Chapter 1

A Discussion on OOMPH-LIB's Block Preconditioning Framework

In this document we discuss `oomph-lib`'s block preconditioning framework. We describe the functionality of the framework by starting with discussing the implementation of a simple block diagonal preconditioner, the LSC preconditioner, and finally a preconditioner for the solution of the constrained Navier-Stokes equations with Lagrange multipliers.

The aim of the block preconditioning framework is to provide a simple environment to facilitate the implementation of distributed block preconditioners which in particular allows existing (block) preconditioners to be reused to create in hierarchical fashion new block preconditioners for multi-physics problems. For example, in the Lagrangian preconditioner (considered in this document) we reuse the existing LSC preconditioner for the subsidiary problems.

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

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

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

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

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

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

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

Block preconditioning requires special enumeration schemes for the unknowns (equivalent to reordering the linear systems) where all the unknowns corresponding to each type of DOF are grouped together and enumerated consecutively. This leads to a natural block structure of the linear systems.

For instance, linear elasticity problems[CITE] involve the solid (the nodal positions in the solid domain) degrees of freedom (DOFs). Consider the two-dimensional case, we begin by reordering the linear system to group together the two types of DOF

$$\begin{bmatrix} S_{xx} & S_{xy} \\ S_{yx} & S_{yy} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}_{\mathbf{x}} \\ \delta\mathbf{x}_{\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{r}_{\mathbf{x}} \\ \mathbf{r}_{\mathbf{y}} \end{bmatrix},$$

then the block diagonal preconditiner of the form

$$P_{diag} = \begin{bmatrix} S_{xx} & \\ & S_{yy} \end{bmatrix}$$

is obtained by omitting the off-diagonal blocks from the Jacobian.

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

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

the two sub-blocks will be solved directly.

1.2 Framework Overview

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

- The classification of the DOFs.
- The application of subsidiary preconditioning operators.

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

1.2.1 Block Preconditionable Elements

The classification of DOFs is implemented at an elemental level. The class `GeneralisedElement` contains two broken virtual functions that must be re-implemented to label the DOFs with their type. The functions 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. For instance the two-dimensional FSI channel with leaflet problem has two types of element:

- `RefineableQTaylorHoodElement<2>` are the fluid elements. They have three types of DOF; x-velocity DOFs are labelled `0`, y-velocity DOFs are labelled `1` and the pressure DOFs are labelled `2`.
- `FSIHermiteBeamElement` are the wall elements and have one type of DOF (the nodal position) labelled `0`.

The linear elasticity elements `MyLinearElasticityElement<DIM>` are made block-preconditionable with a wrapper around the element implemented in the driver code:

```
64  /// \short The number of "blocks" that degrees of freedom in this element
    /// are sub-divided into: The displacement components
66  unsigned ndof_types()
     {
68    return DIM;
     }
70
    /// Create a list of pairs for all unknowns in this element,
72  /// so the first entry in each pair contains the global equation
    /// number of the unknown, while the second one contains the number
74  /// of the "block" that this unknown is associated with.
    /// (Function can obviously only be called if the equation numbering
76  /// scheme has been set up.)
    ///
78  /// The dof type enumeration (in 3D) is as follows:
    /// S_x = 0
80  /// S_y = 1
    /// S_z = 2
82  ///
     void get_dof_numbers_for_unknowns(
84    std::list<std::pair<unsigned long,unsigned> >& block_lookup_list)
     {
86    // number of nodes
      unsigned n_node = this->nnode();
```

```
88
   // temporary pair (used to store block lookup prior to being added to list)
90 std::pair<unsigned,unsigned> block_lookup;

92 // loop over the nodes
   for (unsigned j=0;j<n_node;j++)
94  {
     //loop over displacement components
96   for (unsigned i=0;i<DIM;i++)
      {
98     // determine local eqn number
       int local_eqn_number = this->nodal_local_eqn(j,i);
100
       // ignore pinned values - far away degrees of freedom resulting
102    // from hanging nodes can be ignored since these are be dealt
       // with by the element containing their master nodes
104    if (local_eqn_number >= 0)
        {
106       // store block lookup in temporary pair: Global equation number
          // is the first entry in pair
108       block_lookup.first = this->eqn_number(local_eqn_number);

110       // set block numbers: Block number is the second entry in pair
          block_lookup.second = i;
112
          // add to list
114       block_lookup_list.push_front(block_lookup);
         }
116    }
    }
118 }
```

Thus, in the `MyLinearElasticityElement<2>` we have two types of DOF; corresponding to the x and y nodal displacements. They are classified as DOF types `0` and `1` respectively.

1.2.2 DOF types and block types

In the block diagonal preconditioner for the two-dimensional linear elasticity problem, there are two DOF types, there are also two block types. However, in more complicated preconditioners, such as the LSC preconditioner, there are more DOF types than there are block types. Below we describe relationship between elemental DOF type classification, meshes, the block preconditioner DOF types, and block types:

- **Elemental DOF type classification**: Each element classifies it's own DOF type in the function `get_dof_numbers_for_unknowns(...)`. In the case of `MyLinearElasticityElement<2>` elements, the DOF types are classified as `0` and `1`. For `QTaylorHoodElement<2>` elements, the DOF types are classified as `0` and `1` for the x and y-velocities, and `2` for the pressure p.

- **Role of meshes**: Within the block preconditioning framework, each mesh acts as a container for a set of DOF type classifications. All elements in the same mesh MUST return the same `ndof_types()` value. If two different element types are in the same mesh, and their `ndof_types()` does indeed return the same number, then their elementtal DOF type classifcations will be treated as the same type. For example, the `QTaylorHoodElement<2>` classifies the DOF types as follows:

  - `0` x-velocity
  - `1` y-velocity
  - `2` p-pressure

  If we wish to impose parallel outflow along a boundary, we attach

  `ImposeParallelOutflowElement<ELEMENT>`, see demo problem: Steady finite-Reynolds-number flow through an iliac bifurcation, these `FaceElement`s classifies the bulk DOF types as follows:

  - `0` constrained x-velocity
  - `1` constrained y-velocity
  - `2` L-Lagrange multiplier

  Although the `ndof_types()` for these two different elements are the same, there are clearly six distinct DOF types. To ensure that the block preconditioning framework treats these as different DOF types, we must have two meshes for the two different type of elements. If we put the two elements types in the same mesh, then the block preconditioning framework will not distinguish between the two `0` elemental DOF types, i.e. x-velocity is the same as constrained x-velocity. The same applies to elemental DOF types `1` and `2`. Within the block preconditioning framework, there is a built-in check to throw an error if a mesh passed to the framework contains multiple types of elements. This check can be avoided by setting the optional argument `allow_multiple_element_type_in_mesh` when calling the function `set_mesh(...)` to true, in this case, the framework will check if the `ndof_types()` of all the elements of the same mesh are the same.

- **Block preconditioner DOF types**: The different meshes instructs the block preconditioning framework how to order the elemental DOF types (the `ndof_types()` function provides an offset). For example, consider the above `vmtk` problem, the first mesh (the bulk mesh) says the first

three elemental DOF types are `0` , `1` and `2` (we know that this corresponds to the x and y-velocities and pressure). At this stage the offset is `0` and the block preconditioner DOF types `0+0`, `1+0` and `2+0`. The second mesh (the surface mesh) also has DOF types `0` , `1` and `2`, but because it is a different mesh, the offset is now updated to the sum of the `ndof_types()` of the first element in all previous meshes (hence it is vital that the `ndof_types()` of all the elements in a single mesh is the same, although the elements types may be different), in this case it is `3`. The DOF types in the block preconditioner for the surface mesh are `0+3` , `1+3` and `2+3` (which we know corresponds to the constrained x and y-velocities and the Lagrange multiplier DOF type). It is important to note that the order of the meshes determines the order of the block preconditioner DOF type enumeration. To this end, where the ordering of the block preconditioner DOF types matters (such as the LSC preconditioner), the block preconditioner should handle the ordering of the meshes, the user should use functions such as `set_navier_stokes_mesh(...)` in the case of the `NavierStokesSchurComplementPreconditioner`. We call the ordering of the preconditioner DOF types, determined by the elemental DOF type ordering and the ordering of the meshes, the *natural ordering* of the preconditioner DOF types. For the above example, the natural ordering of the preconditioner DOF types would be:

  - `0` x-velocity
  - `1` y-velocity
  - `2` p-pressure
  - `3` constrained x-velocity
  - `4` constrained y-velocity
  - `5` L-Lagrange multiplier

  Note: Each DOF can be classified more than once. If this is the case, then the classification will be the last mesh visited by the block preconditioning framework. This should not be an issue if you do not have discontinuous boundary conditions.

- **Block types**: The block types are the blocks of sub-matrices the block preconditioner works with. Block types may contain more than one preconditioner DOF types or be as fine grain as the number of preconditioner DOF types. Note: There can not be more block types than there are preconditioner DOF types. For example, in case of the the LSC preconditioner

(in 2D) we have three DOF types (x , y-velocities, and pressure), but the preconditioner only distinguishes between velocity and pressure DOF types, thus we have two block types (velocity block and pressure block). The setup of the block types are handled by the function... `block_setup(...)`. The setup of the blocks and DOF types will be discussed in more detail later on.

1.2.3 Master and Subsidiary Preconditioners

Consider the again the LSC Navier-Stokes preconditioner. If we decide to approximate the **F** block (the momentum block) by it's diagonal blocks, we can pass the block diagonal preconditioner (discussed in (Distributed) General-Purpose Block Preconditioners) to the LSC preconditioner to use as a subsidiary preconditioner via the function `set_f_preconditioner(...)`. We can do the same with the pressure system with the function `set_p_preconditioner(...)`. We refer to these preconditioners as subsidiary preconditioners. `Oomph-lib`'s block preconditioning framework facilitates the the reuse of existing preconditioners as subsidiary preconditioners.

It is important to note that we do not need to consider the `block structure` of subsidiary block preconditioners when developing master preconditioners. However, the master preconditioner must be aware of the preconditioner DOF type ordering of the subsidiary preconditioner. For example, if the LSC preconditioner is a subsidiary preconditioner (as is the case of the FSI preconditioner), the FSI preconditioner must ensure that the last DOF type given to the LSC preconditioner is the pressure DOF type, and the ones before that are the velocity DOF types.

There is only one 'true' master preconditioner, the rest are all subsidiary preconditioners, each subsidiary preconditioner holds a pointer to the preconditioner one level up in the hierarchy. Say, there are three preconditioners, $P1$, $P2$ and $P3$. If we use $P2$ to solve a subsidiary system in $P1$, and $P3$ to solve a subsidiary system in $P2$, then it could be said that $P1$ is the master preconditioner for $P2$, and $P2$ is a master preconditioner for $P3$. However, because $P2$'s `Master_block_preconditioner_pointer_pt` is not null, it is automatically classed as a subsidiary preconditioner. Only the true master preconditioner (so that the `Master_block_preconditioner_pt` is null) holds all the information about the DOF types and look-up schemes. If a subsidiary preconditioner requires information held only by the master preconditioner, it will go to it's 'master' preconditioner. If this 'master' preconditioner is a subsidiary preconditioner, it will again go up the hierarchy to it's master preconditioner.

1.3 Using the Simple Block Preconditioner

We begin our discussion of the implementation details by demonstrating how to use the preconditioner (implemented in the class `SimpleBlockDiagonalPreconditioner`) in an actual driver code

(`two_d_linear_elasticity_with_simple_block_diagonal_preconditioner.cc`).

In the problem constructor, we construct the solver and preconditioner combination. We specify the GMRES iterative solver, and, if available, use the distributed version implemented in `TrilinosAztecOOSolver`.

```
348  // Create the solver.
     #ifdef OOMPH_HAS_TRILINOS
350  TrilinosAztecOOSolver* trilinos_solver_pt = new TrilinosAztecOOSolver;
     trilinos_solver_pt->solver_type() = TrilinosAztecOOSolver::GMRES;
352  Solver_pt = trilinos_solver_pt;
     #else
354  Solver_pt = new GMRES<CRDoubleMatrix>;
     // We use RHS preconditioning. Note that by default,
356  // left hand preconditioning is used.
     static_cast<GMRES<CRDoubleMatrix>*>(Solver_pt)->set_preconditioner_RHS();
358  #endif

360  // Set linear solver
     linear_solver_pt() = Solver_pt;
```

Then we construct an instance of the preconditioner. This linear elasticity problem contains one types of element used for preconditioning (see section 1.2.1). There also exists `LinearElasticityTractionElement`, but in this case the block preconditioner does not require the surface mesh since the bulk mesh contains all the DOFs that needs to be classified and `MyLinearElasticityElement` does classify all the DOF in the bulk mesh. We store the different element types in seperate meshes. Only the bulk mesh is passed to the preconditioner. Finally, the preconditioner is passed to the solver.

```
363  // Create the preconditioner
364  Prec_pt=new SimpleBlockDiagonalPreconditioner<CRDoubleMatrix>;

366  // Block preconditioner can work with just the bulk mesh
     // since its elements contain all the degrees of freedom that
368  // need to be classified.
     Prec_pt->set_nmesh(1);
370  Prec_pt->set_mesh(0,Bulk_mesh_pt);

372  // Set the preconditioner
     Solver_pt->preconditioner_pt()=Prec_pt;
```

In the main function,

```
450  //===start_of_main=========================================================
     /// Driver code for PeriodicLoad linearly elastic problem
452  //=========================================================================
     int main(int argc, char* argv[])
454  {
```

we create an instance of the problem which problem can now be solved in the normal `oomph-lib` fashion:

```
466  // Set up doc info
     DocInfo doc_info;
468
     // Set output directory
470  doc_info.set_directory("RESLT");

472  //Build the problem
     PeriodicLoadProblem<MyLinearElasticityElement<2> >
474   problem(nx,ny,Global_Parameters::Lx, Global_Parameters::Ly);

476  // Solve
     problem.newton_solve();
478
     // Output the solution
480  problem.doc_solution(doc_info);
```

1.4 The implementation of a block diagonal preconditioner

We discuss the implementation of a block preconditioner within `oomph-lib`'s block preconditioning framework. In particular, we will address three fundamental tasks:

- How to identify and classify the DOFs in the underlying `Problem`.
- How to extract subsidiary matrix blocks from the full Jacobian.
- How to recycle existing preconditioning operators within new preconditioners.

We implement the block diagonal preconditioner in the class `SimpleBlockDiagonalPrecond` This class inherits from the base class `BlockPreconditioner` which provides the generic functionality required for common block preconditioning operations.

```
53  template<typename MATRIX>
54   class SimpleBlockDiagonalPreconditioner : public BlockPreconditioner<MATRIX>
    {
```

This preconditioner requires a `Vector` of pointers to `Preconditioner`s for each diagonal block matrix.

```
97   /// \short Vector of SuperLU preconditioner pointers for storing the
98   /// preconditioners for each diagonal block
     Vector<Preconditioner*> Diagonal_block_preconditioner_pt;
```

1.4.1 Constructor for block diagonal preconditioner

The constructor is usually used to initialised variables. In this case, there is nothing to initialise.

```
59   /// Constructor for SimpleBlockDiagonalPreconditioner
60    SimpleBlockDiagonalPreconditioner() : BlockPreconditioner<MATRIX>()
      {
62    } // end_of_constructor
```

1.4.2 setup(...) for block diagonal preconditioner

Like all preconditioners, `BlockPreconditioners` have two key functions, `setup(...)` and `preconditioner_solve(...)` both of which are discussed in more detail in the `oomph-lib` Linear Solvers Tutorial [CITE]. We begin by considering the function `setup(...)`.

```
107  template<typename MATRIX>
108  void SimpleBlockDiagonalPreconditioner<MATRIX>::setup()
     {
```

At the simplest level, `Meshes` are just containers for elements. To reiterate section 1.2.2, storing different element type in separate meshes enables the `BlockPreconditioner` to differentiate between the elemental DOF types of different element types. The order of the meshes determine the order of the preconditioner DOF types. Therefore, in more sophisticated preconditioners, the preconditioner usually handle the calls to `set_nmesh(...)` and `set_mesh(...)` functions. For this simplistic case, the functions `set_nmesh(...)` and `set_mesh(...)` were called in the driver code (see section 1.3).

Passing the meshes to `BlockPreconditioner` gives the framework access to the meshes and allows the preconditioner write access to the number of DOF types associated with the elements in each mesh.

block_setup(...)

The next step is to define a mapping from DOF number to block number. This preconditioner's block type is as fine grain as the preconditioner DOF types. To see this, recall, if the Jacobian (partitioned into DOF types) takes the form

$$J = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{yx} & S_{yy} \end{bmatrix},$$

then the preconditioner is

$$P_{diag} = \begin{bmatrix} S_{xx} & \\ & S_{yy} \end{bmatrix}.$$

It is clear that the blocks the preconditioner works with has to be as fine grain as the DOF types. This is achieved by calling the function `block_setup(...)` with no arguments. By default, this has the same effect as calling `block_setup(...)` with the identity `dof_to_block_map` Vector `= [0 1]`.

```
111  // Set up the block look up scheme
112  this->block_setup();
```

For this simple block diagonal preconditioner, there is method of changing the `dof_to_block_map` Vector. There exists a more sophisticated version of the block diaginal preconditioner in the class `GeneralPurposeBlockPreconditioner`, where there exists a function `set_dof_to_block_map(Vector<unsigned>& dof_to_block_map)` to change the `dof_to_block_map` from the default. Subtilties of the `dof_to_block_map` Vector is discussed below.

block_setup(...): Combining DOF types

If we want a block type to consist of more than one preconditioner DOF type, then we can provide a `dof_to_block_map` to the function `block_setup(...)` describing the mapping we want. For example, if we extend the example above so there are three DOF types,

$$J = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{bmatrix},$$

and instead of simply using the diagonal blocks, we want to use a block preconditioner of the following form

$$\tilde{P}_{diag} = \begin{bmatrix} S_{xx} & S_{xy} & \\ S_{yx} & S_{yy} & \\ & & S_{zz} \end{bmatrix}.$$

That is, we want to group the first two preconditioner DOF types as one block type and the last preconditioner DOF type as a separate block type. This is achieved by passing the `dof_to_block_map = [0 0 1]` to the function `block_setup(...)`, we see a similar `dof_to_block_map` in the implementation of the LSC precondi-

tioner. It is important to note that although the (in C++ index notation) block (0,0) consists of preconditioner DOF types $\begin{bmatrix} S_{xx} & S_{xy} \\ S_{yx} & S_{yy} \end{bmatrix}$, it is not gauranteed that the DOF types are grouped in such a manner in the (0,0) block.

block_setup(...): Re-ordering DOF types

Continuing from the example above, the natural ordering of the preconditioner DOF types (determined by the ordering of the elemental DOF types and the order of the meshes added to the block preconditioner as decribed in section 1.2.2) is:

DOF type name:	S_x	S_y	S_z
Natural ordering:	0	1	2

We have named the DOF types so facilitate the discussion. Suppose we want to re-order the DOF types such that we have the following block order:

New block order:	S_z	S_x	S_y

Then the `dof_to_block_map` `Vector` will be

`dof_to_block_map`:	1	2	0

The index of the `dof_to_block_map` Vector is the DOF type you want to move, then ask yourself 'where do I want to move this DOF type to?', put this value into the current position in the `dof_to_block_map` Vector.

Combining this with the previous concept, it is possible to create a preconditioner of the form

$$\tilde{P}_{diag} = \begin{bmatrix} S_{zz} & S_{zx} & \\ S_{xz} & S_{xx} & \\ & & S_{yy} \end{bmatrix}$$

with the `dof_to_block_map` Vector `[0 1 0]`.

Set up subsidiary preconditioners

The next step is to set up the subsidiary preconditioners. We use the `SuperLU` preconditioner for all the subsidiary systems, we require as many subsidiary preconditioners as there are the number of blocks that the block preconditioner is working with. First we create a new instances of the of `SuperLUPreconditioner` for every block.

```
114   // Number of blocks
      unsigned nblock_types = this->nblock_types();
116
```

```
   // Resize the storage for the diagonal blocks
118 Diagonal_block_preconditioner_pt.resize(nblock_types);

120 // Create the subsidiary preconditioners
   for (unsigned i=0;i<nblock_types;i++)
122  {
      Diagonal_block_preconditioner_pt[i] = new SuperLUPreconditioner;
124  }
```

Now we extract the diagonal blocks and call the `setup(...)` function of the subsidiary preconditioners.

```
126 // Set up the subsidiary preconditioners with the diagonal blocks
   for (unsigned i=0;i<nblock_types;i++)
128  {
      // Get block
130   CRDoubleMatrix block;
      this->get_block(i,i,block);
132
      // Set up preconditioner (i.e. lu-decompose the block)
134   Diagonal_block_preconditioner_pt[i]->setup(&block,this->comm_pt());

136   // Done with this block now, so can go out of scope; LU decomposition
      // is retained in superlu
138  }
```

1.4.3 Preconditioner solve for block diagonal preconditioner

Next we consider the `preconditioner_solve(...)` function which applies the action of the preconditioner to the input vector **y** and returns the result in **z**.

```
142 //=====================================================================
   /// Preconditioner solve for the block diagonal preconditioner
144 //=====================================================================
   template<typename MATRIX>
146  void SimpleBlockDiagonalPreconditioner<MATRIX>::
   preconditioner_solve(const DoubleVector& r, DoubleVector& z)
```

In this section, we implement the application of the preconditioner as described in the section 1.1. First we split the rhs vector into sub-vectors, re-arranged to match the block order of the preconditioner blocks.

```
149   // Split up rhs vector into sub-vectors, re-arranged to match
150   // the matrix blocks
      Vector<DoubleVector> block_r;
152   this->get_block_vectors(r,block_r);
```

Next we loop through the LU decompositions stored in the `Diagonal_block_preconditioner_p` and apply the subsidiary preconditioners.

```
157   // Solution of block solves
158   Vector<DoubleVector> block_z(n_block);
      for (unsigned i = 0; i < n_block; i++)
```

```
160     {
         Diagonal_block_preconditioner_pt[i]->preconditioner_solve(block_r[i],
162                                                                block_z[i]);
     }
```

Finally we return the solution back into **z** in the correct DOF order.

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

1.5 LSC block preconditioner

Theoretical dicussion of the LSC preconditioner can be found in "Finite Elements and Fast Iterative Solvers with Applications in Incompressible Fluid Dynamics" by Howard C. Elman, David J. Silvester, and Andrew J. Wathen, published by Oxford University Press, 2006.

In this section, we partially follow the `oomph-lib` LSC tutorial [CITE oomph-lib lsc tutorial], where we take a closer look at the implementation details to address the following concepts:

- Non trivial grouping of preconditioner DOF types into block types.
- Setting up `MatrixVectorProduct`s within the block preconditioning framework.
- Subsidiary block preconditioners:
  - The `dof_map` for the `turn_into_subsidiary_block_preconditioner(...)` function.
  - Modification to the `preconditioner_solve(...)` function.

1.5.1 Theory

`oomph-lib` currently provides two types of (LBB-stable) Navier-Stokes elements: Taylor-Hood (Q2Q1) and Crouzeix-Raviart (Q2Q-1) elements. These contain two distinct types of degrees of freedom, namely the velocities and pressures.

The least-squares commutator (LSC; formerly BFBT) Navier-Stokes preconditioner employs `oomph-lib`'s block-preconditioning framework to (formally) reorder the linear system to be solved during the Newton iteration into 2x2 blocks, corresponding to the velocity and pressure unknowns. We note that all velocity components are treated as a single block of unknowns. The linear system

therefore has the following block structure

$$\begin{bmatrix} \mathbf{F} & \mathbf{G} \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{z}_u \\ \mathbf{z}_p \end{bmatrix} = \begin{bmatrix} \mathbf{r}_u \\ \mathbf{r}_p \end{bmatrix}.$$

Here $\mathbf{F}$ is the momentum block, $\mathbf{G}$ the discrete gradient operator, and $\mathbf{D}$ the discrete divergence operator. (For unstabilised elements, we have $\mathbf{D} = \mathbf{G}^T$ and in much of the literature the divergence matrix is denoted by $\mathbf{B}$.)

An "exact" preconditioner would solve this system exactly and thus ensure the convergence of any iterative linear solver in a single iteration. However, the application of such a preconditioner would, of course, be exactly as expensive as a direct solve. The LSC/BFBT preconditioner replaces the exact Jacobian by a block-triangular approximation

$$\begin{bmatrix} \mathbf{F} & \mathbf{G} \\ \mathbf{0} & -\mathbf{M}_s \end{bmatrix} \begin{bmatrix} \mathbf{z}_u \\ \mathbf{z}_p \end{bmatrix} = \begin{bmatrix} \mathbf{r}_u \\ \mathbf{r}_p \end{bmatrix},$$

where $\mathbf{M}_s$ is an approximation to the pressure Schur-complement $\mathbf{S} = \mathbf{D}\mathbf{F}^{-1}\mathbf{G}$. This system can be solved in two steps:

1. Solve the second row for $\mathbf{z}_p$ via

$$\mathbf{z}_p = -\mathbf{M}_s^{-1}\mathbf{r}_p$$

2. Given $\mathbf{z}_p$, solve the first row for $\mathbf{z}_u$ via

$$\mathbf{z}_u = \mathbf{F}^{-1}\left(\mathbf{r}_u - \mathbf{G}\mathbf{z}_p\right)$$

In the LSC/BFBT preconditioner, the action of the inverse pressure Schur complement

$$\mathbf{z}_p = -\mathbf{M}_s^{-1}\mathbf{r}_p$$

is approximated by

$$\mathbf{z}_p = -\left(\mathbf{D}\widehat{\mathbf{Q}}^{-1}\mathbf{G}\right)^{-1}\left(\mathbf{D}\widehat{\mathbf{Q}}^{-1}\mathbf{F}\widehat{\mathbf{Q}}^{-1}\mathbf{G}\right)\left(\mathbf{D}\widehat{\mathbf{Q}}^{-1}\mathbf{G}\right)^{-1}\mathbf{r}_p,$$

where $\widehat{\mathbf{Q}}$ is the diagonal of the velocity mass matrix. The evaluation of this expression involves two linear solves involving the matrix

$$\mathbf{P} = \left(\mathbf{D}\widehat{\mathbf{Q}}^{-1}\mathbf{G}\right)$$

which has the character of a matrix arising from the discretisation of a Poisson

problem on the pressure space. We also have to evaluate matrix-vector products with the matrix

$$\mathbf{E} = \mathbf{D}\widehat{\mathbf{Q}}^{-1}\mathbf{F}\widehat{\mathbf{Q}}^{-1}\mathbf{G}$$

In our implementation of the preconditioner, the linear systems can either be solved "exactly", using `SuperLU` (in its incarnation as an exact preconditioner; this is the default) or by any other `Preconditioner` (interpreted as an "inexact solver") specified via the access functions

`NavierStokesSchurComplementPreconditioner::set_f_preconditioner(...)`

or

`NavierStokesSchurComplementPreconditioner::set_p_preconditioner(...)`

1.5.2 Implementation

In this section we focus on aspect of the implementation relevant to the discussion of the block preconditioning framework. Let `dim` be the spatial dimension of the problem, then `NavierStokesSchurComplementPreconditioner` expects `dim` + 1 preconditioner DOF types, `dim` number of DOF types corresponds to the velocity DOF types and 1 DOF type for the pressure.

setup(...) for the LSC preconditioner

We need to group the velocity DOF types as one block type and the pressure DOF type as one DOF type. This is achieved by setting the first `dim` entries of the `dof_to_block_map` Vector to 0 and the last entry to 1.

```
235   Vector<unsigned> dof_to_block_map(ndof_types);
236   dof_to_block_map[ndof_types-1]=1;

238   this->block_setup(dof_to_block_map);
```

For example, in two dimensions, the natural ordering of the DOF types and the `dof_to_block_map` Vector is given below.

DOF type name:	u	v	p
Natural ordering:	0	1	2
`dof_to_block_map`:	0	0	1

The implementation follows the theory in section 1.5.1. We highlight important points to note when implementing a new preconditioner.

When setting up a `MatrixVectorProduct`, it is important to use the setup function `setup_matrix_vector_product(...)`. The first argument is a pointer to the `MatrixVectorProduct`, the second is a pointer to the matrix we wish to

set up with, the third is an unsinged value indicating the block vector which the matrix needs to be multiplied by. This ensures that the column distribution of the `MatrixVectorProduct` is set up properly.

```
493   F_mat_vec_pt = new MatrixVectorProduct;
494   this->setup_matrix_vector_product(F_mat_vec_pt,f_pt,0);
```

If the $\mathbf{F}$ preconditioner is a block preconditioner (used to solve the system $\mathbf{z}_u = \mathbf{F}^{-1}(\mathbf{r}_u - \mathbf{G}\mathbf{z}_p)$), then we must call the function `turn_into_subsidiary_block_preconditioner(...)` of the $\mathbf{F}$ preconditioner. This function takes the pointer of the 'master' preconditioner (in this sense, master is just one level higher in the block preconditioning framework hierarchy), and a Vector describing a mapping of preconditioner DOF types of the master preconditioner and the subsidiary preconditioner. Then the `setup(...)` function of the subsidiary preconditioner is called with a pointer to the whole jacobian (via the access function `matrix_pt()`). If the $\mathbf{F}$ preconditioner is not a block preconditioner, then we do the same as we did in the simple block diagonal preconditioner case.

```
577   if (F_preconditioner_is_block_preconditioner)
578    {
       unsigned ndof_types = this->ndof_types(true);
580    ndof_types--;
       Vector<unsigned> dof_map(ndof_types);
582    for (unsigned i = 0; i < ndof_types; i++)
        {
584      dof_map[i] = i;
        }
586    F_block_preconditioner_pt->
        turn_into_subsidiary_block_preconditioner(this,dof_map);
588
       F_block_preconditioner_pt->setup(matrix_pt(),comm_pt());
590   }
      // otherwise F is not a block preconditioner
592   else
       {
594    F_preconditioner_pt->setup(f_pt,comm_pt());
       delete f_pt; f_pt = 0;
596   }
```

The `dof_map` Vector is different from the `dof_to_block_map` Vector. The `dof_to_block_map` Vector describes the mapping of the preconditioner DOF types to block types within the same block preconditioner, as such, this Vector must have the same size as the number of DOF types the preconditioner expects to be working with. The `dof_map` Vector describes the mapping between the DOF types of a master preconditioner with it's subsidiary preconditioner. Re-ordering DOF types of the subsidiary preconditioner using the `dof_map` Vector is possible and will be demonstrated in the implementation of the Lagrangian preconditioner discussed later in this document.

preconditioner_solve(...) for the LSC preconditioner

The `preconditioner_solve(...)` function for the LSC preconditioner follows the theory discussed in section 1.5.1. When block preconditioner are used as subsidiary preconditioners, we DO NOT return the block vector, as this is handled by the subsidiary block preconditioner.

```
785   // use some Preconditioner's preconditioner solve
786   // and return
      if (F_preconditioner_is_block_preconditioner)
788    {
        return_block_vector(0,another_temp_vec,z);
790     F_preconditioner_pt->preconditioner_solve(z,z);
       }
792   else
       {
794     F_preconditioner_pt->preconditioner_solve(another_temp_vec, temp_vec);
        return_block_vector(0,temp_vec,z);
796    }
```

1.6 Lagrangian block preconditioner

In this section, we use the implementation of the Lagrangian preconditioner to address the following concepts:

- Non-trivial re-ordering of block types.
- Non-trivial re-ordering of subsidiary DOF types.
- Set precomputed preconditioner blocks for subsidiary block preconditioners.

As with previous examples, we highlight the key concepts and neglect the finer implementation details of the preconditioner for the specific problem. The code is (as always) well documented and can be worked through with the theory at hand.

1.6.1 Theory

Suppose we want to impose parallel outflow along a boundary, we attach the block preconditionable elements `ImposeParallelOutflowElement` along the said boundary as discussed in section 1.2.2. This will result in a jacobian of the

following block form (partitioned by DOF types):

$$\left[\begin{array}{cccc|c|c} F_{xx} & F_{xy} & F_{x\hat{x}} & F_{x\hat{y}} & (B_x)^T & \\ F_{yx} & F_{yy} & F_{y\hat{x}} & F_{y\hat{y}} & (B_y)^T & \\ F_{\hat{x}x} & F_{\hat{x}y} & F_{\hat{x}\hat{x}} & F_{\hat{x}\hat{y}} & (B_{\hat{x}})^T & M_x \\ F_{\hat{y}x} & F_{\hat{y}y} & F_{\hat{y}\hat{x}} & F_{\hat{y}\hat{y}} & (B_{\hat{y}})^T & M_y \\ \hline B_x & B_y & B_{\hat{x}} & B_{\hat{y}} & & \\ \hline & & M_x & M_y & & \end{array}\right] \left[\begin{array}{c} \delta\bar{u}_x \\ \delta\bar{u}_y \\ \delta\bar{u}_{\hat{x}} \\ \delta\bar{u}_{\hat{y}} \\ \delta\bar{p} \\ \delta\bar{l} \end{array}\right] = \left[\begin{array}{c} \delta\bar{r}_x \\ \delta\bar{r}_y \\ \delta\bar{r}_{\hat{x}} \\ \delta\bar{r}_{\hat{y}} \\ \delta\bar{r}_p \\ \delta\bar{r}_l \end{array}\right], \tag{1.1}$$

where the block vector $[\bar{u}_x\, \bar{u}_{\hat{x}}]^T$ contains the x coordinates of the unknown nodal positions. Similarly, the block vector $[\bar{u}_y\, \bar{u}_{\hat{y}}]^T$ contains the x coordinates of the unknown nodal positions. The hat represents the nodes affected by the Lagrange multuplier constraint. The Lagrange multiplier block takes the form

$$L = \begin{bmatrix} O & O & M_x & M_y & O \end{bmatrix}.$$

For simplicity, we can re-write the jacobian as

$$\mathcal{J} = \begin{bmatrix} \mathcal{F} & L \\ L & O \end{bmatrix}$$

For this saddle point problem, we seek an augmented preconditioner of the form

$$\mathcal{P} = \begin{bmatrix} \mathcal{F} + L^T W^{-1} L & O \\ O & W \end{bmatrix}$$

The matrix $W \in \mathbb{R}^{n_l \times n_l}$ is chosen to be

$$W = \begin{bmatrix} \frac{1}{\sigma} L L^T \end{bmatrix}, \tag{1.2}$$

to preserve the sparsity of $\mathcal{F}$ and σ is chosen to be the norm of the momentum block to be an effective preconditioner for the above saddle point problem. Application of this preconditioner requires the solution of the two diagonal blocks. We can further approximate the augmented $\hat{\mathcal{F}} = \mathcal{F} + L^T W^{-1} L$ block by the LSC approximation, this is facilitated by the re-use of block preconditioners in `oomph-lib`'s block preconditioning framework.

Detailed theoretical discussion for the Lagrangian preconditioner can be found in [CITE some chapter in my thesis or the yet to be published paper].

1.6.2 Implementation

The Lagrangian preconditioner takes the 'bulk' mesh as the first mesh (the mesh containing the TaylorHood elements). Subsequent meshes are surfaces meshes containing the `ImposeParallelOutflowElement` elements. We have established in section 1.2.2 that this leads to the natural DOF type ordering

DOF type name:	u	v	p	u_c	v_c	L
Natural ordering:	0	1	2	3	4	5

Where u and v are the 'bulk' velocity DOF types in the x and y direction, u_c and v_c are the constained velocity DOF types in the x and y direction DOF type, p is the pressure DOF type and L is the lagrange multiplier DOF type.

setup(...) for the Lagrangian preconditioner

First we need to create the `dof_to_block_map` to create the block ordering as observed in section 1.6.1. Recall from 1.2.2 that to re-order block types, we ask ourselves 'where do I want to move this DOF type to?'. This leads to the following `dof_to_block_map`:

DOF type name:	u	v	p	u_c	v_c	L
Natural ordering:	0	1	2	3	4	5
Desired block ordering:	u	v	u_c	v_c	p	L
`dof_to_block_map`:	0	1	4	2	3	5

The creation of the `dof_to_block_map` Vector is generalised to multiple surface meshes in the `setup(...)` function of the `LagrangeEnforcedflowPreconditioner` class.

```
1078  Vector<unsigned> dof_to_block_map(n_doftypes,0);

1080  // Encapsulate temporary variables
      {
1082   unsigned temp_index = 0;
       unsigned lagrange_entry = N_velocity_doftypes;
1084   for (unsigned mesh_i = 0; mesh_i < nmesh; mesh_i++)
        {
1086     for (unsigned dim_i = 0; dim_i < spatial_dim; dim_i++)
         {
1088       dof_to_block_map[temp_index] = dim_i + mesh_i*spatial_dim;
          temp_index++;
1090      } // for

1092     unsigned ndof_type_in_mesh = this->ndof_types_in_mesh(mesh_i);

1094     for (unsigned doftype_i = spatial_dim;
              doftype_i < ndof_type_in_mesh; doftype_i++)
1096      {
          dof_to_block_map[temp_index] = lagrange_entry;
```

```
      lagrange_entry++;
      temp_index++;
     } // for
    } // for
```

Suppose want to use the LSC block preconditioner to approximate the $\hat{mathcal F}$. First we need to create the `dof_map` which describes which DOF types from the Lagrangian preconditioner the LSC preconditioner works with. We have to work with the DOF types from the natural order and ask ourselves 'which DOF type from the master preconditioner do I want o move into this position?'. The `dof_map` for this problem is given below, along with the `dof_to_block_map` for comparison.

DOF type name:	u	v	p	u_c	v_c	L
Natural ordering:	0	1	2	3	4	5
Desired block ordering:	u	v	u_c	v_c	p	L
`dof_to_block_map`:	0	1	4	2	3	5
`dof_map`:	0	1	3	4	2	

For the `LagrangeEnforcedflowPreconditioner`, the `dof_map` generation is generalised to work with multiple surface meshes.

```
   // Temp velocity and lagrange dof type vectors.
  Vector<unsigned> temp_v_doftypes;
  Vector<unsigned> temp_l_doftypes;

  unsigned increment = 0;
  for(unsigned mesh_i = 0; mesh_i < nmesh; mesh_i++)
   {
    // Store the velocity dof types.
    for(unsigned dim_i = 0; dim_i < spatial_dim; dim_i++)
     {
      temp_v_doftypes.push_back(increment++);
     } // for spatial_dim

    // Store the pressure/lagrange dof types.
    unsigned n_doftype_in_mesh = this->ndof_types_in_mesh(mesh_i);

    for(unsigned l_i = spatial_dim; l_i < n_doftype_in_mesh; l_i++)
     {
      temp_l_doftypes.push_back(increment++);
     }
   } // for nmesh

  // Concatenate the vectors.
  Subsidiary_list_bcpl.clear();
  Subsidiary_list_bcpl.reserve(temp_v_doftypes.size() +
                               temp_l_doftypes.size());
  Subsidiary_list_bcpl.insert(Subsidiary_list_bcpl.end(),
                              temp_v_doftypes.begin(),
                              temp_v_doftypes.end());
  Subsidiary_list_bcpl.insert(Subsidiary_list_bcpl.end(),
                              temp_l_doftypes.begin(),
                              temp_l_doftypes.end());
```

The above code actually generates the list `[0 1 3 4 2 5]`, where the first `dim*nmesh` entries corresponds to the $\mathcal{F}$ DOF types. Therefore we fill the `dof_map` with only the required DOF types.

```
1704    Vector<unsigned> ns_dof_map(N_fluid_doftypes,0);
        for (unsigned i = 0; i < N_fluid_doftypes; i++)
1706     {
          ns_dof_map[i]= Subsidiary_list_bcpl[i];
1708     }
```

Then we proceed to call the `turn_into_subsidiary_block_preconditioner(...)` function

```
1733    // Set the dof_map
1734    navier_stokes_block_preconditioner_pt
         ->turn_into_subsidiary_block_preconditioner(this, ns_dof_map);
```

There exists two problems if we want to re-use the LSC block preconditioner:

1. The LSC preconditioner expects `dim` + 1 DOF types. We our Navier-Stokes block consists of `dim*nmesh` + 1 DOF types.

2. The function `get_block(...)` extracts the block matrix from the original jacobian. We want the LSC preconditioner to operate on the modified $\hat{\mathcal{F}}$ block.

There exists one function to solve both problems. The function `set_precomputed_blocks(...)` takes a `DenseMatrix` consisting of pointers to the (possibly modified) precomputed preconditioner blocks and (yet another!) mapping (`dof_to_dof_map`) between the DOF types of the master preconditioner and the subsidiary preconditioner. This mapping describes which DOF type of the master preconditioner should be treated as a single DOF type in the subsidiary preconditioner. For the above example, the `dof_to_dof_map` Vector would be the two dimension Vector:

```
0 [0 2]
1 [1 3]
2 [4]
```

Recall that we now have the DOF type ordering

	0	1	2	3	4	5
Desired block ordering:	u	v	u_c	v_c	p	L

so the `dof_to_dof_map` says to the LSC preconditioner 'treat DOF types 0 and 2 from the master preconditioner as DOF type 0, treat the DOF types 1 and 3 from the master preconditioner as DOF type 1, and treat DOF type 4 from the master preconditioner as DOF type 2'. Again, the implementation of the

`LagrangeEnforcedflowPreconditioner` is fully generalised, but will produce the above results.

In the below code, we fill the `f_subblock_pt` with the (modified) velocity blocks and the rest of the pressure block B to form the blocks required for $\hat{\mathcal{F}}$.

```
1674    // Get the augmented velocity blocks.
        DenseMatrix<CRDoubleMatrix* > f_subblock_pt(N_fluid_doftypes,
1676                                                N_fluid_doftypes,0);
        // put in v_aug_pt:
1678    for(unsigned v_i = 0; v_i < N_velocity_doftypes; v_i++)
         {
1680      for(unsigned v_j = 0; v_j < N_velocity_doftypes; v_j++)
          {
1682       f_subblock_pt(v_i,v_j) = v_aug_pt(v_i,v_j);
          }
1684     }

1686    // Fill in the pressure block B plus 1
        for(unsigned col_i = 0; col_i < N_fluid_doftypes; col_i++)
1688     {
          f_subblock_pt(N_velocity_doftypes,col_i) = new CRDoubleMatrix;
1690
          this->get_block(N_velocity_doftypes,col_i,
1692                        *f_subblock_pt(N_velocity_doftypes,col_i));
         }
1694
        // Fill in the pressure block B^T
1696    for(unsigned row_i = 0; row_i < N_velocity_doftypes; row_i++)
         {
1698      f_subblock_pt(row_i,N_velocity_doftypes) = new CRDoubleMatrix;

1700      this->get_block(row_i,N_velocity_doftypes,
                          *f_subblock_pt(row_i,N_velocity_doftypes));
1702     }
```

The code below describes how we create the `dof_to_dof_map`.

```
1747    Vector<Vector<unsigned> > dof_to_dof_map;
1748
        // First do the velocities
1750    for (unsigned direction = 0; direction < spatial_dim; direction++)
         {
1752      Vector<unsigned> dir_doftypes_vec(nmesh,0);
          for (unsigned mesh_i = 0; mesh_i < nmesh; mesh_i++)
1754       {
            dir_doftypes_vec[mesh_i] = spatial_dim*mesh_i+direction;
1756       }

1758       // Push it in!
           dof_to_dof_map.push_back(dir_doftypes_vec);
1760     }

1762    // The pressure DOF type is located at N_velocity_doftypes
        Vector<unsigned> ns_p_vec(1,0);
1764    ns_p_vec[0] = N_velocity_doftypes;

1766    // Push it in!
        dof_to_dof_map.push_back(ns_p_vec);
```

Now, we set the precomputed blocks and call `setup(...)`.

```
1769      navier_stokes_block_preconditioner_pt
1770       ->set_precomputed_blocks(f_subblock_pt,dof_to_dof_map);

1772      navier_stokes_block_preconditioner_pt
           ->setup(matrix_pt(), comm_pt());
```

1.7 Source Files

The following source files were used for our discussion of `oomph-lib`'s block preconditioning framework.

`demo_drivers/linear_solvers/two_d_linear_elasticity_with_simple_block_diagonal_preconditioner.cc`

`demo_drivers/linear_solvers/simple_block_preconditioners.h`

`src/navier_stokes/navier_stokes_preconditioners.h`

`src/navier_stokes/navier_stokes_preconditioners.cc`

`src/navier_stokes/lagrange_enforced_flow_preconditioner.h`

1.8 Under the hood

In this section we take a closer look at innards of the block preconditioning framework. In particular, the mechanism for setting precomputed preconditioner blocks. This feature was developed to allow the re-use of block preconditioners when the master preconditioner has modified the preconditioner blocks. This knowledge is not required for developing new preconditioners, but it may be useful for maintaining the block preconditioning framework.

The following discussion assumes good knowledge of the distributed data structures used within `oomph-lib`, in particular, how data is distributed. If you are not already familiar with these concepts, please refer to the `oomph-lib` tutorials 'Parallel processing' and 'Distributed Linear Algebra infrastructure'.

`BlockPreconditioner`s are `DistributedLinearAlgebraObject`s. Their `LinearAlgebraDistribution` can be accessed via the function `preconditioner_matrix_distribution_pt()`. This `LinearAlgebraDistribution` describes the distribution of the preconditioner, which is the concatenation of inidividual block distributions *without communication*.

The concatenation of `DistributedLinearAlgebraObject`s without communication means to 'combine' several objects (of the same type) into a new object whilst keeping the data on the processor it already resides. For reference, see the following functions:

- `CRDoubleMatrixHelpers::concatenate_without_communication(...)`

- `DoubleVectorHelpers::concatenate_without_communication(...)`
- `LinearAlgebraDistributionHelpers::concatenate(...)`

The distributions of the inidividual preconditioner blocks are uniformly distributed. However, the distribution of the preconditioner (which is a concatenation of the distributions of it's preconditioner blocks) is not uniformly distributed. This means one cannot generate a uniformly distributed matrix and use it as a preconditioner within a particular preconditioner, since the preconditioner expects the preconditioner matrix to have the same distribution as the concatenation of the distributions of the individual preconditioner blocks.

The distribution of the individual preconditioner blocks is stored within each block preconditioner and can be accessed via the function `block_distribution_pt(...)`.

1.8.1 Setting precomputed preconditioner blocks

We revisit the Lagrangian block preconditioner example discussed in section 1.6 and assume we have a two dimensional problem with one bulk mesh containing `QTaylorHoodElement`s and one surface mesh containing `ImposeParallelOutflowElement`s. After re-ordering the block types as discribed in section 1.6.2, we have the following block order:

	0	1	2	3	4	5
Desired block ordering:	u	v	u_c	v_c	p	L

The blocks corresponding to the constrained velocities, namely blocks (2,2), (2,3), (3,2) and (3,3) have been modified by the Lagrangian preconditioner. We want to apply the LSC preconditioner to the (modified) Navier-Stokes blocks 0-4, so we pass pointers to these blocks to the LSC preconditioner.

Within the LSC preconditioner (which splits the block types into velocity and pressure), when we call `get_block(0,0)`, the preconditioning framework would know that preconditioner blocks have been precomputed and will return a concatenation (without communication) of the following blocks

$$\begin{bmatrix} (0,0) & (0,2) & (0,1) & (0,3) \\ (2,0) & (2,2) & (2,1) & (2,3) \\ (1,0) & (1,2) & (1,1) & (1,3) \\ (3,0) & (3,2) & (3,1) & (3,3) \end{bmatrix}.$$

Since the concatenation is without communication, the distribution of the resuling block(0,0) is not uniform. But the inidivual blocks of a preconditioner always

have a uniform distribution! More importantly, this mean that the block vector 0 from the LSC preconditioner also have a uniform distribution, so we cannot use the above preconditioner block since it's distribution is different from that of the vector we with to apply the matrix operation to.

We ensure that the distribution of the preconditioner block matrix is the same as the distribution of the vector we wish to apply the matrix to by the following method: If preconditioner blocks have been precomputed then

- use the identity `dof_to_block_map` for the `block_setup(...)` function.
- when the function `get_block(...)` is called, use the lookup scheme set by `dof_to_dof_map` to concatenate the relevant *precomputed* preconditioner blocks (without communication).
- when the function `get_block_vector(...)` is called, use the lookup scheme set by `dof_to_dof_map` to extract the relevant block vector and concatenate them (without communication).
- when the function `return_block_vector(...)` is called, split the vector into the relevant block vectors and return each block vector back to the vector we wish to return the entries to.

As long as the precomputed preconditioner blocks are ordered in the same manner as the underlaying blocks of the LSC preconditioner (the subsidiary preconditioner), then the distribution is the same for the concatenated precomputed block and the concatenated block vectors. The order of the underlaying blocks of the subsidiary preconditioner is determined by the `dof_map` vector passed to the function `turn_into_subsidiary_block_preconditioner(...)`.

This scheme has a few subtitles:

- If preconditioner blocks have been precomputed for a preconditioner, then the number of block types for the preconditioner is the most fine grain. This means that the number of preconditioner DOF types equal to the number of block types.
- The function `nblock_types(...)` still returns the number of block types the preconditioner expects without precomputed preconditioner blocks.
- The function `ndof_types(...)` still returns the number of DOF types the preconditioner expects without precomputed preconditioner blocks.

1.8.2 Limitations and future development

There is no mechanism to split a matrix which has been concatenated (with or without communication). This means, if the LSC preconditioner was to modify it's preconditioner blocks (having already been set precomputed preconditioner blocks from a master preconditioner), there is no way (yet!) to split up the matrices into the most fine grain block type to pass them down to a another subsidiary preconditioner. The current scheme only allows for one preconditioner in the *hierarchical chain* to modify the preconditioner blocks and pass them down to subsidiary block preconditioners. Two subsidiary block preconditioners can modify the preconditioner blocks and pass them down to it's own subsidiary preconditioner(s) if they are operating on completely different subsidiary systems.

There exists another approach to the whole problem. When the preconditioner blocks are modified by the master preconditioner and to be used by a subsidiary block preconditioner, the master preconditioner copies the enteries back into a copy of the jacobian. Thus we do not have to concatenate the precomputed block matrices nor do we have to always work with the most fine grain block type. Implementation of this feature was considered but due to time constraints, was not implemented; there is currently no book keeping information of where the DOF came from and there would still be the problem of combining DOF types in the subsidiary preconditioner.