Chapter 1

Change log

This document provides an overview of the major changes between different "official" releases of oomph-lib.

As of 2016 (i.e. $>$ the 1.0.$*$ release) we include the svn revision number as part of the release identifier since this makes it easier for us to publish incremental updates to the library (and to do so more frequently than we used to!). Whenever such a new release is made available we automatically update the snapshot of the corresponding svn log which contains the commit messages of all changes made since "day one":

svn log [Note that this link will not work from a local installation of oomph-lib.]

[For historical reasons we also provide access to the complete log from our previous (non-public) svn repository. This covers all changes from the initial release of the library (svn revision 132) up to the move to the new public repository.]

You may also want to consult the entries in our (sadly rather under-used) bugzilla-based bug-and-feature-tracking system, accessible online at

http://oomph-lib.maths.man.ac.uk/bugzilla

for known bugs and their resolution.

The remainder of this document covers the following issues:

1.1 Policy on interface changes

We obviously try to avoid interface changes as much as possible as they may force users to adjust their code when upgrading to a new version. We will only change interfaces (names of objects and/or member functions, or the number or order of their arguments) if

1. The previously chosen name turned out to be too ambiguous. For instance, the change from
   `Problem::actions_before_solve()`

   to
   `Problem::actions_before_newton_solve()`

   became necessary because of the addition of further nonlinear (but non-Newton) solvers (e.g. the segregated FSI solver). Since such solvers may call the Newton solver themselves, a more fine-grained control over what we mean by "solving" was required.

2. The previously chosen name violated our own coding conventions. For instance,
   `FiniteElement::n_nodal_position_type()`

   had to be changed to

   `FiniteElement::nnodal_position_type()`

   because we do not want underscores after the leading "n" in an access function that returns the number of certain objects.

3. The order of the function arguments violated our own coding conventions. For instance,
   `FiniteElement::get_x(Vector<double>& s, const unsigned& t,...)`

   had to be changed to
   `FiniteElement::get_x(const unsigned& t, Vector<double>& s,...)`

   because the discrete "time" argument always comes first.

Changes to function or object names are easy to detect because the compiler will not find the old version when linking against a new version of the library, encouraging the user to consult this list to (hopefully) find the appropriate replacement.
Changes to the order of function arguments can be very dangerous: if the order of two arguments of the same type is exchanged the compiler cannot detect the change to the interface but the code is likely to compute different results. To facilitate the detection of such changes, the new version of such functions contains an explicit warnings that can be enabled by compiling the library with the macro `WARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTERFACES`. (For a gcc compiler this is done with the compiler flag `-DWARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTE↩RFACES`).
Here is an example of a function in which the order of the two unsigned arguments was changed. If compiled with `WARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTERFACES`, the code will issue an appropriate warning every time this function is called.

```
// Function of two unsigned arguments whose order has changed
// since the previous release
void SomeClass::some_function(const unsigned& t, const unsigned& i)
{
  [...]
 #ifdef WARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTERFACES
   // Throw an oomph-lib warning to alert the user to the change
   // in the order of the arguments
   OomphLibWarning("Warning: Order of interfaces has changed",
                   "SomeClass::some_function(...)",
                   OOMPH_EXCEPTION_LOCATION);
 #endif
  [...]
}
```

Obviously, you should only compile the library with `WARN_ABOUT_SUBTLY_CHANGED_OOMPH_INTERFACES` if you encounter problems after upgrading to a new version.

1.1.1 Changes between version 0.9 and 1.0.∗

1.1.1.1 Major new functionality

- Provided the capability to solve the Helmholtz equation (with various ways of implementing the Sommerfeld radiation condition), the time-harmonic equations of linear elasticity, and the interaction between these two systems of equations in time-harmonic acoustic fluid structure interaction problems. All equations are implemented in cartesian and cylindrical polar coordinates (in the latter case, using an azimuthal Fourier decomposition for the solution).

- Provided block preconditioners for the solution of large-displacement fluid-structure interaction problems, based the coupling of full nonlinear 2D/3D elasticity and the Navier-Stokes equations.

- Implemented the linearised Navier-Stokes equations (with the option to study their solutions as perturbations relative to a base state that is itself governed by the full Navier-Stokes equations).

- Provided elements for the solution of the axisymmetric equations of linear elasticity.

- Finished what is hopefully the final rewrite of the (parallel) block preconditioning framework and its extensive tutorial.

- Provided elements for the solution of the Foeppl-von-Karman equations in various formulations (using either an Airy-stress function or the in-plane displacements) and in various coordinate systems.

- Provided elements for Darcy porous media flow and Biot-type poro-elasticity, including the ability to use the latter elements in fluid-structure interaction problems which are governed by the equations of poro-elasticity coupled to the Navier-Stokes equations.

- Implemented the ability adapt unstructured meshes in 2D (in serial and in parallel) and (in a parallel setting) to distribute such meshes over multiple processors, and to perform load balancing in spatially adaptive simulations.

- Provided elements for the solution of elements for the solution of the Navier-Stokes equations with generalised Newtonian constitutive equations (i.e. non-Newtonian behaviour in which the viscosity depends on the invariants of the rate-of-strain tensor).

- Provided machinery to output paraview (vtu and vtp) files directly (implemented for selected elements; broken virtual functions in base classes provide instructions what to do for others).

- Provided machinery to establish what the equations/unknowns in a fully-built problem represent. Calling `Problem::describe_dofs()` trawls recursively through the oomph-lib object hierarchy to describe what each degree of freedom represents (from the Problem's, Meshes' and Nodes' point of view). Very useful for debugging (e.g. "Why are all the entries in the n-th row/column in the Jacobian zero?" or "The n-th

residual is enormous -- which equation does it correspond to?", etc.)

- Provided machinery for hp adaptivity and p-type spectral elements with mortar constraints in 2D and 3D.

- Added interfaces to the Trilinos ANASAZI eigensolver to allow parallel solution of eigenproblems.

- Basic implementation of discontinuous Galerkin methods for general flux transport problems.

- Many new detailed tutorials explaining:

  - How to use \t oomph-lib in parallel and how to (optionally) distribute problems over multiple processors and to perform load balancing when performing spatially-adaptive, distributed simulations.

  - How to use oomph-lib's block preconditioning framework.

  - How to solve large-displacement fluid-structure interaction problems, based the coupling of full non-linear 2D/3D elasticity and the Navier-Stokes equations, incl. the use of preconditioners and spatial adaptivity.

  - How to generate unstructured 2D meshes whose curvlinear boundaries are described my `GeomObjects`; these boundaries are respected under mesh refinement, i.e. the mesh converges to the exact domain geometry, rather than using the geometry defined by the initial, coarse discretisation.

  - How to simulate steady and unsteady free-surface flows in various geometries.

  - How to simulate the transport of immersed solid particles in viscous flow.

  - How to implement and solve surface transport equations, such as insoluble and soluble surfactant problems.

  - How to solve the Helmholtz equations in a variety of coordinate systems and how to apply the Sommerfeld radiation condition by ABCs, DTNs, or PMLs.

  - How to solve time-harmonic acoustic fluid-structure interaction problems in a variety of coordinate systems.

  - ...

Have a look through the extensive list of example codes to see what else is available.

1.1.2 Changes between version 0.85 and 0.9

1.1.2.1 Major new functionality

- Significantly extended oomph-lib's parallel processing capabilities. The library's solver and (block) preconditioning framework is now fully parallelised; the parallel assembly of the Jacobian and residual vector has been optimised, and problems can be now distributed by domain decomposition techniques. A new tutorial and various new demo driver codes demo driver codes provide an overview of these capabilities.

- Completed the documentation of oomph-lib's solid mechanics (large-displacement elasticity) capabilities. We provide an overview of the theory and implementation, as well as numerous tutorials discussing the solution of steady and unsteady, 2D and 3D problems with compressible and incompressible materials on structured and unstructured meshes.

- Developed a tutorial that shows how to use the Vascular Modeling Toolkit (VMTK) to generate oomph-lib meshes from medical images. The methodology is used in the following driver codes:

  - The inflation of a blood vessel.

  - Finite Reynolds number flow through a (rigid) iliac bifurcation.

  - Finite Reynolds number flow through an elastic iliac bifurcation.

- Extended FSI capabilities so that such problems can be solved on unstructured meshes. We provide new tutorials that demonstrate this capability for 2D and 3D problems.

- Introduced ElementWithExternalElement as base class for multi-physics elements in which elements in different meshes/ domains interact via source/load functions (e.g. in fluid-structure interaction or thermal convection problems). See the new tutorials discussing the serial and parallel solution of multi-physics problems by multi-domain approaches.

- Parallelised and optimised the MeshAsGeomObject class to use bin structures to accelerate the search in locate_zeta(...).

- FiniteElements are now GeomObjects within which their local coordinates act as their intrinsic coordinates.

- Extended the Z2ErrorEstimator so that the elemental error estimate can be based on the maximum of various distinct fluxes – useful for multi-field problems such as Boussinesq convection. See the tutorial for details.

- Added a structured, domain-based mesh for the discretisation of tube-like domains. See the list of structured meshes for details.

- Provided a new Lagrange-multiplier-based element, that allows the imposition of parallel in- or outflow from boundaries that are not aligned with coordinate planes. This is discussed in a separate tutorial.

- Provided a new mechanism that allows multiple FaceElements to be attached to same node, while providing the ability to distinguish between different Lagrange multipliers stored at the same node. This is discussed in a separate tutorial.

- Significantly extended oomph-lib's bifurcation tracking routines (no tutorials yet, though).

- Provided general framework and driver codes for discontinuous Galerkin methods and explicit timesteppers (no tutorials yet, though).

1.1.2.2 Major interface changes

- SolidFiniteElement::add_jacobian_for_solid_ic()

  becomes
  SolidFiniteElement::fill_in_jacobian_for_solid_ic()

- SolidFiniteElement::add_residuals_for_ic()

  becomes
  SolidFiniteElement::fill_in_residuals_for_solid_ic()

- SolidFiniteElement::get_residuals_for_ic()

  becomes
  SolidFiniteElement::fill_in_residuals_for_solid_ic()

- Changed order of arguments in PVDEquationsBase::BodyForceFctPt so that time goes first (as it should!).

- Not exactly an interface change but important: AlgebraicMeshes and MacroElementNode↩ UpdateMeshes **should** now specify the GeomObjects involved in their node update functions. (This **must** be done if such meshes are to be used in parallel computations involving domain decomposition). The

declaration of the GeomObjects should use the functions AlgebraicMesh::add_geom_object_list_pt(...) and MacroElementNodeUpdateMesh::set_geom_object_vector_pt(...), respectively.

- Added integration point arguments to all get_source-type functions in elements to allow for overloading in multi-domain calculations. For example,

```
NavierStokesEquations<DIM>::get_body_force_nst(const double& time,
                                               const Vector<double> &s,
                                               const Vector<double> &x,
                                               Vector<double> &result)
```

has been changed to

```
NavierStokesEquations<DIM>::get_body_force_nst(const double& time,
                                               const unsigned& ipt,
                                               const Vector<double> &s,
                                               const Vector<double> &x,
                                               Vector<double> &result)
```

- GeomObject::drdt(...) is now GeomObject::dposition_dt(...) to make it consistent with all other functions of that type.

- FSI_functions::setup_fluid_load_info_for_solid_elements(...) now requires a pointer to the Problem to be specified as the first argument.

- Specification of wall and fluid meshes in FSIPreconditioner changed from read/write access functions FSIPreconditioner::navier_stokes_mesh_pt() and FSIPreconditioner::wall_mesh_pt() to FSIPreconditioner::set_navier_stokes_mesh(...) and FSIPreconditioner::set_wall_mesh(...), respectively.

- Specification of fluid mesh in NavierStokesLSCPreconditioner changed from read/write access function FSIPreconditioner::navier_stokes_mesh_pt() to FSIPreconditioner::set_navier_stokes_mesh(...).

- Removed all explicit references to MPI_COMM_WORLD and the parameters MPI_Helpers::Nproc MPI_Helpers::My_rank. The total number of processors and the rank of each processor should be obtained from the appropriate OomphCommunicator.

- Replaced MPI_Helpers::setup(...) with MPI_Helpers::init(...) and added MPI_Helpers::finalize(). These should be called instead of MPI's own init() and finalize() functions.

- Introduced `DoubleVector` – a distributable vector storing double precision numbers.

- Merged `CRDoubleMatrix` and `DistributableCRDoubleMatrix` into `CRDoubleMatrix`. (Note: `CRDoubleMatrix` is the only distributable matrix within `oomph-lib`.)

- Merged `SuperLU` and `SuperLU_dist` into `SuperLUSolver`. If `oomph-lib` is compiled with MPI support, the parallel version of the (exact) preconditioner is used automatically. This behaviour can be over-ruled with the member function
  `SuperLUSolver::set_solver_type(...)`
  whose argument must be one of the three options listed in the enumeration `SuperLUSolver::Type`. This allows the serial version of the solver to be used even if `oomph-lib` is compiled with MPI support.

- Merged `SuperLUPreconditioner` and `SuperLUDistPreconditioner` into `SuperLUPreconditioner`. If `oomph-lib` is compiled with MPI support, the parallel version of the solver is used automatically.

- Updated all solvers/preconditioners to handle new `CRDoubleMatrix` and `DoubleVector`.

- In `NavierStokesLSCPreconditioner`

  ```
  void set_f_preconditioner(Preconditioner& new_f_preconditioner)
  ```

  and

  ```
  void set_p_preconditioner(Preconditioner& new_f_preconditioner)
  ```

  were changed to

  ```
  void set_f_preconditioner(Preconditioner* new_f_preconditioner_pt)
  ```

  and

  ```
  void set_p_preconditioner(Preconditioner* new_f_preconditioner_pt)
  ```

  respectively, to make them consistent with the rest of the code.

- Changed
  `FiniteElement::get_block_numbers_for_unknowns(...)`
  to
  `FiniteElement::get_dof_numbers_for_unknowns(...)`
  and
  `FiniteElement::nblock_types()`
  to

`FiniteElement::ndof_types()`

- Nearly an interface change: We now provide the option to exclude/wipe the validata from the distribution (during `make dist`). To make sure that the self-test procedure doesn't break if there's no validata, configure now assesses the availability of validata by checking the existence of
`demo_drivers/poisson/one_d_poisson/validata/one_d_poisson_results.dat.gz`

  If this is not found, we'll assume that there's no validata anywhere. In that case, "make check" will still compile and run the demo drivers but not use fpdiff.py to compare them against the validata. The behaviour is thus similar to what's done if we don't have python. We therefore modified the "no_python" argument that used to be passed to all validate.sh scripts to suppress the execution of fpdiff.py and replaced it by "no_fpdiff". As a result all validate.sh scripts had to be changed. MAKE SURE YOU UPDATE ANY NEW ONES BEFORE ADDING THEM TO THE DISTRIBUTION!

  To reflect the fact that the fpdiff-ing can now be suppressed for multiple reasons, we renamed
`bin/validate_no_python.sh`

  to
`bin/dummy_validate.sh`

- All the bools in calls to the various `node_update(...)` functions were changed to `const bool&` (rather than `bool`).

- The constructors for all constitutive equations now take pointers to constitutive parameters rather than the parameters themselves. (With the previous approach, changing Young's modulus, say, required the construction of a new constitutive equation object that then had to be passed to all elements again etc.)

- Got rid of all other instances where time argument was passed by copy rather than by constant reference – as long as grep-ing for the pattern '(double t' detected it. In practice this involved various Navier-Stokes traction, body force and source functions.

1.1.3 Changes between version 0.8 and 0.85

1.1.3.1 Major new functionality

- Added a variety of iterative linear solvers and general-purpose preconditioners.

- Added wrappers to the powerful serial and parallel iterative solvers/preconditioners from the Hyre and Trilinos libraries.

- Developed a general block preconditioning framework and used it implement problem-specific preconditioners for Navier-Stokes and fluid-structure-interaction problems.

  - Using oomph-lib's Least-Squares-Commutator Navier-Stokes preconditioner.

  - Using oomph-lib's FSI preconditioner.

- Provided segregated solver capabilities for fluid-structure interaction.

- Added a number of additional tutorials/demo codes illustrating oomph-lib's fluid-structure interaction capabilities:

  - Turek & Hron's FSI benchmark problem of flow past a "flag".

  - Flow in a channel with an elastic leaflet.

  - Using oomph-lib's segregated solvers for fluid-structure-interaction problems.

  - Using oomph-lib's FSI preconditioner.

- MPI capability is now fully integrated into the library (rather than being kept in a separate mpi directory/sub-library) but not yet complete (or documented). Segments of code that involve MPI calls are surrounded by #ifdef OOMPH_HAS_MPI [...] #endif. They are only compiled if the -enable-MPI flag is specified at the configure stage.

- Assuming you have pdflatex installed on your machine, the documentation is now not only built in html format (pretty but hard to print) but we also create associated pdf files. These are accessible via the link at the bottom of relevant html page.

- oomph-lib's build script, autogen.sh now allows for parallel compilation. This can lead to considerable speedups on multicore processors that are now widely available. To build oomph-lib in parallel, using up four threads run autogen.sh as follows
```
./autogen.sh --jobs=4
```
  or (if you are re-building)
```
./autogen.sh --jobs=4 --rebuild
```

- Angelo Simone has written a python script that converts oomph-lib's output to the vtu format that can be read by paraview, an open-source 3D plotting package. This is discussed in separate tutorial.

- ...and much more (eigenproblems, bifurcation tracking, advection-diffusion-reaction equations, displacement-based linear elasticity, impedance-type outflow boundary conditions for Navier-Stokes problems, Lagrange-multiplier-based `FaceElements` to apply non-trivial boundaries to beams and shells, ...). Feel free to have a look around the distribution. You're welcome to use it all, but please remember that you use any functionality for which no documentation is available at your own risk; while we reserve the right to change interfaces for all objects in the library, such changes are almost certain to happen for objects that are not yet documented (in the form of tutorials).

1.1.3.2 Major interface changes

- `Problem::actions_before_solve()` becomes `Problem::actions_before_newton_solve()`
- `Problem::actions_after_solve()` becomes `Problem::actions_after_newton_solve()`
- `TCrouzeixRaviartElement<DIM>` becomes `TCrouzeixRaviartElement<DIM>`
- `QCrouzeixRaviartElement<DIM>` becomes `QCrouzeixRaviartElement<DIM>`
- `RefineableQCrouzeixRaviartElement<DIM>` becomes `RefineableQCrouzeixRaviartElement<DIM>`
- Fixed various violations of our naming conventions; see Policy on interface changes.
- Changed the interface of to the `FaceElements` so that the faces are represented by an integer `face_index`, rather than the combination `s_fixed_index` and `s_limit` which was not sufficiently general.

1.2 PDF file

A `pdf version` of this document is available.