Preprint of:

`oomph-lib` – An *O*bject-*O*riented *M*ulti-*Ph*ysics Finite-Element *Lib*rary

Matthias Heil & Andrew L. Hazel
School of Mathematics, University of Manchester, Manchester
M13 9PL, UK

Chapter in the book: "Fluid-Structure Interaction"
Editors: M. Schafer und H.-J. Bungartz
Springer (Lecture Notes on Computational Science and Engineering)

oomph-lib – An *O*bject-*O*riented *M*ulti-*Ph*ysics Finite-Element *Lib*rary

Matthias Heil and Andrew L. Hazel

School of Mathematics, University of Manchester, Manchester M13 9PL, UK

Abstract. This paper discusses certain aspects of the design and implementation of oomph-lib, an object-oriented multi-physics finite-element library, available as open-source software at http://www.oomph-lib.org. The main aim of the library is to provide an environment that facilitates the robust, adaptive solution of multi-physics problems by monolithic discretisations, while maximising the potential for code re-use. This is achieved by the extensive use of object-oriented programming techniques, including multiple inheritance, function overloading and template (generic) programming, which allow existing objects to be (re-)used in many different ways without having to change their original implementation.

These ideas are illustrated by considering some specific issues that arise when implementing monolithic finite-element discretisations of large-displacement fluid-structure-interaction problems within an Arbitrary Lagrangian Eulerian (ALE) framework. We also discuss the development of wrapper classes that permit the generic and efficient evaluation of the so-called "shape derivatives", the derivatives of the discretised fluid equations with respect to those solid mechanics degrees of freedom that affect the nodal positions in the fluid mesh. Finally, we apply the methodology in two examples.

1 Introduction

The development of efficient and robust methods for the numerical solution of multi-physics problems, such as large-displacement fluid-structure interactions, involves numerous challenges. One of the key issues is how best to combine existing "optimal" methodologies for the solution of the constituent single-physics problems in a coupled framework.

The two main approaches are "partitioned" and "monolithic" solvers. In a partitioned approach, existing single-physics codes are coupled via a global fixed-point (Picard) iteration and the single-physics codes are treated as "black-box" modules, whose internal data structures are regarded as inaccessible. The approach facilitates (in fact, requires) code re-use and is the only feasible approach if the source code for the single-physics solvers is unavailable, e.g. commercial software packages. The disadvantage of this approach is that the Picard iteration often converges very slowly, or not at all, even when good initial guesses are available. Under-relaxation or Aitken extrapolation may improve the convergence characteristics (see, e.g., [3] and, more recently, [10]), but in many cases (especially in time-dependent problems with strong

fluid-structure interaction) even these methods are not sufficient to ensure convergence.

Monolithic solvers are based on the fully-coupled discretisation of the governing equations, allowing (but also demanding) complete control over every aspect of the implementation. This approach allows the complete system of nonlinear algebraic equations that results from the coupled discretisation to be solved using Newton's method. If good initial guesses for the solution are available, e.g. from continuation methods or when time-stepping, the Newton iteration converges quadratically, leading to a robust method for solving the coupled problem.

A monolithic discretisation allows direct access to the code's internal data structures and facilitates the implementation of non-standard boundary conditions, or other "exotic" constraints. Furthermore, preconditioners for the iterative solution of the linear systems that must be solved during the Newton iteration may be derived directly from the governing equations; see, e.g., [4]. While these characteristics make monolithic solvers attractive, their implementation is often regarded as (too) labour intensive, and code re-use is perceived to be difficult to achieve.

In this paper we shall discuss the design and implementation of `oomph-lib`, an object-oriented multi-physics finite-element library, available as open-source software at `http://www.oomph-lib.org`. The main aim of the library is to provide an environment that facilitates the monolithic discretisation of multi-physics problems while maximising the potential for code re-use. This is achieved by the extensive use of object-oriented programming techniques, including multiple inheritance, function overloading and template (generic) programming, which allow existing objects to be (re-)used in many different ways without having to change their original implementation.

We shall illustrate these techniques by considering some specific issues that arise when implementing monolithic finite-element discretisations of large-displacement fluid-structure-interaction problems (and many other free boundary problems) within an Arbitrary Lagrangian Eulerian (ALE) framework:

1. It must be possible for the "load terms" in the solid mechanics finite elements to depend on unknowns in the coupled problem because the traction that the fluid exerts onto the solid must be determined as part of the overall coupled solution.
2. The solution of the equations of solid mechanics determines the shape of the fluid domain. We, therefore, require clearly defined interfaces that allow the transfer of geometric information between the solid mechanics elements and the procedures that generate the (fluid) mesh, and update its nodal positions in response to changes in the shape and position of the domain boundary.
3. The discretised fluid equations are affected by changes in the nodal positions within the fluid mesh, which are determined indirectly (via the node

update procedures referred to in 2.) by the solid mechanics degrees of freedom. A monolithic discretisation of the coupled problem requires the efficient evaluation of the so-called "shape derivatives" — the derivatives of the discretised fluid equations with respect to those solid mechanics degrees of freedom that affect the nodal positions in the fluid mesh.

In order to maximise the potential for code re-use, it is desirable to provide this functionality without having to re-implement any existing fluid or solid elements or any mesh generation/update procedures.

The outline of this paper is as follows: after a brief discussion of `oomph-lib`'s general design objectives in Section 2, Section 3 provides an overview of `oomph-lib`'s data structures and discusses the library's fundamental objects: `Data`, `Node`, `Element`, `Mesh` and `Problem`. In Section 4 we illustrate how multiple inheritance, combining the `GeneralisedElement` and `GeomObject` base classes, is used to represent domain boundaries whose positions are determined as part of the solution. Section 5 explains the mesh generation process and illustrates how `oomph-lib`'s mesh adaptation procedures allow the fully-automatic spatial adaptation of meshes in domains that are bounded by curvilinear boundaries. In Section 6 we describe the joint use of template programming and multiple inheritance to create wrapper classes that "upgrade" existing elements to elements that allow the generic and efficient evaluation of the "shape-derivatives". Finally, Section 7 presents several examples: a "toy" free-boundary problem in which the solution of a 2D Poisson equation is coupled to the shape of the domain boundary; and an unsteady large-displacement fluid-structure-interaction problem: finite-Reynolds-number flow in a rapidly oscillating elastic tube.

2 The Overall Design

2.1 General Design Objectives

The main aim of the library is to provide a general framework for the discretisation and the robust, adaptive solution of a wide range of multi- (and single-)physics problems. The library provides fully-functional elements for a wide range of "classical" partial differential equations (the Poisson, Advection-Diffusion, and the Navier–Stokes equations; the Principle of Virtual Displacements (PVD) for solid mechanics; etc.) and it is easy to formulate new elements for other, more "exotic" problems. Furthermore, it is straightforward to combine existing single-physics elements to create hybrid elements that can be used in multi-physics simulations.

"Raw speed" is regarded as less important than robustness and generality, but this is not an excuse for inefficiency. The use of appropriate data structures and "easy-to-use" spatial and temporal adaptivity are a key feature of the library.

Generic tasks such as equation numbering, the assembly and solution of the system of coupled nonlinear algebraic equations, timestepping, etc. are

fully implemented and may be executed via simple and intuitive high-level interfaces. This allows the "user" to concentrate on the problem formulation, performed by writing C++ "driver" codes that specify the discretisation of a (mathematical) problem as a `Problem` object.

2.2 The Overall Framework

Within `oomph-lib`, all problems are regarded as nonlinear and it is assumed that any continuous (sub-)problems will be discretised in time and space, i.e. the problem's (approximate) solution must be represented by M discrete values V_j ($j = 1, ..., M$), e.g. the nodal values in a finite-element mesh. Boundary conditions and other constraints prescribe some of these values, and so only a subset of the M values are unknown. We shall denote these unknowns by U_i ($i = 1, ..., N$) and assume that they are determined by a system of N non-linear algebraic equations that may be written in residual form:

$$\mathcal{R}_i\left(U_1, ..., U_N\right) = 0 \quad \text{for } i = 1, ..., N. \tag{1}$$

By default, `oomph-lib` uses Newton's method to solve the system (1). The method requires the provision of an initial guess for the unknowns, and the repeated solution of the linear system

$$\sum_{i=1}^{N} \mathcal{J}_{ij}\ \delta U_j = -\mathcal{R}_i \quad \text{for } i = 1, ..., N, \tag{2}$$

where

$$\mathcal{J}_{ij} = \frac{\partial \mathcal{R}_i}{\partial U_j} \quad \text{for } i, j = 1, ..., N \tag{3}$$

is the Jacobian matrix. The solution of the linear system is followed by an update of the unknowns,

$$U_i := U_i + \delta U_i \quad \text{for } i = 1, ..., N. \tag{4}$$

Steps (2) and (4) are repeated until the infinity norm of the residual vector, $||\boldsymbol{\mathcal{R}}||_\infty$, is sufficiently small. Within this framework, linear problems are special cases for which Newton's method converges in a single iteration.

The adaptive solution of a given problem involves three main tasks:

1. The (repeated) "assembly" of the global Jacobian matrix and residual vector
   `oomph-lib` employs a finite-element-type framework in which each "element" provides a contribution to the global Jacobian matrix, $\boldsymbol{\mathcal{J}}$, and the global residual vector, $\boldsymbol{\mathcal{R}}$, as illustrated in Fig. 1. We note that `oomph-lib`'s definition of an "element" is very general. While the elemental residual vectors and Jacobian matrices *may* arise from finite-element discretisations, they could equally well represent finite-difference stencils or algebraic constraints.

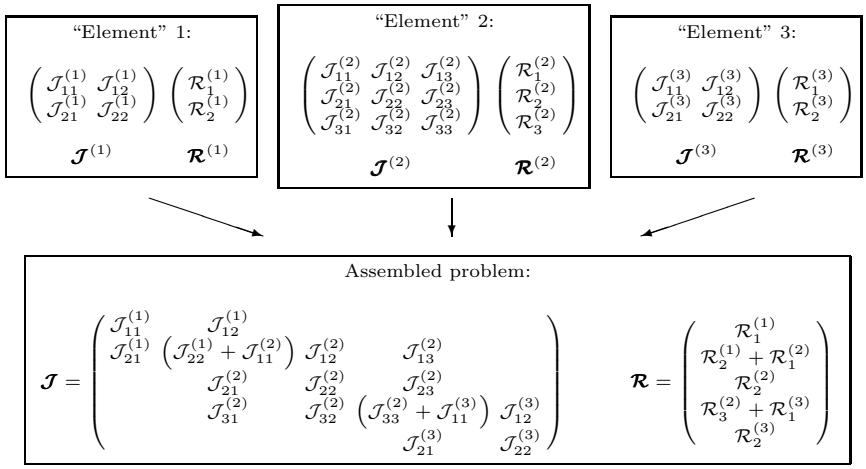

Fig. 1. Schematic illustration of the "assembly" process. Each "element" provides a contribution to the global Jacobian matrix $\boldsymbol{\mathcal{J}}$ and residual vector $\boldsymbol{\mathcal{R}}$. The elemental contributions may arise from a finite-element discretisation but they could equally well represent finite-difference stencils or algebraic constraints.

2. The solution of the linear systems
   The Newton iteration requires the repeated solution of the linear systems (2) that are (at least formally) specified by the global Jacobian matrix $\boldsymbol{\mathcal{J}}$ and residual vector $\boldsymbol{\mathcal{R}}$. (Not all linear solvers actually require the assembly of these objects. For instance, frontal solvers perform the LU decomposition of the Jacobian matrix "on the fly".) In cases where the assembled Jacobian matrix is required, the assembly can be performed serially or in parallel, using an MPI-based implementation of the assembly process. The solution of the linear systems is performed by `LinearSolver` objects, most of which currently represent wrappers to state-of-the-art third-party direct solvers such as the frontal solver `MA42` from the HSL library [7] or the serial and parallel versions of `SuperLU` [2,9]. `IterativeSolver` and `Preconditioner` classes are currently under development.

3. Error estimation and problem adaptation
   Following the solution of the discretised problem with a given spatial discretisation, `oomph-lib`'s `ErrorEstimator` objects may be used to determine error estimates for each "element" in the mesh. `oomph-lib` provides fully automatic mesh adaptation procedures which refine (or unrefine) the mesh in regions in which the estimated error is larger (or smaller) than certain user-specified thresholds. The procedures are implemented via high-level interfaces so that a simple modification to the driver code

suffices to compute a fully adaptive solution to a given problem; see the two example driver codes shown in Fig. 3 below.

3 The Data Structure

3.1 The Fundamental Objects

Fig. 2 presents an overview of the relation between oomph-lib's fundamental objects: Data, Node, Element, Mesh and Problem.

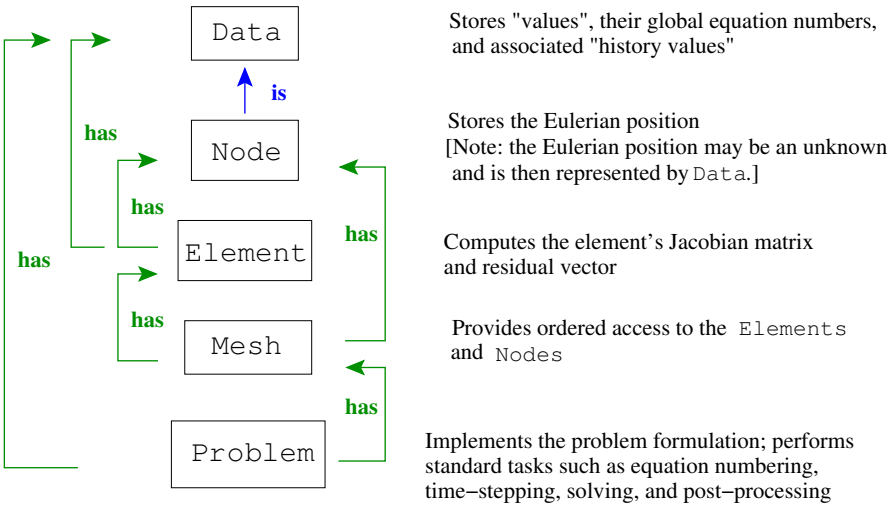

Fig. 2. Overview of the relation between oomph-lib's fundamental objects.

Data: The ultimate aim of any oomph-lib computation is the determination of the M values V_i $(i = 1, ..., M)$ that represent the solution to the discretised problem. These values are either prescribed ("pinned") by boundary conditions, or are unknowns. Each of the N unknown values is associated with a unique global (equation) number in the range 1 to N. oomph-lib's Data object provides storage for values and their global equation numbers.

In many problems, the values represent components of vectors and it is often desirable to combine related values in a single object. For instance, in the finite-element discretisation of a 3D Navier-Stokes problem, each node stores three velocity components. Data therefore provides storage for multiple values. Furthermore, in time-dependent problems, the implicit approximation

of time-derivatives requires the storage of auxiliary "history values". For instance, in a backward Euler time-discretisation, the value of the unknown at the previous timestep is required to evaluate an approximation of the value's time-derivative. `Data` provides storage for such history values, and stores a (pointer to a) `TimeStepper` object that translates the history values into approximations of the values' time-derivatives.

`Nodes:` `Nodes` are `Data`, i.e. they store values, but they also store the node's spatial (Eulerian) coordinates. In solid mechanics problems, the nodal coordinates can themselves be unknowns and in that case they are represented by `Data`.

`Elements:` The main role of `Elements` is to provide contributions to the global Jacobian matrix and the residual vector. The elemental contributions typically depend on a subset of the problem's `Data`, which are accessed via pointers, stored in the `Element`. Within an `Element`, we distinguish between three different types of `Data`: **(i)** Internal `Data` contains values that are local to the element, such as the pressure in a Navier-Stokes element with a discontinuous pressure representation; **(ii)** Nodal `Data` is usually shared with other elements and all elements that share a given `Node` make contributions to the global equations that determine its values; **(iii)** External `Data` contains values that *affect* the element's residual vector and its Jacobian matrix but are not determined *by* it. For instance, in a fluid-structure-interaction problem, the load that acts on a solid-mechanics finite element *affects* its residual but is determined *by* the adjacent fluid element(s).

`Meshes:` The main role of a `Mesh` is to provide ordered access to its `Nodes` and `Elements`. A `Mesh` also provides storage for (and access to) lookup schemes that identify the `Nodes` that are located on domain boundaries.

`Problem:` To solve a given (mathematical) problem with `oomph-lib`, its discretisation must be specified in a suitable `Problem` object. This usually involves the specification of the `Mesh` and the `Element` types, followed by the application of boundary conditions. If spatial adaptivity is required, an `ErrorEstimator` object must also be specified. The error estimator is used by `oomph-lib`'s automatic mesh adaptation procedures to determine which elements should be refined or unrefined. The `Problem` base class implements generic tasks such as equation numbering, the solution of the non-linear algebraic equations by Newton's method, time-stepping, error estimation and spatial adaptation, etc. Typically, the problem specification is provided in the constructor, in which case the driver code can be as simple as the ones shown in Fig. 3. Note the trivial change required to enable spatial adaptivity.

```
// Driver code solves problem
// on a fixed mesh
main()
{
  // Create the problem object
  ReallyHardProblem problem;


  // Solve the problem on
  // the specified mesh
  problem.newton_solve();

  // Document the solution
  problem.doc_solution();
}
```

```
// Driver code solves problem
// with spatial adaptivity
main()
{
  // Create the problem object
  ReallyHardProblem problem;

  // Solve, adapt the mesh,
  // re-solve, ... up to
  // three times
  problem.newton_solve(3);

  // Document the solution
  problem.doc_solution();
}
```

Fig. 3. Two simple driver codes illustrate `oomph-lib`'s high-level interfaces. Note that fully-automatic spatial adaptivity is enabled by a trivial change to the driver code.

3.2 An Example of Object Hierarchies: The Inheritance Structure for Elements

Most of `oomph-lib`'s fundamental objects are implemented in a hierarchical structure to maximise the potential for code re-use. Typically, abstract base classes are employed to **(i)** define interfaces for functions that all objects of this type must have, but that cannot be implemented in generality; and **(ii)** to implement concrete functions that perform generic tasks common to all such objects. Templating is used extensively to define families of related objects.

As an example, Fig. 4 illustrates the typical inheritance structure for finite elements. As discussed above, the minimum requirement for all elements is that they must be able to compute their contribution to the global Jacobian matrix and the residual vector. Interfaces for these tasks are defined[1] in the base class `GeneralisedElement`. For instance, the computation of the elemental Jacobian matrix must be implemented in the function `GeneralisedElement::get_jacobian(...)`. The class also provides storage for the (pointers to the) element's external and internal `Data`. (`GeneralisedElements` do not necessarily have `Nodes`; see Section 4.1 for an example). Finally, the class implements various generic tasks, such as the setup of the local/global equation numbering scheme for the values associated with the `Data` objects that affect the element.

The next level in the element hierarchy are `FiniteElements`. All `FiniteElements` have `Nodes`, and the `FiniteElement` class provides pointer-based

[1] The is achieved by implementing them as "pure virtual" C++ functions.

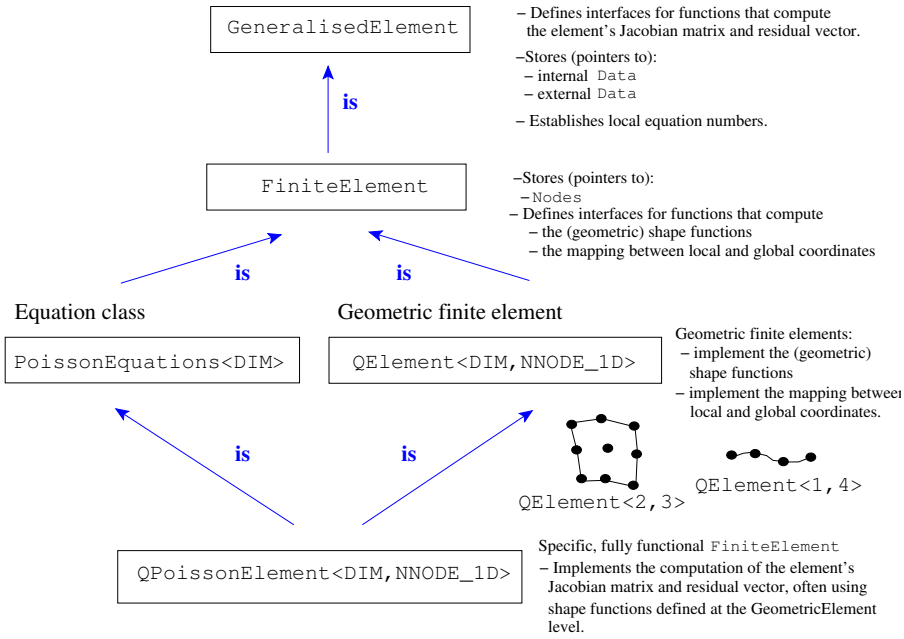

Fig. 4. Typical inheritance structure for `FiniteElements`.

access to these. Furthermore, all `FiniteElements` have (geometric) shape functions which are used to compute the mapping between the element's local and global (Eulerian) coordinates. The number and functional form of these shape functions depend on the specific element geometry, therefore the `FiniteElement` class only defines abstract interfaces for these functions.

Shape functions are implemented in specific "geometric" `FiniteElements`, such as the `QElement` family of 1D line, 2D quad and 3D brick elements. `QElements` are templated by the spatial dimension and the number of nodes along the element's 1D edges so that `QElement<1,4>` represents a four-node line element, while `QElement<3,2>` is an eight-node brick element, etc.

"Equation classes", such as `PoissonEquations`, are also derived directly from the `FiniteElement` class and implement the computation of the element's Jacobian matrix and its residual vector for a specific mathematical problem, based on the weak form of the partial differential equation (PDE). Within the equation classes, we only define the interfaces to the functions that compute the shape functions (used to represent the element geometry), the basis functions (used to represent the unknown functions) and the test functions. Their full specification is delayed until the next and final level of the element hierarchy. Templating is again used to implement equations in dimension-independent form, wherever possible. Table 1 provides a partial

• PoissonEquations<DIM>
• AdvectionDiffusionEquations<DIM>
• UnsteadyHeatEquations<DIM>
• LinearWaveEquations<DIM>
• NavierStokesEquations<DIM>
• AxisymmetricNavierStokesEquations
• PVDEquations<DIM>
• PVDEquationsWithPressure<DIM>
• KirchhoffLoveBeamEquations
• KirchhoffLoveShellEquations

Table 1. Partial list of oomph-lib's equation classes. These may be combined (by multiple inheritance) with any geometric finite element that provides sufficient inter-element continuity to form fully-functional finite elements. The presence of a template argument, DIM, indicates the dimension-independent implementation of the equations. The two PVD equation elements implement the principal of virtual displacements in the displacement and displacement/pressure formulations, respectively.

list of currently implemented equation classes. oomph-lib's documentation provides instructions and numerous "worked examples" that illustrate how to create additional equation classes.

Finally, fully functional elements are constructed via multiple inheritance, by combining a specific geometric FiniteElement with a specific equation class. The (geometric) shape functions, provided by the geometric FiniteElement class implement the abstract shape functions defined in the equation class. For isoparametric Galerkin finite elements, the geometric shape functions are also used as the basis and test functions; for Petrov-Galerkin methods or for elements that use different interpolations for different variables (e.g. velocity and pressure in mixed Navier-Stokes elements), additional basis and test functions may be specified when the specific element is defined. Again, templating is used to create families of elements. For instance, the QPoissonElement<DIM,NNODE_1D> represents the family of isoparametric, Galerkin finite elements that discretise the DIM-dimensional Poisson equation on line, quad or brick elements with $\texttt{NNODE\_1D}^{\texttt{DIM}}$ nodes.

The hierarchical implementation maximises the potential for code re-use, because any equation class may be combined with any geometric element, provided the degree of inter-element continuity of the geometric element is consistent with the differentiability requirements imposed by the weak form of the PDE represented by the equation class. The distinction between equation classes and geometric elements also facilitates the generic implementation of mesh generation and adaptation procedures, which both operate on the level of geometric FiniteElements.

4 GeneralisedElements and GeomObjects – How to Represent Unknown Domain Boundaries

The inheritance structure discussed in the previous section contains objects that arise naturally in the course of the finite-element discretisation of "classical" PDE problems. However, `oomph-lib` does not require the PDEs to be discretised by finite-element methods. The `GeneralisedElements`' contributions to the global residual vector and the Jacobian matrix may equally well represent finite-difference stencils or algebraic constraints. We shall now illustrate how this allows the representation of unknown domain boundaries in fluid-structure-interaction problems.

4.1 An Example of a GeneralisedElement

Fig. 5(a) shows a very simple example of an object that may be encountered in a fluid-structure-interaction problem: a circular ring of radius R whose centre is located at (X_c, Y_c). The ring is mounted on an elastic foundation (a spring of stiffness k), and is loaded by an external force f. The vertical displacement of the ring is governed by the algebraic equilibrium equation

$$f = k\, Y_c. \tag{5}$$

If the ring represents a boundary in a fluid-structure-interaction problem,

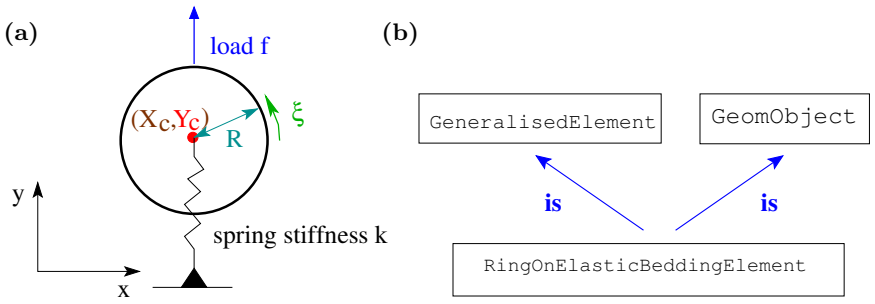

Fig. 5. A ring on an elastic foundation and its implementation as a `GeneralisedElement` and a `GeomObject`.

f would be the (resultant) vertical force that the surrounding fluid exerts onto the ring. To allow the determination of the ring's vertical displacement, Y_c, as part of the overall solution, the ring must be represented by a `GeneralisedElement` – a `RingOnElasticBeddingElement`, say. For this purpose we represent Y_c as the element's internal `Data` whose single unknown value is determined by the residual equation (5). In a fluid-structure-interaction problem, the load f is an unknown. Its magnitude *affects* the

element's residual equation, but is not determined *by* the element, so we represent the load as external `Data`. If the load f is prescribed, a situation that would arise if the `RingOnElasticBeddingElement` was used in a (trivial) single-physics problem, "pinning" the value that represents f (using the `Data` member function `Data::pin(...)`) automatically excludes it from the element's list of unknowns. Similarly, the vertical position of the ring may be fixed by "pinning" the value that represents Y_c.

The entries in the element's residual vector contain the element's contribution to the global equations that determine the values of its (up to) two unknowns. If both Y_c and f are unknown, the first entry in the residual vector is given by the residual of the equilibrium equation (5). The element does not make a direct contribution to the equation that determines the external load, therefore we set the second entry to zero. The 2×2 elemental Jacobian matrix contains the derivatives of the two components of the residual vector with respect to the corresponding unknowns, so we have

$$\boldsymbol{\mathcal{R}}^{(E)} = \begin{pmatrix} f - k\ Y_c \\ 0 \end{pmatrix} \quad \text{and} \quad \boldsymbol{\mathcal{J}}^{(E)} = \begin{pmatrix} -k & 1 \\ 0 & 0 \end{pmatrix}. \tag{6}$$

If either f or Y_c are pinned, the element contains only a single unknown and its Jacobian matrix and residual vector are reduced to the appropriate 1×1 sub-blocks. If both values are pinned, the element does not make any contribution to global Jacobian matrix and residual vector.

4.2 An Example of a `GeomObject`

If used in a fluid-structure-interaction problem, the `RingOnElasticBedding-Element` defines the boundary of the fluid domain. Hence its position and shape must be accessible (via standard interfaces) to `oomph-lib`'s mesh generation and mesh update procedures. `oomph-lib` provides an abstract base class, `GeomObject`, that defines the common functionality of all objects that describe geometric features. It is assumed that the shape of a `GeomObject` may be specified explicitly by a position vector $\mathbf{R}(\boldsymbol{\xi})$, parameterised by a vector of intrinsic (Lagrangian) coordinates, $\boldsymbol{\xi}$, where $dim(\mathbf{R}) \geq dim(\boldsymbol{\xi})$. For instance, the ring's shape may be represented by a 2D position vector $\mathbf{R}$, parametrised by the 1D Lagrangian coordinate ξ;

$$\mathbf{R}(\xi) = \begin{pmatrix} X_c + R\cos(\xi) \\ Y_c + R\sin(\xi) \end{pmatrix}. \tag{7}$$

This parametrisation must be implemented in the `GeomObject`'s member function `GeomObject::position(xi,r)`, which computes the position vector `r` as a function of the vector of the intrinsic coordinates `xi`.

Multiple inheritance allows the `RingOnElasticBeddingElement` to exist as both a `GeneralisedElement` and a `GeomObject`, as indicated by the inheritance diagram in Fig. 5(b). Its role as a `GeneralisedElement` allows us

to determine its vertical height, Y_c, as part of the overall solution process; its role as a `GeomObject` allows us to use it for the parametrisation of the domain boundary, e.g. during mesh generation.

In fluid-structure-interaction problems, the (solid mechanics) unknowns that determine the position of the domain boundary affect the residuals and Jacobian matrices of the elements in the fluid mesh. The monolithic solution of the coupled problem via Newton's method requires the evaluation of the derivatives of the fluid mechanics residuals with respect to the (solid mechanics) unknowns that determine the shape of the fluid domain — the so-called "shape derivatives". To facilitate such computations, the `GeomObject` class provides storage for (pointers to) those `Data` objects whose values affect the object's shape and position. We refer to these as "geometric `Data`" and note that they should be identified and declared whenever a specific `GeomObject` is implemented. For instance, in the above example, the internal `Data` object that stores the value of Y_c represents the `RingOnElasticBeddingElement`'s only geometric `Data`.

Similar inheritance structures are implemented for "real" solid mechanics elements. For instance, in the 2D fluid-structure-interaction problem to be discussed in Section 7, the fluid domain is bounded by a thin-walled elastic ring. The ring is discretised by a surface mesh of `KirchhoffLoveBeamElements`. The shape of a deformed beam element is defined by interpolation between its nodal coordinates (represented by the `Node`'s positional `Data`), using the element's geometric shape functions. The element's 1D local coordinate, therefore, parametrises its 2D shape and allows it to be implemented as a `GeomObject` that can be used to define domain boundaries in fluid-structure-interaction problems. The positional `Data` stored at the element's `Nodes` is the `GeomObject`'s geometric `Data`. Fig. 6 illustrates the inheritance structure for this element.

5 Mesh Generation and Adaptation in Domains with Curvilinear Boundaries

In the previous Section we demonstrated how `GeomObjects` provide standardised interfaces for the specification of domain boundaries, and illustrated how multiple inheritance may be used to deal with domain boundaries whose positions must be determined as part of the overall solution. We now discuss how the geometric information provided by `GeomObjects` is used to create and adapt meshes in domains with arbitrary, curvilinear boundaries. The methodology employed during the mesh generation allows a sparse update of the nodal positions in response to changes in the boundary shape — a key requirement for the efficient solution of fluid-structure-interaction problems using monolithic schemes.

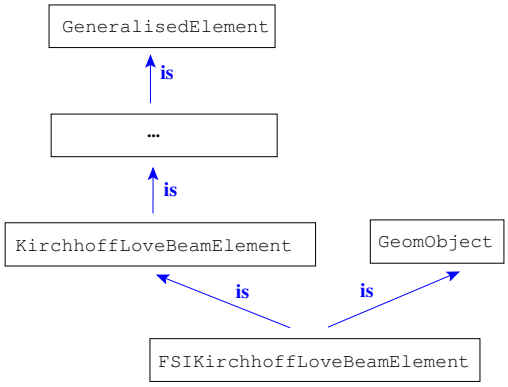

Fig. 6. Inheritance structure illustrating how a `KirchhoffLoveBeamElement` is "upgraded" to an element that may be used in fluid-structure-interaction problems.

5.1 Two Simple Examples

We first illustrate `oomph-lib`'s mesh adaptation capabilities in two simple single-physics problems. To begin, consider the 2D Poisson problem

$$\nabla^2 u = 1 \quad \text{in } D_{\text{fish}} \quad \text{subject to} \quad u = 0 \text{ on } \partial D_{\text{fish}}, \tag{8}$$

where D_{fish} is the fish-shaped domain, shown in Fig. 7(a). The "fish body" is bounded by two circular arcs of radius R, whose centres are located at $(X_c, \pm Y_c)$; the remaining domain boundaries are straight lines. The plots in Figs. 7(b-e) show contours of the solution, computed on the meshes that are generated at successive stages of the fully-automatic mesh-adaptation process. Note that `oomph-lib` requires only the provision of a very coarse initial mesh, here containing just four nine-node quad elements of type `QPoisson-Element<2,3>`.

Following the initial solution, `oomph-lib`'s mesh adaptation procedures refine the mesh, first uniformly throughout the domain, then predominantly near the inward facing corners, where the solution of Poisson's equation is singular.

Fig. 8 shows a 3D example: steady entry flow into a circular cylindrical tube at a Reynolds number of $Re = 200$. The axial velocity profiles illustrate how the flow develops from the entry profile $\mathbf{u} = (1 - r^{20})\mathbf{e}_z$ towards the parabolic Hagen-Poiseuille profile. This computation was started with a very coarse initial mesh, comprising six axial layers, each containing three elements. The automatic mesh adaptations then refined the mesh, most strongly near the inflow where a thin boundary layer develops on the tube wall.

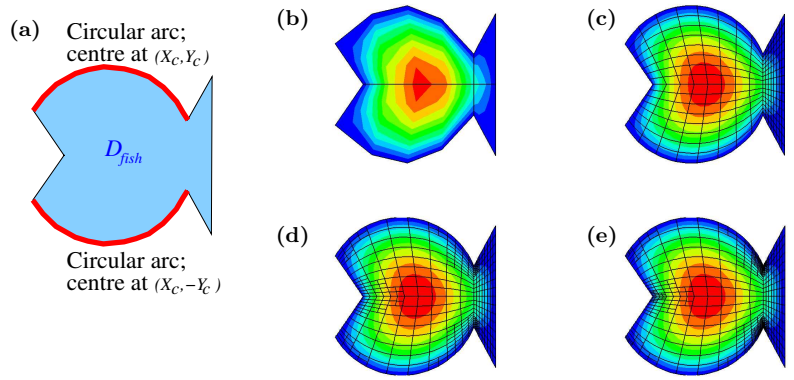

Fig. 7. The solution of a 2D Poisson equation in a fish-shaped domain. Fig. (a) shows the problem sketch; Figs. (b)-(e) show contours of the computed solution, obtained on the meshes that are generated by `oomph-lib`'s automatic mesh adaptation procedures.

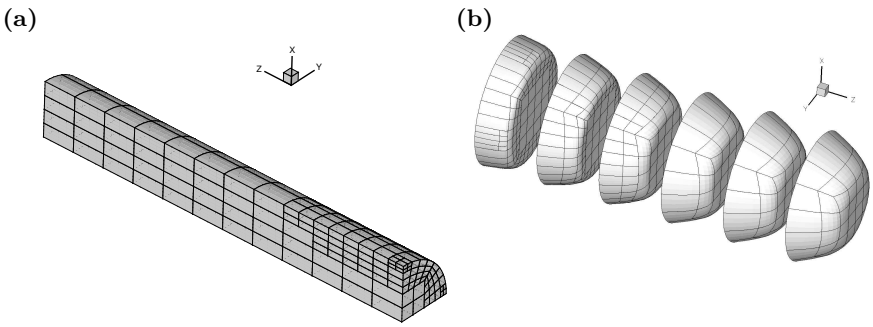

Fig. 8. Adaptive computation of finite-Reynolds-number entry flow into a circular cylindrical tube at $Re = 200$. (a) The adapted mesh (flow is from right to left). (b) Axial velocity profiles (flow is from left to right).

5.2 Some Details of the Implementation

oomph-lib's fully-automatic mesh adaptation routines use generic high-level interfaces to the procedures that implement the adaptation for specific types of meshes (e.g. meshes consisting of quad or brick elements). The adaptation involves the following specific steps:

1. Compute an error estimate for all elements in the mesh. This task is performed by a specific `ErrorEstimator` object. Error estimation in the two examples shown above was performed with oomph-lib's `Z2ErrorEstimator` which provides an implementation of Zhu & Zienkiewicz's flux recovery procedure [12].
2. Select all elements whose error estimate exceeds (or falls below) certain user-specified thresholds for refinement (or unrefinement).
3. Split all elements that are scheduled for refinement into "son" elements and collapse groups of elements that are scheduled for unrefinement into their "father" element, provided *all* elements in the group are scheduled for unrefinement.
4. Delete any nodes that might have become obsolete, and create new ones where required. Interpolate the previously computed solution onto the new nodes and apply the correct boundary conditions for any newly created nodes that are located on a domain boundary.
5. Identify any "hanging nodes", i.e. nodes on the edges or faces of elements that are not shared by adjacent (larger) elements. Inter-element continuity of the solution is ensured by constraining the nodal values and coordinates at such nodes so that they represent suitable linear combinations of these quantities at the associated "master nodes"; see Fig. 9(c). This is achieved through the implementation of the access functions to the nodal values and coordinates. For instance, at non-hanging nodes, the function `Node::value(j)` returns the `j`-th nodal value itself; at hanging nodes, it returns the weighted averages of the `j`-th values at the "master nodes".
6. Re-generate the equation numbering scheme.

Provided an `ErrorEstimator` object has been specified, the above steps are performed completely automatically by a call to the function `Problem::adapt()`. On return from this function, the adapted problem may be re-solved immediately.

At present, oomph-lib provides implementations of these procedures for meshes that contain 2D quad and 3D brick elements. Generalised quadtree and octree data structures are used to store the refinement pattern, and to identify efficiently the elements' edge and face neighbours during the determination of the hanging nodes and their associated "master nodes".

5.3 How to Resolve Curvilinear Boundaries: `Domains` and `MacroElements`

One particular aspect of the implementation requires a more detailed discussion. How do `oomph-lib`'s mesh adaptation procedures determine the position of newly created nodes in domains with curvilinear boundaries? Fig. 9 illustrates the potential problem. A quarter-circle domain has been discretised

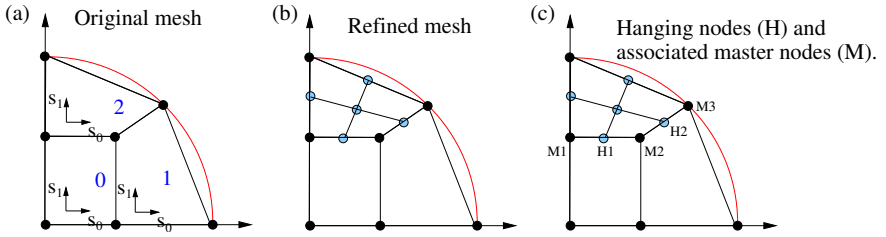

Fig. 9. (a,b): Adaptive mesh refinement without `MacroElements`. The positions of newly created nodes are determined by interpolation, using the "father" element's geometric shape functions. Mesh refinement does not improve the representation of the curvilinear domain boundary. (c): Hanging nodes and associated master nodes: M1 and M2 are master nodes for H1; M2 and M3 are master nodes for H2.

with a very coarse initial mesh, containing three four-node quad elements. Assume that the error estimation indicates that element 2 should be refined. During the refinement, four new "son" elements and five new nodes are created. By default, the nodal values and coordinates of newly created nodes are obtained by interpolation from the corresponding quantities in the "father" element, using the "father" elements shape and basis functions. This procedure is perfectly adequate for meshes in domains with polygonal boundaries. However, in problems with curvilinear domain boundaries, we must ensure that the refined meshes provide a progressively more accurate representation of the exact boundary shape.

In order to achieve this, `oomph-lib` requires domains with curvilinear boundaries to be represented by objects that are derived from the abstract base class `Domain`. All specific `Domain` objects decompose an "exact" domain into a number of macro elements which must have the same topology and be parametrised by the same set of local coordinates as the geometric finite elements in the coarse initial mesh, as illustrated in Fig. 10. A `Domain` object defines the boundaries of its constituent macro elements, given by either the exact curvilinear boundaries (typically represented by `GeomObjects`), or arbitrary (usually straight/planar) internal edges/faces. Common interfaces for macro elements are defined in the abstract base class `MacroElement`. All macro elements must implement the member function

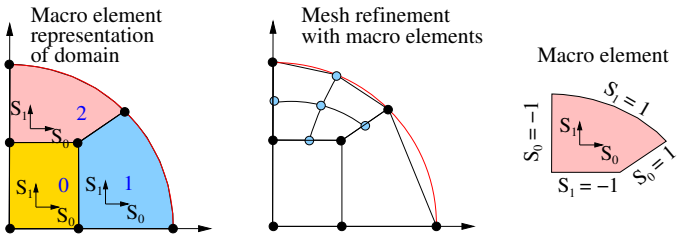

Fig. 10. `MacroElements` decompose a `Domain` into a number of subdomains which have the same topology as the corresponding `FiniteElements` in the coarse initial mesh.

`MacroElement::macro_map(S,r)` which establishes the mapping between the macro element's vector of local coordinates, `S`, and the global (Eulerian) position vector, `r`, to a point inside it. The `QMacroElement<DIM>` family provides an implementation of this mapping for 2D quad- and 3D brick-shaped macro elements and may be used with geometric finite elements from the `QElement` family.

The only non-trivial task to be performed when creating a new `Domain` object is the parametrisation of its macro element boundaries; not an overly onerous task, given that a domain may (and indeed should) be parametrised by very few macro elements. Once a physical domain is represented by a `Domain` object, each `FiniteElement` in the coarse initial mesh is associated with one of the `Domain`'s macro elements. The `FiniteElement`'s macro element representation is then employed **(i)** to determine the position of the nodes in the coarse initial mesh, and **(ii)** to determine the position of newly created nodes during mesh refinement.

6 Evaluation of "Shape Derivatives"

6.1 Macro-Element-Based Node Updates

The macro-element-based representation of the domain may also be used to update the nodal positions in response to changes in the domain boundary. The update may be performed on a node-by-node basis, if we allow each `Node` to store **(i)** a pointer to the macro element in which it is located[2], and **(ii)** its local coordinates in that macro element. Thus each `Node` is able to determine (or update) "its own" position by a call to the macro-element mapping `MacroElement::macro_map(S,r)`. (To avoid the allocation of unnecessary storage in problems that do not involve moving boundaries, the storage for

[2] The macro-element mappings of adjacent `MacroElements` are continuous, therefore `Nodes` that are located at the interface between two `MacroElements` may be associated with either one.

these quantities is provided in the class `MacroElementUpdateNode`, derived from `oomph-lib`'s `Node` class.) Once `Nodes` can "update their own positions", the generic and efficient evaluation of shape derivatives in fluid-structure interaction (or any other free-boundary) problems is possible.

6.2 The Generic Evaluation of "Shape Derivatives"

For simplicity we shall illustrate the methodology by considering a "toy" free-boundary problem: the solution of a 2D Poisson problem, coupled to the position of the boundary. Recall that `oomph-lib`'s `QPoissonElement<DIM, NNODE_1D>` is a single-physics element that implements the discretisation of the `DIM`-dimensional Poisson equation $\nabla^2 u(x_i) = g(x_i)$, via an isoparametric Galerkin approach in which the element's $N_u^{(E)}$ geometric shape functions $\psi_i(x_j)$ $(i = 1, ..., N_u^{(E)})$ are also used as test and basis functions. The $N_u^{(E)}$ components of the element's residual vector are given by

$$\mathcal{R}_i^{(E)} = \int_E \left(\sum_{j=1}^{N_u^{(E)}} U_j^{(E)} \sum_{k=1}^{\texttt{DIM}} \frac{\partial \psi_j}{\partial x_k} \frac{\partial \psi_i}{\partial x_k} + g(x_1, ..., x_{\texttt{DIM}})\, \psi_i \right) dV \qquad \text{for } i = 1, ..., N_u^{(E)}, \tag{9}$$

and depend on the element's $N_u^{(E)}$ unknown nodal values, $U_j^{(E)}$ $(j = 1, ..., N_u^{(E)})$. The element's $N_u^{(E)} \times N_u^{(E)}$ Jacobian matrix contains the derivatives of the residual vector with respect to these unknowns,

$$\mathcal{J}_{ij}^{(E)} = \frac{\partial \mathcal{R}_i^{(E)}}{\partial U_j^{(E)}} = \int_E \left(\sum_{k=1}^{\texttt{DIM}} \frac{\partial \psi_j}{\partial x_k} \frac{\partial \psi_i}{\partial x_k} \right) dV \qquad \text{for } i, j = 1, ..., N_u^{(E)}. \tag{10}$$

In a free-boundary problem, the residual also depends on the nodal positions which are determined (via the element's macro element representation) by the position of the domain boundary. As discussed above, unknown domain boundaries are represented by `GeomObjects`, whose shape and position is specified by their geometric `Data`. We shall denote the set of $N_G^{(E)}$ geometric unknowns, represented by the geometric `Data` that affect the nodal positions in an element, by $G_i^{(E)}$ $(i = 1, ..., N_G^{(E)})$.

In order to use an existing `FiniteElement` in a free-boundary problem, the geometric unknowns $G_i^{(E)}$ $(i = 1, ..., N_G^{(E)})$ that determine the element's nodal positions must be added to the list of unknowns that affect the element's residual vector. This requires the extension of the element's Jacobian matrix and residual vector to

$$\boldsymbol{\mathcal{J}}^{(E)} = \begin{pmatrix} \boldsymbol{\mathcal{J}}_{[DD]}^{(E)} & \boldsymbol{\mathcal{J}}_{[DB]}^{(E)} \\ \boldsymbol{\mathcal{J}}_{[BD]}^{(E)} & \boldsymbol{\mathcal{J}}_{[BB]}^{(E)} \end{pmatrix} \quad \text{and} \quad \boldsymbol{\mathcal{R}}^{(E)} = \begin{pmatrix} \boldsymbol{\mathcal{R}}_{[D]}^{(E)} \\ \boldsymbol{\mathcal{R}}_{[B]}^{(E)} \end{pmatrix}, \tag{11}$$

where the subscripts "D" and "B" indicate entries that correspond to equations/unknowns that are associated with the equation being solved inside the domain and those that determine the shape of its boundary, respectively.

`oomph-lib` provides a templated wrapper class,

```
template<class ELEMENT>
class MacroElementNodeUpdateElement<ELEMENT> : public virtual ELEMENT
```

that computes the augmented quantities in (11) in complete generality. Given any existing finite element, specified by the template parameter `ELEMENT`, the `MacroElementNodeUpdateElement` class automatically incorporates the dependence of the element's residual vector on the geometric unknowns involved in its macro-element-based node update functions into the computation of the element's Jacobian matrix.

To explain the implementation, we consider the origin of the various terms in (11) for our "free boundary" Poisson element. The vector $\boldsymbol{\mathcal{R}}^{(E)}_{[D]}$ contains the residuals of the discretised Poisson equation, evaluated for the current values of nodal unknowns, $U^{(E)}_i$ $(i = 1, ..., N^{(E)}_U)$, and the geometric unknowns $G^{(E)}_i$ $(i = 1, ..., N^{(E)}_G)$. The main diagonal block $\boldsymbol{\mathcal{J}}^{(E)}_{[DD]}$ contains the derivatives of $\boldsymbol{\mathcal{R}}^{(E)}_{[D]}$ with respect to the element's nodal values. $\boldsymbol{\mathcal{J}}^{(E)}_{[DD]}$ and $\boldsymbol{\mathcal{R}}^{(E)}_{[D]}$ are therefore given by the Jacobian matrix and the residual vector of the single-physics element, as specified in (9) and (10). In the `MacroElementNodeUpdateElement` these may be obtained directly by calling `ELEMENT::get_jacobian(...)`. The Poisson element does not make a direct contribution to the equations that determine the shape of the domain boundary, therefore we set $\boldsymbol{\mathcal{R}}^{(E)}_{[B]} = \mathbf{0}$, which implies that $\boldsymbol{\mathcal{J}}^{(E)}_{[BD]} = \boldsymbol{\mathcal{J}}^{(E)}_{[BB]} = \mathbf{0}$. Hence, the only non-trivial entry in the augmented element's Jacobian matrix is the off-diagonal block $\boldsymbol{\mathcal{J}}^{(E)}_{[DB]}$. It contains the derivatives of the residual vector of the underlying element with respect to the geometric unknowns $G^{(E)}_i$ $(i = 1, ..., N^{(E)}_G)$ — the "shape derivatives". For our "free boundary" Poisson element these are given by

$$\mathcal{J}^{(E)}_{[DB]\ ij} = \frac{\partial}{\partial G_j} \int_E \left(\sum_{l=1}^{N^{(E)}_u} U^{(E)}_l \sum_{k=1}^{\mathtt{DIM}} \frac{\partial \psi_l}{\partial \underline{x_k}} \frac{\partial \psi_i}{\partial \underline{x_k}} + g(\underline{x_1, ..., x_{\mathtt{DIM}}})\ \psi_i \right) \underline{dV}$$

$$\text{for } i = 1, ..., N^{(E)}_U,\ j = 1, ..., N^{(E)}_G. \quad (12)$$

In Eqn. (12) the underlined quantities are affected by a change in the element's nodal positions, and hence by a change in the geometric unknowns $G^{(E)}_i$ $(i = 1, ..., N^{(E)}_G)$. A change in the geometric unknowns affects the Jacobian of the mapping between local and global coordinates, contained in the differential dV; the derivatives of the shape functions with respect to the global coordinates; and the argument to the source function, $g(x_i)$. The

analytical evaluation of the derivatives of these quantities with respect to the geometric unknowns would result in extremely lengthy algebraic expressions. Furthermore, the precise form of the derivatives is element-specific and would also depend on the macro-element mapping. To permit the evaluation of these terms for any template argument, `ELEMENT`, and any `MacroElement`, `oomph-lib` approximates the derivatives using finite differences,

$$\mathcal{J}^{(E)}_{[DB]\ ij} = \frac{\partial \mathcal{R}_i^{(E)}\left(U_1,...,U_{N_U^{(E)}};G_1,...,G_{N_G^{(E)}}\right)}{\partial G_j}$$
$$\approx \frac{\mathcal{R}_i^{(E)}\left(...,G_j+\epsilon_{FD},...\right)-\mathcal{R}_i^{(E)}\left(...,G_j,...\right)}{\epsilon_{FD}}, \qquad (13)$$

where $\epsilon_{FD} \ll 1$. The evaluation of the finite-difference expressions is a sparse operation because the element's list of geometric unknowns includes only unknowns that actually change the position of at least one of its nodes, implying that only non-zero entries in the Jacobian matrix are computed.

The implementation of the above steps is completely generic, allowing the wrapper class to be used with *any* of `oomph-lib`'s existing finite elements and macro elements. In addition, because the adaptive solution of problems in domains with curvilinear boundaries already requires a macro-element-based representation of the domain, "upgrading" an existing fixed-domain problem to a free-boundary problem is trivial. In fact, it is only necessary to pass a list of (the pointers to) those `GeomObjects` that determine the boundaries of a given macro element to the associated `FiniteElement` when the coarse initial mesh is created. `oomph-lib` automatically extracts the geometric unknowns from the `GeomObject`'s geometric `Data` and includes them in the list of the element's unknowns. Moreover, during mesh refinement, the relevant information is automatically passed to the "son" elements when a coarse element is split.

6.3 Other Node Update Approaches in `oomph-lib`

The generic implementation of the `MacroElementNodeUpdateElement` as a templated wrapper class is only possible because `MacroElementUpdateNodes` are able to update their own positions in response to changes in the shape of the domain boundaries. `oomph-lib` provides a number of alternative node update strategies and associated wrapper elements:

`SpineElement<ELEMENT>:` A generalisation of Kistler & Scriven's "Method of Spines" [8], often used for free-surface fluids problems.

`AlgebraicElement<ELEMENT>:` A generalisation of the `MacroElementNodeUpdateElement` class, discussed above. These elements increase the sparsity of the node update operations in cases where a domain is bounded by many `GeomObjects` and are explained in more detail in Section 7.2. These elements are more efficient than `MacroElementNodeUpdateElement` but require more "user" input.

All elements discussed so far update the nodal positions based on algebraic update functions. oomph-lib also provides the

`PseudoElasticNodeUpdateElement<ELEMENT,SOLID_ELEMENT>`: A doubly-templated wrapper class that uses the equations of solid mechanics (discretised by the solid mechanics element specified by the second template parameter) to update the nodal positions. This element is easiest to use because it requires neither `MacroElements` nor any algebraic update functions. However, it is much more computationally expensive than the other wrapper classes, because it introduces a large number of additional unknowns into the problem.

7 Examples

We present several examples that illustrate the application of the methodologies discussed in the previous sections. Fully-documented demo codes for the solution of the example problems are available from `http://www.oomph-lib.org`.

7.1 A "Toy" Free-Boundary Problem: The Solution of Poisson's Equation, Coupled to the Position of the Domain Boundary

In our first example we combine the two simple single-physics problems of Section 4.1 and Section 5.1 into a "toy" free-boundary problem: Two rings on elastic foundations define the upper and lower curvilinear boundaries of the fish-shaped domain, while $u_{\rm ctrl}$, the solution of Poisson's equation at a pre-selected control node, specifies the load that drives the rings' vertical displacements, as shown in Fig. 11

In oomph-lib, the solution of the coupled problem only requires a few trivial changes to the single-physics (Poisson) code:

- Replace the element type, `QPoissonElement<3,2>`, by `MacroElementNodeUpdateElement<QPoissonElement<3,2> >`.
- Pass the pointers to the `RingOnElasticBeddingElements` (which are already used during the macro-element-based mesh generation in the single-physics code) to the `MacroElementNodeUpdateElements` to indicate that their geometric `Data` affects their nodal positions. During this step, the `RingOnElasticBeddingElements` are used in their role as `GeomObjects`.
- Pass the pointer to the control `Node` in the fish mesh to the `RingOnElasticBeddingElements` to specify the "load". During this step, the control `Node` is used in its role as `Data`.
- Add the `RingOnElasticBeddingElements` to the fish mesh. During this step, the `RingOnElasticBeddingElements` are used in their role as `GeneralisedElements`.

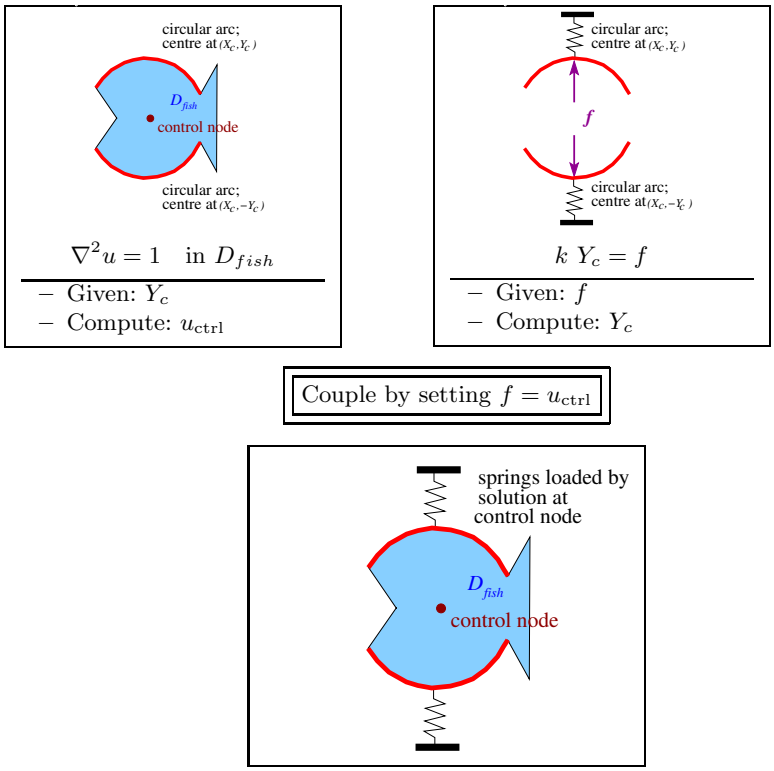

Fig. 11. Sketch illustrating the combination of two simple single-physics problems into a coupled "free boundary" Poisson problem.

Fig. 12 compares the results of a sequence of single-physics computations in which Y_c is prescribed, to the solution of the coupled problem. An increase in Y_c increases the distance between the two `RingOnElasticBeddingElements` that define the upper and lower curvilinear boundaries of the fish-shaped domain. Figs. 12(a-d) shows that this increases the amplitude of the solution of Poisson's equation, causing $u_{\rm ctrl}(Y_c)$ to increase with Y_c, as shown by the solid line in Fig. 12(e).

For a spring stiffness of $k = 1$, the solution of the coupled problem should be located at the intersection of $u_{\rm ctrl}(Y_c)$ with the line $u_{\rm ctrl} = Y_c$ (the dashed line). This is in perfect agreement with `oomph-lib`'s solution of the coupled problem, represented by the square marker.

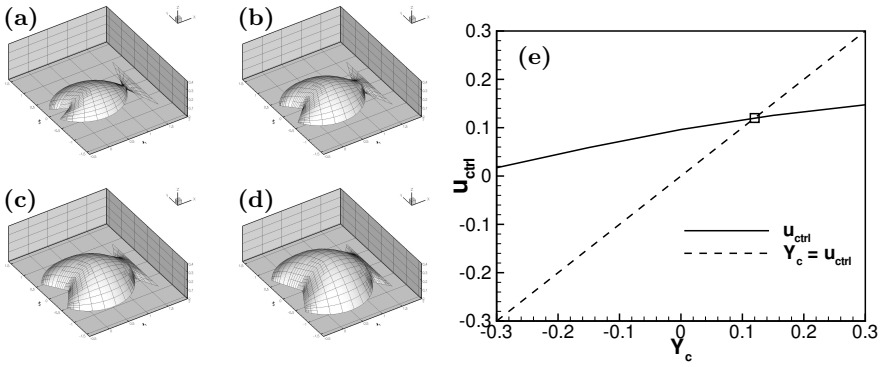

Fig. 12. (a-d) The single-physics solutions of Poisson's equation for various values of Y_c. (e) The solution of Poisson's equation at a control node, $u_{\rm ctrl}$ as a function of Y_c (solid line), and the solution of the coupled problem for $k = 1$ (square marker). The solution is located at the intersection of $u_{\rm ctrl}(Y_c)$ with the line $u_{\rm ctrl} = Y_c$ (the dashed line).

7.2 A Fluid-Structure-Interaction Problem: Flow in Collapsible Tubes

I. Background

Our next example is concerned with the classical biomechanical fluid-structure-interaction problem of flow in collapsible tubes. Many physiological flow problems (e.g. blood flow in the veins and arteries) involve finite-Reynolds-number flows in elastic vessels (see, e.g., [5] for a recent review). Experimental studies of such flows, reviewed in [1], are often performed using a Starling Resistor, a thin-walled elastic tube mounted on two rigid tubes and enclosed in a pressure

chamber. Viscous fluid is pumped through the tube, while the external pressure in the chamber is kept constant. The tube wall is loaded by the external pressure and the fluid traction. When the compressive load exceeds a critical threshold, the tube buckles non-axisymmetrically, undergoing large deflections and inducing strong fluid-structure interaction. The non-axisymmetric collapse is often followed by the development of large-amplitude, self-excited oscillations. The mechanism responsible for the development of these oscillations is not fully understood. This is partly because the theoretical or computational analysis of the problem involves the solution of the 3D unsteady Navier–Stokes equations, coupled to the equations of large-displacement thin-shell theory, a formidable task. However, scaling arguments may be used to simplify the problem in particular regions of parameter space. Here we consider the case in which the Reynolds number associated with the mean flow through the tube is large, and the tube wall performs high-frequency, small-amplitude oscillations. If the 3D unsteady flow $\mathbf{u}(\mathbf{x},t)$ in the tube is decomposed into a steady and a time-periodic unsteady component, so that $\mathbf{u}(\mathbf{x},t) = \overline{\mathbf{u}}(\mathbf{x}) + \widehat{\mathbf{u}}(\mathbf{x},t)$, it may be shown that the unsteady component $\widehat{\mathbf{u}}(\mathbf{x},t)$ uncouples from the mean flow (see [6] for details). Furthermore, because the amplitude of the wall deformation is small, the oscillation causes small changes in the tube's cross-sectional area and only drives small axial flows. This implies that $\widehat{\mathbf{u}}(\mathbf{x},t)$ is dominated by its transverse components,

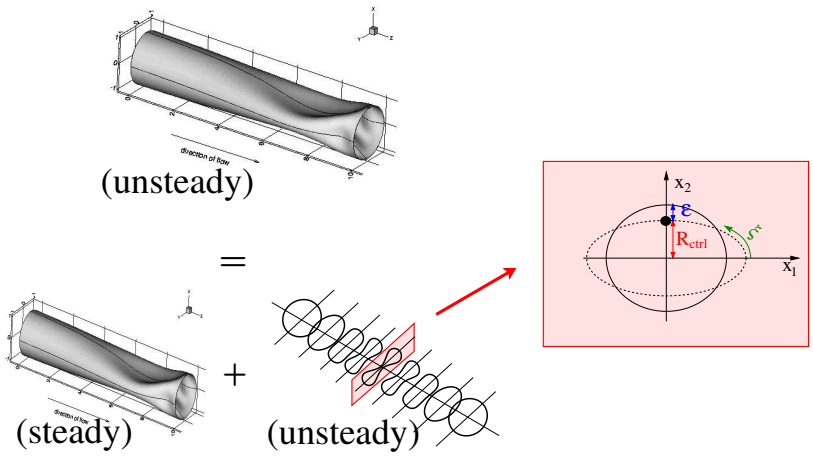

Fig. 13. Large-Reynolds-number flows in collapsible tubes that perform high-frequency, small-amplitude oscillations may be decomposed into steady and time-periodic unsteady components. The time-periodic unsteady flows are dominated by their transverse velocity components and may be determined independently in the cross-sections of the tube.

allowing $\widehat{\mathbf{u}}(\mathbf{x}, t)$ to be determined by computing the 2D flows that develop within the tube's individual cross-sections, as indicated by the sketch in Fig. 13.

II. Prescribed wall motion – `MacroElement`-based node update

We start by analysing the 2D internal fluid flows generated by the non-axisymmetric deflections of a circular ring performing a prescribed high-frequency oscillation that resembles the *in vacuo* oscillations of an elastic ring in its N-th fundamental mode. If we denote the amplitude of the oscillations by ϵ, the time-dependent wall shape, parametrised by the Lagrangian coordinate ξ, shown in Fig. 13, is given by

$$\mathbf{R}(\xi, t) = R_0 \begin{pmatrix} \cos(\xi) \\ \sin(\xi) \end{pmatrix} + \epsilon \ \mathbf{V}_N(\xi) \ \sin(\Omega t), \tag{14}$$

where the wall displacement field, $\mathbf{V}_N(\xi)$, has the form

$$\mathbf{V}_N(\xi) = \begin{pmatrix} \cos(N\xi)\cos(\xi) - \mathbb{A}\sin(N\xi)\sin(\xi) \\ \cos(N\xi)\sin(\xi) + \mathbb{A}\sin(N\xi)\cos(\xi) \end{pmatrix}, \tag{15}$$

see [11]. To investigate this problem with `oomph-lib`, we represent the wall shape by a `GeomObject` and use it to create the coarse initial fluid mesh, shown in the leftmost plot in Fig. 14. The fluid mesh initially contains only three Crozier-Raviart (Q2Q-1) Navier-Stokes elements. Before starting the computation, we perform three uniform refinements, using the `Problem` member function `Problem::refine_uniformly()`, and assign the initial conditions on the resulting mesh, shown in the second mesh plot. The remaining mesh plots in Fig. 14 illustrate how `oomph-lib`'s automatic adaptation procedures adjust the fluid mesh throughout the simulation.

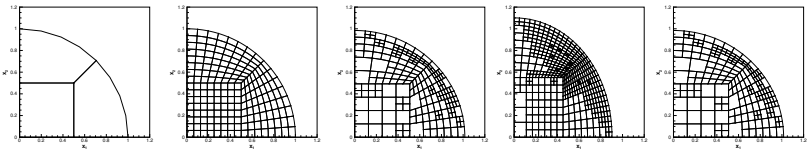

Fig. 14. Mesh adaptation during the simulation of 2D flows that are driven by the motion of the oscillating ring. The simulation is started with the uniform mesh that is obtained by performing three levels of uniform refinement of the coarse initial mesh. The remaining plots illustrate how `oomph-lib`'s automatic mesh adaptation procedures adjust the mesh throughout the simulation.

Because the wall shape is prescribed by equations (14) and (15), the current problem does not involve any "proper" fluid-structure interaction. However, the small but finite change in the ring's cross-sectional area, induced

by the prescribed wall displacement field (15), would violate the discrete mass conservation enforced by the discretised continuity equation. It is therefore necessary to adjust the shape of the boundary so that the total area of the computational domain is maintained, e.g. by allowing the ring's mean radius, R_0, to vary. In principle, R_0 could be determined via the constraint on the area of the computational domain. However, a more elegant (and easier-to-implement) approach is to give the ring some nominal elasticity, so that its mean radius is determined by the "equilibrium equation"

$$R_0 - 1 = p_{\rm ctrl} \tag{16}$$

where $p_{\rm ctrl}$ is the fluid pressure at a certain fixed position, e.g. at the origin. In this approach, variations in the fluid pressure (which, in an incompressible fluid, is only determined up to an arbitrary constant) adjust R_0 so that the area of the computational domain is conserved.

The implementation of this approach is straightforward. We treat R_0 as the `GeomObject`'s geometric `Data` whose single unknown value is determined by the "equilibrium equation" (16). The resulting free-boundary problem may be solved exactly as the free-boundary Poisson problem considered in the previous example.

Fig. 15 compares the computational results (instantaneous streamlines and pressure contours at two characteristic phases during the oscillation) with Heil & Waters' [6] asymptotic predictions. The two upper plots show the velocity field at an instant when the moving wall approaches its undeformed, axisymmetric configuration. The fluid is accelerated and the velocity field resembles an unsteady stagnation point flow. A thin boundary layer exists near the wall but it has little effect on the overall flow field. The two lower plots show the flow during the second half of the periodic oscillation, when the wall approaches its most strongly deformed configuration. A strong adverse pressure gradient decelerates the fluid and an area of "flow separation" appears to form near the wall.

The flow is characterised by the amplitude of the wall oscillation, ϵ, and the Womersley number, α^2, an unsteady Reynolds number. The computations were performed for $\epsilon = 0.1$ and $\alpha^2 = 100$. Given that the asymptotic predictions are only valid in the limit of small amplitude, $\epsilon \ll 1$, and large Womersley number, $\alpha^2 \gg 1$, the agreement between the two sets of results is very satisfying.

III. Full fluid-structure interaction

III.a Node update with `AlgebraicNodes`

Next, we consider the problem with "full" fluid-structure interaction. For this purpose, we replace the `GeomObject` that prescribes the wall motion by a surface mesh of `FSIKirchhoffLoveBeamElements`, loaded by the traction exerted by the adjacent fluid elements. The implementation of the fluid-mesh update in response to changes in the shape of the domain boundary could,

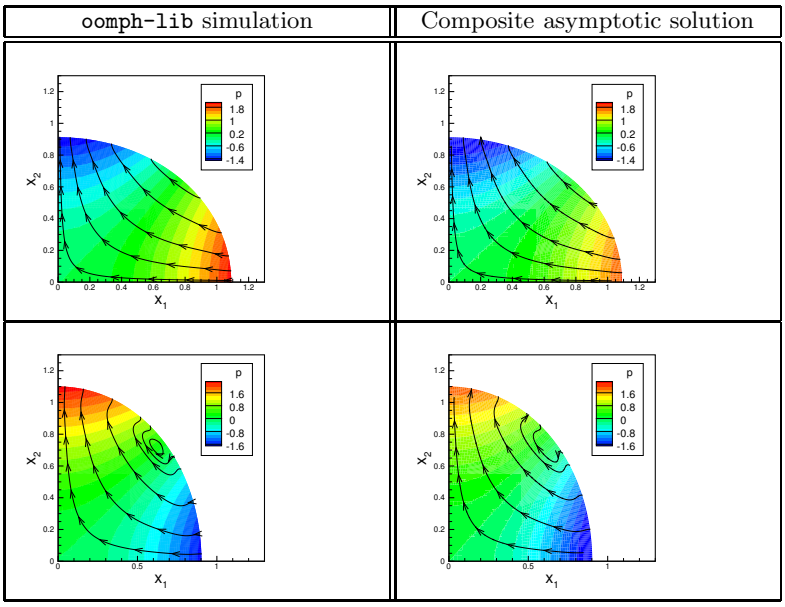

Fig. 15. Instantaneous streamlines and contours of the pressure at two characteristic phases of the oscillation. The plots on the left show the computed results, those on the right show the analytical predictions of reference [6].

.

in principle, be performed by the macro-element-based mesh update used in all previous examples. For this purpose, we would have to combine the individual `FSIKirchhoffLoveBeamElements` into a single `GeomObject` whose geometric `Data` contains the positional `Data` of *all* nodes in the wall mesh. Using this representation, the change in the nodal positions of *any* node in the wall mesh potentially induces a change in the position of *all* fluid nodes.

To avoid this undesirable feature, `oomph-lib` provides an alternative mesh update strategy that allows node updates to be performed much more sparsely. The idea is illustrated in Fig. 16. Consider the coarse three-element mesh in

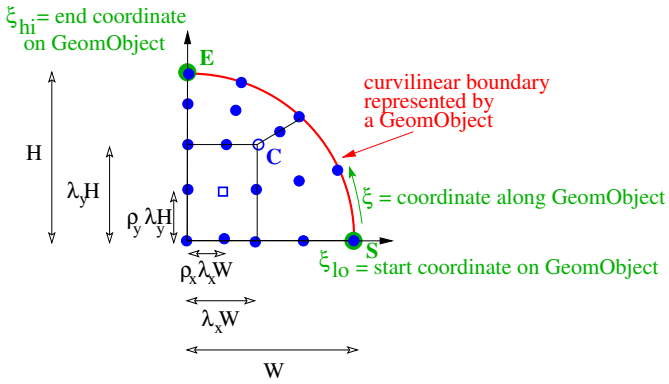

Fig. 16. Sketch illustrating the sparse node update procedures for a "quarter circle" mesh.

a "quarter circle domain" whose curved boundary is represented by one (or possibly more) `GeomObjects`. One strategy for distributing the nodes is to place the central node, "C", at fixed fractions, λ_x and λ_y, of the domain's width W and height H, respectively. This defines the boundary of the central element. Its constituent nodes can then be located at fixed fractions, ρ_x and ρ_y, of its width and height. Similarly, the nodes in the two elements adjacent to the curvilinear boundary may be placed on straight lines that connect the central element to reference points on the curvilinear boundary, identified by their intrinsic coordinate $\xi_{\rm ref}$ on the `GeomObject`. If the curvilinear boundary is represented by multiple `GeomObjects` (as in the fluid-structure-interaction problem), the reference points on the curvilinear boundary may be identified by a pointer to one of those `GeomObjects`, and the reference point's intrinsic coordinate, $\xi_{\rm ref}$, within it.

To update the nodal positions in response to changes in the domain boundary, each `Node` therefore requires (at most) two types of data: **(i)** Pointer(s) to the `GeomObject`(s) that affect its position, and **(ii)** a certain number of parameters such as $\lambda_x, \lambda_y, \rho_x, \rho_y$ and $\xi_{\rm ref}$, that identify reference

points on the GeomObjects, and the Node's relative position to these. Storage for this "node update data" is provided in the AlgebraicNode class, which is derived from the Node class.

Using this mesh-update strategy, the position of each fluid node in the "quarter circle domain" depends on no more than three FSIKirchhoffLoveBeamElements. As a result, the shape-derivative sub-matrix in the global Jacobian matrix is much sparser than that generated by the macro-element-based node update.

It is important to note that, as in the case of the macro-element-based node updates, oomph-lib only requires the specification of the "node update data" on a coarse initial mesh. Once created, the mesh may be refined with oomph-lib's mesh adaptation procedures which automatically determine the "node update data" for any newly created AlgebraicNodes, based on the data stored at the previously existing nodes.

The computation of the "shape derivatives" is again performed fully-automatically by a templated wrapper class, AlgebraicElement<ELEMENT>, which determines the geometric unknowns that affect the element's nodal positions by extracting the geometric Data from the GeomObjects that are stored in the "node update data" of its constituent AlgebraicNodes.

III.B Results

Fig. 17 shows a result from the numerical simulation of the fully-coupled fluid-structure-interaction problem. The computations were performed with

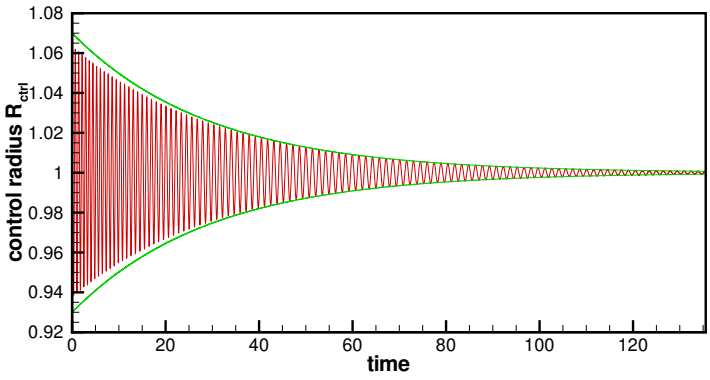

Fig. 17. Time history of the control radius, $R_{ctrl}(t)$ for the fully-coupled fluid-structure-interaction problem at $\alpha^2 = 200$. At large times, the amplitude of the oscillation, $\widehat{R_{\rm ctrl}}$, decays exponentially, i.e. $\widehat{R_{\rm ctrl}} \sim \exp(-\lambda t)$, as shown by the envelope.

the AlgebraicElement-version of the 2D Crozier-Raviart Navier-Stokes el-

ement used in the previous example. The simulations were started from an initial configuration in which the ring and the fluid are at rest. The oscillation was initiated by subjecting the ring to a short transient load perturbation of the form

$$\mathbf{f}_{\rm transient} = \begin{cases} \mathbf{0} & \text{for } t < 0 \\ p_{\cos} \cos(N\xi)\ \mathbf{N} & \text{for } 0 \le t \le 0.3 \\ \mathbf{0} & \text{for } t > 0.3, \end{cases} \tag{17}$$

where $\mathbf{N}$ is the unit normal vector on the ring. Fig. 17 shows a plot of the control radius $R_{\rm ctrl}$ (identified in the sketch in Fig. 13) as a function of time. The transient perturbation deforms the ring non-axisymmetrically, with a maximum amplitude of approximately 6% of its undeformed radius. Subsequently, the ring performs slowly decaying oscillations about its axisymmetric equilibrium state. Heil & Waters' theoretical analysis [6] demonstrates that the period of the oscillations is controlled by a dynamic balance between fluid inertia and the wall's elastic restoring forces, while viscous dissipation causes the oscillations to decay over a timescale that is much larger than the period of the oscillations. The frequency of the oscillations decreases slightly and ultimately approaches a constant value. At this stage, the system performs damped harmonic oscillations whose amplitude decays exponentially, as shown by the envelope in Fig. 17. The period and decay rate of the oscillation observed in the computations is in excellent agreement with Heil & Waters' theoretical predictions, obtained from a multiple-scales analysis of the problem.

8 Acknowledgements

MH gratefully acknowledges the financial support from the EPSRC for an Advanced Research Fellowship.

References

1. C. D. Bertram. Experimental studies of collapsible tubes. In T. J. Pedley and P. W. Carpenter, editors, *Flow in Collapsible Tubes and Past Other Highly Compliant Boundaries*, pages 51–65, Dordrecht, Netherlands, 2003. Kluwer.
2. J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.
3. M. Heil. Stokes flow in an elastic tube – a large-displacement fluid-structure interaction problem. *International Journal for Numerical Methods in Fluids*, 28:243–265, 1998.
4. M. Heil. 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.

5. M. Heil and O. E. Jensen. Flows in deformable tubes and channels – theoretical models and biological applications. In T. J. Pedley and P. W. Carpenter, editors, *Flow in Collapsible Tubes and Past Other Highly Compliant Boundaries*, pages 15–50, Dordrecht, Netherlands, 2003. Kluwer.
6. M. Heil and S.L. Waters. Transverse flows in rapidly oscillating, elastic cylindrical shells. *Journal of Fluid Mechanics*, 547:185–214, 2006.
7. HSL2004. A collection of Fortran codes for large scale scientific computation, 2004. http://www.cse.clrc.ac.uk/nag/hsl/hsl.shtml.
8. S. F. Kistler and L. E. Scriven. Coating flows. In J.R.A. Pearson and S.M. Richardson, editors, *Computational Analysis of Polymer Processing*. Applied Science Publishers, London, 1983.
9. X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
10. D. P. Mok and W. A. Wall. Partitioned analysis schemes for the transient interaction of incompressible flows and nonlinear flexible structures. In W. A. Wall, K.-U. Bletzinger, and K. Schweizerhof, editors, *Trends in Computational Structural Mechanics*, Barcelona, Spain, 2001. CIMNE, Barcelona.
11. W. Soedel. *Vibrations of shells and plates*. Marcel Dekker, New York, 1993.
12. O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33:1331–1364, 1992.