Chapter 1

oomph-lib's fluid-structure-interaction (FSI) preconditioner

The purpose of this tutorial is to show how to use `oomph-lib's` FSI preconditioner for the efficient monolithic solution of fluid-structure interaction problems. We illustrate the use of the preconditioner for the collapsible channel problem described in the `tutorial demonstrating the use of oomph-lib's segregated FSI solver.` The test problem used is discussed in detail in

`Heil, M., Hazel, A.L. & Boyle, J. (2008): Solvers for large-displacement fluid-structure interaction problems: Segregated vs. monolithic approaches. Computational Mechanics.`

where we contrast the relative performance of segregated and monolithic solvers.

1.1 Theory

The monolithic discretisation of fluid-structure interaction problems in which the fluid node-update in response to changes in the wall shape is performed by one of `oomph-lib's` algebraic node update techniques (such as the ones implemented in the `AlgebraicMesh` and `MacroElementNodeUpdateMesh` classes; see the relevant tutorials for a discussion of the `algebraic` and `macro-element-based` node update techniques) leads to Jacobian matrices that contain three types of degree of freedom, namely the fluid velocities, the fluid pressures, and the solid mechanics degrees of freedom.
`oomph-lib's` FSI preconditioner employs the library's `block-preconditioning framework` to (formally) re-order the linear system to be solved during the Newton iteration into 3x3 blocks. We note that all fluid velocity components and all solid degrees of freedom are treated as single blocks of unknowns. The linear system therefore has the following block structure

$$\left(\begin{array}{cc|c} \mathbf{F} & \mathbf{G} & \mathbf{C}_{us} \\ \mathbf{D} & \mathbf{0} & \mathbf{C}_{ps} \\ \hline \mathbf{C}_{su} & \mathbf{C}_{sp} & \mathbf{S} \end{array}\right) \left(\begin{array}{c} \delta\mathbf{u} \\ \delta\mathbf{p} \\ \hline \delta\mathbf{s} \end{array}\right) = - \left(\begin{array}{c} \mathbf{r}_u \\ \mathbf{r}_p \\ \hline \mathbf{r}_s \end{array}\right)$$

Here the on-diagonal block matrices

$$\left(\begin{array}{cc} \mathbf{F} & \mathbf{G} \\ \mathbf{D} & \mathbf{0} \end{array}\right) \qquad (1)$$

and

$$(\mathbf{S}) \qquad (2)$$

are the Jacobian matrices from the corresponding single-physics (fluid and solid) problems. The off-diagonal matrix blocks $\mathbf{C}_{**}$ arise from the interaction between fluid and solid equations: $\mathbf{C}_{us}$ and $\mathbf{C}_{ps}$ contain the so-called ``shape derivatives'' — the derivatives of the Navier–Stokes residuals with respect to the solid displacements that affect

the nodal positions in the fluid mesh. Similarly, $\mathbf{C}_{su}$ and $\mathbf{C}_{sp}$ contain the derivatives of the solid residuals with respect to the fluid variables; this interaction arises through the fluid loading on the wall. `oomph-lib's` algebraic node-update strategy ensures that the interaction matrices are very sparse. The maximum fill level for the examples presented in Heil, Hazel & Boyle's (2008) paper is about 3% and such (relatively) large values only arose in computations with very coarse meshes; the much finer meshes used in typical production runs resulted in much sparser matrices.

We showed in an earlier paper [Heil, M. "An efficient solver for the fully-coupled solution of large-displacement fluid-structure interaction problems". Computer Methods in Applied Mechanics and Engineering **193**, 1-23, (2004)] that the use of block-triangular approximations to the global Jacobian matrix, obtained by neglecting the fluid-solid or solid-fluid interaction blocks,

$$\mathcal{P}_1 = \left(\begin{array}{cc|c} \mathbf{F} & \mathbf{G} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{C}_{su} & \mathbf{C}_{sp} & \mathbf{S} \end{array}\right) \quad \text{and} \quad \mathcal{P}_2 = \left(\begin{array}{cc|c} \mathbf{F} & \mathbf{G} & \mathbf{C}_{us} \\ \mathbf{D} & \mathbf{0} & \mathbf{C}_{ps} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{S} \end{array}\right) \quad \text{and} \quad \mathcal{P}_3 = \left(\begin{array}{cc|c} \mathbf{F} & \mathbf{G} & \mathbf{0} \\ \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{S} \end{array}\right)$$

in the Newton method seriously degrades its performance, resulting in the loss of its quadratic convergence and thus one of its the most attractive features. However, the block-triangular approximations were shown to be excellent preconditioners for the solution of the linear system by Krylov subspace methods. Because of their block-triangular structure, each application of the preconditioners involves linear solves with each of the two single-physics systems (1) and (2), and matrix-vector products with the retained interaction matrices.

The current implementation of the FSI preconditioner within `oomph-lib` employs Elman, Silvester & Wathen's ``least squares commutator'' (LSC) preconditioner, discussed in another tutorial to approximately solve the linear system involving the fluid matrix (1).

1.2 An example

To demonstrate how to use the preconditioner, here are the relevant extracts from the driver code `fsi_chan_precond_driver.cc` which solves the collapsible channel problem discussed in another tutorial. As explained in the Linear Solvers Tutorial switching to an iterative linear solver is typically performed in the `Problem` constructor and involves a few straightforward steps:

1. **Create an instance of the IterativeLinearSolver and pass it to the Problem**

   We create an instance of `GMRES`,

```
// Build iterative linear solver
GMRES<CRDoubleMatrix>* iterative_linear_solver_pt =
  new GMRES<CRDoubleMatrix>;
```

   set the maximum number of iterations to 100, and increase the GMRES convergence tolerance to 10^{-6} as experiments showed this to give the fastest overall solve times; see Further comments and exercises.

```
// Set maximum number of iterations
iterative_linear_solver_pt->max_iter() = 100;

// Set tolerance
iterative_linear_solver_pt->tolerance() = 1.0e-6;
```

   Finally, we pass the pointer to the iterative linear solver to the problem.

```
  this->linear_solver_pt()=iterative_linear_solver_pt;
```

2. **Create an instance of the Preconditioner and pass it to the IterativeLinearSolver**

We start by creating an instance of the FSI preconditioner.

```
// Create an instance of the FSI preconditioner -- pass the pointer
// to the problem
FSIPreconditioner* prec_pt=new FSIPreconditioner(this);
```

Next we identify the meshes that contain the fluid and solid elements – this information is required by `oomph-lib's` block-preconditioning framework to identify the types of the various degrees of freedom in the Jacobian matrix. Identifying the fluid elements is straightforward as they are already contained in a distinct (sub-)mesh, accessible via the member function `bulk_mesh_pt()`:

```
// Set Navier Stokes mesh:
prec_pt->set_navier_stokes_mesh(this->bulk_mesh_pt());
```

The `FSIHermiteBeamElement` elements used for the discretisation of the flexible channel wall are also contained in their own (sub-)mesh, accessible via the member function `wall_mesh_pt()`. If displacement control is used, the `DisplacementControlElement` introduces a further unknown into the problem: the adjustable external pressure; see the brief discussion of this in another tutorial. We classify the external pressure as a solid mechanics degree of freedom and therefore add the `Displacement↩ ControlElement` to a combined solid mesh (constructed from a vector of pointers to its constituent sub-meshes):

```
// Build a compound mesh that contains all solid elements:

// Create a vector of pointers to submeshes. Start with the solid
// mesh itself.
Vector<Mesh*> s_mesh_pt(1);
s_mesh_pt[0]=this->wall_mesh_pt();

// Add the displacement control mesh if required
if (this->Displ_control)
 {
  s_mesh_pt.push_back(this->Displ_control_mesh_pt);
 }
// Build compound mesh from vector of solid submeshes
Mesh* combined_solid_mesh_pt = new Mesh(s_mesh_pt);
```

Finally, we pass the pointer to the combined solid mesh to the FSI preconditioner to identify the solid degrees of freedom (the optional boolean flag indicates that we allow the mesh to contain multiple element types):

```
// Set solid mesh and tolerate multiple element types this is mesh.
prec_pt->set_wall_mesh(combined_solid_mesh_pt,true);
```

The preconditioner is now fully functional and we pass
a pointer to it to the preconditioner:

```
// Pass preconditioner to iterative linear solver
iterative_linear_solver_pt->preconditioner_pt()= prec_pt;
```

3. **Customise the Preconditioner (if required)**

The `FSIPreconditioner` allows preconditioning to be performed with either one of the three block-triangular approximations to the Jacobian:

(a) The lower-triangular block preconditioner $\mathcal{P}_1$ which retains the interaction matrices $\mathbf{C}_{s*}$ that represent the effect of the fluid onto the solid:

```
// Choose preconditioner that retains fluid on solid terms
prec_pt->use_block_triangular_version_with_fluid_on_solid();
```

(b) The upper-triangular block preconditioner $\mathcal{P}_2$ which retains the interaction matrices $\mathbf{C}_{*s}$ that represent the effect of the solid onto the fluid:

```
// Choose preconditioner that retains solid on fluid terms
prec_pt->use_block_triangular_version_with_solid_on_fluid();
```

(c) The block-diagonal block preconditioner $\mathcal{P}_3$ which suppresses all the interaction matrices:

```
// Use block-diagonal preconditioner
prec_pt->use_block_diagonal_version();
```

The linear systems involving the fluid block are solved (approximately) by `oomph-lib's` Least-Squares-Commutator Navier-Stokes preconditioner `NavierStokesLSCPreconditioner`, discussed in `another tutorial.` The behaviour of this (sub-)preconditioner may be customised too. For instance, to employ the `Hypre` AMG solver to solve the linear systems involving the pressure Schur complement matrix $\mathbf{P}$ in the `NavierStokesLSCPreconditioner`, we use the procedure discussed `earlier`:

```
#ifdef OOMPH_HAS_HYPRE
//If we are using MPI, then only use HYPRE if it has been initialised
#ifdef OOMPH_HAS_MPI
      if(MPI_Helpers::mpi_has_been_initialised())
#endif
       {

        // By default, the LSC Preconditioner uses SuperLU as
        // an exact preconditioner (i.e. a solver) for the
        // momentum and Schur complement blocks.
        // Can overwrite this by passing pointers to
        // other preconditioners that perform the (approximate)
        // solves of these blocks.

        // Create internal preconditioners used on Schur block
        HyprePreconditioner* P_matrix_preconditioner_pt =
         new HyprePreconditioner;

        // Set defaults parameters for use as preconditioner on Poisson-type
        // problem
        Hypre_default_settings::set_defaults_for_2D_poisson_problem(
         P_matrix_preconditioner_pt);

        // Use Hypre for the Schur complement block
        prec_pt->navier_stokes_preconditioner_pt()->
         set_p_preconditioner(P_matrix_preconditioner_pt);

        // Shut up
        P_matrix_preconditioner_pt->disable_doc_time();
       }
#endif // endif for we have hypre...
```

1.3 Further comments and exercises

1. **The convergence tolerance for the iterative linear solver**

   Since the iterative linear solver operates within an "outer" (Newton) iteration it has occasionally been suggested to adjust the convergence tolerance for the iterative linear solver, depending on the progress of the "outer" (Newton) iteration: The idea is that there is little point in "over-solving" the linear system (i.e. solving

it to very high precision) if the Newton method is still "far" from converged. (Only) during the final stages of the Newton iteration is an accurate solution of linear systems essential, otherwise the Newton iteration stagnates. This idea can be made rigorous (see, e.g., R.S. Dembo, S.C. Eisenstat and T. Steilhaug "Inexact Newton methods" SIAM J. Numer. Anal. **19** (1982), 400-408).

In practice we found that the method does not offer particularly great savings in CPU time; see e.g. `Heil (2004).` This is because the Newton method tends to converge in very few steps. Hence the need to perform the occasional additional Newton step (because the convergence tolerance of the "inner" iterative linear solver
was "just" not tight enough) is hardly ever compensated for by
the reduced number of iterations in the iterative linear solver.

The GMRES convergence tolerance of 10^{-6} chosen here was found (by trial and error) to give optimal overall solve times, but this choice is problem-dependent and unless you are willing to perform systematic preliminary investigations, we recommend using the default convergence tolerance of 10^{-8}, as defined in the `IterativeLinearSolver` base class.

2. **Explore the behaviour of preconditioner(s)**

   The shell script `steady_precond.bash` may be used to explore the performance of the various FSI preconditioners. It solves the monolithically-discretised, steady fluid-structure interaction problem described in `Heil, Hazel & Boyle's (2008) paper` with a variety of solver/preconditioner combinations:

   (a) The direct solver, `SuperLUSolver`.

   (b) `GMRES`, with the three FSI preconditioners $\mathcal{P}_1, \mathcal{P}_2$ or $\mathcal{P}_3$, discussed above, using various (sub-)solver combinations (`SuperLUSolver` or (if available) `Hypre`) for the solution of linear (sub-)systems.

   (c) GMRES with various "sanity-check" preconditioners:

   - The `ExactFSIPreconditioner:` A preconditioner formed from the full Jacobian matrix by re-arranging the entries into the appropriate block structure. This is an exact preconditioner (and its application is therefore just as costly as a direct solve) and leads to GMRES convergence in a single iteration.

   - The `SimpleFSIPreconditioner:` A preconditioner that uses the block-triangular matrices $\mathcal{P}_1, \mathcal{P}_2$ or $\mathcal{P}_3$, stored as full matrices (i.e. without performing any block elimination).

   There is a second shell script `unsteady_precond.bash` that can be used to perform similar parameter studies for the corresponding unsteady problem discussed in `Heil, Hazel & Boyle's (2008) paper.`

3. **Explore the FSI preconditioner in other problems**

   The FSI preconditioner is also used in the driver codes for other demo problems:

- Hron & Turek's FSI benchmark of flow past a flag.
- The problem of flow in a channel that is partially obstructed by an elastic leaflet.
- Your own? If so, let us know how it works.

1.4 Source files for this tutorial

- The source files for this tutorial are located in the directory:

demo_drivers/interaction/fsi_channel_seg_and_precond

- The driver code is:

demo_drivers/interaction/fsi_channel_seg_and_precond/fsi_chan_precond_driver.cc

1.5 PDF file

A pdf version of this document is available.