================
/// Preconditioner solve for the diagonal preconditioner:
/// Apply preconditioner to r and return z, so that P z = r, where
/// P is the block diagonal matrix constructed from the original
/// linear system.
//=============================================================================
template<typename MATRIX>
void Diagonal<MATRIX>::
preconditioner_solve(const DoubleVector& r, DoubleVector& z)
{
 // Get number of blocks
 unsigned nblock_types = this->nblock_types();
 // Split up rhs vector into sub-vectors, re-arranged to match
 // the matrix blocks
 Vector<DoubleVector> block_r;
 this->get_block_vectors(r,block_r);
```

We then provide storage for the five solution vectors and compute them by applying the subsidiary preconditioners' `preconditioner_solve(...)` function:

```
 // Solution of block solves
 Vector<DoubleVector> block_z(nblock_types);
 for (unsigned i = 0; i < nblock_types; i++)
  {
   Diagonal_block_preconditioner_pt[i]->preconditioner_solve(block_r[i],
                                                             block_z[i]);
  }
```

Finally the solutions in `block_z` are returned into the full-length solution vector `z` via a call to `return_block_vectors(...)`.

```
 // Copy solution in block vectors block_z back to z
 this->return_block_vectors(block_z,z);
}
```

1.3.1.5 The clean_up_my_memory() function

This function (which is called by the `setup()` function and the destructor) frees the memory that is allocated when a new linear system is solved – here the subsidiary preconditioners and their associated data (the LU decompositions of the diagonal blocks).

```
//=======================start_of_clean_up_for_simple=========================
/// The clean up function.
//=============================================================================
template<typename MATRIX>
void Diagonal<MATRIX>::clean_up_my_memory()
{
 // Delete diagonal preconditioners (approximate solvers)
 unsigned n_block = Diagonal_block_preconditioner_pt.size();
 for (unsigned i=0;i<n_block;i++)
  {
   if(Diagonal_block_preconditioner_pt[i]!=0)
    {
     delete Diagonal_block_preconditioner_pt[i];
     Diagonal_block_preconditioner_pt[i]=0;
    }
  }
} // End of clean_up_my_memory function.
```

1.3.1.6 Comments and Exercises

- The function `get_block_vectors(r,block_r)` extracts the five (or, in general, `nblock_type()`) block vectors `block_r` from the full-length vector `r`. The sizes of the block vectors (and the permutation

of their entries relative to their order in the full length vector `r`) match that of the matrix blocks. There is an alternative function `get_block_vector(...)` (note the missing s) which extracts a single block vector. An equivalent version exists for the `return_block_vector`[s] functions.

- In the example above we used an "exact preconditioner" (direct solver) to solve the five linear systems associated with the diagonal blocks. However, the (approximate) solution of these linear systems can be performed by any other matrix-based preconditioner, such as `oomph-lib's` diagonal preconditioner, `Matrix↩ BasedDiagPreconditioner`, discussed in another tutorial. The setup and application of this preconditioner is obviously much faster than for the `SuperLUPreconditioner`. Its setup merely requires the extraction of the diagonal entries and storage of their inverses (rather than the computation of the LU decomposition), while the application simply requires the multiplication of the input vector by the pre-computed inverses of the diagonal entries (rather than a back-substitution). However, the preconditioner is clearly not "as good" and therefore results in a larger number of iterations in the iterative linear solver. In fact, using the diagonal preconditioner for the approximate solution of the five linear systems involving the diagonal blocks is mathematically equivalent to using the diagonal preconditioner on the entire matrix. Try it out!

1.3.2 A block upper triangular preconditioner

NEW FEATURES: How to set up matrix vector products with off-diagonal blocks

1.3.2.1 Theory

Next we consider the implementation of an upper triangular preconditioner, formed by retaining only the blocks in the upper right hand part of $\mathbf{J}$, including the diagonals.

$$\mathbf{P} = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ & J_{22} & J_{23} & J_{24} & J_{25} \\ & & J_{33} & J_{34} & J_{35} \\ & & & J_{44} & J_{45} \\ & & & & J_{55} \end{pmatrix}. \tag{6}$$

The application of this preconditioner (i.e. the solution of the linear system $\mathbf{P}\mathbf{y} = \mathbf{z}$ for $\mathbf{y}$) again requires the solution of five much smaller linear systems

$$\begin{aligned} \mathbf{J}_{11}\,\mathbf{y}_1 &= \widetilde{\mathbf{z}}_1 = \mathbf{z}_1 - \mathbf{J}_{15}\,\mathbf{y}_5 - \mathbf{J}_{14}\,\mathbf{y}_4 - \mathbf{J}_{13}\,\mathbf{y}_3 - \mathbf{J}_{12}\,\mathbf{y}_2, \\ \mathbf{J}_{22}\,\mathbf{y}_2 &= \widetilde{\mathbf{z}}_2 = \mathbf{z}_2 - \mathbf{J}_{25}\,\mathbf{y}_5 - \mathbf{J}_{24}\,\mathbf{y}_4 - \mathbf{J}_{23}\,\mathbf{y}_3, \\ \mathbf{J}_{33}\,\mathbf{y}_3 &= \widetilde{\mathbf{z}}_3 = \mathbf{z}_3 - \mathbf{J}_{35}\,\mathbf{y}_5 - \mathbf{J}_{34}\,\mathbf{y}_4, \\ \mathbf{J}_{44}\,\mathbf{y}_4 &= \widetilde{\mathbf{z}}_4 = \mathbf{z}_4 - \mathbf{J}_{45}\,\mathbf{y}_5, \\ \mathbf{J}_{55}\,\mathbf{y}_5 &= \widetilde{\mathbf{z}}_5 = \mathbf{z}_5, \end{aligned} \tag{7}$$

where we have again assumed that the two vectors $\mathbf{y}$ and $\mathbf{z}$ are re-ordered into ``block vectors'' in the same way as the vectors $\delta\mathbf{x}$ and $\mathbf{r}$ in "the original linear system" (2) are re-ordered into the ``block vectors'' in (3).
The main difference to the block diagonal preconditioner considered before is that the right hand sides of the linear systems have to be modified. We start by solving the final equation for $\mathbf{y}_5$. We then multiply this vector by the off-diagonal block $\mathbf{J}_{45}$, subtract the result from $\mathbf{z}_4$ and use the result of this operation as the right-hand-side for the linear system that determines $\mathbf{y}_4$, etc.
The implementation of the preconditioning operations in (7) can again be subdivided into two distinct `setup()` and `preconditioner_solve(...)` phases. Assuming that the linear systems in (7) are solved exactly by a direct solver (an ``exact preconditioner'') that can pre-compute and store the LU decomposition of the diagonal matrix blocks, the `setup()` phase involves the following operations [text in square brackets refers to their `oomph-lib` specific implementation]:

- Set up any data structures/lookup tables that are required to extract matrix blocks from the original matrix $\mathbf{J}$ [by calling the `BlockPreconditioner::block_setup()` function].
- Extract the five diagonal blocks $\mathbf{J}_{ii}$ (for $i = 1, ..., 5$) [using the `BlockPreconditioner::get_↩ block(...)` function].
- Compute and store the LU decomposition of the diagonal blocks to allow the rapid solution of the systems $\mathbf{J}_{ii}\,\mathbf{y}_i = \widetilde{\mathbf{z}}_i$ (for $i = 1, ..., 5$) during the `preconditioner_solve(...)` phase by back-substitution. [This is done by calling the `setup(...)` function of the subsidiary preconditioner/inexact solver. Following this, the diagonal matrix blocks are longer required and can be deleted.]

- Extract the relevant off-diagonal blocks from $\mathbf{J}$ and create `MatrixVectorProduct` operators. [The matrix vector products are set up using the `setup_matrix_vector_product(...)` function. As with the subsidiary preconditioners, the `MatrixVectorProduct` operators retain their own copy of any required data, so the off-diagonal matrix blocks can be deleted (or be allowed to go out of scope) following the setup.]

1.3.2.2 Implementation as a BlockPreconditioner

Here is a sample implementation of the upper triangular block preconditioner as a class `UpperTriangular`, derived from the `BlockPreconditioner` base class. The class provides storage for the subsidiary preconditioners that solve the linear systems associated with the diagonal blocks, and the `MatrixVectorProduct` operators. We also implement the `setup()` and `preconditioner_solve(...)` functions, and provide a helper function `clean_up_my_memory()` which does what it says. As before we also provide an access function which allows the user to specify the pointer to the `Mesh` that contains the `MultiPoissonElements` which classify the dofs.

```
//==========================start_of_upper_triangular_class=====================
/// \short Upper triangular preconditioner for a system
/// with any number of dof types.
//==============================================================================
 template<typename MATRIX>
 class UpperTriangular : public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor.
  UpperTriangular() : BlockPreconditioner<MATRIX>()
   {
    Multi_poisson_mesh_pt=0;
   }

  /// Destructor - delete the preconditioner matrices
  virtual ~UpperTriangular()
   {
    this->clean_up_my_memory();
   }

  /// clean up the memory
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  UpperTriangular(const UpperTriangular&)
   {
    BrokenCopy::broken_copy("UpperTriangular");
   }

  /// Broken assignment operator
  void operator=(const UpperTriangular&)
   {
    BrokenCopy::broken_assign("UpperTriangular");
   }

  /// Apply preconditioner to r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  void setup();

  // Use the version in the Preconditioner base class for the alternative
  // setup function that takes a matrix pointer as an argument.
  using Preconditioner::setup;

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private:

  /// Pointers to matrix vector product operators for the off diagonals
  DenseMatrix<MatrixVectorProduct*> Off_diagonal_matrix_vector_product_pt;

  /// \short Vector of pointers to preconditioners/inexact solvers
  /// for each diagonal block
  Vector<Preconditioner*> Block_preconditioner_pt;

  /// \short Pointer to mesh with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.2.3 The setup() function

As before, we start by cleaning up the memory, set the pointer to the mesh, and set up the generic block preconditioner functionality by calling `block_setup()`.

```
 //=======================start_of_setup_for_upper_triangular_class===========
 /// The setup function.
 //=============================================================================
 template<typename MATRIX>
 void UpperTriangular<MATRIX>::setup()
 {
  // clean the memory
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err « "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
  // Set up the block look up schemes
  this->block_setup();
```

We provide storage for the (pointers to the) matrix vector products and the subsidiary preconditioners.

```
  // Number of block types
  unsigned nblock_types = this->nblock_types();
  // Storage for the pointers to the off diagonal matrix vector products
  // and the the subsidiary preconditioners (inexact solvers) for the diagonal
  // blocks
  Off_diagonal_matrix_vector_product_pt.resize(nblock_types,nblock_types,0);
  Block_preconditioner_pt.resize(nblock_types);
```

Next we create the subsidiary preconditioners which we will use to solve the linear systems involving the diagonal blocks.

```
  // Build the preconditioners and matrix vector products
  for(unsigned i = 0; i < nblock_types; i++)
   {
    // Create the subsidiary preconditioners
    Block_preconditioner_pt[i] = new SuperLUPreconditioner;

    // Put in braces so block matrix goes out of scope when done...
    {
     // Get block -- this makes a deep copy of the relevant entries in the
     // full Jacobian (i.e. the matrix of the linear system we're
     // actually trying to solve); we can do with this copy whatever
     // we want...
     CRDoubleMatrix block;
     this->get_block(i,i,block);

     // Set up preconditioner (i.e. lu-decompose the block)
     Block_preconditioner_pt[i]->setup(&block);

     // Done with this block now, so the diagonal block that we extracted
     // above can go out of scope. Its LU decomposition (which is the only
     // thing we need to apply the preconditioner in the
     // preconditioner_solve(...) function) is retained in the associated
     // sub-preconditioner/(in)exact solver(SuperLU).
    } // end of brace to make block go out of scope
```

We then extract the relevant off-diagonal blocks (those above the diagonal) from the full matrix, create a `MatrixVectorProduct` operator for each and use the `BlockPreconditioner::setup_matrix_vector_product(...)` function to make them fully functional. Note that the final argument to this function (the column index of the off-diagonal block in its block enumeration within the current preconditioner) is required to set up additional lookup tables that are required to ensure the correct operation of this object in cases when the preconditioner operates in parallel. The details are messy and not worth explaining here – just do it!

```
    // Next set up the off diagonal mat vec operators
    for(unsigned j=i+1;j<nblock_types;j++)
     {
      // Get the off diagonal block
      CRDoubleMatrix block_matrix = this->get_block(i,j);
      // Create a matrix vector product operator
      Off_diagonal_matrix_vector_product_pt(i,j) = new MatrixVectorProduct;
      // Setup the matrix vector product for the currrent block matrix
      // and specify the column in the "big matrix" as final argument.
      // This is needed for things to work properly in parallel -- don't ask!
      this->setup_matrix_vector_product(
       Off_diagonal_matrix_vector_product_pt(i,j),&block_matrix,j);
      // Done with this block now, so the diagonal block that we extracted
      // above can go out of scope. The MatrixVectorProduct operator retains
```

```
     // its own copy of whatever data it needs.
    } // End for loop over j
  } // End for loop over i
} // End setup(...)
```

1.3.2.4 The preconditioner_solve() function

As in the block diagonal preconditioner, we start by extracting the block vectors from the full-length vector, `r`.

```
//=============================================================================
/// Preconditioner solve for the upper triangular preconditioner:
/// Apply preconditioner to r and return z, so that P z = r, where
/// P is the block diagonal matrix constructed from the original
/// linear system.
//=============================================================================
template<typename MATRIX> void UpperTriangular<MATRIX>::
preconditioner_solve(const DoubleVector& r, DoubleVector& z)
{
 // Get number of blocks
 unsigned n_block = this->nblock_types();
 // vector of vectors for each section of rhs vector
 Vector<DoubleVector> block_r;

 // rearrange the vector r into the vector of block vectors block_r
 this->get_block_vectors(r,block_r);
```

Next we provide storage for the solution vectors and work backwards through the (block)-rows of the (block-)linear system (7). Following each linear solve we update the right-hand-side of the next linear system, as discussed above.

```
 // Vector of vectors for the solution block vectors
 Vector<DoubleVector> block_z(n_block);
 // Required to be an int due to an unsigned being unable to be compared to a
 // negative number (because it would roll over).
 for (int i=n_block-1;i>-1;i--)
  {
   // Back substitute
   for (unsigned j=i+1;j<n_block;j++)
    {
     DoubleVector temp;
     Off_diagonal_matrix_vector_product_pt(i,j)->multiply(block_z[j],temp);
     block_r[i] -= temp;
    } // End for over j
   // Solve on the block
   this->Block_preconditioner_pt[i]->
    preconditioner_solve(block_r[i], block_z[i]);
  } // End for over i
```

Finally, the solutions in `block_z` are combined via `return_block_vectors(...)` which places the results back into the full-length vector `z` that is returned by this function.

```
 // Copy solution in block vectors block_r back to z
 this->return_block_vectors(block_z,z);
}
```

1.3.2.5 The clean_up_my_memory() function

This function again deletes any data that was allocated in the setup function – here the subsidiary preconditioners (and their LU decompositions) and the matrix-vector product operators.

```
//========================start_of_clean_up_for_upper_triangular_class========
/// The clean up function.
//=============================================================================
template<typename MATRIX>
void UpperTriangular<MATRIX>::clean_up_my_memory()
{
 // Delete anything in Off_diagonal_matrix_vector_products
 for(unsigned i=0,ni=Off_diagonal_matrix_vector_product_pt.nrow();i<ni;i++)
  {
   for(unsigned j=0,nj=Off_diagonal_matrix_vector_product_pt.ncol();j<nj;j++)
    {
     if(Off_diagonal_matrix_vector_product_pt(i,j) != 0)
      {
       delete Off_diagonal_matrix_vector_product_pt(i,j);
       Off_diagonal_matrix_vector_product_pt(i,j) = 0;
      }
    }
  }
 // Delete preconditioners (approximate solvers)
 unsigned n_block = Block_preconditioner_pt.size();
 for (unsigned i=0;i<n_block;i++)
  {
   if(Block_preconditioner_pt[i]!=0)
    {
     delete Block_preconditioner_pt[i];
     Block_preconditioner_pt[i]=0;
    }
  }
```

```
} // End of clean_up_my_memory function.
```

1.3.3 Combining multiple dof types into compound blocks. Part 1

NEW FEATURES: How to combine multiple dof types into compound blocks

1.3.3.1 Theory

So far we have illustrated how to implement block preconditioners for cases where the dof types (as identified by the elements) correspond directly to the block types. This is appropriate for our model PDE system (1) in which the five fields (and the governing equations) are all of the same type. In many applications, particularly in multi-physics problems, it may be desirable to combine similar/related dof types into single blocks. For instance, in a 2D fluid-structure interaction problem, we may wish to distinguish between the two solid (x and y solid displacements) and three fluid (x and y fluid velocities and the pressure) dofs and employ subsidiary preconditioners that act directly on the two distinct solid and fluid blocks. A basic block diagonal preconditioner for such a problem that ignores the coupling between fluid and solid dofs has the following structure

$$\mathbf{P} = \left(\begin{array}{cc|ccc} J_{11} & J_{12} & & & \\ J_{21} & J_{22} & & & \\ \hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & J_{53} & J_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & \\ \hline & B_{22} \end{array}\right)$$

where B_{11} and B_{22} are the blocks formed from the corresponding "dof blocks" (the J_{ij} matrices). The application of this preconditioner (i.e. the solution of the linear system $\mathbf{P}\mathbf{y} = \mathbf{z}$ for $\mathbf{y}$) requires the solution of the two smaller linear systems

$$\left(\begin{array}{cc} J_{11} & J_{12} \\ J_{21} & J_{22} \end{array}\right)\left(\begin{array}{c} y_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} z_1 \\ z_2 \end{array}\right) \quad \text{or} \quad B_{11}\, Y_1 = Z_1 \tag{8}$$

and

$$\left(\begin{array}{ccc} J_{33} & J_{34} & J_{35} \\ J_{43} & J_{44} & J_{45} \\ J_{53} & J_{54} & J_{55} \end{array}\right)\left(\begin{array}{c} y_3 \\ y_4 \\ y_5 \end{array}\right) = \left(\begin{array}{c} z_3 \\ z_4 \\ z_5 \end{array}\right) \quad \text{or} \quad B_{22}\, Y_2 = Z_2. \tag{9}$$

A key feature of the block preconditioning framework is the ability to combine dof types in this manner so that the preconditioner can operate directly with blocks B_{11} and B_{22} and the corresponding block vectors Y_1, Y_2, Z_1 and Z_2.

Assuming again that the linear systems in (8) and (9) are solved exactly by a direct solver (an ``exact preconditioner'') that can pre-compute and store the LU decomposition of the diagonal matrix blocks, B_{11} and B_{22}, the `setup()` phase involves the following operations [text in square brackets refers to their `oomph-lib` specific implementation]:

- Set up any data structures/lookup tables that are required to extract the matrix blocks B_{11} and B_{22} and the associated block vectors [by calling the `BlockPreconditioner::block_setup(...)` function – this time with arguments that specify the mapping between "dof types" and "block types"].
- Extract the two diagonal blocks, B_{11} and B_{22} [using the `BlockPreconditioner::get_block(...)` function].
- Compute and store the LU decomposition of the diagonal blocks to allow the rapid solution of the systems during the `preconditioner_solve(...)` phase by back-substitution. [This is done by calling the `setup(...)` function of the subsidiary preconditioner/inexact solver. Following this, the diagonal matrix blocks are longer required and can be deleted.]

Once the `setup()` phase has been completed, the solution of the linear system $\mathbf{P}\mathbf{y} = \mathbf{z}$ by the `preconditioner_solve(...)` function involves the following steps:

- Extract the two ``block vectors'' $\mathbf{Z}_i$ (for $i = 1, 2$) from the vector $\mathbf{z}$ [using the `BlockPreconditioner::get_block_vectors(...)` function].
- Solve the linear systems $\mathbf{B}_{ii}\, \mathbf{Y}_i = \mathbf{Z}_i$ for the vectors $\mathbf{Y}_i$ (for $i = 1, 2$) using the precomputed LU decomposition of the diagonal blocks $\mathbf{B}_{ii}$ (for $i = 1, 2$) created during the `setup()` phase.
- Combine the two ``block vectors'' $\mathbf{Y}_i$ (for $i = 1, ..., 2$) to the full-length vector $\mathbf{y}$ [using the `BlockPreconditioner::return_block_vectors(...)` function].

1.3.3.2 Implementation as a BlockPreconditioner

The implementation of the preconditioner closely follows that of the block diagonal preconditioner discussed above, the main difference being that the current preconditioner only ever operates with exactly two blocks. Therefore we store pointers to the two subsidiary preconditioners (rather than a vector of pointers that can store an arbitrary number of these).

```
//======================start_of_two_plus_three_class=========================
/// \short Block diagonal preconditioner for system with 5 dof types
/// assembled into a 2x2 block system, with (0,0) block containing the
/// first two dof types and the (1,1) block the remaining three dof types.
//============================================================================
 template<typename MATRIX>
 class TwoPlusThree : public BlockPreconditioner<MATRIX>
  {
    public :

  /// Constructor for TwoPlusThree
  TwoPlusThree() : BlockPreconditioner<MATRIX>(),
   First_subsidiary_preconditioner_pt(0),
   Second_subsidiary_preconditioner_pt(0)
    {
     Multi_poisson_mesh_pt=0;
    } // end_of_constructor

  /// Destructor - delete the diagonal solvers (subsidiary preconditioners)
  ~TwoPlusThree()
   {
    this->clean_up_my_memory();
   }
  /// clean up the memory
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  TwoPlusThree
  (const TwoPlusThree&)
   {
    BrokenCopy::broken_copy("TwoPlusThree");
   }

  /// Broken assignment operator
  void operator=(const TwoPlusThree&)
   {
    BrokenCopy::broken_assign("TwoPlusThree");
   }

  /// Apply preconditioner to r, i.e. return z such that P z = r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  virtual void setup();

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private :

  /// \short Pointer to preconditioners/inexact solver
  /// for (0,0) block
  Preconditioner* First_subsidiary_preconditioner_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (1,1) block
  Preconditioner* Second_subsidiary_preconditioner_pt;

  /// \short Pointer to mesh with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.3.3 The setup() function

As usual, we start by freeing up any previously allocated memory, and set the pointer to the mesh:

```
 //====================start_of_setup_for_two_plus_three=====================
 /// The setup function.
 //============================================================================
 template<typename MATRIX>
 void TwoPlusThree<MATRIX>::setup()
 {
  // Clean up memory.
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
```

```
   {
    std::stringstream err;
    err « "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
```

Since this preconditioner assumes explicitly that the problem involves five dof types we check that this is actually the case.

```
  // How many dof types do we have?
  unsigned n_dof_types = this->ndof_types();
#ifdef PARANOID
  // This preconditioner only works for 5 dof types
  if (n_dof_types!=5)
   {
    std::stringstream tmp;
    tmp « "This preconditioner only works for problems with 5 dof types\n"
        « "Yours has " « n_dof_types;
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

To indicate that several dof types are to be combined into single blocks, we specify the mapping between dof types and block types as an argument to the `block_setup(...)` function This is done by creating vector of length `ndof_type()` in which each entry indicates the block that the corresponding dof is supposed to end up in:

```
  // Combine into two blocks, one containing dof types 0 and 1, the
  // final one dof types 2-4. In general we want:
  // dof_to_block_map[dof_type] = block type
  Vector<unsigned> dof_to_block_map(n_dof_types);
  dof_to_block_map[0]=0;
  dof_to_block_map[1]=0;
  dof_to_block_map[2]=1;
  dof_to_block_map[3]=1;
  dof_to_block_map[4]=1;
  this->block_setup(dof_to_block_map);
```

To show that this actually worked, we output the number of blocks (which should be – and indeed is – equal to two).

```
  // Show that it worked ok:
  oomph_info « "Preconditioner has "
             « this->nblock_types() « " block types\n";
```

Next we create the two subsidiary preconditioners and call their `setup(...)` functions, passing the two diagonal blocks $\mathbf{B}_{11}$ and $\mathbf{B}_{22}$ to them.

```
  // Create the subsidiary preconditioners
  First_subsidiary_preconditioner_pt= new SuperLUPreconditioner;
  Second_subsidiary_preconditioner_pt= new SuperLUPreconditioner;

  // Set diagonal solvers/preconditioners; put in own scope
  // so variable block goes out of scope
  {
   CRDoubleMatrix block;
   this->get_block(0,0,block);
   // Set up preconditioner (i.e. lu-decompose the block)
   First_subsidiary_preconditioner_pt->setup(&block);
  }
  {
   CRDoubleMatrix block;
   this->get_block(1,1,block);

   // Set up preconditioner (i.e. lu-decompose the block)
   Second_subsidiary_preconditioner_pt->setup(&block);
  }
 } // End of setup
```

1.3.3.4 The preconditioner_solve() function

The `preconditioner_solve(...)` function is equivalent to that in the `Diagonal` preconditioner discussed above, though here it simply acts on a 2x2 block system.

```
 //=============================================================================
 /// Preconditioner solve for the two plus three diagonal preconditioner:
 /// Apply preconditioner to r and return z, so that P r = z, where
 /// P is the block diagonal matrix constructed from the original
 /// linear system.
 //=============================================================================
 template<typename MATRIX>
 void TwoPlusThree<MATRIX>::
 preconditioner_solve(const DoubleVector& r, DoubleVector& z)
```

```
{
 // Get number of blocks
 unsigned n_block = this->nblock_types();
 // Split up rhs vector into sub-vectors, arranged to match the matrix blocks.
 Vector<DoubleVector> block_r;
 this->get_block_vectors(r,block_r);
 // Create storage for solution of block solves
 Vector<DoubleVector> block_z(n_block);
 // Solve (0,0) diagonal block system
 First_subsidiary_preconditioner_pt->preconditioner_solve(block_r[0],
                                                          block_z[0]);

 // Solve (1,1) diagonal block system
 Second_subsidiary_preconditioner_pt->preconditioner_solve(block_r[1],
                                                           block_z[1]);

 // Copy solution in block vectors block_z back to z
 this->return_block_vectors(block_z,z);
}
```

1.3.3.5 The clean_up_my_memory() function

This function again deletes the allocated storage – here the subsidiary preconditioners.

```
//====================start_of_clean_up_for_two_plus_three====================
/// The clean up function.
//============================================================================
template<typename MATRIX>
void TwoPlusThree<MATRIX>::clean_up_my_memory()
{
 //Clean up subsidiary preconditioners.
 if(First_subsidiary_preconditioner_pt!=0)
  {
   delete First_subsidiary_preconditioner_pt;
   First_subsidiary_preconditioner_pt = 0;
  }
 if(Second_subsidiary_preconditioner_pt!=0)
  {
   delete Second_subsidiary_preconditioner_pt;
   Second_subsidiary_preconditioner_pt = 0;
  }
} // End of clean_up_my_memory function.
```

1.3.4 Combining multiple dof types into compound blocks. Part 2: How to deal with off-diagonal blocks

NEW FEATURES: How to set up matrix vector products when multiple dof types have been combined into compound blocks

1.3.4.1 Theory

The extension of the preconditioner introduced in the previous section to block-triangular form is straightforward: We use the same dof-to-block mapping as before but now retain the off-diagonal block B_{12} so that the preconditioner has the structure:

$$\mathbf{P} = \left(\begin{array}{cc|ccc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{13} & J_{14} & J_{15} \\ \hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & J_{53} & J_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline & B_{22} \end{array}\right). \tag{10}$$

In the FSI context where B_{11} and B_{22} represent the solid and fluid sub-blocks, respectively, the inclusion of the off-diagonal block B_{12} incorporates the effect of fluid dofs (via pressure and shear stress) onto the solid equations. Since this captures "more of the physics" the preconditioner can be expected to be better than its block diagonal counterpart.

The application of the preconditioner (i.e. the solution of the linear system $\mathbf{Py} = \mathbf{z}$ for $\mathbf{y}$) requires the solution of the two smaller linear systems

$$\left(\begin{array}{cc} J_{11} & J_{12} \\ J_{21} & J_{22} \end{array}\right)\left(\begin{array}{c} y_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} z_1 \\ z_2 \end{array}\right) - \left(\begin{array}{ccc} J_{13} & J_{14} & J_{15} \\ J_{23} & J_{24} & J_{25} \end{array}\right)\left(\begin{array}{c} y_3 \\ y_4 \\ y_5 \end{array}\right) \quad \text{or} \quad B_{11}Y_1 = Z_1 - B_{12}Y_2 \tag{11}$$

and

$$\begin{pmatrix} J_{33} & J_{34} & J_{35} \\ J_{43} & J_{44} & J_{45} \\ J_{53} & J_{54} & J_{55} \end{pmatrix} \begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix} \qquad \text{or} \qquad B_{22} Y_2 = Z_2. \tag{12}$$

1.3.4.2 Implementation as a BlockPreconditioner

The implementation is very similar to that in the previous example – we simply provide additional storage for the (single) matrix vector product operator required for the multiplication with B_{12} when updating the right-hand-side in equation (11).

```
//=================start_of_two_plus_three_upper_triangular_class==============
/// \short Upper triangular two plus three triangular preconditioner for a
/// system with  5 dof types.
//============================================================================
 template<typename MATRIX>
 class TwoPlusThreeUpperTriangular
  : public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor.
  TwoPlusThreeUpperTriangular()  :
   BlockPreconditioner<MATRIX>(),
    Off_diagonal_matrix_vector_product_pt(0),
    First_subsidiary_preconditioner_pt(0),
    Second_subsidiary_preconditioner_pt(0)
   {
    Multi_poisson_mesh_pt=0;
   }

  /// Destructor - delete the preconditioner matrices
  virtual ~TwoPlusThreeUpperTriangular()
   {
    this->clean_up_my_memory();
   }

  /// clean up the memory
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  TwoPlusThreeUpperTriangular(const TwoPlusThreeUpperTriangular&)
   {
    BrokenCopy::broken_copy("TwoPlusThreeUpperTriangular");
   }

  /// Broken assignment operator
  void operator=(const TwoPlusThreeUpperTriangular&)
   {
    BrokenCopy::broken_assign("TwoPlusThreeUpperTriangular");
   }

  /// Apply preconditioner to r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  void setup();

  // Use the version in the Preconditioner base class for the alternative
  // setup function that takes a matrix pointer as an argument.
  using Preconditioner::setup;

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private:

  /// Pointer to matrix vector product operator for the single off diagonals
  MatrixVectorProduct* Off_diagonal_matrix_vector_product_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (0,0) block
  Preconditioner* First_subsidiary_preconditioner_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (1,1) block
  Preconditioner* Second_subsidiary_preconditioner_pt;

  /// \short Pointer to mesh with preconditionable elements used
```

```
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.4.3 The setup() function

As before, we start by freeing up any previously allocated memory and set the pointer to the mesh,

```
 //==============start_of_setup_for_two_plus_three_upper_triangular_class=======
 /// The setup function.
 //============================================================================
 template<typename MATRIX>
 void TwoPlusThreeUpperTriangular<MATRIX>::setup()
 {
  // clean the memory
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err << "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
```

and check that the number of dof types is correct.

```
  // Get number of degrees of freedom.
  unsigned n_dof_types = this->ndof_types();
#ifdef PARANOID
  // This preconditioner only works for 5 dof types
  if (n_dof_types!=5)
   {
    std::stringstream tmp;
    tmp << "This preconditioner only works for problems with 5 dof types\n"
        << "Yours has " << n_dof_types;
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

The block setup is again performed with a dof-to-block mapping that results in a block preconditioner with 2x2 blocks.

```
  // Combine into two major blocks, one containing dof types 0 and 1, the
  // final one dof types 2-4. In general we want
  // dof_to_block_map[dof_type] = block_type
  Vector<unsigned> dof_to_block_map(n_dof_types);
  dof_to_block_map[0]=0;
  dof_to_block_map[1]=0;
  dof_to_block_map[2]=1;
  dof_to_block_map[3]=1;
  dof_to_block_map[4]=1;
  this->block_setup(dof_to_block_map);
```

We create the two subsidiary preconditioners and pass the two diagonal blocks B_{11} and B_{22} to their `setup()` functions. As before, the deep copies of these matrices are then allowed to go out of scope, freeing up the memory, since the subsidiary preconditioners retain whatever information they require.

```
  // Create the subsidiary preconditioners
  First_subsidiary_preconditioner_pt= new SuperLUPreconditioner;
  Second_subsidiary_preconditioner_pt= new SuperLUPreconditioner;
  // Set diagonal solvers/preconditioners; put in own scope
  // so block goes out of scope
  {
   CRDoubleMatrix block;
   this->get_block(0,0,block);

   // Set up preconditioner (i.e. lu-decompose the block)
   First_subsidiary_preconditioner_pt->setup(&block);
  }
  {
   CRDoubleMatrix block;
   this->get_block(1,1,block);

   // Set up preconditioner (i.e. lu-decompose the block)
   Second_subsidiary_preconditioner_pt->setup(&block);
  } // end setup of last subsidiary preconditioner
```

Finally we create and set up the off-diagonal vector product. Note that the block column index refers to the block enumeration, so the block column index of B_{12} is 1 (in a C++ zero-based enumeration!).

```
  // next setup the off diagonal mat vec operators
  {
   // Get the block
```

```
  CRDoubleMatrix block_matrix = this->get_block(0,1);
  // Create matrix vector product
  Off_diagonal_matrix_vector_product_pt = new MatrixVectorProduct;
  // Set it up -- note that the block column index refers to the
  // block enumeration (not the dof enumeration)
  unsigned block_column_index=1;
  this->setup_matrix_vector_product(
   Off_diagonal_matrix_vector_product_pt,&block_matrix,block_column_index);
 }
```

1.3.4.4 The preconditioner_solve() function

The application of the preconditioner is the exact equivalent of that of the general-purpose block triangular preconditioner discussed above, restricted to a 2x2 system:

```
//=============================================================================
/// Preconditioner solve for the two plus three upper block triangular
/// preconditioner:
/// Apply preconditioner to r and return z, so that P z = r, where
/// P is the block diagonal matrix constructed from the original
/// linear system.
//=============================================================================
template<typename MATRIX>
void TwoPlusThreeUpperTriangular<MATRIX>::
preconditioner_solve(const DoubleVector& r, DoubleVector& z)
{
 // Get number of blocks
 unsigned n_block = this->nblock_types();
 // Split up rhs vector into sub-vectors, rarranged to match the matrix blocks.
 Vector<DoubleVector> block_r;
 this->get_block_vectors(r,block_r);
 // Create storage for solution of block solves
 Vector<DoubleVector> block_z(n_block);
 // Solve (1,1) diagonal block system
 Second_subsidiary_preconditioner_pt->preconditioner_solve(block_r[1],
                                                           block_z[1]);
 // Solve (0,1) off diagonal.
 // Substitute
 DoubleVector temp;
 Off_diagonal_matrix_vector_product_pt->multiply(block_z[1],temp);
 block_r[0] -= temp;
 // Solve (0,0) diagonal block system
 First_subsidiary_preconditioner_pt->preconditioner_solve(block_r[0],
                                                          block_z[0]);
 // Copy solution in block vectors block_z back to z
 this->return_block_vectors(block_z,z);
}
```

1.3.4.5 The clean_up_my_memory() function

As before, this function frees up any memory that has been allocated in the `setup()` function.

```
//===========start_of_clean_up_for_two_plus_three_upper_triangular_class======
/// The clean up function.
//============================================================================
template<typename MATRIX>
void TwoPlusThreeUpperTriangular<MATRIX>::clean_up_my_memory()
{
 // Delete of diagonal matrix vector product
 if (Off_diagonal_matrix_vector_product_pt != 0)
  {
   delete Off_diagonal_matrix_vector_product_pt;
   Off_diagonal_matrix_vector_product_pt = 0;
  }
 //Clean up subsidiary preconditioners.
 if(First_subsidiary_preconditioner_pt!=0)
  {
   delete First_subsidiary_preconditioner_pt;
   First_subsidiary_preconditioner_pt = 0;
  }
 if(Second_subsidiary_preconditioner_pt!=0)
  {
   delete Second_subsidiary_preconditioner_pt;
   Second_subsidiary_preconditioner_pt = 0;
  }
} // End of clean_up_my_memory function.
```

1.3.5 Using subsidiary block preconditioners

NEW FEATURES: How to use subsidiary block preconditioners to (approximately) solve linear systems constructed from subsets of dof-blocks.

1.3.5.1 Theory

The two previous examples were motivated by the observation that in multi-physics problems (such as fluid-structure interaction) it is natural to combine "related" dof blocks into compound block matrices. We showed that the block preconditioning framework makes it easy to extract such matrices from the original system matrix and demonstrated how to solve linear systems involving these matrices with separate subsidiary preconditioners.
One problem with this approach is that, once a compound matrix has been created (by the `get_block(...)` function), all information about its dof types is lost, making it impossible to employ block preconditioners as subsidiary preconditioners.
We will now revisit the 2x2 block triangular preconditioner described in the previous example and demonstrate how to employ subsidiary block preconditioners to (approximately) solve linear systems involving matrices formed (formally) by compound matrices that are constructed from multiple dof-level blocks. From a mathematical point of view, the structure of the preconditioner therefore remains unchanged and is given by

$$\mathbf{P} = \left(\begin{array}{cc|ccc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{13} & J_{14} & J_{15} \\ \hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & J_{53} & J_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline & B_{22} \end{array}\right). \tag{13}$$

We will continue to use a dof-to-block mapping to view this as the 2x2 block matrix shown on the right. This makes it easy to extract the compound off-diagonal block B_{12} from the system matrix when setting up the matrix-vector product (as before). The setup of the subsidiary block preconditioners used to (approximately) solve the linear systems involving B_{11} and B_{22} is handled differently:

- When calling the subsidiary block preconditioner's `setup(...)` function we pass a pointer to the entire system matrix, i.e. the matrix containing, formally, all the dof-level blocks in equation (3).
- We then turn the preconditioner into a subsidiary block preconditioner, using its member function `turn↩_into_subsidiary_block_preconditioner(...)` whose arguments specify which of the dof-level blocks in the current (master) preconditioner are to be used by the subsidiary block preconditioner.

The subsidiary block preconditioner is thus given access to all the information required to extract the relevant data directly from the original system matrix (and any associated full-length vectors). It is in fact a key design principle of the block preconditioning framework that **subsidiary block preconditioners are given access to the "full size" matrices and vectors, but only operate on the subset of data that they are "in charge of".**
When employing subsidiary block preconditioners for the approximate solution of the two smaller linear systems

$$\begin{aligned} \underbrace{\begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix}}_{B_{11}} \underbrace{\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}}_{Y_1} &= \underbrace{\begin{pmatrix} z_1 \\ z_2 \end{pmatrix}}_{\mathbf{Z}_1} - \underbrace{\begin{pmatrix} J_{13} & J_{14} & J_{15} \\ J_{23} & J_{24} & J_{25} \end{pmatrix}}_{B_{12}} \underbrace{\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix}}_{\mathbf{Y}_2} \\ \underbrace{\begin{pmatrix} J_{33} & J_{34} & J_{35} \\ J_{43} & J_{44} & J_{45} \\ J_{53} & J_{54} & J_{55} \end{pmatrix}}_{B_{22}} \underbrace{\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix}}_{\mathbf{Y}_2} &= \underbrace{\begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix}}_{\mathbf{Z}_2}, \end{aligned} \tag{14}$$

the subsidiary preconditioners that operate on the linear systems involving B_{11} and B_{22} therefore retain access to the relevant dof-level blocks. Hence, if we employ the block triangular preconditioner discussed above to (approximately) solve the two linear systems in equation (14), the complete preconditioning operation is described by the following equations:

$$\begin{pmatrix} J_{11} & J_{12} \\ & J_{22} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \underbrace{\underbrace{\begin{pmatrix} z_1 \\ z_2 \end{pmatrix}}_{\mathbf{Z}_1} - \underbrace{\begin{pmatrix} J_{13} & J_{14} & J_{15} \\ J_{23} & J_{24} & J_{25} \end{pmatrix}}_{B_{12}} \underbrace{\begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix}}_{\mathbf{Y}_2}}_{\widehat{\mathbf{Z}}_1}$$

$$\begin{pmatrix} J_{33} & J_{34} & J_{35} \\ & J_{44} & J_{45} \\ & & J_{55} \end{pmatrix} \begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix}.$$

Note that when we wrote the block triangular preconditioner we did not have to be aware of the fact that it may subsequently be used as a subsidiary block preconditioner. The internal data structures implemented in the `Block_Preconditioner` base class ensure that when we call `get_block(0,0,block_matrix)` in the subsidiary block preconditioner acting on B_{22}, `block_matrix` will receive a deep copy of J_{33}, extracted from the full system matrix. Similarly, a call to `get_block_vectors(r,block_r)` will extract the three block vectors $\mathbf{r}_3$, $\mathbf{r}_4$ and $\mathbf{r}_5$ from the full-length vector $\mathbf{r}$, while `return_block_vectors(block_z,z)` will return the three solution vectors $\mathbf{z_3}$, $\mathbf{z_4}$ and $\mathbf{z_5}$ to the appropriate entries in the full-length vector $\mathbf{z}$.
The implementation of the preconditioning operations can again be subdivided into two distinct `setup()` and `preconditioner_solve(...)` phases.

- Set up the data structures/lookup tables that map dof types 0 and 1 to block 0 and dof types 2, 3 and 4 to block 1 [by calling the `BlockPreconditioner::block_setup(...)` function with arguments that specify the mapping between "dof types" and "block types" as before].
- Create two instances of the block triangular preconditioner (or any other block preconditioner) and turn them into the subsidiary preconditioners for the current (master) preconditioner, specifying which dof types in the master preconditioner the subsidiary block preconditioners are to work with.
- Extract the compound off-diagonal block B_{12} and create a `MatrixVectorProduct` operator.

Once the `setup()` phase has been completed, the solution of the linear system $\mathbf{Py} = \mathbf{z}$ by the `preconditioner_solve(...)` function involves the following steps:

- Solve the linear systems $\mathbf{B}_{22}\ \mathbf{Y}_2 = \mathbf{Z}_2$ using the subsidiary block preconditioner that works with $\mathbf{B}_{22}$. [The subsidiary block preconditioner's `preconditioner_solve(...)` function is given access to the full-size vectors $\mathbf{z}$ and $\mathbf{y}$ and extracts/returns $\mathbf{Z}_2$ and $\mathbf{Y}_2$ directly from/into these.]
- Extract the solution vector $\mathbf{Y}_2$ from the just undated full-length vector $\mathbf{y}$, perform the matrix vector product with B_{12} and store the result in a temporary vector $\mathbf{t}$.
- Extract the block vector $\mathbf{Z}_1$ from the full-length vector $\mathbf{z}$, subtract $\mathbf{t}$ from it, and return the result, $\widehat{\mathbf{Z}}_1 = \mathbf{Z}_1 - B_{12}\mathbf{Z}_2$ into the appropriate entries into the full-length vector $\mathbf{z}$.
- Solve the linear systems $\mathbf{B}_{11}\ \mathbf{Y}_1 = \widehat{\mathbf{Z}}_1 = \mathbf{Z}_1 - B_{12}\mathbf{Z}_2$ using the subsidiary block preconditioner that works with $\mathbf{B}_{11}$. [The subsidiary block preconditioner's `preconditioner_solve(...)` function is given access to the full-size vectors $\mathbf{z}$ and $\mathbf{y}$ and extracts/returns $\widehat{\mathbf{Z}}_1$ and $\mathbf{Y}_1$ directly from/into these; recall that the relevant entries in $\mathbf{z}$ have been over-written in the previous step so that $\widehat{\mathbf{Z}}_1$ contains the updated right hand side.]

1.3.5.2 Implementation as a BlockPreconditioner

The implementation of the preconditioner is completely equivalent to the corresponding block triangular preconditioner considered in the previous example:

```
//=========start_of_two_plus_three_upper_triangular_with_sub_class=============
/// \short Upper block triangular with subsidiary block preconditioners
/// for a system with 5 dof types.
//============================================================================
 template<typename MATRIX>
 class TwoPlusThreeUpperTriangularWithOneLevelSubsidiary
  : public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor.
  TwoPlusThreeUpperTriangularWithOneLevelSubsidiary() :
   BlockPreconditioner<MATRIX>(),
    Off_diagonal_matrix_vector_product_pt(0),
    First_subsidiary_preconditioner_pt(0),
    Second_subsidiary_preconditioner_pt(0)
   {
    Multi_poisson_mesh_pt=0;
   }

  /// Destructor - delete the preconditioner matrices
  virtual ~TwoPlusThreeUpperTriangularWithOneLevelSubsidiary()
   {
    this->clean_up_my_memory();
   }

  /// Clean up the memory
```

```
  void clean_up_my_memory();

  /// Broken copy constructor
  TwoPlusThreeUpperTriangularWithOneLevelSubsidiary
  (const TwoPlusThreeUpperTriangularWithOneLevelSubsidiary&)
   {
    BrokenCopy::broken_copy
     ("TwoPlusThreeUpperTriangularWithOneLevelSubsidiary");
   }

  /// Broken assignment operator
  void operator=(const
                 TwoPlusThreeUpperTriangularWithOneLevelSubsidiary&)
   {
    BrokenCopy::broken_assign(
     "TwoPlusThreeUpperTriangularWithOneLevelSubsidiary");
   }

  /// Apply preconditioner to r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  void setup();
  // Use the version in the Preconditioner base class for the alternative
  // setup function that takes a matrix pointer as an argument.
  using Preconditioner::setup;

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private:

  /// Pointer to matrix vector product operators for the off diagonal block
  MatrixVectorProduct* Off_diagonal_matrix_vector_product_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (0,0) block
  Preconditioner* First_subsidiary_preconditioner_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (1,1) block
  Preconditioner* Second_subsidiary_preconditioner_pt;

  /// \short Pointer to mesh with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.5.3 The setup() function

As usual we free up any memory and set the pointer to the mesh:

```
 //=======start_of_setup_for_two_plus_three_upper_triangular_with_sub_class====
 /// The setup function.
 //============================================================================
 template<typename MATRIX>
 void TwoPlusThreeUpperTriangularWithOneLevelSubsidiary<MATRIX>::setup()
 {
  // clean the memory
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err << "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
```

We check that the number of dof types is appropriate for this preconditioner:

```
  // number of dof types
  unsigned n_dof_types = this->ndof_types();
#ifdef PARANOID
  // This preconditioner only works for 5 dof types
  if (n_dof_types!=5)
   {
    std::stringstream tmp;
    tmp << "This preconditioner only works for problems with 5 dof types\n"
        << "Yours has " << n_dof_types;
```

```
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

Next we define the block structure of the preconditioner, using a dof-to-block mapping to combine dofs 0 and 1 into block 0, and dofs 2, 3 and 4 into block 1:

```
  // Combine "dof blocks" into two compound blocks, one containing dof
  // types 0 and 1, the final one dof types 2-4. In general we want:
  // dof_to_block_map[dof_type] = block type
  Vector<unsigned> dof_to_block_map(n_dof_types);
  dof_to_block_map[0]=0;
  dof_to_block_map[1]=0;
  dof_to_block_map[2]=1;
  dof_to_block_map[3]=1;
  dof_to_block_map[4]=1;
  this->block_setup(dof_to_block_map);
```

Next we create the block triangular preconditioner used to (approximately) solve linear systems involving the compound "top left" 2x2 block:

```
  // Create the subsidiary block preconditioners.
  {
   // Block upper triangular block preconditioner for compound
   // 2x2 top left block in "big" 5x5 matrix
   UpperTriangular<CRDoubleMatrix>* block_prec_pt=
    new UpperTriangular<CRDoubleMatrix>;
   First_subsidiary_preconditioner_pt=block_prec_pt;
```

Next we specify the pointer to the mesh that contains the elements that classify the degrees of freedom. We note, that, strictly speaking this is not necessary since the preconditioner will only be used as a subsidiary preconditioner – the enumeration of the dof types is always handled by the top-most master preconditioner. One (or more) mesh pointers must be set for the master preconditioner, and, if compiled in `PARANOID` mode, `oomph-lib` will throw an error if this is not done. Some (but not all!) `oomph-lib` developers regard it as "good practice" to set the mesh pointer anyway, so one is less likely to forget...

```
   // Set mesh
   block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);
```

We turn this preconditioner into a subsidiary block preconditioner, specifying the pointer to the current (master) preconditioner and the mapping between dof types in the present and the subsidiary block preconditioners (here the identity):

```
   // Turn into a subsidiary preconditioner, declaring which
   // of the five dof types in the present (master) preconditioner
   // correspond to the dof types in the subsidiary block preconditioner:
   // dof_map[dof_block_ID_in_subsdiary] = dof_block_ID_in_master. Also
   // pass pointer to present (master) preconditioner.
   unsigned n_sub_dof_types=2;
   Vector<unsigned> dof_map(n_sub_dof_types);
   dof_map[0]=0;
   dof_map[1]=1;
   block_prec_pt->turn_into_subsidiary_block_preconditioner(this,dof_map);
```

When calling the subsidiary block preconditioners `setup(...)` function we pass a pointer to the full matrix:

```
   // Setup: Pass pointer to full-size matrix!
   block_prec_pt->setup(this->matrix_pt());
  }
```

The second subsidiary block preconditioner (for the 3x3 "bottom right" compound matrix) is created similarly, though the mapping between dof-types is now no longer the identity but maps dof types 2, 3 and 4 in the current (master) preconditioner to dof types 0, 1 and 2 in the subsidiary block preconditioner:

```
  {
   // Block upper triangular for 3x3 bottom right block in "big" 5x5 matrix
   UpperTriangular<CRDoubleMatrix>* block_prec_pt=
    new UpperTriangular<CRDoubleMatrix>;
   Second_subsidiary_preconditioner_pt=block_prec_pt;

   // Set mesh
   block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);

   // Turn second_sub into a subsidiary preconditioner, declaring which
   // of the five dof types in the present (master) preconditioner
   // correspond to the dof types in the subsidiary block preconditioner:
   // dof_map[dof_block_ID_in_subsdiary] = dof_block_ID_in_master. Also
   // pass pointer to present (master) preconditioner.
   unsigned n_sub_dof_types=3;
   Vector<unsigned> dof_map(n_sub_dof_types);
   dof_map[0]=2;
   dof_map[1]=3;
   dof_map[2]=4;
   block_prec_pt->turn_into_subsidiary_block_preconditioner(this,dof_map);
   // Setup: Pass pointer to full-size matrix!
   block_prec_pt->setup(this->matrix_pt());
  }
```

The setup of the matrix-vector product with the off-diagonal matrix is unchanged from the previous example:

```
  // Setup the off-diagonal mat vec operator
  {
   // Get the off-diagonal block: the top-right block in the present
   // block preconditioner (which views the system matrix as comprising
   // 2x2 blocks).
   CRDoubleMatrix block_matrix = this->get_block(0,1);

   // Create matrix vector product
   Off_diagonal_matrix_vector_product_pt = new MatrixVectorProduct;

   // Setup: Final argument indicates block column in the present
   // block preconditioner (which views the system matrix as comprising
   // 2x2 blocks).
   unsigned block_column_index=1;
   this->setup_matrix_vector_product(
    Off_diagonal_matrix_vector_product_pt,&block_matrix,block_column_index);
  }

 }
```

1.3.5.4 The preconditioner_solve() function

As discussed in the theory section, we start by (approximately) solving the system $B_{22}Y_2 = Z_2$, using the second subsidiary block preconditioner which automatically extracts Z_2 from the full length vector `z` and returns the result Y_2 into the appropriate entries of the full length vector `y`.

```
 //=============================================================================
 /// Preconditioner solve
 //=============================================================================
 template<typename MATRIX>
 void TwoPlusThreeUpperTriangularWithOneLevelSubsidiary<MATRIX>::
 preconditioner_solve(const DoubleVector& z, DoubleVector& y)
 {
  // Solve "bottom right" (1,1) diagonal block system, using the
  // subsidiary block preconditioner that acts on the
  // "bottom right" 3x3 sub-system (only!). The subsidiary preconditioner
  // will extract the relevant (3x1) "sub-vectors" from the "big" (5x1)
  // vector z and treat it as the rhs, r, of P y = z
  // where P is 3x3 a block matrix. Once the system is solved,
  // the result is automatically put back into the appropriate places
  // of the "big" (5x1) vector y:
  Second_subsidiary_preconditioner_pt->preconditioner_solve(z,y);
```

We now extract the block vector Y_2 from the full-length vector `y`,

```
  // Now extract the "bottom" (3x1) block vector from the full-size (5x1)
  // solution vector that we've just computed -- note that index 1
  // refers to the block enumeration in the current preconditioner
  // (which has two blocks!)
  DoubleVector block_y;
  this->get_block_vector(1,y,block_y);
```

multiply it by B_{12}, using the `MatrixVectorProduct` operator,

```
  // Evaluate matrix vector product of just-extracted (3x1) solution
  // vector with off-diagonal block and store in temporary vector
  DoubleVector temp;
  Off_diagonal_matrix_vector_product_pt->multiply(block_y,temp);
```

and subtract the result from Z_1 (which we extract from the full length vector `z`):

```
  // Extract "upper" (2x1) block vector from full-size (5x1) rhs
  // vector (as passed into this function)...
  DoubleVector block_z;
  this->get_block_vector(0,z,block_z);
  // ...and subtract matrix vector product computed above
  block_z -= temp;
```

`block_z` now contains the updated right hand side, $\widehat{\mathbf{Z}_1}$, for the linear system to be (approximately) solved by the first subsidiary block preconditioner. We therefore return $\widehat{\mathbf{Z}_1}$ to the appropriate entries into a full length vector of the same size as right hand side vector `z` :

```
  // Block solve for first diagonal block. Since the associated subsidiary
  // preconditioner is a block preconditioner itself, it will extract
  // the required (2x1) block from a "big" (5x1) rhs vector.
  // Therefore we first put the actual (2x1) rhs vector block_z into a
  // "big" (5x1) vector big_z whose row distribution matches that of the
  // "big" right hand side vector, z, that was passed into this function.
  DoubleVector big_z(z.distribution_pt());
  this->return_block_vector(0,block_z,big_z);
```

We then pass this vector to first subsidiary preconditioner which updates the appropriate entries in the full-length solution vector `y` which can therefore be returned directly by this function:

```
  // Now apply the subsidiary block preconditioner that acts on the
  // "upper left" (2x2) sub-system (only!). The subsidiary preconditioner
  // will extract the relevant (2x1) block vector from the "big" (5x1)
  // vector big_r and treat it as the rhs, z, of its P y = z
  // where P is upper left 2x2 block diagonal of the big system.
```

```
 // Once the system is solved, the result is automatically put back
 // into the appropriate places of the "big" (5x1) vector y which is
 // returned by the current function, so no further action is required.
 First_subsidiary_preconditioner_pt->preconditioner_solve(big_z,y);
}
```

1.3.5.5 The clean_up_my_memory() function

As usual, we use this helper function to free up any memory allocated in the `setup()` function to avoid memory leaks.

```
//====start_of_clean_up_for_two_plus_three_upper_triangular_with_sub_class=====
/// The clean up function.
//=============================================================================
template<typename MATRIX>
void TwoPlusThreeUpperTriangularWithOneLevelSubsidiary<MATRIX>::
clean_up_my_memory()
{
 // Delete off-diagonal matrix vector product
 if(Off_diagonal_matrix_vector_product_pt!= 0)
  {
   delete Off_diagonal_matrix_vector_product_pt;
   Off_diagonal_matrix_vector_product_pt = 0;
  }
 //Clean up subsidiary preconditioners.
 if(First_subsidiary_preconditioner_pt!=0)
  {
   delete First_subsidiary_preconditioner_pt;
   First_subsidiary_preconditioner_pt = 0;
  }
 if(Second_subsidiary_preconditioner_pt!=0)
  {
   delete Second_subsidiary_preconditioner_pt;
   Second_subsidiary_preconditioner_pt = 0;
  }
} // End of clean_up_my_memory function.
```

1.3.6 Replacing/modifying blocks

NEW FEATURES: How to replace/modify matrix blocks

1.3.6.1 Theory

So far, we have demonstrated how to extract matrix blocks from the full-sized system matrix (typically the Jacobian matrix used in Newton's method) and how to apply a preconditioner via operations involving these blocks. Many preconditioners do not operate directly with the matrix blocks themselves, but on matrices that are derived from them. For instance, `oomph-lib`'s `Schur complement Navier-Stokes preconditioner` operates on an (approximate) Schur complement; augmentation preconditioners involve operations on matrices that are obtained by the addition of a diagonal matrix to some of the matrix blocks; etc. Within a given preconditioner such derived matrices are typically pre-computed by the preconditioner's `setup()` function and then stored as private member data which makes them available to the `preconditioner_solve()` function. Unfortunately, this approach does not work if the modified block is to be used in a subsidiary block preconditioner because, as discussed in the previous example, by default the subsidiary block preconditioner will extract its block matrices directly from the full-size system matrix and will therefore ignore any (local) modifications made by its master preconditioner(s). What is therefore required is a method that indicates to the block preconditioning framework that a given sub-block is not to be extracted from the full system matrix but to be represented by suitable replacement matrix.
We demonstrate this methodology by re-visiting the preconditioner considered in the previous example, namely

$$\mathbf{P}_{\text{previous}} = \left(\begin{array}{cc|ccc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{13} & J_{14} & J_{15} \\ \hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & J_{53} & J_{54} & J_{55} \end{array}\right). \tag{15}$$

However, here we want to modify the off-diagonal blocks by "replacing" each block J_{ij} (for $i \neq j$) by a "replacement matrix" R_{ij} so that the preconditioner becomes

$$\mathbf{P} = \left(\begin{array}{cc|ccc} J_{11} & R_{12} & R_{13} & R_{14} & R_{15} \\ R_{21} & J_{22} & R_{23} & R_{24} & R_{25} \\ \hline & & J_{33} & R_{34} & R_{35} \\ & & R_{43} & J_{44} & R_{45} \\ & & R_{53} & R_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline & B_{22} \end{array}\right). \tag{16}$$

The application of this preconditioner (i.e. the solution of the linear system $\mathbf{Py} = \mathbf{z}$ for $\mathbf{y}$) still requires the solution of the two smaller linear systems

$$\underbrace{\begin{pmatrix} J_{11} & R_{12} \\ R_{21} & J_{22} \end{pmatrix}}_{B_{11}} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} - \underbrace{\begin{pmatrix} R_{13} & R_{14} & R_{15} \\ R_{23} & R_{24} & R_{25} \end{pmatrix}}_{B_{12}} \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix}$$

$$\underbrace{\begin{pmatrix} J_{33} & R_{34} & R_{35} \\ R_{43} & J_{44} & R_{45} \\ R_{53} & R_{54} & J_{55} \end{pmatrix}}_{B_{22}} \begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix} \tag{17}$$

where we have again assumed that the two vectors $\mathbf{y}$ and $\mathbf{z}$ are re-ordered into `'block vectors'' in the same way as the vectors $\delta\mathbf{x}$ and $\mathbf{r}$ in "the original linear system" (3) are re-ordered into the `'block vectors'' in (17). We wish to continue to solve the linear systems involving the compound matrices B_{11} and B_{22} (which involve "replaced" blocks) by two subsidiary block preconditioners (which operate on 3x3 and 2x2 dof blocks, respectively).
In the specific example below we replace all of the diagonal matrices by suitably sized zero matrices, so that the actual preconditioning operation is defined by the following linear systems

$$\underbrace{\begin{pmatrix} J_{11} & \\ & J_{22} \end{pmatrix}}_{B_{11}} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} - \underbrace{\begin{pmatrix} & \\ & \end{pmatrix}}_{B_{12}} \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix}$$

$$\underbrace{\begin{pmatrix} J_{33} & & \\ & J_{44} & \\ & & J_{55} \end{pmatrix}}_{B_{22}} \begin{pmatrix} y_3 \\ y_4 \\ y_5 \end{pmatrix} = \begin{pmatrix} z_3 \\ z_4 \\ z_5 \end{pmatrix}$$

which, in effect, turns the preconditioner into the block-diagonal preconditioner considered at the very beginning of this tutorial.

1.3.6.2 Implementation as a BlockPreconditioner

The implementation of the preconditioner is completely equivalent to the preconditioner considered in the previous example. The only additional feature is the provision a matrix of pointers to the replacement matrices, `Replacement_matrix_pt`.

```
//=============start_of_two_plus_three_upper_triangular_with_replace_class=====
/// \short Block diagonal preconditioner for system with 5 dof types
/// assembled into a 2x2 block system, with (0,0) block containing
/// the first two dof types, the (1,1) block the remaining dof types.
/// The blocks are solved by upper block triangular preconditioners.
/// However, the overall system is modified by replacing all off-diagonal
/// blocks by replacement matrices (zero matrices, so the preconditioner
/// again behaves like a 5x5 block diagonal preconditioner).
//======================================================================
 template<typename MATRIX>
 class TwoPlusThreeUpperTriangularWithReplace :
  public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor for TwoPlusThreeUpperTriangularWithReplace
  TwoPlusThreeUpperTriangularWithReplace() :
   BlockPreconditioner<MATRIX>(),
   First_subsidiary_preconditioner_pt(0),
   Second_subsidiary_preconditioner_pt(0),
   Off_diagonal_matrix_vector_product_pt(0)
   {
    Multi_poisson_mesh_pt=0;
   } // end_of_constructor


  /// Destructor clean up memory
  ~TwoPlusThreeUpperTriangularWithReplace()
   {
    this->clean_up_my_memory();
   }

  /// Clean up the memory
```

```
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  TwoPlusThreeUpperTriangularWithReplace
  (const TwoPlusThreeUpperTriangularWithReplace&)
   {
    BrokenCopy::
     broken_copy("TwoPlusThreeUpperTriangularWithReplace");
   }

  /// Broken assignment operator
  void operator=(const TwoPlusThreeUpperTriangularWithReplace&)
   {
    BrokenCopy::
     broken_assign("TwoPlusThreeUpperTriangularWithReplace");
   }

  /// Apply preconditioner to r, i.e. return z such that P z = r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  void setup();

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
   private :

  /// \short Pointer to preconditioners/inexact solver
  /// for compound  (0,0) block
  Preconditioner* First_subsidiary_preconditioner_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for compound (1,1) block
  Preconditioner* Second_subsidiary_preconditioner_pt;

  /// \short Matrix vector product operator with the compound
  /// (0,1) off diagonal block.
  MatrixVectorProduct* Off_diagonal_matrix_vector_product_pt;
  // Matrix of pointers to replacement matrix blocks
  DenseMatrix<CRDoubleMatrix*> Replacement_matrix_pt;

  /// \short Pointer to mesh with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;

 };
```

1.3.6.3 The setup() function

As usual, we start by cleaning up any memory using a call to a `clean_up_my_memory()` function, and set the pointer to the mesh

```
 //==start_of_setup_for_two_plus_three_upper_triangular_with_replace===========
 /// The setup function.
 //============================================================================
 template<typename MATRIX>
 void TwoPlusThreeUpperTriangularWithReplace<MATRIX>::setup()
 {
  // Clean up memory.
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err « "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
```

Next we check that the number of dof types is 5, as the preconditioner is designed to only work for that number.

```
  // How many dof types do we have?
  const unsigned n_dof_types = this->ndof_types();
#ifdef PARANOID
  // This preconditioner only works for 5 dof types
  if (n_dof_types!=5)
   {
    std::stringstream tmp;
```

```
    tmp « "This preconditioner only works for problems with 5 dof types\n"
        « "Yours has " « n_dof_types;
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

The block setup follows exactly the same pattern as in the previous example: Dof types 0 and 1 are combined into compound block 0, while dof types 2, 3 and 4 are combined into compound block 1.
On return from the block setup function we should therefore have two block types:

```
  // Call block setup with the Vector [0,0,1,1,1] to:
  // Merge DOF types 0 and 1 into block type 0
  // Merge DOF types 2, 3, and 4 into block type 1.
  Vector<unsigned> dof_to_block_map(n_dof_types,0);
  dof_to_block_map[0] = 0;
  dof_to_block_map[1] = 0;
  dof_to_block_map[2] = 1;
  dof_to_block_map[3] = 1;
  dof_to_block_map[4] = 1;
  this->block_setup(dof_to_block_map);
#ifdef PARANOID
  // We should now have two block types -- do we?
  const unsigned nblocks = this->nblock_types();
  if (nblocks!=2)
   {
    std::stringstream tmp;
    tmp « "Expected number of block types is 2.\n"
        « "Yours has " « nblocks « ".\n"
        « "Perhaps your argument to block_setup(...) is not correct.\n";
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

Now we perform the replacement of the off-diagonal dof blocks. (Note that there are still five of these. Dof-blocks and compound blocks are not the same – if you get them confused you will get into trouble!). We allocate storage for the pointers to the replacement matrices and loop over the off-diagonal blocks:

```
  // Now replace all the off-diagonal DOF blocks.
  // Storage for the replacement DOF blocks
  Replacement_matrix_pt.resize(n_dof_types,n_dof_types,0);
  // Set off-diagonal DOF blocks to zero, loop over the number of DOF blocks.
  // NOTE: There are two (compound) blocks, but the replacement functionality
  // works with DOF blocks.
  for(unsigned i=0;i<n_dof_types;i++)
   {
    for(unsigned j=0;j<n_dof_types;j++)
     {
      if(i!=j)
       {
```

Given that the replacement matrices are zero matrices, we could simply create them without ever looking at the original blocks. Sadly the creation of zero matrices turns out to be slightly more painful than one would wish because they have to be created as a (possibly distributed) `CRDoubleMatrix`. The relevant code is contained in the source code but we won't discuss it here since the more common situation is one where we actually want to modify the already existing entries of an already existing block matrix. Therefore we simply extract the matrix and set its initially nonzero entries to zero (admittedly a bit silly – we now have a sparse matrix full of zeroes, but it's just a demonstration!):

```
        // Modify matrix
        bool modify_existing_matrix=true;
        if (modify_existing_matrix)
         {
          // Get the dof-block and make a deep copy of it
          Replacement_matrix_pt(i,j)=new CRDoubleMatrix;
          this->get_dof_level_block(i,j,(*Replacement_matrix_pt(i,j)));

          // Set all its entries to zero
          unsigned nnz=Replacement_matrix_pt(i,j)->nnz();
          for (unsigned k=0;k<nnz;k++)
           {
            Replacement_matrix_pt(i,j)->value()[k]=0.0;
           }
         } // done -- quite wasteful, we're actually storing lots of zeroes, but
           // this is just an example!
```

We then pass the pointer to the replacement dof block to the block preconditioner

```
        // Replace (i,j)-th dof block
        this->set_replacement_dof_block(i,j,Replacement_matrix_pt(i,j));
       }
     }// end for loop of j
   }// end for loop of i
```

The rest of the setup works exactly as in the previous example, only this time, the subsidiary preconditioners and

the matrix vector products will work with the replacement dof blocks that we've just defined.
We create and set up the first subsidiary block preconditioner which operates on our dof types 0 and 1 (and treats them as its own dof types 0 and 1):

```
  // First subsidiary precond is a block triangular preconditioner
  {
   UpperTriangular<CRDoubleMatrix>* block_prec_pt=
    new UpperTriangular<CRDoubleMatrix>;
   First_subsidiary_preconditioner_pt=block_prec_pt;

   // Set mesh
   block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);

   // Turn it into a subsidiary preconditioner, declaring which
   // of the five dof types in the present (master) preconditioner
   // correspond to the dof types in the subsidiary block preconditioner
   unsigned n_sub_dof_types=2;
   Vector<unsigned> dof_map(n_sub_dof_types);
   dof_map[0]=0;
   dof_map[1]=1;
   block_prec_pt->turn_into_subsidiary_block_preconditioner(this,dof_map);
    // Perform setup. Note that because the subsidiary
   // preconditioner is a block preconditioner itself it is given
   // the pointer to the "full" matrix
   block_prec_pt->setup(this->matrix_pt());
  }
```

The second subsidiary block preconditioner which operates on our dof types 2, 3 and 4 (and treats them as its own dof types 0, 1 and 2):

```
  // Second subsidiary precond is a block triangular preconditioner
  {
   UpperTriangular<CRDoubleMatrix>* block_prec_pt=
    new UpperTriangular<CRDoubleMatrix>;
   Second_subsidiary_preconditioner_pt=block_prec_pt;

   // Set mesh
   block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);

   // Turn it into a subsidiary preconditioner, declaring which
   // of the five dof types in the present (master) preconditioner
   // correspond to the dof types in the subsidiary block preconditioner
   unsigned n_sub_dof_types=3;
   Vector<unsigned> dof_map(n_sub_dof_types);
   dof_map[0]=2;
   dof_map[1]=3;
   dof_map[2]=4;
   block_prec_pt->turn_into_subsidiary_block_preconditioner(this,dof_map);

   // Perform setup. Note that because the subsidiary
   // preconditioner is a block preconditioner itself it is given
   // the pointer to the "full" matrix
   block_prec_pt->setup(this->matrix_pt());
  }
```

Finally, we create the matrix vector product operator:

```
  // Next setup the off diagonal mat vec operators:
  {
   // Get the block
   CRDoubleMatrix block_matrix = this->get_block(0,1);

   // Create matrix vector product operator
   Off_diagonal_matrix_vector_product_pt = new MatrixVectorProduct;

   // Setup: Final argument indicates block column in the present
   // block preconditioner (which views the system matrix as comprising
   // 2x2 blocks).
   unsigned block_column_index=1;
   this->setup_matrix_vector_product(
   Off_diagonal_matrix_vector_product_pt,&block_matrix,block_column_index);
   // Extracted block can now go out of scope since the matrix vector
   // product retains whatever information it needs
  }
 }
```

1.3.6.4 The preconditioner_solve() function

The `preconditioner_solve()` function is completely identical to the one used in the previous preconditioner, so we omit the code listing – the subsidiary preconditioners and the matrix vector product operator work in the same way but now simply operate on the replacement dof blocks where they have been set.

1.3.6.5 The clean_up_my_memory() function

Memory is cleaned up as before, so we omit the code listing.

1.3.7 Coarsening/combining dof types

NEW FEATURES: How to coarsen/combine dof types for use by subsidiary block preconditioners.

1.3.7.1 Theory

In the examples presented so far we have demonstrated how to combine various dof-blocks into compound blocks in order to facilitate the application of certain preconditioning operations. For instance, in many of the previous examples we performed a matrix vector product using the compound matrix B_{12} that was (formally) formed by the concatenation of the 2x3 "top right" off-diagonal dof blocks in the full-sized system.
We also showed how subsidiary block preconditioners which operate on a specific number of dof blocks can be instructed to operate on selected dof types from the full-sized system. Our standard example for this was a 2D Navier-Stokes preconditioner which operates on three dof types (two fluid velocities and one pressure) and is used as a subsidiary block preconditioner in an FSI problem that also involves additional dofs associated with the solid mechanics (e.g. the two solid displacement components). This was done by informing the subsidiary preconditioner which of the dof types in the full-sized system to regard as "its own" when calling its `turn_into_subsidiary_block_preconditioner(...)` function. This implies that the subsidiary block preconditioner remains unaware of any compound blocks that may have been formed in its master preconditioner. The functionality presented so far only allows us to associate dof-blocks in the master preconditioner with dof blocks in the subsidiary block preconditioner. It is therefore not possible (without further functionality which we explain in this example) to use a subsidiary block preconditioner if the dof-types in the master preconditioner are "too fine-grained". This arises, for instance, in Navier-Stokes problems where the master preconditioner sub-divides the two components of the fluid velocity into degrees of freedom on the domain boundary and those in the interior. It is then necessary to make the subsidiary preconditioner act on the combined dof types, a process that we describe as "coarsening".
We illustrate the procedure by returning, yet again, to our 5x5 block linear system that we wish to precondition with

$$\mathbf{P}_{\text{previous}} = \left(\begin{array}{cc|ccc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{13} & J_{14} & J_{15} \\ \hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & J_{53} & J_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline & B_{22} \end{array}\right). \tag{18}$$

However, now we wish to solve the two linear systems involving the compound matrices B_{11} and B_{22} with a 2x2 upper triangular subsidiary block preconditioner. To make this possible, we "coarsen" the dof types such that the subsidiary block preconditioner acting on B_{22} treats the global dof types 3 and 4 as a single dof type so that the block structure can be viewed as

$$\mathbf{P} = \left(\begin{array}{cc||ccc} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ J_{21} & J_{22} & J_{23} & J_{24} & J_{25} \\ \hline\hline & & J_{33} & J_{34} & J_{35} \\ & & J_{43} & J_{44} & J_{45} \\ & & \hline J_{53} & J_{54} & J_{55} \end{array}\right) = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline & B_{22} \end{array}\right). \tag{19}$$

If we now use a 2x2 upper triangular block preconditioner to (approximately) solve the linear systems involving the diagonal blocks B_{11} and B_{22} the preconditioner is given (mathematically) by

$$\mathbf{P} = \begin{pmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} \\ & J_{22} & J_{23} & J_{24} & J_{25} \\ & & J_{33} & J_{34} & J_{35} \\ & & J_{34} & J_{44} & J_{45} \\ & & & & J_{55} \end{pmatrix}.$$

[Note that, In the actual implementation discussed below, we also set the off diagonal dof-blocks to zero, using the replacement methodology discussed in the previous example. The preconditioner therefore becomes mathematically equivalent to the $5x5$ block diagonal preconditioner discussed at the very beginning of this tutorial.]

1.3.7.2 Implementation as a BlockPreconditioner

The implementation of the preconditioner is completely equivalent to the preconditioner considered in the previous examples:

```
//==================start_of_coarse_two_plus_two_plus_one_class================
```

```
/// \short Block diagonal preconditioner for system with 5 dof types
/// assembled into a 2x2 block system, with the (0,0) block containing
/// the first two dof types, the (1,1) block containing the three remaining
/// ones.
//======================================================================
 template<typename MATRIX>
 class CoarseTwoPlusTwoPlusOne :
  public BlockPreconditioner<MATRIX>
 {

 public :

  /// Constructor for CoarseTwoPlusTwoPlusOne
   CoarseTwoPlusTwoPlusOne() :
  BlockPreconditioner<MATRIX>(),
   First_subsidiary_preconditioner_pt(0),
   Second_subsidiary_preconditioner_pt(0),
   Off_diagonal_matrix_vector_product_pt(0)
    {
     Multi_poisson_mesh_pt=0;
    } // end_of_constructor


  /// Destructor - delete the diagonal solvers (subsidiary preconditioners)
  ~CoarseTwoPlusTwoPlusOne()
   {
    this->clean_up_my_memory();
   }

  /// clean up the memory
  virtual void clean_up_my_memory();

  /// Broken copy constructor
  CoarseTwoPlusTwoPlusOne
  (const CoarseTwoPlusTwoPlusOne&)
   {
    BrokenCopy::broken_copy(
     "CoarseTwoPlusTwoPlusOne");
   }

  /// Broken assignment operator
  void operator=(const
                 CoarseTwoPlusTwoPlusOne&)
   {
    BrokenCopy::broken_assign(
     "CoarseTwoPlusTwoPlusOne");
   }

  /// Apply preconditioner to r, i.e. return z such that P z = r
  void preconditioner_solve(const DoubleVector &r, DoubleVector &z);

  /// \short Setup the preconditioner
  virtual void setup();

  /// Specify the mesh that contains multi-poisson elements
  void set_multi_poisson_mesh(Mesh* multi_poisson_mesh_pt)
  {
   Multi_poisson_mesh_pt=multi_poisson_mesh_pt;
  }
 private :

  /// \short Pointer to preconditioners/inexact solver
  /// for (0,0) block
  Preconditioner* First_subsidiary_preconditioner_pt;

  /// \short Pointer to preconditioners/inexact solver
  /// for (1,1) block
  Preconditioner* Second_subsidiary_preconditioner_pt;
  // Matrix of pointers to replacement matrix blocks
  DenseMatrix<CRDoubleMatrix*> Replacement_matrix_pt;

  /// Matrix vector product operator
  MatrixVectorProduct* Off_diagonal_matrix_vector_product_pt;

  /// \short Pointer to mesh with preconditionable elements used
  /// for classification of dof types.
  Mesh* Multi_poisson_mesh_pt;
 };
```

1.3.7.3 The setup() function

As usual we clean up any previously allocated memory and set the pointer to the mesh:

```
 //===============start_of_setup_for_coarse_two_plus_two_plus_one=============
 /// The setup function.
 //======================================================================
```

```
 template<typename MATRIX>
 void CoarseTwoPlusTwoPlusOne<MATRIX>::setup()
 {
  // Clean up memory
  this->clean_up_my_memory();
#ifdef PARANOID
  if (Multi_poisson_mesh_pt == 0)
   {
    std::stringstream err;
    err « "Please set pointer to mesh using set_multi_poisson_mesh(...).\n";
    throw OomphLibError(err.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif

  // The preconditioner works with one mesh; set it!
  this->set_nmesh(1);
  this->set_mesh(0,Multi_poisson_mesh_pt);
```

Next we check that the number of degrees of freedom is 5, as the preconditioner is designed to only work for that number.

```
  // This preconditioner only works for 5 dof types
  unsigned n_dof_types = this->ndof_types();
#ifdef PARANOID
  if (n_dof_types!=5)
   {
    std::stringstream tmp;
    tmp « "This preconditioner only works for problems with 5 dof types\n"
        « "Yours has " « n_dof_types;
    throw OomphLibError(tmp.str(),
                        OOMPH_CURRENT_FUNCTION,
                        OOMPH_EXCEPTION_LOCATION);
   }
#endif
```

The block setup follows exactly the same pattern as in the previous examples: Dof types 0 and 1 are combined into compound block 0, while dof types 2, 3 and 4 are combined into compound block 1.

```
  // Call block setup with the Vector [0,0,1,1,1] to:
  // Merge DOF types 0 and 1 into block type 0.
  // Merge DOF types 2, 3 and 4 into block type 1.
  Vector<unsigned> dof_to_block_map(n_dof_types,0);
  dof_to_block_map[0] = 0;
  dof_to_block_map[1] = 0;
  dof_to_block_map[2] = 1;
  dof_to_block_map[3] = 1;
  dof_to_block_map[4] = 1;
  this->block_setup(dof_to_block_map);
```

[We omit the code listing the replacement of the off-diagonal dof blocks with zero matrices since it is identical to what we already discussed in the previous example.]

Next we create the two subsidiary preconditioners that (approximately) solve the linear systems involving the diagonal blocks B_{11} and B_{22}. The first subsidiary preconditioner is a standard upper triangular block preconditioner which acts on the compound block formed by dof types 0 and 1:

```
  // Create the subsidiary preconditioners
  //--------------------------------------
  {
   // First subsidiary precond is a block diagonal preconditioner itself.
   UpperTriangular<CRDoubleMatrix>* block_prec_pt=
    new UpperTriangular<CRDoubleMatrix>;
   First_subsidiary_preconditioner_pt=block_prec_pt;
   // Set mesh
   block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);

   // Turn first_sub into a subsidiary preconditioner, declaring which
   // of the five dof types in the present (master) preconditioner
   // correspond to the dof types in the subsidiary block preconditioner
   const unsigned n_sub_dof_types=2;
   Vector<unsigned> dof_map(n_sub_dof_types);
   dof_map[0]=0;
   dof_map[1]=1;
   block_prec_pt->turn_into_subsidiary_block_preconditioner(this,dof_map);
   // Perform setup. Note that because the subsidiary
   // preconditioner is a block preconditioner itself it is given
   // the pointer to the "full" matrix
   block_prec_pt->setup(this->matrix_pt());
  }
```

The second subsidiary preconditioner is more interesting. It's a block preconditioner that only operates on a 2x2 block system, yet we want to use it to solve the linear system involving the compound block formed the three dof types 2, 3 and 4. To do this we wish to combine the dof blocks associated with dof types 2 and 3 into a single block. We start by setting the mesh pointer and by setting up the usual mapping that identifies the dof types (in the current preconditioner) that we wish the subsidiary preconditioner to act on.

```
  // Second subsidiary preconditioner is also a block preconditioner
```

```
 {
  SimpleTwoDofOnly<CRDoubleMatrix>* block_prec_pt=
   new SimpleTwoDofOnly<CRDoubleMatrix>;
  Second_subsidiary_preconditioner_pt=block_prec_pt;

  // Set mesh
  block_prec_pt->set_multi_poisson_mesh(Multi_poisson_mesh_pt);

  // This is the usual mapping between the subsidiary and master dof types.
  Vector<unsigned> dof_map(3);
  dof_map[0]=2;
  dof_map[1]=3;
  dof_map[2]=4;
```

To combine/coarsen dof types 2 and 3 (in the current preconditioner) into a single dof type for the subsidiary preconditioner we create a vector of vectors, `doftype_coarsening` whose entries are to be interpreted as `doftype_coarsening[coarsened_dof_type][i]=dof_type` where `i` ranges from 0 to the number of dof types (minus one, because of the zero-based indexing...) in the enumeration of the subsidiary preconditioner that are to be combined/coarsened into dof type `coarsened_dof_type`:

```
  // The subsidiary block preconditioner SimpleTwoDofOnly accepts only two
  // dof types. We therefore have to "coarsen" the 3 dof types into two
  // by specifying the vector of vectors doftype_coarsening whose
  // entries are to be interpreted as
  //
  //    doftype_coarsening[coarsened_dof_type][i]=dof_type
  //
  // where i ranges from 0 to the number of dof types (minus one, because
  // of the zero-based indexing...) that are to be
  // combined/coarsened into dof type dof_type_in_coarsed_block_preconditioner
  // Number of dof types the subsidiary block preconditioner expects.
  const unsigned n_sub_dof_types=2;
  Vector<Vector<unsigned> > doftype_coarsening(n_sub_dof_types);

  // Subsidiary dof type 0 contains 2 dof types.
  doftype_coarsening[0].resize(2);
  // Coarsen subsidiary dof types 0 and 1 into subsidiary dof type 0.
  doftype_coarsening[0][0]=0;
  doftype_coarsening[0][1]=1;

  // Subsidiary dof type 1 contains 1 dof types.
  doftype_coarsening[1].resize(1);

  // Subsidiary Dof type 1 contains subsidiary dof type 2.
  doftype_coarsening[1][0]=2;
```

We pass both lookup schemes to the function that turns the preconditioner into a subsidiary block preconditioner and then call its own setup function, as usual.

```
  // Turn into subdiary preconditioner
  block_prec_pt->
   turn_into_subsidiary_block_preconditioner(this,dof_map,
                                             doftype_coarsening);
  // Perform setup. Note that because the subsidiary
  // preconditioner is a block preconditioner itself it is given
  // the pointer to the "full" matrix
  block_prec_pt->setup(this->matrix_pt());
 }
```

Finally, we set up the of diagonal matrix-vector product which acts on the compound (0,1) block (formed from dof types {0,1}x{2,3,4}) in the current preconditioner.

```
 // Set up off diagonal matrix vector product
 {
  // Get the off diagonal block.
  CRDoubleMatrix block_matrix = this->get_block(0,1);

  // Create matrix vector product operator
  Off_diagonal_matrix_vector_product_pt = new MatrixVectorProduct;

  // Setup: Final argument indicates block column in the present
  // block preconditioner (which views the system matrix as comprising
  // 2x2 blocks).
  unsigned block_column_index=1;
  this->setup_matrix_vector_product(
   Off_diagonal_matrix_vector_product_pt,&block_matrix,block_column_index);
  // extracted block can now go out of scope; the matrix vector product
  // retains its own (deep) copy.
 }

}// End of setup
```

1.3.7.4 The preconditioner_solve() function

The `preconditioner_solve()` function is completely identical to the one used in the previous example, so we omit the code listing.

1.3.7.5 The clean_up_my_memory() function

Memory is cleaned up as before, so we omit the code listing.

1.3.8 Using multiple meshes – explained for a genuine fluid-structure interaction problem

NEW FEATURES: How to use multiple meshes

1.3.8.1 Theory

Finally, we demonstrate the use of multiple meshes by discussing a simple implementation of the FSI preconditioner described in the `FSI Preconditioner Tutorial`. We refer to the tutorial discussing the `FSI channel with leaflet problem` for the overall problem setup.
FSI problems involve fluid (velocities and pressures from the Navier-Stokes equations) and solid (the nodal positions in the solid domain) degrees of freedom (dofs). We begin by reordering the linear system to group together the two types of dof

$$\begin{bmatrix} F & C_{fs} \\ C_{sf} & S \end{bmatrix} \begin{bmatrix} \delta\mathbf{f} \\ \delta\mathbf{s} \end{bmatrix} = - \begin{bmatrix} \mathbf{r_f} \\ \mathbf{r_s} \end{bmatrix},$$

where $\mathbf{f}$ and $\mathbf{s}$ denote the fluid and solid dofs, F is the Navier-Stokes Jacobian (representing the derivatives of the discretised fluid equations with respect to the fluid dofs), S is the solid Jacobian, and the blocks C_{fs} and C_{sf} arise from the interaction between fluid and solid equations.
The Navier Stokes Jacobian F has its own block structure. Decomposing the fluid dofs into velocity and pressure dofs so that

$$\mathbf{f} = \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix},$$

we obtain the well known saddle-point structure of F

$$F = \begin{bmatrix} A & B^T \\ B & \end{bmatrix},$$

where A is the momentum block, B^T the discrete gradient operator, and B the discrete divergence operator (see `Navier Stokes Preconditioner Tutorial`).
This FSI preconditioner takes the form of a block triangular preconditioner. Here we only consider the lower block triangular version

$$P_{FSI} = \begin{bmatrix} F & \\ C_{sf} & S \end{bmatrix}$$

obtained by omitting the C_{fs} block from the Jacobian.
The application of the preconditioner requires the solution of the linear system

$$\begin{bmatrix} F & \\ C_{sf} & S \end{bmatrix} \begin{bmatrix} \mathbf{z_f} \\ \mathbf{z_s} \end{bmatrix} = \begin{bmatrix} \mathbf{y_f} \\ \mathbf{y_s} \end{bmatrix}.$$

However, for preconditioning purposes this system does not have to be solved exactly. We therefore replace the solution of the linear systems involving the diagonal blocks (representing the single-physics fluid and solid Jacobians F and S) by existing preconditioners (interpreted as inexact solvers). Formally, we write this as

$$\begin{bmatrix} \tilde{F} & \\ C_{sf} & \tilde{S} \end{bmatrix} \begin{bmatrix} \mathbf{z_f} \\ \mathbf{z_s} \end{bmatrix} = \begin{bmatrix} \mathbf{y_f} \\ \mathbf{y_s} \end{bmatrix}. \qquad (20)$$

where $\tilde{F}$ is the fluid preconditioner and $\tilde{S}$ the solid preconditioner, both used as subsidiary preconditioners.
The application of the preconditioner can be accomplished in four distinct steps:

1. Apply the fluid preconditioner $\tilde{F}$ to the fluid dofs of the RHS vector $\mathbf{y_f}$ and store the result in the fluid solution $\mathbf{z_f} = \tilde{F}^{-1}\mathbf{y_f}$.
2. Multiply the fluid-solid coupling matrix C_{sf} with the fluid solution $\mathbf{z_f}$ and store the result in the temporary vector $\mathbf{w} = C_{sf}\mathbf{z_f}$.
3. Subtract $\mathbf{w}$ from the solid dofs of the RHS vector $\mathbf{y_s}$ and store the result in the temporary $\mathbf{w}$ to complete the action of the C_{sf} matrix vector product, $\mathbf{w} = \mathbf{y_s} - \mathbf{w}$.
4. Apply the solid preconditioner $\tilde{S}$ to the temporary $\mathbf{w}$ to compute the solid solution $\mathbf{z_s} = \tilde{S}^{-1}\mathbf{w}$.

This is, of course, extremely similar to the methodology explained in the section Using subsidiary block preconditioners, the main difference being that the fluid and solid dofs are classified by two different elements. In the two-dimensional `FSI channel with leaflet problem` these are:

- The fluid elements are of type `RefineableQTaylorHoodElement<2>`. These elements have three types of dof; x-velocity dofs are labelled `0`, y-velocity dofs are labelled `1` and the pressure dofs are labelled `2`.

- The solid elements are of type `FSIHermiteBeamElement`.
  They have one type of dof (the nodal position) labelled `0`.

When classifying the dofs we specify the elements via two separate meshes, the first one containing the pointers to the fluid elements, the second one the pointers to the solid elements. This means that in the global enumeration of the dof types the fluid dofs appear before the solid dofs.

1.3.8.2 The Implementation of the FSI Preconditioner

We implement the FSI preconditioner in the class `SimpleFSIPreconditioner`. This class inherits from the base class `BlockPreconditioner` which provides the generic functionality required for common block preconditioning operations.
The overall structure of the class is similar to that of the preconditioners considered before, the main difference being that we now store pointers to two meshes.

```
//=start_of_simple_fsi_preconditioner============================================
/// \short Simple FSI preconditioner. A block uppper triangular preconditioner
/// for the 2x2 FSI block system -- DOFs are decomposed into fluid DOFs and
/// solid DOFs. The fluid subisidiary system is solved with the
/// Navier Stokes Preconditioner and the solid subsidiary system with the
//================================================================================
 template<typename MATRIX>
class SimpleFSIPreconditioner
 : public virtual BlockPreconditioner<MATRIX>
{

public :

 /// \short Constructor for SimpleFSIPreconditioner
 SimpleFSIPreconditioner(Problem* problem_pt)
  : BlockPreconditioner<MATRIX>(), Navier_stokes_preconditioner_pt(0),
    Solid_preconditioner_pt(0), Fluid_solid_coupling_matvec_pt(0),
    Navier_stokes_mesh_pt(0), Solid_mesh_pt(0)
  {
   // Create the Navier Stokes Schur Complement preconditioner
   Navier_stokes_preconditioner_pt =
    new NavierStokesSchurComplementPreconditioner(problem_pt);
   // Create the Solid preconditioner
   Solid_preconditioner_pt = new SuperLUPreconditioner;
   // Create the matrix-vector product operator
   Fluid_solid_coupling_matvec_pt = new MatrixVectorProduct;
  }// end_of_constructor


 /// Destructor: Clean up.
 ~SimpleFSIPreconditioner()
  {
   //Delete the Navier-Stokes preconditioner
   delete Navier_stokes_preconditioner_pt; Navier_stokes_preconditioner_pt = 0;

   //Delete the solid preconditioner
   delete Solid_preconditioner_pt; Solid_preconditioner_pt = 0;

   // Delete the matrix vector product operator
```

```
   delete Fluid_solid_coupling_matvec_pt; Fluid_solid_coupling_matvec_pt = 0;
  }

 /// Broken copy constructor
 SimpleFSIPreconditioner(const SimpleFSIPreconditioner&)
  {
   BrokenCopy::broken_copy("SimpleFSIPreconditioner");
  }


 /// \short Access function to mesh containing the block-preconditionable
 /// Navier-Stokes elements.
 void set_navier_stokes_mesh(Mesh* mesh_pt)
  {
   Navier_stokes_mesh_pt = mesh_pt;
  }

 /// \short Access function to mesh containing the block-preconditionable
 /// FSI solid elements.
 void set_solid_mesh(Mesh* mesh_pt)
  {
   Solid_mesh_pt = mesh_pt;
  }

 /// \short Setup the preconditioner
 void setup();

 /// \short Apply preconditioner to r
 void preconditioner_solve(const DoubleVector &r,
                           DoubleVector &z);
private:

 /// Pointer the Navier Stokes preconditioner.
 NavierStokesSchurComplementPreconditioner* Navier_stokes_preconditioner_pt;

 /// Pointer to the solid preconditioner.
 Preconditioner* Solid_preconditioner_pt;

 /// Pointer to the fluid onto solid matrix vector product.
 MatrixVectorProduct* Fluid_solid_coupling_matvec_pt;

 /// Pointer to the navier stokes mesh.
 Mesh* Navier_stokes_mesh_pt;

 /// Pointer to the solid mesh.
 Mesh* Solid_mesh_pt;
};
```

1.3.8.3 Preconditioner Setup

We start by setting up the meshes, choosing the fluid mesh to be mesh `0` and the solid mesh to be mesh `1`. The preconditioner therefore has four dof types enumerated in mesh order:

- `0` fluid x velocity (dof type 0 in mesh 0)
- `1` fluid y velocity (dof type 1 in mesh 0)
- `2` fluid pressure (dof type 2 in mesh 0)
- `3` solid (dof type 0 in mesh 1)

```
//=start_of_setup=====================================================================
/// Setup the preconditioner.
//====================================================================================
 template<typename MATRIX>
 void SimpleFSIPreconditioner<MATRIX>::setup()
{
 // setup the meshes for BlockPreconditioner and get the number of types of
 // DOF assoicated with each Mesh.
 // Mesh 0 is the fluid mesh, and hence DOFs 0 to n_fluid_dof_type-1
 // are the fluid DOFs. Mesh 1 is the solid mesh and therefore DOFs
 // n_fluid_dof_type to n_total_dof_type-1 are solid DOFs
 // set the mesh pointers
 this->set_nmesh(2);
 this->set_mesh(0,Navier_stokes_mesh_pt);
 this->set_mesh(1,Solid_mesh_pt);

 unsigned n_fluid_dof_type = this->ndof_types_in_mesh(0);
 unsigned n_total_dof_type = n_fluid_dof_type + this->ndof_types_in_mesh(1);
```

Next we define the mapping from dof number to block number. The preconditioner has two block types – fluid and solid – therefore we group the fluid dofs into block type `0` and the solid dofs into block type `1`. We define a

map from dof type to block type in a vector (the vector indices denote the dof type and the vector elements denote the block type) and pass it to `block_setup(...)` to complete the setup of the `BlockPreconditioner` infrastructure.

```
// This fsi preconditioner has two types of block -- fluid and solid.
// Create a map from DOF number to block type. The fluid block is labelled
// 0 and the solid block 1.
Vector<unsigned> dof_to_block_map(n_total_dof_type,0);
for (unsigned i = n_fluid_dof_type; i < n_total_dof_type; i++)
 {
  dof_to_block_map[i] = 1;
 }

// Call the BlockPreconditioner method block_setup(...) to assemble the data
// structures required for block preconditioning.
this->block_setup(dof_to_block_map);
```

Next we set up the subsidiary operators required by the preconditioner. We start with the solid subsidiary preconditioner ($\tilde{S}$). We extract the solid matrix block S from the Jacobian using the `BlockPreconditioner` method `get_block(...)` and then set up the solid subsidiary preconditioner:

```
// First the solid preconditioner
//===============================

// get the solid block matrix (1,1)
CRDoubleMatrix* solid_matrix_pt = new CRDoubleMatrix;
this->get_block(1,1,*solid_matrix_pt);

// setup the solid preconditioner
// (perform the LU decomposition)
Solid_preconditioner_pt->setup(solid_matrix_pt);
delete solid_matrix_pt; solid_matrix_pt = 0;
```

Note that, compared to the previous examples, we have used an alternative, pointer-based version of the `get_block(...)` function. However, as before, the block matrix can be deleted once the subsidiary preconditioner has been set up since the latter retains whatever data it requires.

The fluid subsidiary preconditioner ($\tilde{F}$) a block preconditioner itself. Its setup is therefore performed in two steps:

1. First we turn the `NavierStokesSchurComplementPreconditioner` into a subsidiary block preconditioner. We assemble a list a fluid dof types in the current (master) preconditioner, and pass this list to the Navier-Stokes preconditioner to indicate that dof type `i` in the master FSI preconditioner is dof type `i` in the subsidiary fluid preconditioner (for `i` = `0`, `1`, `2`) (Note that the fact that this mapping is the identity mapping is a result of choosing the fluid mesh to be mesh `0`; in general the index of `ns_dof_list` corresponds to the dof type number in the Navier Stokes subsidiary preconditioner and the value corresponds to the index in this master preconditioner).

   ```
   // Next the fluid preconditioner
   //==============================

   // Specify the relationship between the enumeration of DOF types in the
   // master preconditioner and the Schur complement subsidiary preconditioner
   // so that ns_dof_type[i_nst] contains i_master
   Vector<unsigned> ns_dof_list(n_fluid_dof_type);
   for (unsigned i = 0; i < n_fluid_dof_type; i++)
    {
     ns_dof_list[i] = i;
    }

   // Turn the NavierStokesSchurComplement preconditioner into a subsidiary
   // preconditioner of this (FSI) preconditioner
   Navier_stokes_preconditioner_pt->
    turn_into_subsidiary_block_preconditioner(this,ns_dof_list);
   ```

2. Next we set up the `NavierStokesSchurComplementPreconditioner`. We pass the Navier-Stokes mesh to the the subsidiary preconditioner and set up the preconditioner. Note that the pointer to the full FSI Jacobian is passed to the subsidiary block preconditioner. This allows the subsidiary preconditioner to extract the relevant sub-blocks, using the lookup schemes established by the call to `turn_into_subsidiary_block_preconditioner(...)`.

   ```
   // Set up the NavierStokesSchurComplement preconditioner.
   // (Pass it a pointer to the Navier Stokes mesh)
   Navier_stokes_preconditioner_pt->
    set_navier_stokes_mesh(Navier_stokes_mesh_pt);
   // Navier Stokes preconditioner is a subsidiary block preconditioner.
   // It therefore needs a pointer to the full matrix.
   Navier_stokes_preconditioner_pt->setup(this->matrix_pt());
   ```

Finally, we set up is the matrix-vector product. This mirrors the set up of the solid subsidiary preconditioner. First the subsidiary matrix is extracted from the Jacobian and then the operator is set up:

```
// Finally the fluid onto solid matrix vector product operator
//============================================================
```

```
 // Similar to the solid preconditioner get the matrix
 CRDoubleMatrix* fluid_onto_solid_matrix_pt = new CRDoubleMatrix;
 this->get_block(1,0,*fluid_onto_solid_matrix_pt);

 // And setup the matrix vector product operator
 this->setup_matrix_vector_product(Fluid_solid_coupling_matvec_pt,
                                   fluid_onto_solid_matrix_pt,
                                   0);
 // Clean up
 delete fluid_onto_solid_matrix_pt; fluid_onto_solid_matrix_pt = 0;
```

Again, the extracted block can be deleted since the matrix vector product operator retains the relevant data. The FSI preconditioner is now ready to be used.

1.3.9 Preconditioner Solve

The `preconditioner_solve(...)` method applies
the preconditioner to the input vector $\mathbf{y}$ and returns the result in $\mathbf{z}$.
We start by applying the Navier-Stokes preconditioner $\tilde{F}$ to the fluid elements `y_f` of `y`. Since $\tilde{F}$ is a subsidiary block preconditioner we apply it to the full-length `y` and `z` vectors which contain both the fluid and solid unknowns. The block preconditioning infrastructure utilised within the `NavierStokesSchurComplement↩ Preconditioner` will ensure that the preconditioner only operates on fluid dofs.

```
//=start_of_preconditioner_solve=====================================================
/// Apply preconditioner.
//====================================================================================
 template<typename MATRIX>
 void SimpleFSIPreconditioner<MATRIX>::preconditioner_solve(
  const DoubleVector &y, DoubleVector &z)
{
 // Fluid Subsidiary Preconditioner
 //================================
 // Start by applying the Fluid subsidiary preconditioner
 // The fluid subsidiary preconditioner is a block preconditioner and
 // hence we pass it the global residual and solution vectors (y and z)
 Navier_stokes_preconditioner_pt->preconditioner_solve(y,z);
```

The fluid elements `z_f` of the vector `z` will now have been updated to contain the action of the SchurComplement preconditioner on the fluid elements `y_f` of the vector `y`.
To apply the fluid-solid coupling matrix vector product C_{sf}, we copy the fluid elements from `z` into another vector `z_f`. We then apply the matrix-vector product operator to `z_f` and store the result in a vector `w`. Finally, we subtract `w` from the solid residuals `y_s` and store the result in `w` to complete the application of the matrix-vector product.

```
 // Fluid Onto Solid Matrix Vector Product Operator
 //================================================
 // The vector z_f contains the result of the action of the
 // NavierStokesPreconditioner on a subset of  the elements of z.
 // Remember the fluid block index is 0 and the solid block index is 1.
 DoubleVector z_f;
 this->get_block_vector(0,z,z_f);
 // Apply the matrix vector product to z_f and store the results in w
 DoubleVector w;
 Fluid_solid_coupling_matvec_pt->multiply(z_f,w);
 // The vector y_s contains the solid residuals
 DoubleVector y_s;
 this->get_block_vector(1,y,y_s);
 // Subtract the action of the fluid onto solid matrix vector product from y_s
 y_s -= w;
 w = y_s;
```

Finally, we apply the solid subsidiary preconditioner $\tilde{S}$ to `w` and return the result to `z`. We note that because the solid subsidiary preconditioner is not a block preconditioner, the preconditioner solve method must be called with the solid block vectors. The result is then copied to the full-length vector `z` which contains the fluid and solid dofs.

```
 // Solid Subsidiary Preconditioner
 //================================
 // Apply the solid preconditioner to s and return the result to the
 // global solution vector z
 DoubleVector z_s;
 Solid_preconditioner_pt->preconditioner_solve(w,z_s);
 this->return_block_vector(1,z_s,z);
}
```

1.4 Parallelisation

We note that the above discussion did not address the parallelisation of the preconditioners. This is because all the required parallel features are "hidden" within the block preconditioning framework which relies heavily on the library's `distributed linear algebra infrastructure.` Any of the preconditioners discussed in this tutorial can therefore be used without change when `oomph-lib` is compiled with MPI support and if the the

executable is run on multiple processes.

1.5 Source files for this tutorial

- The source file for the simple block diagonal preconditioner for the linear elasticity problem is

  demo_drivers/linear_solvers/simple_block_preconditioners.h

- The driver code demonstrating the use of the simple block diagonal preconditioner for the linear elasticity problem is

  demo_drivers/linear_solvers/two_d_linear_elasticity_with_simple_block↩
  _diagonal_preconditioner.cc

- The source files for the "multi-poisson" preconditioners and the serial driver codes are located in the directory:

  demo_drivers/linear_solvers/

- The serial "multi-poisson" driver code (which demonstrates the use of the various "multi-poisson" preconditioners discussed above) is:

  demo_drivers/poisson/two_d_multi_poisson.cc

- The parallel counterpart is here (note that, as claimed, this code uses exactly the same preconditioners as the serial version):

  demo_drivers/mpi/solvers/two_d_multi_poisson.cc

- The (parallel) driver code which demonstrates the implementation and use of the simple FSI preconditioner (for the "channel with leaflet" problem) is here:

  demo_drivers/mpi/solvers/fsi_channel_with_leaflet.cc

1.6 PDF file

A pdf version of this document is available.