Computer Physics Communications 192 (2015) 205–219

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Nektar++: An open-source spectral/*hp* element framework☆

C.D. Cantwell[a,*], D. Moxey[a], A. Comerford[a], A. Bolis[a], G. Rocco[a], G. Mengaldo[a], D. De Grazia[a], S. Yakovlev[b], J.-E. Lombard[a], D. Ekelschot[a], B. Jordi[a], H. Xu[a], Y. Mohamied[a], C. Eskilsson[c], B. Nelson[b], P. Vos[a], C. Biotto[a], R.M. Kirby[b], S.J. Sherwin[a]

[a] *Department of Aeronautics, Imperial College London, London, UK*

[b] *School of Computing and Scientific Computing and Imaging (SCI) Institute, University of Utah, Salt Lake City, UT, USA*

[c] *Department of Shipping and Marine Technology, Chalmers University of Technology, Gothenburg, Sweden*

ARTICLE INFO

Article history:
Received 22 September 2014
Received in revised form
23 January 2015
Accepted 13 February 2015
Available online 24 February 2015

Keywords:
High-order finite elements
Spectral/*hp* elements
Continuous Galerkin method
Discontinuous Galerkin method
FEM

ABSTRACT

Nektar++ is an open-source software framework designed to support the development of high-performance scalable solvers for partial differential equations using the spectral/*hp* element method. High-order methods are gaining prominence in several engineering and biomedical applications due to their improved accuracy over low-order techniques at reduced computational cost for a given number of degrees of freedom. However, their proliferation is often limited by their complexity, which makes these methods challenging to implement and use. *Nektar++* is an initiative to overcome this limitation by encapsulating the mathematical complexities of the underlying method within an efficient C++ framework, making the techniques more accessible to the broader scientific and industrial communities. The software supports a variety of discretisation techniques and implementation strategies, supporting methods research as well as application-focused computation, and the multi-layered structure of the framework allows the user to embrace as much or as little of the complexity as they need. The libraries capture the mathematical constructs of spectral/*hp* element methods, while the associated collection of pre-written PDE solvers provides out-of-the-box application-level functionality and a template for users who wish to develop solutions for addressing questions in their own scientific domains.

Program summary

Program title: Nektar++

Catalogue identifier: AEVV_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEVV_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: MIT

No. of lines in distributed program, including test data, etc.: 1052456

No. of bytes in distributed program, including test data, etc.: 42851367

Distribution format: tar.gz

Programming language: C++.

Computer: Any PC workstation or cluster.

Operating system: Linux/UNIX, OS X, Microsoft Windows.

RAM: 512 MB

Classification: 12.

External routines: Boost, FFTW, MPI, BLAS, LAPACK and METIS (www.cs.umn.edu)

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: c.cantwell@imperial.ac.uk (C.D. Cantwell).

http://dx.doi.org/10.1016/j.cpc.2015.02.008

Nature of problem:
The Nektar++ framework is designed to enable the discretisation and solution of time-independent or time-dependent partial differential equations.

Solution method:
Spectral/*hp* element method

Running time:
The tests provided take a few minutes to run. Runtime in general depends on mesh size and total integration time.

1. Introduction

Finite element methods (FEM) are commonplace among a wide range of engineering and biomedical disciplines for the solution of partial differential equations (PDEs) on complex geometries. However, low-order formulations often struggle to capture certain complex solution characteristics without the use of excessive mesh refinement due to numerical dissipation. In contrast, spectral techniques offer improved numerical characteristics, but are typically restricted to relatively simple regular domains.

High-order finite element methods, such as the traditional spectral element method [1], the p-type method [2] and the more recent spectral/*hp* element method [3], exhibit the convergence properties of spectral methods while retaining the geometric flexibility of traditional linear FEM. They potentially offer greater efficiency on modern CPU architectures as well as more exotic platforms such as many-core general-purpose graphics processing units (GPGPUs). The data structures which arise from using high-order methods are more compact and localised than their linear finite element counterparts, for a fixed number of degrees of freedom, providing increased cache coherency and reduced memory accesses, which is increasingly the primary bottleneck of modern computer systems.

The methods have had greatest prominence in the structural mechanics community and subsequently the academic fluid dynamics community. They are also showing promise in other areas of engineering, biomedical and environmental research. The most common concern cited with respect to using high-order finite element techniques outside of academia is the implementational complexity, stemming from the complex data structures, necessary to produce a computationally efficient implementation. This is a considerable hurdle which has limited their widespread uptake in many application domains and industries.

Nektar++ is a cross-platform spectral/*hp* element framework which aims to make high-order finite element methods accessible to the broader community. This is achieved by providing a structured hierarchy of C++ components, encapsulating the complexities of these methods, which can be readily applied to a range of application areas. These components are distributed in the form of cross-platform software libraries, enabling rapid development of solvers for use in a wide variety of computing environments. The code accommodates both small research problems, suitable for desktop computers, and large-scale industrial simulations, requiring modern HPC infrastructure, where there is a need to maintain efficiency and scalability up to many thousands of processor cores.

A number of software packages already exist for fluid dynamics which implement high-order finite element methods, although these packages are typically targeted at a specific domain or provide limited high-order capabilities as an extension. The *Nektar flow solver* is the predecessor to Nektar++ and implements the spectral/*hp* element method for solving the incompressible and compressible Navier–Stokes equations in both 2D and 3D. While it is widely used and the implementation is computationally efficient on small parallel problems, achieving scaling on large HPC clusters is challenging. Semtex [4] implements the 2D spectral element method coupled with a Fourier expansion in the third direction. The implementation is highly efficient, but can only be parallelised through Fourier-mode decomposition. Nek5000 [5] is a 3D spectral element code, based on hexahedral elements, which has been used for massively parallel simulations up to 300,000 cores. Hermes [6] implements hp-FEM for two-dimensional problems and has been used in a number of application areas. Limited high-order finite element capabilities are also included in a number of general purpose PDE frameworks including the DUNE project [7] and deal.II [8]. A number of codes also implement high-order finite element methods on GPGPUs including nudg++, which implements a nodal discontinuous Galerkin scheme [9], and PyFR [10], which supports a range of flux reconstruction techniques.

Nektar++ provides a single codebase with the following key features:

- Arbitrary-order spectral/*hp* element discretisations in one, two and three dimensions;
- Support for variable polynomial order in space and heterogeneous polynomial order within two- and three-dimensional elements;
- High-order representation of the geometry;
- Continuous Galerkin, discontinuous Galerkin and hybridised discontinuous Galerkin projections;
- Support for a Fourier extension of the spectral element mesh;
- Support for a range of linear solvers and preconditioners;
- Multiple implementation strategies for achieving linear algebra performance on a range of platforms;
- Efficient parallel communication using MPI showing strong scaling up to 2048-cores on Archer, the UK national HPC system;
- A range of time integration schemes implemented using generalised linear methods; and
- Cross-platform support for Linux, OS X and Windows operating systems.

In addition to the core functionality listed above, Nektar++ includes a number of solvers covering a range of application areas. A range of pre-processing and post-processing utilities are also included with support for popular mesh and visualisation formats, and an extensive test suite ensures the robustness of the core functionality.

The purpose of this paper is to expose the novel aspects of the code and document the structure of the library. We illustrate its use through a broad range of example applications which should enable other scientists to build on and extend *Nektar++* for use in their own applications. We begin by outlining the mathematical formulation of the spectral/*hp* element method and discuss the implementation of the framework. We then present a number

of example applications and conclude with a discussion of our development strategy and future direction.

2. Methods

In this section we introduce the mathematical foundations of the spectral/*hp* element methods implemented in Nektar++. A more detailed overview of the mathematical theory can be found in [3] and is beyond the scope of this paper. *Nektar++* supports both continuous and discontinuous discretisations in one, two and three dimensions, but the majority of the formulation which follows is generic to all cases, except where stated.

We consider the numerical solution of partial differential equations (PDEs) of the form $\mathcal{L}u = 0$ on a domain Ω, which may be geometrically complex, for some solution u. Practically, Ω takes the form of a d-dimensional finite element mesh ($d \leq 3$), consisting of elements Ω_e such that $\Omega = \bigcup \Omega_e$ and $\Omega_{e_1} \cap \Omega_{e_2} = \partial\Omega_{e_1 e_2}$ is the empty set or an interface of dimension $\bar{d} < d$. The domain may be embedded in a space of equal or higher dimension, $\hat{d} \geq d$. We will solve the PDE problem in the weak sense and, in general, $u|_{\Omega_e}$ must be smooth and have at least a first-order derivative; we therefore require that $u|_{\Omega_e}$ is in the Sobolev space $W^{1,2}(\Omega_e)$. For a continuous discretisation, we additionally impose continuity along element interfaces.

Our problem is cast in the weak form and, for illustrative purposes, we assume that it can be expressed as follows: *find* $u \in H^1(\Omega)$ *such that*

$$a(u, v) = l(v) \quad \forall v \in H^1(\Omega),$$

where $a(\cdot,\cdot)$ is a symmetric bilinear form, $l(\cdot)$ is a linear form, and $H^1(\Omega)$ is formally defined as

$$H^1(\Omega) := W^{1,2}(\Omega) = \{v \in L^2(\Omega) \mid D^\alpha u \in L^2(\Omega)\ \forall\, |\alpha| \leq 1\}.$$

To solve this problem numerically, we consider solutions in a finite-dimensional subspace $V_N \subset H^1(\Omega)$ and cast our problem as: *find* $u^\delta \in V_N$ *such that*

$$a(u^\delta, v^\delta) = l(v^\delta) \tag{1}$$

$\forall v^\delta \in V_N$, augmented with appropriate boundary conditions. For a projection which enforces continuity across elements, we impose the additional constraint that $V_N \subset C^0$. We assume the solution can be represented as $u^\delta(\mathbf{x}) = \sum_n \hat{u}_n \Phi_n(\mathbf{x})$, a weighted sum of N trial functions $\Phi_n(\mathbf{x})$ defined on Ω and our problem becomes that of finding the coefficients $\hat{u}_n$. The approximation u^δ does not directly give rise to unique choices for the coefficients $\hat{u}_n$. To achieve this we place a restriction on $R = \mathcal{L}u^\delta$ that its L^2 inner product, with respect to the test functions $\Psi_n(\mathbf{x})$, is zero. For a Galerkin projection we choose the test functions to be the same as the trial functions, that is $\Psi_n = \Phi_n$.

To construct the global basis Φ_n we first consider the contributions from each element in the domain. Each Ω_e is mapped from a standard reference space $\mathcal{E} \subset [-1, 1]^d$ by a parametric mapping $\chi_e : \Omega_e \to \mathcal{E}$ given by $\mathbf{x} = \chi_e(\boldsymbol{\xi})$, where $\mathcal{E}$ is one of the supported region shapes, and $\boldsymbol{\xi}$ are d-dimensional coordinates representing positions in a reference element, distinguishing them from $\mathbf{x}$ which are $\hat{d}$-dimensional coordinates in the Cartesian coordinate space. The mapping need not necessarily exhibit a constant Jacobian, supporting deformed and curved elements through an isoparametric mapping. The reference spaces implemented in *Nektar++* are listed in Table 1. On triangular, tetrahedral, prismatic and pyramid elements, one or more of the coordinate directions are collapsed creating singular vertices within these regions [11,12]. Operations, such as calculating derivatives, map the coordinate system to a non-collapsed coordinate system through a Duffy transformation [13] – for example, $\omega_{\mathcal{T}} : \mathcal{T} \to \mathcal{Q}$ maps the triangular region $\mathcal{T}$ to the quadrilateral region $\mathcal{Q}$ – to allow these methods to be well-defined.

Table 1
List of supported elemental reference regions.

Name	Class	Domain definition
Segment	`StdSeg`	$\mathcal{S} = \{\xi_1 \in [-1, 1]\}$
Quadrilateral	`StdQuad`	$\mathcal{Q} = \{\boldsymbol{\xi} \in [-1, 1]^2\}$
Triangle	`StdTri`	$\mathcal{T} = \{\boldsymbol{\xi} \in [-1, 1]^2 \mid \xi_1 + \xi_2 \leq 0\}$
Hexahedron	`StdHex`	$\mathcal{H} = \{\boldsymbol{\xi} \in [-1, 1]^3\}$
Prism	`StdPrism`	$\mathcal{R} = \{\boldsymbol{\xi} \in [-1, 1]^3 \mid \xi_1 \leq 1, \xi_2 + \xi_3 \leq 0\}$
Pyramid	`StdPyr`	$\mathcal{P} = \{\boldsymbol{\xi} \in [-1, 1]^3 \mid \xi_1 + \xi_3 \leq 0, \xi_2 + \xi_3 \leq 0\}$
Tetrahedron	`StdTet`	$\mathcal{A} = \{\boldsymbol{\xi} \in [-1, 1]^3 \mid \xi_1 + \xi_2 + \xi_3 \leq -1\}$

A local polynomial basis is constructed on each reference element with which to represent solutions. A one-dimensional order-P basis is a set of polynomials $\phi_p(\xi)$, $0 \leq p \leq P$, defined on the reference segment, $\mathcal{S}$. The choice of basis is usually made based on its mathematical or numerical properties and may be modal or nodal in nature. For two- and three-dimensional regions, a tensorial basis may be used, where the polynomial space is constructed as the tensor-product of one-dimensional bases on segments, quadrilaterals or hexahedral regions. In particular, a common choice is to use a modified hierarchical Jacobi polynomial basis, given as a function of one variable by

$$\phi_p(\xi) = \begin{cases} \dfrac{1-\xi}{2} & p = 0, \\ \left(\dfrac{1-\xi}{2}\right)\left(\dfrac{1+\xi}{2}\right) P_{p-1}^{1,1}(\xi) & 0 < p < P, \\ \dfrac{1+\xi}{2} & p = P \end{cases}$$

which supports boundary–interior decomposition and therefore improves numerical efficiency when solving the globally assembled system. Equivalently, ϕ_p could be defined by the Lagrange polynomials through the Gauss–Lobatto–Legendre quadrature points which would lead to a traditional spectral element method.

On a physical element Ω_e the discrete approximation u^δ to the solution u may be expressed as

$$u^\delta(\mathbf{x}) = \sum_n \hat{u}_n \phi_n\left([\chi_e]^{-1}(\mathbf{x})\right)$$

where $\hat{u}_n$ are the coefficients from Eq. (1), obtained through projecting u onto the discrete space. Therefore, we restrict our solution space to

$$V := \left\{u \in H^1(\Omega) \mid u|_{\Omega_e} \in \mathcal{P}_P(\Omega_e)\right\},$$

where $\mathcal{P}_P(\Omega_e)$ is the space of order-P polynomials on Ω_e.

Elemental contributions to the solution may be assembled to form a global solution through an assembly operator. In a continuous Galerkin setting, the assembly operator sums contributions from neighbouring elements to enforce the C^0-continuity requirement. In a discontinuous Galerkin formulation, such mappings transfer flux values from the element interfaces into the global solution vector.

3. Implementation

In this section, we provide an architectural overview of *Nektar++*, sufficient to enable other scientists to leverage the framework to develop application-specific PDE solvers using high-order methods. In doing so we summarise the salient features of the code and how the mathematical constructions from Section 2 are represented in the library.

In designing *Nektar++*, strong emphasis has been placed on ensuring the code structure strongly mirrors the mathematical

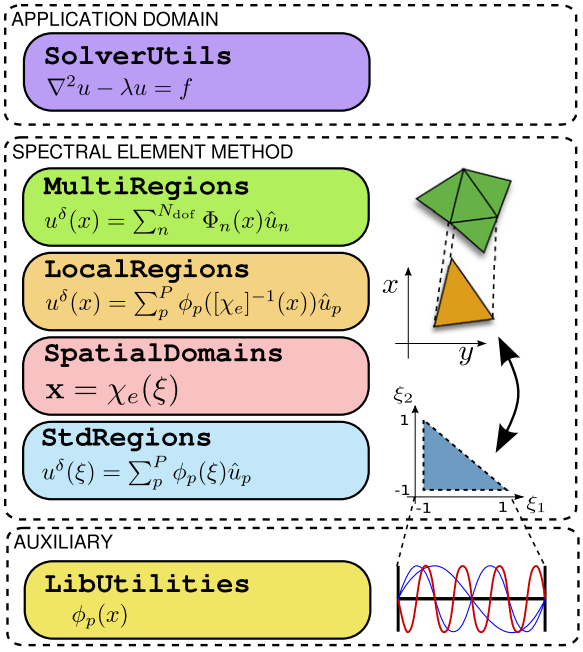

Fig. 1. Nektar++ architecture diagram. Shows the relationship between the individual libraries in the framework and the mathematical constructs to which they relate.

formulation. The implementation is partitioned into a set of six libraries, each of which encapsulates one aspect of the above construction. This division maintains separation between mathematically distinct parts of the formulation, provides an intuitive means to organise the code and enables developers to easily interject at the level most appropriate for their needs. The overall architecture of *Nektar++* is illustrated in Fig. 1. In summary, the six libraries cover the following aspects of the mathematical formulation given in Section 2:

- `LibUtilites`: elemental basis functions ψ_p, point distributions ξ_j and basic building blocks such as I/O handling and mesh partitioning,
- `StdRegions`: reference regions $\mathcal{E}$ along with integration, differentiation and other core operations on these regions,
- `SpatialDomains`: the mappings χ_e, the geometric factors $\frac{\partial \chi}{\partial \xi}$, and Jacobians of the mappings,
- `LocalRegions`: physical regions in the domain, composing a reference region $\mathcal{E}$ with a map χ_e, extensions of core operations onto these regions,
- `MultiRegions`: list of physical regions comprising Ω, global assembly maps which may optionally enforce continuity, construction and solution of global linear systems, extension of local core operations to domain-level operations,
- `SolverUtils`: building blocks for developing complete solvers.

We now outline in detail the different aspects of the implementation, the division of functionality across the libraries, the relationships between them and how together they construct the Helmholtz operator $\mathbf{H}$ which forms an essential component in solving many elliptic PDE problems.

3.1. Input format

Nektar++ uses one or more XML-structured text files as input for simulations. These describe both the discretisation of the domain (the finite element mesh) and the specification of the PDE problem in terms of the necessary boundary conditions, variables and parameters required to solve a specific problem. A large number of example XML input files are provided with the source code in the `Examples` and `Tests` subdirectories of the library demonstration programs and solvers. The input format and comprehensive list of the available options are documented in full in the user guide (see Appendix A.14).

The Nektar++ mesh specification format is of a hierarchical type in which one-dimensional edges are defined in terms of the vertices they connect, two-dimensional faces are defined in terms of the bounding edges and three-dimensional elements in terms of the bounding faces. A composite is defined as a collection of mesh entities which have a common shape, but need not necessarily be connected. Composites are used for specifying the extent of the domain and for defining boundary regions on which constraints can be imposed (see Section 3.7). Meshes are typically generated by third-party mesh generation packages and the necessary XML specification is generated by the *Nektar++* `MeshConvert` utility.

3.2. `LibUtilities` library

The primary function of the `LibUtilities` library is to provide the fundamental mathematical and software constructs necessary to implement the spectral/*hp* element method. In particular, this includes the description of Q-point coordinate distributions, ξ_j, on the standard segment $\mathcal{S}$ and the construction of suitable polynomial bases, ψ_i, to span $\mathcal{P}_P(\mathcal{S})$. Each type of basis is encapsulated in a class which, when augmented with a point distribution, provides the $P \times Q$ basis matrix $\mathbf{B}_{\mathcal{S}}[i][j] = \psi_i(\xi_j)$, the $P \times Q$ basis derivative matrix $\mathbf{DB}_{\mathcal{S}}[i][j] = \frac{\partial \psi_i}{\partial \xi}|_{\xi_j}$, the $Q \times Q$ diagonal quadrature weight matrix $\mathbf{W}_{\mathcal{S}}[i][i] = w_i$ and the Q coordinates ξ_i of the points on $\mathcal{S}$. As well as providing these basic mathematical objects, the `LibUtilities` library also provides a range of other generic functionality. In particular, parallel communication routines, Fourier transforms, linear algebra containers and design pattern implementations are all incorporated into this part of the framework.

3.3. `StdRegions` library

Reference regions $\mathcal{E} \subset [-1, 1]^d$ provide core element-level operations and are implemented in the `StdRegions` library. Classes are defined for each of the reference regions, as given in Table 1, and class inheritance is used to share common functionality. An example of the class hierarchy for two-dimensional elements is shown in Fig. 2(a). We assume each region is equipped with a basis ϕ_n extended from one of the basis functions ψ_p. In two- and three-dimensional elements, the basis is often constructed as a tensor product of one-dimensional hierarchical bases; for example, $\phi_{n(p_1,p_2)}(\xi_1, \xi_2) = \psi_{p_1}(\xi_1) \otimes \psi_{p_2}(\xi_2)$ for a quadrilateral region or $\phi_{n(p_1,p_2)}(\xi_1, \xi_2) = \psi_{p_1} \otimes \psi_{p_1,p_2}$ for a triangular region. However, nodal and non-tensorial bases are also supported. In a similar way, coordinates in two- and three-dimensional elements are given by $\xi_{q=q(\mathbf{i})}$. Expansion orders may be different for each of the ψ_{p_i} bases, although constraints are imposed on simplex regions to ensure a complete polynomial space. The core operators defined on the reference element are then:

- `BwdTrans`: $\hat{\mathbf{u}} \mapsto u(\boldsymbol{\xi}) = \sum_n \hat{u}_n \phi_n(\boldsymbol{\xi})$,

  which evaluates the solution represented by $\hat{u}$, on the basis ϕ_n, at the quadrature points $\boldsymbol{\xi}$. This operation requires the basis matrix $\mathbf{B}$ and therefore evaluates the result as $\mathbf{u} = \mathbf{B}\hat{\mathbf{u}}$.
- `IProductWRTBase`: $f \mapsto \hat{f}_n = \int_{\mathcal{E}} f(\xi)\phi_n(\xi)\, \mathrm{d}\boldsymbol{\xi}$,

  which computes the inner product of a function with respect to the basis. The discrete approximation of integration, Gaussian quadrature, leads to $\hat{\mathbf{f}}[i] \approx \sum_q w_q f(\xi_q)\phi_i(\xi_q)$ which can be

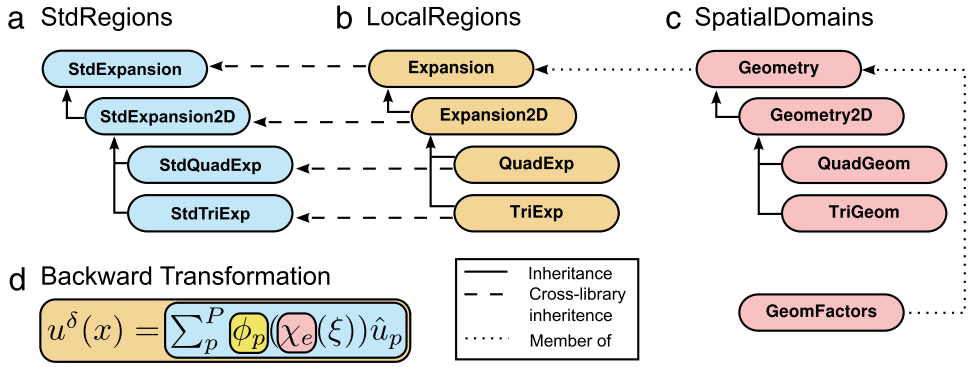

Fig. 2. Example of class inheritance structure for two-dimensional regions. Each `LocalRegions` class (b) inherits base functionality from the corresponding `StdRegions` class (a). Inheritance is used to minimise code duplication. The `Expansion` class contains member pointers to `Geometry` and `GeomFactors` objects (c). Operations, such as a transform from coefficient to physical space (d), on a physical element are constructed as a composition of functionality from the different libraries.

expressed in matrix form as $\hat{\mathbf{f}} = \mathbf{B}^\top\mathbf{W}\mathbf{f}$, where $\mathbf{W}$ is a diagonal matrix containing the integration weights w_q.

- `PhysDeriv`: $u \mapsto \frac{\partial u}{\partial \xi_i}$,

  which computes the derivative of u with respect to the d coordinates of the element, with matrix representation $\mathbf{u}' = \mathbf{D}\mathbf{u}$.

These operators can be combined to produce more complex operators such as the mass matrix,

$$\mathbf{M} = \mathbf{B}^\top\mathbf{W}\mathbf{B}.$$

With this we can project a solution onto the discrete space using the `FwdTrans` operation. This requires solving the projection problem $u^\delta(\boldsymbol{\xi}) = f(\boldsymbol{\xi})$. In the weak sense, this has the form

$$\int_\varepsilon v^\delta(\boldsymbol{\xi})u^\delta(\boldsymbol{\xi})\,\mathrm{d}\boldsymbol{\xi} = \int_\varepsilon v^\delta(\boldsymbol{\xi})f(\boldsymbol{\xi})\,\mathrm{d}\boldsymbol{\xi}$$

and is equivalent to solving for $\hat{\mathbf{u}}$ the linear system

$$\mathbf{M}\hat{\mathbf{u}} = \mathbf{B}^\top\mathbf{W}\mathbf{f}.$$

The `StdRegions` classes also describe the mapping of basis functions to the vertices, edges and faces of the element, which are necessary to assemble elemental contributions and construct a global system for the domain.

3.4. *SpatialDomains library*

The classes defined in `SpatialDomains` fall into three main hierarchies which together describe the geometric information needed to represent the problem domain and the elemental entities which comprise it. `Geometry` classes capture the physical geometry of an individual element Ω_e. There are separate classes for each elemental region and the class hierarchy follows a parallel structure to the `StdRegions` classes, shown in Fig. 2(c). The `GeomFactors` class, instantiated by each geometry object, defines the parametric mapping χ_e between the physical geometry of the element Ω_e and the corresponding standard region. This mapping is implemented in a generic manner and does not require dimension-dependent subclasses.

`MeshGraph` classes read the mesh definition from the input file and construct the domain Ω, instantiating a corresponding `Geometry` object for each Ω_e. The derived classes are again dimension-dependent. Finally, the `BoundaryConditions` class manages the association of specific mathematical constraints to each boundary of the domain for a given variable.

3.5. *LocalRegions library*

Physical regions, shown in Fig. 2(b), are an extension of a reference element augmented with geometric information and a mapping between the two regions. As such, the physical element types implemented in the `LocalRegions` library inherit their `StdRegions` counterparts and override core operations, such as integration and differentiation, to incorporate the geometric information. For example, the inner product operation becomes

$$f(x) \mapsto \hat{f}_i = \int_{\Omega_e} f(x)\phi_i(x)\,\mathrm{d}x$$

which, in discrete form is evaluated as $\hat{\mathbf{f}}[i] \approx \sum_q J w_q f(\xi_q)\phi_i(\xi_q)$. Here we have incorporated J, the determinant of the Jacobian of χ_e. Similarly, for differentiation, the chain rule gives rise to $u \mapsto \frac{\partial u}{\partial x_i} = \sum_{j=1}^n \frac{\partial \xi_j}{\partial x_i}\frac{\partial u}{\partial \xi_j}$. Both J and the terms $\frac{\partial \xi_j}{\partial x_i}$ are provided by the `GeomFactors` class.

To identify the relationship between the different libraries and their respective contributions to the above core operations, Fig. 2(d) illustrates how those libraries examined so far contribute to the implementation of the backward transform on a given element Ω_e.

3.6. *MultiRegions library*

So far, operations have only been defined on the physical elemental regions; to define operators on the entire domain the elemental regions are assembled. The `MultiRegions` library encapsulates the global representation of a solution across a domain comprising of one or more elemental regions. While the same type of basis functions must be used throughout the domain, the order of the polynomials for each expansion may vary between elements. Assembly is the process of summing local elemental contributions to construct a global representation of the solution on the domain. The information to construct this mapping is derived from the elemental mappings of modes to vertices, edges and faces of the element. Mathematically, this operation can be represented as a highly sparse matrix, but it is practically implemented as an injective map in the `AssemblyMap` classes. Different maps are used for different projections; in particular, the `AssemblyMapCG` class supports the exchange of neighbouring contributions in continuous Galerkin projections, while the `AssemblyMapDG` supports the mapping of elemental data onto the trace space in the discontinuous Galerkin method.

The resulting assembly of the elemental matrix contributions leads to a global linear system in the `GlobalLinSys` classes. This

may be solved using a variety of linear algebra techniques including direct Cholesky factorisation and iterative preconditioned conjugate gradient. Substructuring (multi-level static condensation) allows for a more efficient solution of the matrix system. As well as a traditional Jacobi preconditioner, specialist `Preconditioner` classes tailored to high-order methods are also available [14]. These include a coarse-space preconditioner, block preconditioner and low-energy preconditioner [15]. Performance of the conjugate gradient solver is dependent on both the efficiency of the matrix–vector operation and inter-process communication. The rich parameter space of a high-order elemental discretisation may be leveraged by providing multiple implementations of the core operators, each of which perform efficiently across a subset of the parameter space on different hardware architectures. This has been extensively explored in the literature [16–18]. The gather–scatter operation necessary for evaluating operations in parallel is implemented in the *gslib* library [19], developed within Nek5000. Finally, a PetSc interface is available which provides access to a range of additional solvers.

3.7. Boundary conditions

Boundary-value problems require the imposition of constraints at the boundaries of the domain. Although these conditions are frequently Dirichlet, Neumann or Robin constraints, depending upon the application area, other more complex conditions can be implemented by the user if needed. The specification of boundary conditions in the input file is generic to support this.

- Boundary regions are defined using one or more mesh composites. For example, the following XML describes two regions constructed from three composites on which different constraints are to be imposed:

```
<B ID="0"> C[1,3] </B>
<B ID="1"> C[2]   </B>
```

- The conditions to be imposed on each boundary region for each variable are described using XML element tags to indicate the underlying type of condition:
  - `D`: Dirichlet;
  - `N`: Neumann;
  - `R`: Robin.

  For example, the following excerpt defines an in-flow boundary using a high-order Neumann boundary condition on the pressure:

```
<REGION REF="0">
    <D VAR="u" VALUE="0"/>
    <D VAR="v" VALUE="0"/>
    <D VAR="w" VALUE="y*(1-y)"/>
    <N VAR="p" USERDEFINEDTYPE="H" VALUE="0"/>
</REGION>
```

  The `USERDEFINEDTYPE` attribute specifies the user-implemented condition to be used. The `REF` attribute corresponds to the `ID` of the boundary region.

The list of boundary regions and their constraints is managed by the `BoundaryConditions` data structure in the `SpatialDomains` library. The enforcement of the boundary conditions on the solution is implemented at the `MultiRegions` level of the code (and above) during the construction and use of domain-wide operators. User-defined boundary conditions are implemented in specific solvers. For example both high-order Neumann boundary conditions and a radiation boundary condition are supported by the incompressible Navier–Stokes solver.

3.8. `SolverUtils` library

The `SolverUtils` library provides the top-level building blocks for constructing application-specific solvers. This includes core functionality, such as IO, time-stepping [20] and common initialisation routines, useful in quickly constructing a solver using the *Nektar++* framework. It contains a library of application-independent modules for implementing diffusion and advection terms as well as a number of `Driver` modules which implement general high-level algorithms, such as an Arnoldi method for performing various stability analyses [21].

3.9. Solvers

Nektar++ includes a number of pre-written solvers for some common PDEs, developed for our own research. Some examples, outlined in the next section, include incompressible and compressible Navier–Stokes equations, the cardiac electrophysiology monodomain equation, shallow water equations and a solver for advection–diffusion–reaction problems. The modular nature of the code, combined with the mathematically motivated class hierarchy allows the code to be adapted and extended to rapidly address a range of application and numerical questions.

To illustrate the use of the framework, we first consider the solution of the Helmholtz equation, since this is a fundamental operation in the solution of many elliptic partial differential equations. The problem is described by the following PDE and associated boundary conditions:

$$\begin{aligned} &\nabla^2 u - \lambda u + f = 0 && \text{on } \Omega, \\ &u = g_d && \text{on } \partial\Omega, \\ &\frac{\partial u}{\partial x} = g_n && \text{on } \partial\Omega. \end{aligned}$$

We put this into the weak form and after integration by parts, this gives,

$$\int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} + \lambda \int_\Omega uv \, d\mathbf{x} = \int_\Omega fv \, d\mathbf{x} + \int_{\partial\Omega} v \nabla u \cdot \mathbf{n} \, d\mathbf{x}. \quad (2)$$

Approximating u and v with their finite-dimensional counterparts and substituting into Eq. (2) we obtain

$$\begin{aligned} &\sum_m \sum_n \hat{u}_n \hat{v}_m \int_\Omega \nabla \Phi_m \cdot \nabla \Phi_n + \lambda \sum_m \sum_n \hat{u}_n \hat{v}_m \int_\Omega \Phi_n \cdot \Phi_n \\ &\quad = \sum_n \hat{v}_n \int_\Omega f \cdot \Phi_n + \sum_n \sum_m \hat{u}_n \hat{v}_m \int_{\partial\Omega} \Phi_n \cdot \nabla \Phi_n, \end{aligned}$$

which can be expressed in matrix form as

$$\hat{\mathbf{v}}^\top (\mathbf{DB})^\top \mathbf{W} (\mathbf{DB}) \hat{\mathbf{u}} + \lambda \hat{\mathbf{v}}^\top \mathbf{M} \hat{\mathbf{u}} = \hat{\mathbf{v}}^\top \mathbf{B}^\top \mathbf{W} \mathbf{f}.$$

The matrix $\mathbf{H} = (\mathbf{DB})^\top \mathbf{W} (\mathbf{DB}) + \lambda \mathbf{M}$ is the Helmholtz matrix, and the system $\mathbf{H}\hat{\mathbf{u}} = \hat{\mathbf{f}}$, where $\hat{\mathbf{f}} = \mathbf{B}^\top \mathbf{W} \mathbf{f}$ is the projection of f onto V_N, is then solved for $\hat{\mathbf{u}}$.

The above is implemented in *Nektar++* through the `HelmSolve` function, which takes physical values of the forcing function f and produces the solution coefficients u, as

```
field->HelmSolve(forcing->GetPhys(),
                 field->UpdateCoeffs(),
                 NullFlagList,
                 factors);
```

where `factors` is a data-structure which allows us to prescribe the value of λ.

3.10. Implementing solvers using Nektar++

To conclude this section, we outline how one can construct a time-dependent solver for the unsteady diffusion problem using

the *Nektar++* framework, solving

$$\frac{\partial u}{\partial t} = \nabla^2 u \qquad \text{on } \Omega$$
$$u = g_D \qquad \text{on } \partial\Omega.$$

This can be implemented with the following key steps as outlined below. Only the key statements are shown to illustrate the use of the library. Full source code for this example is included in Appendix A.12.

- Create a `SessionReader` object to load and parse the input files. This provides access for the rest of the library to the XML input files supplied by the user. It also initiates MPI communication if needed.

```
session = LibUtilities::SessionReader
            ::CreateInstance(argc, argv);
```

- Create `MeshGraph` and field objects to generate the hierarchical mesh entity data structures and allocate storage for the global solution. The mesh provides access to the geometric information about each element and the connectivity of those elements to form the domain. In contrast, the field object represents a solution on the domain and provides the finite element operators.

```
var   = session->GetVariable(0);
mesh  = SpatialDomains::MeshGraph::Read(session);
field = MemoryManager<MultiRegions::ContField2D>
            ::AllocateSharedPtr(session, mesh, var);
```

- Get the coordinates of the quadrature points on all elements and evaluate the initial condition

```
field->GetCoords(x0,x1,x2);
icond->Evaluate(x0, x1, x2, 0.0, field->UpdatePhys());
```

- Perform backward Euler time integration of the initial condition for the number of steps specified in the session file, where `epsilon` is the coefficient of diffusion.

```
for (int n = 0; n < nSteps; ++n)
{
    Vmath::Smul(nq, -1.0/delta_t/epsilon,
                    field->GetPhys(),    1,
                    field->UpdatePhys(), 1);

    field->HelmSolve(field->GetPhys(),
                     field->UpdateCoeffs(),
                     NullFlagList,
                     factors);

    field->BwdTrans (field->GetCoeffs(),
                     field->UpdatePhys());
}
```

- Write out the solution to a file which can be post-processed and visualised.

```
fld->Write(outFile, FieldDef, FieldData);
```

  Here, `fld` is a *Nektar++* field format I/O object.

A second example is provided in the supplementary material Appendix A.13 which elicits the use of the time-integration framework to support more general (implicit) methods.

4. Applications

In this section we illustrate, through the use of the pre-written solvers, key aspects of the *Nektar++* framework through a number of example scientific problems spanning a broad range of application areas. The source files used to generate the figures in this section are available in the supplementary material (apart from Fig. 3, due to commercial considerations). Although these problems primarily relate to the modelling of external and internal flow phenomena, the framework is not limited to this domain and some examples extend into broader areas of biomedical engineering.

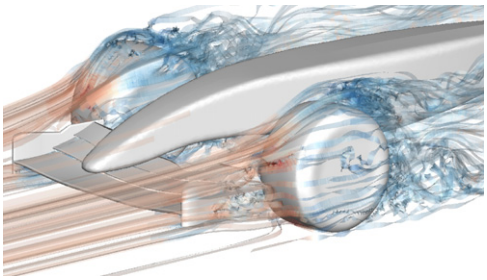

Fig. 3. Flow past a front section of a Formula 1 racing car at a Reynolds number of 2.2×10^5. Streamlines show the flow trajectory and are coloured by pressure. The simulation has 13 million degrees of freedom at polynomial order $P = 3$.

4.1. External aerodynamics

One of the most challenging problems in next-generation aerodynamics is capturing highly resolved transient flow past bluff bodies using Direct Numerical Simulation (DNS). For example, understanding and controlling the behaviour of vortices generated by the front-wing section of a racing car (see Fig. 3) is critical in ensuring the stability and traction of the vehicle. Apart from the computational complexity of the simulation, a number of mesh generation challenges need to be overcome to accurately capture the flow dynamics. These include high-order curvilinear meshing of the CAD geometry and the generation of sufficiently fine elements adjacent to the vehicle surfaces to resolve the thin boundary layers and accurately predict separation of the flow.

Using the `IncNavierStokesSolver`, flow is modelled under the incompressible Navier–Stokes equations,

$$\frac{\partial u}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla p + \nu\nabla^2 u, \tag{3a}$$
$$\nabla\cdot\mathbf{u} = 0, \tag{3b}$$
$$\left(\mathbf{u}, \frac{\partial p}{\partial n}\right)\Big|_{\partial\Omega_w} = \mathbf{0}, \tag{3c}$$
$$\left(\frac{\partial \mathbf{u}}{\partial n}, p\right)\Big|_{\partial\Omega_o} = \mathbf{0}, \tag{3d}$$
$$(\mathbf{u}, p)\,|_{\partial\Omega_i} = (\mathbf{f}, 0), \tag{3e}$$

where $\mathbf{u}$ is the velocity of the flow, p is the pressure and ν is the kinematic viscosity. No-slip boundary conditions are applied to the front wing, body and rotating wheels (Ω_w), a high-order outflow boundary condition [22] is imposed on the outlet (Ω_o) and a constant free-stream velocity f is applied on the inlet and far-field boundaries of the domain (Ω_i). Due to the high Reynolds number of Re $= 2.2 \times 10^5$, based on the chord of the front-wing main plane, the solution is particularly sensitive to the spatial resolution, requiring the use of techniques such as spectral vanishing viscosity (SVV) [23] and dealiasing in order to efficiently maintain the stability of the solution. To improve convergence of the conjugate gradient solver, we apply a low-energy block preconditioner [15] to the linear systems for velocity and pressure. We apply, in addition, a coarse-space preconditioner, implemented as a Cholesky factorisation [14], on the pressure field using an additive Schwarz approach. A high-order operator-splitting scheme [24] is used to decouple the system into four linear equations together with consistent pressure boundary conditions [25], although care must be taken to avoid instabilities arising from this formulation [26]. Further examples of the capabilities of the incompressible Navier–Stokes solver are given in Section 4.7.

In contrast, the `CompressibleFlowSolver` encapsulates two different sets of equations forming the cornerstone of aerodynamics problems: the compressible Euler and compressible

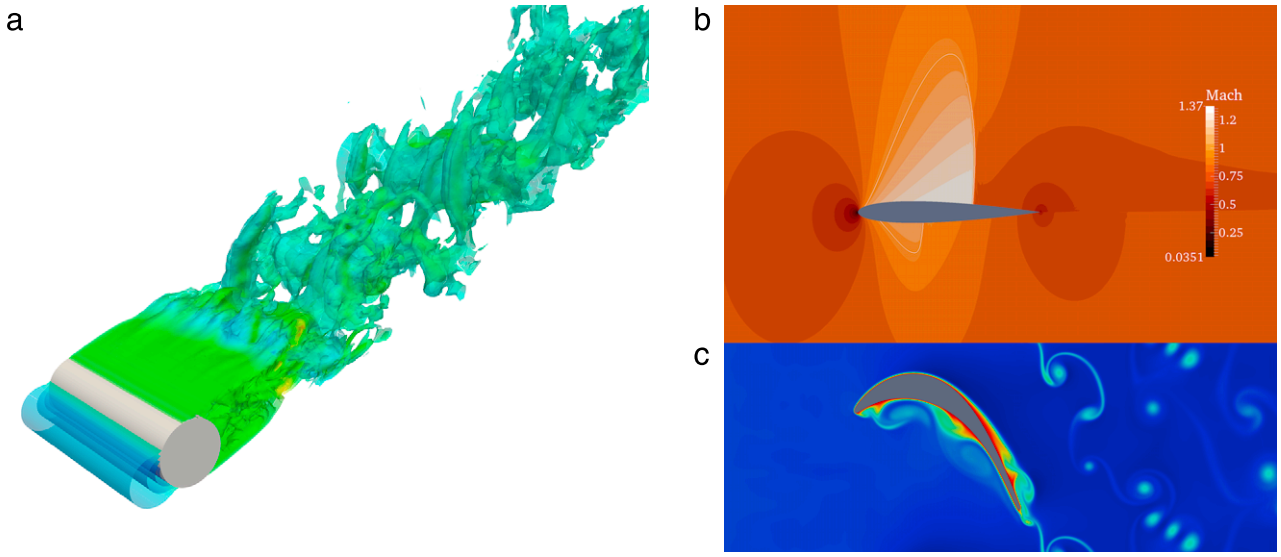

Fig. 4. Examples of external aerodynamics problems solved using the `CompressibleFlowSolver`. (a) Flow over a cylinder at Re = 3900. (b) Euler simulation of flow over a NACA0012 aerofoil at $\mathrm{Ma}_\infty = 0.8$. (c) Temperature of flow passing over a T106C low-pressure turbine blade at Re = 80,000. Simulation input files are provided in Appendices A.15, A.16 and A.17, respectively.

Navier–Stokes equations. In these equations, the compressibility of the flow is taken into account, leading to a conservative hyperbolic formulation,

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{U}) = \nabla \cdot \mathbf{F}_v(\mathbf{U}),$$

where $\mathbf{U} = [\rho, \rho u, \rho v, \rho w, E]^\top$ is the vector of conserved variables in terms of density ρ, velocity $(u_1, u_2, u_3) = (u, v, w)$, E is the specific total energy, and

$$\mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u & \rho v & \rho w \\ p + \rho u^2 & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ u(E+p) & u(E+p) & v(E+p) \end{bmatrix},$$

where p is the pressure. To close the system we need to specify an equation of state, in this case the ideal gas law $p = \rho RT$ where T is the temperature and R is the gas constant. For the Euler equations, the tensor of viscous forces $\mathbf{F}_v(\mathbf{U}) = \mathbf{0}$, while for Navier–Stokes

$$\mathbf{F}_v(\mathbf{U}) = \begin{bmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \tau_{yy} & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \tau_{zz} \\ A & B & C \end{bmatrix},$$

with

$$A = u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\partial_x T,$$
$$B = u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\partial_y T,$$
$$C = u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\partial_z T,$$

where in tensor notation the stress tensor $\tau_{x_i x_j} = 2\mu(\partial_{x_i} u_i + \partial_{x_i} u_j - \frac{1}{3}\partial_{x_k} u_k \delta_{ij})$, μ is the dynamic viscosity calculated using Sutherland's law and k is the thermal conductivity.

To discretise these equations in space, we adopt an approach which allows for the resolution of discontinuities and shocks that may appear in the solution at transonic and supersonic flow speeds. We therefore use approximations to our solution comprised of functions which are not continuous across element boundaries. Traditionally, we follow a Galerkin approach by utilising the variational form of the equations in order to obtain the discontinuous Galerkin method. One of the key features of *Nektar++* is the ability to select a wide range of numerical options, and to this end we support both discontinuous Galerkin and flux reconstruction spatial discretisations, which have various numerical equivalences [27] but may possess different performance characteristics. In the flux reconstruction formulation, we instead use the equation in differential form in combination with correction functions which impose continuity of fluxes between elements.

In either case, information is transferred between elements by solving a one-dimensional Riemann problem at the interface between two elements. For the non-viscous terms there is support for a wide variety of Riemann solvers, including an exact solution or a number of approximate solvers such as HLLC, Roe and Lax–Friedrichs solvers [28]. For the viscous terms, we utilise a local discontinuous Galerkin method (or the equivalent flux reconstruction version). Boundary conditions are implemented in a weak form by modifying the fluxes for both the non-viscous and viscous terms [29]. Various versions of the discontinuous Galerkin method which are available throughout the literature, mostly relating to the choices of modal functions and quadrature points, can also be readily selected by setting appropriate options in the input file.

Given the complexity and highly nonlinear form of these equations, we adopt a fully explicit formulation to discretise the equations in time, allowing us to use any of the explicit timestepping algorithms implemented through the general linear methods framework [20], including 2nd and 4th order Runge–Kutta methods. Finally, in order to stabilise the flow in the presence of discontinuities we utilise a shock capturing technique which makes use of artificial viscosity to damp oscillations in the solution, in conjunction with a discontinuity sensor adapted from the approach taken in [30] to decide where the addition of artificial viscosity is needed.

Fig. 4 shows representative results from compressible flow simulations of a number of industrially relevant test cases. We first highlight two simulations which utilise the Navier–Stokes equations. Fig. 4(a) demonstrates the three-dimensional version of the compressible solver showing flow over a cylinder at Re = 3900. In this figure we visualise isocontours of the pressure field and colour the field according to the density ρ. To demonstrate the shock capturing techniques available in the code, Fig. 4(b) shows the results of an Euler simulation for flow over a NACA0012 aerofoil at a farfield Mach number $\mathrm{Ma}_\infty = 0.8$ and a 1.5° angle of attack. The transonic Mach number of this flow leads to the development of a strong and weak shock along the upper and lower surfaces of

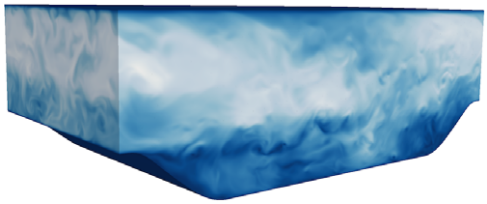

Fig. 5. Contours of velocity magnitude in a periodic hill simulation at Re = 2800. Simulation input files available in Appendix A.18.

the wing respectively. This figure shows isocontours of the Mach number where the presence of the shocks are clearly identified. Finally, in Fig. 4(c), we visualise the temperature field from flow passing over a T106C low-pressure turbine blade at Re = 80,000 to highlight applications to high Reynolds number flow simulations.

4.2. Transitional turbulent flow dynamics

Transient problems in which turbulence dominates the flow domain, or in which the transition to turbulence dominates the simulation, remain some of the most challenging problems to resolve in computational fluid simulations. Here, accurate numerical schemes and high resolution of the domain is critical. Moreover, any choice of scheme must be efficient in order to obtain results in computationally feasible time-scales. Traditionally, highly resolved turbulence simulations, such as the Taylor–Green vortex problem, lie firmly in the class of spectral methods. However, spectral methods typically lead to strong geometry restrictions which limits the domain of interest to simple shapes such as cuboids or cylinders.

Whilst spectral element methods may seem the ideal choice for such simulations, particularly when the domain of interest is geometrically complex, they can be more computationally expensive in comparison to spectral methods. However, when the domain of interest has a geometrically homogeneous component – that is, the domain can be seen to be the tensor product of a two-dimensional 'complex' part and a one-dimensional segment – we can combine both the spectral element and traditional spectral methods to create a highly efficient and spectrally accurate technique [4].

We consider the application of this methodology to the problem of flow over a periodic hill, depicted in Fig. 5, where the flow is periodic in both streamwise and spanwise directions. This case is a well-established benchmark problem in the DNS and LES communities [31], and is challenging to resolve due to the smooth detachment of the fluid from the surface and recirculation region. Here we consider a Reynolds number of 2800, normalised by the bulk velocity at the hill crest and the height of the hill, with an appropriate body forcing term to drive the flow over the periodic hill configuration.

The periodicity of this problem makes it an ideal candidate for the hybrid technique described above. We therefore construct a two-dimensional mesh of 3626 quadrilateral elements at polynomial order $P = 6$, and exploit the domain symmetry with a Fourier pseudospectral method consisting of 160 collocation points in the spanwise direction to perform the simulation. This yields a resolution of 20.9M degrees of freedom per field variable and allows us to obtain excellent agreement with the benchmark statistics, which are available in reference [32].

4.3. Flow stability

In addition to direct numerical simulation of the full non-linear incompressible Navier–Stokes equations, the `IncNavierStokesSolver` supports global flow stability analysis through the linearised Navier–Stokes equations with respect to a steady or periodic base flow. This process identifies whether a steady flow is susceptible to a fundamental change of state when perturbed by an infinitesimal disturbance. The linearisation takes the form

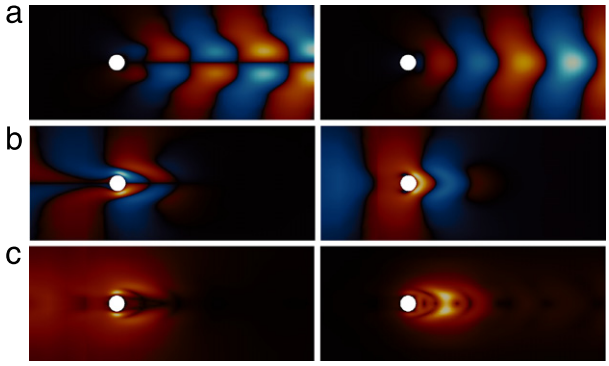

Fig. 6. Linear stability analyses of two-dimensional flow past a circular cylinder at Re = 42. Illustrative plots of (a) streamwise (left) and transverse (right) components of velocity for the dominant direct mode, (b) streamwise (left) and transverse (right) velocity for the dominant adjoint mode and (c) structural sensitivity to base flow modification (left) and local feedback (right). Simulation input files are provided in Appendix A.19.

$$\frac{\partial \mathbf{u}'}{\partial t} + (\mathbf{u}' \cdot \nabla)\mathbf{U} + (\mathbf{U} \cdot \nabla)\mathbf{u}' = -\nabla p' + \nu \nabla^2 \mathbf{u}' \tag{4a}$$

$$\nabla \cdot \mathbf{u}' = 0, \tag{4b}$$

where $\mathbf{U}$ is the base flow and $\mathbf{u}'$ is now the perturbation. The time-independent base flow is computed through evolving Eqs. (3) to steady-state with appropriate boundary conditions. Time-periodic base flows are sampled at regular intervals and interpolated.

The linear evolution of a perturbation under Eqs. (4) can be expressed as

$$u'(t) = \mathcal{A}(t)u'(0),$$

for some initial state $u'(0)$, and we seek, for some arbitrary time T, the dominant eigenvalues and eigenmodes of the operator $\mathcal{A}(T)$, which are solutions to the equation

$$\mathcal{A}(T)\tilde{u}_j = \lambda_j \tilde{u}_j.$$

The sign of the leading eigenvalues λ_j are used to establish the global stability of the flow. An iterative Arnoldi method [33] is applied to a discretisation $\mathbf{M}$ of $\mathcal{A}(T)$. Repeated actions of $\mathbf{M}$ are applied to the discrete initial state $\mathbf{u}_0$ using the same time-integration code as for the non-linear equations [21]. The resulting sequence of vectors spans a Krylov subspace of $\mathbf{M}$ and, through a partial Hessenberg reduction, the leading eigenvalues and eigenvectors can be efficiently determined. The same approach can be applied to the adjoint form of the linearised Navier–Stokes evolution operator $\mathcal{A}^*(T)$ to examine the receptivity of the flow and, in combination with the direct mode, identify the sensitivity to base flow modification and local feedback. The direct and adjoint methods can also be combined to identify convective instabilities over different time horizons τ in a flow by computing the leading eigenmodes of $(\mathcal{A}^*\mathcal{A})(\tau)$. This is referred to as transient growth analysis.

To illustrate the linear analysis capabilities of *Nektar++*, we use the example of two-dimensional flow past a circular cylinder at Re = 42, just below the critical Reynolds number for the onset of the Bénard–von Kármán vortex street. This is a well-established test case for which significant analysis is available in the literature. We show in Fig. 6 the leading eigenmodes for the direct ($\mathcal{A}$) and adjoint ($\mathcal{A}^*$) linear operators for both the streamwise and cross-stream components of velocity. The modes are characterised by the asymmetry in the streamwise component and symmetry in the cross-stream component. We also note the spatial distribution

of the modes with the leading direct modes extending far downstream of the cylinder, while the adjoint modes are predominantly localised upstream but close to the cylinder. This separation is a result of the non-normality of the $\mathcal{A}$ operator. We also show the structural sensitivity of the flow to base flow modification and local feedback. The latter highlights regions where localised forcing would have greatest impact on the flow.

4.4. Shallow water modelling

The `ShallowWaterSolver` simulates depth-averaged wave equations, often referred to as "long-wave" approximations. These equations are often used for engineering applications where the vertical dimension of the flow is small compared to the horizontal. Examples of applications include tidal flow, river flooding and nearshore phenomena such as wave-induced circulation and wave disturbances in ports.

The governing equations are derived from potential flow: the Laplace equation inside the flow domain and appropriate boundary conditions at the free surface and bottom. The two key steps are (i) the expansion of the velocity potential with respect to the vertical coordinate and (ii) the integration of the Laplace equation over the fluid depth. This results in sets of equations expressed in horizontal dimensions only. Depending on the order of truncation in nonlinearity and dispersion, numerous long-wave equations with different kinematic behaviour have been derived over the years [34–36].

Many depth-averaged equations can be written in a generic form as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{U}) + \mathbf{D}(\mathbf{U}) = \mathbf{S}(\mathbf{U}) \,, \tag{5}$$

where $\mathbf{U} = [H \,, Hu \,, Hv]^T$ is the vector of conserved variables. The horizontal velocity is denoted by $\mathbf{u} = [u(\mathbf{x}, t) \,, v(\mathbf{x}, t)]^T$, $H(\mathbf{x}, t) = \eta(\mathbf{x}, t) + d(\mathbf{x})$ is the total water depth, η is the free surface elevation and d the still water depth. The flux vector $\mathbf{F}(\mathbf{U})$ is given as

$$\mathbf{F}(\mathbf{U}) = \begin{bmatrix} Hu & Hv \\ Hu^2 + gH^2/2 & Huv \\ Huv & Hv^2 + gH^2/2 \end{bmatrix} , \tag{6}$$

in which g is the acceleration due to gravity. The source term $\mathbf{S}(\mathbf{U})$ contains forcing due to, for example, Coriolis effects, bed-slopes and bottom friction. Importantly, $\mathbf{D}(\mathbf{U})$ contains all the dispersive terms. The actual form of the dispersive terms differs between different wave equations and the term can be highly complex with many high-order mixed and spatial derivatives.

At present, the `ShallowWaterSolver` supports the non-dispersive shallow-water equations (SWE) and the weakly dispersive Boussinesq equations of Peregrine [34]. The SWE are recovered if $\mathbf{D}(\mathbf{U}) \equiv \mathbf{0}$ while for the Peregrine equation the expression is:

$$\mathbf{D}(\mathbf{U}) = \partial_t \begin{bmatrix} 0 \\ (d^3/6)\partial_x (\nabla \cdot (H\mathbf{u}/d)) - (d^2/2)\partial_x (\nabla \cdot (H\mathbf{u})) \\ (d^3/6)\partial_y (\nabla \cdot (H\mathbf{u}/d)) - (d^2/2)\partial_y (\nabla \cdot (H\mathbf{u})) \end{bmatrix} . \tag{7}$$

The Boussinesq equations are solved using the wave continuity approach [37]. The momentum equations are first recast into a scalar Helmholtz type equation and solved for the auxiliary variable $z = \nabla \cdot \partial_t (H\mathbf{u})$. The conservative variables are recovered in a subsequent step.

A frequently used test-case for Boussinesq models is the scattering of a solitary wave impinging a vertical cylinder. Here a solitary wave with nonlinearity $\epsilon = 0.1$ is propagating over a still water depth of 1 m ($\epsilon = A/d$, where A is the wave amplitude). The initial solitary wave condition is given by Laitone's first order solution. The cylinder has a diameter of 4 m, giving a Keulegan–Carpenter number well below unity and diffraction number on the order of 2. Hence, the viscous effects are small while the diffraction and scattering are significant.

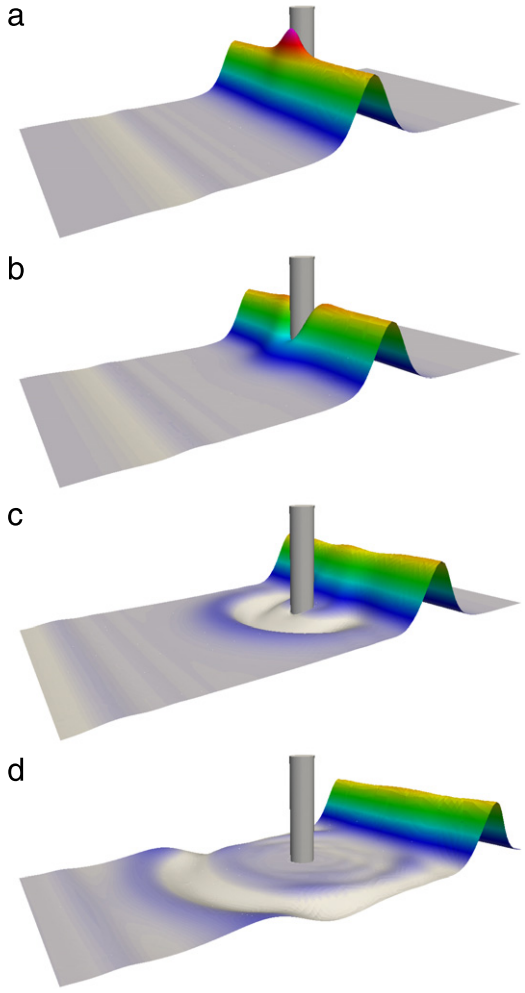

Fig. 7. Solitary wave impinging a stationary cylinder. The colours and surface deformation illustrate the height of the surface at times (a) $t = 4.5$ s; (b) $t = 5.5$ s; (c) $t = 8.5$ s; and (d) $t = 12.5$ s. Simulation input files are provided in Appendix A.20.

We compute the solution in the domain $x \in [-25 \,, 50]$ metres and $y \in [-19.2 \,, 19.2]$ metres, discretised into 552 triangles using $P = 5$. Snapshots of the free surface elevation at four different times are shown in Fig. 7. In Fig. 7(a) the solitary wave reaches its maximum run-up on the cylinder, while in Fig. 7(b) the peak of solitary wave has reached the centre of the cylinder and a depression in the free surface around the cylinder is clearly visible. The propagation of the scattered waves, and those later reflected from the side walls, are seen in Figs. 7(c) and (d).

4.5. Cardiac electrophysiology

The cardiac electrical system in the heart is the signalling mechanism used to ensure coordinated contraction and efficient pumping of blood. Conduction occurs due to a complex sequence of active ion exchanges between intracellular and extracellular spaces, initiated due to a potential difference between the inside and outside of the cell exceeding a threshold, producing an action potential. This causes a potential difference across boundaries with adjacent cells, resulting in a flow of ions between cells and triggering an action potential in the adjacent cell. Disease, age and infarction lead to interruption of this signalling process and may produce abnormal conduction patterns known as arrhythmias. Clinically,

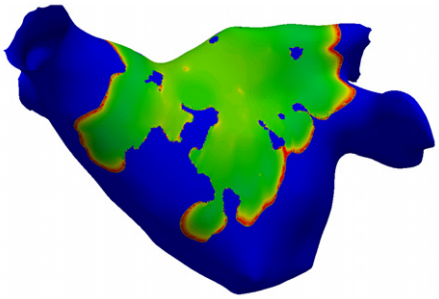

Fig. 8. Illustrative simulation of a depolarising electrochemical wavefront on a two-dimensional manifold representation of a human left atrium. Blue areas denote regions of unexcited (polarised) tissue, while green denotes areas of excited (depolarised) cells. The red areas highlight the wavefront. Simulation input files are provided in Appendix A.21. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

these can be treated using catheter ablation, however accurately selecting the most effective substrate modification to restore normal rhythm is often particularly challenging and may benefit from insight derived from computer modelling.

The `CardiacEPSolver` models the conduction process using the monodomain equations

$$\beta\left(C_m\frac{\partial u}{\partial t}+I_{ion}\right)=\nabla\cdot\boldsymbol{\sigma}\nabla u,$$

where the I_{ion} term captures the complex movement of ions in and out of cells and is itself modelled as a set of ordinary differential equations. Additionally, σ captures the potentially heterogeneous and anisotropic nature of the tissue which governs the speed of electrical propagation. While full 3D simulations of myocardium are traditionally performed, the left atrium is sufficiently thin that it can be reasonably represented as a two-dimensional manifold embedded in three dimensions and solved at significantly reduced computational cost [38].

An example of conduction propagation over the left atrium using the monodomain equations is illustrated in Fig. 8. Electrophysiological characteristics vary spatially, with regions of scar and fibrosis more resistive to activation, resulting in activation patterns of greater complexity. The geometry is derived from segmented magnetic resonance imaging (MRI) data, while tissue heterogeneity is prescribed based on late gadolinium-enhanced MRI. A human atrial ionic model [39] is used to compute the I_{ion} term and represents the exchanges of ions between the interior and exterior of the cell, along with other cellular biophysics.

4.6. Arterial pulse-wave propagation

1D modelling of the vasculature (arterial network) represents an insightful and efficient tool for tackling problems encountered in arterial biomechanics as well as other engineering problems. In particular, 3D modelling of the vasculature is relatively expensive. 1D modelling provides an alternative in which the modelling assumptions provide a good balance between physiological accuracy and computational efficiency. To describe the flow and pressure in this network we consider the conservation of mass and momentum applied to an impermeable, deformable tube filled with an incompressible fluid, the nonlinear system of partial differential equations presented in non-conservative form is given by

$$\frac{\partial\mathbf{U}}{\partial t}+\mathbf{H}\frac{\partial\mathbf{U}}{\partial x}=\mathbf{S},\tag{8}$$

$$\mathbf{U}=\begin{bmatrix}U\\A\end{bmatrix},\qquad\mathbf{H}=\begin{bmatrix}U & A\\ \rho\dfrac{\partial p}{\partial A} & U\end{bmatrix},\qquad\mathbf{S}=\begin{bmatrix}0\\ \dfrac{1}{\rho}\left(\dfrac{f}{A}-s\right)\end{bmatrix},$$

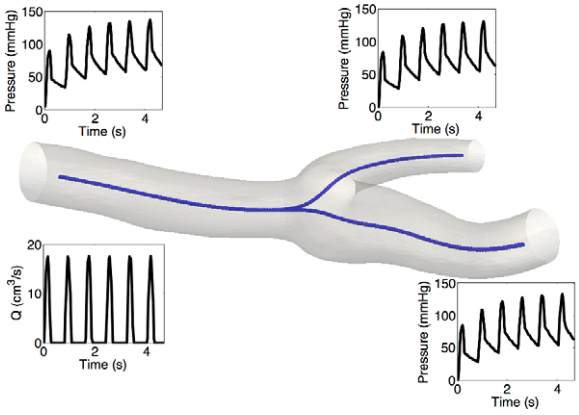

Fig. 9. Geometry used for the simulation. Insets show how flow and pressure vary with time and different locations in the geometry. Simulation input files are provided in Appendix A.22.

in which A is the Area (related to pressure), x is the axial coordinate along the vessel, $U(x,t)$ the axial velocity, $P(x,t)$ is the pressure in the tube, ρ is the density and finally f the frictional force per unit length. The unknowns in Eq. (8) are u, A and p; hence, we must provide an explicit algebraic relationship to close this system. Typically, closure is provided by an algebraic relationship between A and p.

For a thin elastic tube this is given by

$$p=p_0+\beta\left(\sqrt{A}-\sqrt{A_0}\right),\quad\beta=\frac{\sqrt{\pi}hE}{(1-\nu^2)A_0},\tag{9}$$

where p_0 is the external pressure, A_0 is the initial cross-sectional area, E is the Young's modulus, h is the vessel wall thickness and ν is the Poisson's ratio. Other more elaborate pressure–area relationships are currently being implemented into the framework. Application of Riemann's method of characteristics to Eqs. (8) and (9) indicates that velocity and area are propagated through the system by forward and backward travelling waves. These waves are reflected within the network by appropriate treatment of interfaces and boundaries (see for example [40,41]). The final system of equations are discretised in the *Nektar++* framework using a discontinuous Galerkin projection.

To illustrate the capabilities of the `PulseWaveSolver`, a 1D geometry is created by extracting the centreline directly from a 3D segmentation of a carotid bifurcation. The extracted centreline with the segmented geometry overlaid is shown in Fig. 9. At the inlet a half-sinusoidal flow profile is applied during the systolic phase, whist during the diastolic phase a no-flow condition is applied. Although this profile is not representative of the carotid wave, it is useful for demonstrating essential dynamics of the system e.g. reflection of backward travelling waves only during the diastolic phase. At the outflow RCR boundary conditions are utilised [41]. The RCR model is an electrical analogy consisting of two resistors (total peripheral resistance) and a capacitor (peripheral compliance). This boundary condition takes into account the effects of the peripheral vessels (e.g. small arteries, arterioles and capillaries) on pulse wave propagation. The pressure and flow results are shown in Fig. 9. The insets demonstrate that the pressure needs about 4 cycles to reach a periodic state. The boundary condition is responsible for establishing the correct pressure–flow relationship on the outflow and throughout the domain.

4.7. Vascular mass transport

To conclude this section, we illustrate the flexibility of the software in combining two solvers to understand mass transport in

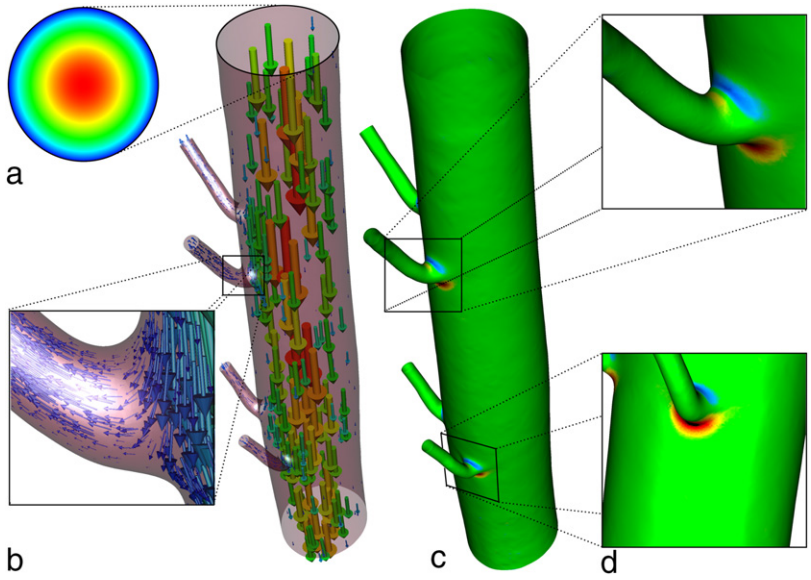

Fig. 10. Calculation of mass transport in an intercostal pair. (a) Inflow boundary condition for fluid simulation computed by solving a Poisson problem on the inflow surface. (b) Flow computed using the incompressible Navier–Stokes equations. (c) Mass transport of low diffusion coefficient species simulated using the advection–diffusion solver with $\mathrm{Pe} = 7.5 \times 10^5$. Non-dimensional gradient of concentration (Sherwood number) at the wall is shown. (d) Detailed view of non-dimensional concentration gradient at intercostal branches. Simulation input files are provided in Appendix A.21.

the aorta. We consider the simulation of blood-flow through a pair of intercostal branches in the descending aorta. The three-dimensional geometry is derived from CT scans and is meshed using a combination of tetrahedral and prismatic elements. Prisms are used to better capture the boundary layer close to the wall, while tetrahedra fill the remaining interior of the domain. The embedded manifold discretisation code can be used to compute boundary conditions for three-dimensional simulations where the complexity of the geometry precludes the use of analytic conditions. The resulting inflow condition is shown in Fig. 10(a).

The flow is modelled using the incompressible Navier–Stokes equations (see Eqs. (3) in Section 4.1). In this case, the inlet flow-profile $\mathbf{f}$ in Eq. (3)(e) is the solution of the Poisson problem, computed using the `ADRSolver`, on the two-dimensional inlet boundary surface for a prescribed body force. The resulting profile is imposed on the three-dimensional flow problem as illustrated and the steady-state velocity field from the flow simulation at Reynolds number $\mathrm{Re} = 300$ is shown in Fig. 10(b). A single boundary layer of prismatic elements is used for this simulation and both the prismatic and tetrahedral elements use a polynomial expansion order of $P = 4$. This is sufficient to capture the boundary layer at the walls.

We next solve the advection–diffusion equation,

$$\nabla \cdot (-D\nabla c + c\mathbf{u}) = 0,$$

to model transport of oxygen along the arterial wall, again using the `ADRSolver`. Here, c is the concentration and $\mathbf{u}$ is the steady-state flow solution obtained previously. $D = 1/\mathrm{Pe}$ is the diffusivity of the species considered where we use $\mathrm{Pe} = 7.5 \times 10^5$ for oxygen. This value corresponds to a Schmidt number (relative size of mass transfer and momentum boundary layers) of 3000, which is typical of species such as free oxygen or adenosine triphosphate (ATP). For most of the domain, the non-dimensional concentration remains constant at $c = 1$. However, a particularly high gradient in concentration forms at the wall. Biologically, it is this non-dimensional concentration gradient, the Sherwood number (Sh), in the vicinity of the cells that line the arterial wall which is of particular interest. This is given by

$$\mathrm{Sh} = 2\nabla c \cdot \mathbf{n},$$

where c is the non-dimensional concentration and $\mathbf{n}$ is the local wall normal.

The existing mesh used for the flow simulation is unable to resolve the concentration gradients close to the wall. We therefore refine the boundary layer in the wall-normal direction, with element heights following a geometric progression, using an isoparametric refinement technique [42,43] to naturally curve the resulting subelements in such a way as to guarantee their validity. This technique is implemented in the `MeshConvert` utility, which acts as both a way to convert meshes from other formats such as Gmsh [44] and Star-CCM+, but also to apply a variety of processing steps to the mesh in order to make it suitable for high-order computation.

To minimise computational cost, we reduce the polynomial order of the prisms in the directions parallel to the wall, since the concentration shows negligible variation in these directions. This exploits the rich nature of the spectral/*hp* discretisation and removes the need for a potentially expensive remeshing and interpolation step. Furthermore, the domain-interior tetrahedra may also be discarded and a Dirichlet $c = 1$ condition imposed on the resulting interior prism boundary.

Fig. 10(c) shows the resulting Sherwood number distribution on the surface of the arterial wall. Regions of reduced mass flux are observed upstream of the intercostal branches, while elevated mass flux are observed downstream. Fig. 10(d) shows close-ups of the branches to illustrate this. These patterns are driven directly by the blood flow mechanics in these regions. In particular, upstream of the intercostal branch the mass transfer boundary layer grows due to a growth of the momentum boundary layer as it negotiates the sharp bend into the branch, forcing flow away from the apex of the bend; the bulk of the flow continues down the aorta with only a small proportion entering the branch. Progressing into the branch, as shown in the top inset of Fig. 10(d), the mass flux is slightly elevated as the boundary layer shrinks and the flow is directed towards the wall of the branch, causing the mass transfer boundary

layer to shrink. In the main aorta, distal to the intercostal branch, mass flux to the arterial wall is elevated. This is associated with a flow stagnation region that forms due to the impingement of flow crossing the branch mouth onto the wall, as illustrated in Fig. 10(b). Below the impingement zone the boundary layer grows, leading to a reduction in the mass flux.

5. Discussion & future directions

The *Nektar++* framework provides a feature-rich platform with which to develop numerical methods and solvers in the context of spectral/*hp* element methods. It has been designed in such a way that the libraries reflect the mathematical abstractions of the method, to simplify uptake for new users, as well as being written in a modular manner to improve the robustness of the code, minimise duplication of functionality and promote sustainability.

Development & tools

The development of a complex and extensive software project such as *Nektar++* necessitates the adoption of certain development practices to enable developers to easily write new code without breaking the existing code for other users of the framework. The code is managed using the git distributed version control system [45] due to its performance, enhanced support for branching, as well as allowing off-line development. All development is performed in branches and only after rigorous multi-architecture testing and internal peer-review is new code merged into the main codebase, thereby always maintaining a stable distribution. New bugs and feature requests are recorded using the Trac [46] issue-management system. To enable cross-platform compatibility *Nektar++* uses CMake [47] to manage the creation of build scripts, which also allows the automatic download and compilation of additional third-party libraries and simplifies the configuration and installation for the end-user. Boost [48] data structures and algorithms are used throughout the code to simplify complex data management, improve code modularity and avoid the introduction of memory leaks. While the templated nature of many of the Boost libraries significantly adds to compilation times, we consider the benefits to code robustness justify its use.

Testing is a critical part of the development cycle for any software project and regression tests ensure new features do not break existing functionality, ensuring the code base remains stable when new features are implemented. Continuous integration using a publicly accessible buildbot service [49], builds and executes these tests after each update to the master branch of the code, across a range of operating systems, architectures and configuration options. The system may also be used by developers to test other branches prior to inclusion in the main codebase.

Nektar++ makes extensive use of C++ programming patterns to decouple and manage components of the code and the creation of objects at runtime. As well as limiting inter-dependencies within the source code, it improves compilation times, enforces modularity and simplifies compile-time selection of features and functionality. Design patterns formalise many aspects of writing high-quality, robust code and we briefly outline some of the key patterns used within *Nektar++*.

The *Template method pattern* provides separation between algorithms and specific implementation. A general algorithm is implemented in a C++ base class, while particular aspects of the algorithm implementation are overridden in derived classes through the use of protected virtual functions. These derived classes could correspond to specific element shapes or Galerkin projections, for example. The *Factory method pattern* allows dynamic key-based object creation at runtime, without prescribing the particular implementation choice a priori within the code at

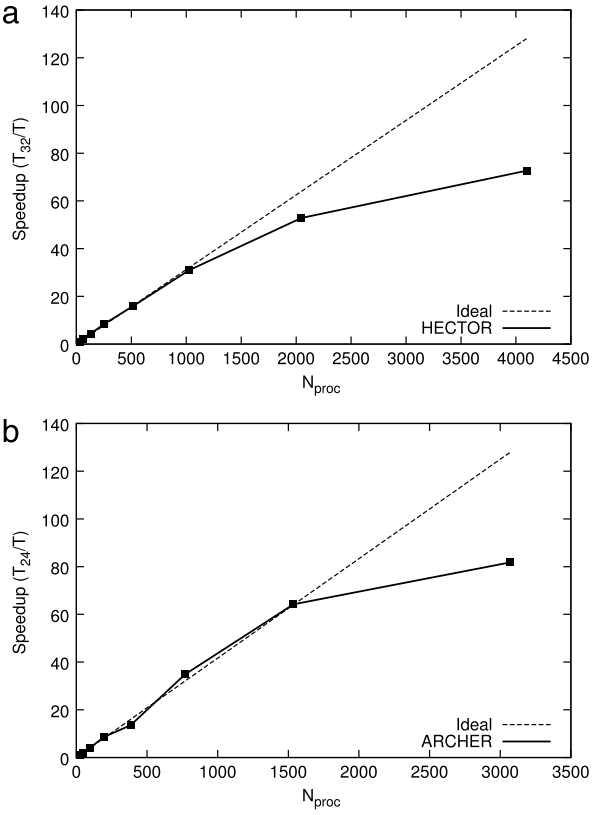

Fig. 11. Strong scaling of the *Nektar++* diffusion solver on an intercostal pair simulation similar to the one presented in Section 4.7. (a) Performance scaling on HECToR where each node has 32 cores and (b) performance scaling on ARCHER where each node has 24 cores. In both cases, results are normalised by the performance on a single node.

compilation time. The technique is used extensively within the libraries as a means to decouple components of the code and manage multiple implementations of an algorithm. Additional implementation modules can be added to the code at a later date without needing to modify those routines which instantiate the objects. Finally, *Managers* provide a templated mechanism to keep track of large numbers of similarly-typed objects during program execution and avoid duplication where possible and so minimise memory usage. Their underlying data structure is a static map. For each manager, a functor is held which can be used to instantiate objects which have not previously been allocated.

Performance

Modern high-performance computing has transformed in recent years as processor clock speeds have reached practical limitations and vendors have been forced to increase parallelism in order to support greater computation. This has resulted in an increase in the number of cores per processor die and subsequently an effective reduction in available cache per core. It is becoming ever more expensive to move data within a computer and modern software must therefore be engineered to optimise algorithms to make the most of data while it is on the CPU. High-order methods naturally increase data locality by producing tightly coupled blocks of data. This enables greater cache coherency and therefore supports a larger number of floating point operations to be achieved per cache line than conventional linear finite element methods, when using high polynomial orders.

For each of the key finite element operators, the *Nektar++* architecture supports multiple implementations. This allows the code

to be targeted at a variety of architectures in an efficient manner based on the choice of input parameters, such as polynomial orders and mesh configuration. Although mathematically equivalent, these implementations may lead to slight differences in the numerical result due to the order of floating-point operation evaluation. However, for most applications, double-precision arithmetic provides more than sufficient precision to ensure this is not a practical problem.

Nektar++ is designed to work on a wide range of computer systems from laptops to large high-performance compute clusters. The code has been tested on large clusters such as HECToR and ARCHER, the UK National High Performance Computing Facilities, and shows excellent scaling for both two-dimensional and three-dimensional problems. Fig. 11 shows strong scaling for the implicit diffusion solve for an intercostal flow simulation similar to that presented in Section 4.7. This mesh contained approximately 8,000 tetrahedral elements, resulting in only 2 or 3 elements per core in the most parallel case, which accounts for the reduced efficiency at higher core counts.

Future work

Although the current version of *Nektar++* supports variable and heterogeneous choices of polynomial order, it does not yet support adaptive polynomial order during time advancement (*p*-adaptivity). This is one of the next features to be implemented in the code. Mesh refinement (*h*-adaptivity) is a well-established technique in many other finite element research codes, and we believe *hp*-adaptivity will provide substantial performance benefits in a wide range of application areas.

Finally, it is becoming increasingly costly for individual institutions to purchase and maintain the necessary large-scale HPC infrastructures to support cutting-edge research. In recent years cloud computing has become increasingly prevalent and is potentially the approach by which extensive computational resources may be obtained for simulation in the future. *Nektar++* is embracing this infrastructure shift through the development of Nekkloud [50] which removes the complexities of maintaining and deploying numerical code onto cloud platforms.

6. Availability

Nektar++ is open-source software, released under the MIT license, and is freely available from the project website (http://www.nektar.info). While the git repository is freely accessible, discrete releases are made at milestones in the project and are available to download as compressed tar archives, or as binary packages for a range of operating systems. These releases are considered to contain relatively complete functionality compared to the repository master branch.

Acknowledgements

Nektar++ has been developed over a number of years and we would like to thank the many people who have made contributions to the specific application codes distributed with the libraries. In particular, we would like to acknowledge the contribution of Christian Roth for initial developments on the pulse-wave solver, Kilian Lackhove for work on extending the acoustic perturbation equations solver and Rheeda Ali, Eugene Chang and Caroline Roney for contributing to the cardiac electrophysiology solver.

The development of Nektar++ has been supported by a number of funding agencies including Engineering and Physical Sciences Research Council (grants EP/L000407/1, EP/K037536/1, EP/K038788/1, EP/L000261/1, EP/I037946/1, EP/H000208/1, EP/I030239/1, EP/H050507/1, EP/D044073/1, EP/C539834/1), the British Heart Foundation (grants FS/11/22/28745 and RG/10/11/28457), the Royal Society of Engineering, McLaren Racing, the National Science Foundation (grants IIS-1212806, OCI-1148291), the Army Research Office (grant W911NF121037), the Air Force Office of Scientific Research (grant FA9550-08-1-0156) and the Department of Energy (grant DE-EE0004449).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.cpc.2015.02.008.

References

[1] A.T. Patera, J. Comput. Phys. 54 (3) (1984) 468–488.
[2] I. Babuska, B.A. Szabo, I.N. Katz, SIAM J. Numer. Anal. 18 (3) (1981) 515–545.
[3] G.E. Karniadakis, S.J. Sherwin, Spectral/hp Element Methods for CFD, Oxford University Press, 2005.
[4] H.M. Blackburn, S. Sherwin, J. Comput. Phys. 197 (2) (2004) 759–778.
[5] P. Fischer, J. Kruse, J. Mullen, H. Tufo, J. Lottes, S. Kerkemeier, Nek5000–open source spectral element CFD solver, Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, see https://nek5000.mcs.anl.gov/index.php/MainPage.
[6] T. Vejchodskỳ, P. Šolín, M. Zítka, Math. Comput. Simul. 76 (1) (2007) 223–228.
[7] A. Dedner, R. Klöfkorn, M. Nolte, M. Ohlberger, Computing 90 (3–4) (2010) 165–196.
[8] W. Bangerth, R. Hartmann, G. Kanschat, ACM Trans. Math. Softw. 33 (4) (2007) 24.
[9] J.S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications, Vol. 54, Springer, 2007.
[10] F. Witherden, A. Farrington, P. Vincent, Comput. Phys. Comm. 185 (2014) 3028–3040. http://dx.doi.org/10.1016/j.cpc.2014.07.011.
[11] M. Dubiner, J. Sci. Comput. 6 (4) (1991) 345–390.
[12] S.J. Sherwin, G.E. Karniadakis, Comput. Methods Appl. Mech. Engrg. 123 (1–4) (1995) 189–229.
[13] M.G. Duffy, SIAM J. Numer. Anal. 19 (6) (1982) 1260–1262.
[14] H.M. Tufo, P.F. Fischer, J. Parallel Distrib. Comput. 61 (2) (2001) 151–177.
[15] S.J. Sherwin, M. Casarin, J. Comput. Phys. 171 (1) (2001) 394–417.
[16] P.E. Vos, S.J. Sherwin, R.M. Kirby, J. Comput. Phys. 229 (13) (2010) 5161–5181.
[17] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, P.H.J. Kelly, Comput. & Fluids 43 (2011) 23–28.
[18] C.D. Cantwell, S.J. Sherwin, R.M. Kirby, P.H.J. Kelly, Math. Mod. Nat. Phenom. 6 (2011) 84–96.
[19] P. Fischer, J. Lottes, D. Pointer, A. Siegel, Petascale Algorithms for Reactor Hydrodynamics, in: Journal of Physics: Conference Series, vol. 125, IOP Publishing, 2008, p. 012076.
[20] P.E. Vos, C. Eskilsson, A. Bolis, S. Chun, R.M. Kirby, S.J. Sherwin, Int. J. Comput. Fluid Dyn. 25 (3) (2011) 107–125.
[21] D. Barkley, H. Blackburn, S.J. Sherwin, Internat. J. Numer. Methods Fluids 57 (9) (2008) 1435–1458.
[22] S. Dong, G.E. Karniadakis, C. Chryssostomidis, J. Comput. Phys. 261 (2014) 83–105.
[23] R.M. Kirby, S.J. Sherwin, Comput. Methods Appl. Mech. Eng. 195 (23) (2006) 3128–3144.
[24] G.E. Karniadakis, M. Israeli, S.A. Orszag, J. Comput. Phys. 97 (2) (1991) 414–443.
[25] S.A. Orszag, M. Israeli, M.O. Deville, J. Sci. Comput. 1 (1) (1986) 75–111.
[26] E. Ferrer, D. Moxey, S.J. Sherwin, R.H.J. Willden, Commun. Comput. Phys. 16 (3) (2014) 817–840. http://dx.doi.org/10.4208/cicp.290114.170414a.
[27] D. de Grazia, G. Mengaldo, D. Moxey, P.E. Vincent, S.J. Sherwin, International journal for numerical methods in fluids 75 (12) (2014) 860–877. http://dx.doi.org/10.1002/fld.3915.
[28] E.F. Toro, Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction, third ed., Springer, Berlin, New York, 2009.
[29] G. Mengaldo, D. De Grazia, J. Peiro, A. Farrington, F. Witherden, P.E. Vincent, S.J. Sherwin, 7th AIAA Theoretical Fluid Mechanics Conference, AIAA Aviation, American Institute of Aeronautics and Astronautics, 2014.
[30] P.-O. Persson, J. Peraire, Sub-cell shock capturing for discontinuous Galerkin methods, AIAA 112.
[31] M. Breuer, N. Peller, C. Rapp, M. Manhart, Comput. & Fluids 38 (2) (2009) 433–457.
[32] ERCOFTAC QNET-CFD Database for test case UFR 3-30, 2D Periodic Hill Flow: database of numerical and experimental results 2014. URL http://qnet-ercoftac.cfms.org.uk/w/index.php/UFR_3-30_References.
[33] W.E. Arnoldi, Quart. Appl. Math. 9 (1) (1951) 17–29.
[34] D.H. Peregrine, J. Fluid Mech. 27 (1967) 815–827.
[35] P. Madsen, H. Schäffer, Philos. Trans. R. Soc. Lond. Ser. A 356 (1998) 3123–3184.
[36] M. Brocchini, Philos. Trans. R. Soc. Lond. Ser. A 469.
[37] C. Eskilsson, S. Sherwin, J. Comput. Phys. 212 (2006) 566–589.
[38] C.D. Cantwell, S. Yakovlev, R.M. Kirby, N.S. Peters, S.J. Sherwin, J. Comput. Phys. 257 (2014) 813–829.
[39] M. Courtemanche, R.J. Ramirez, S. Nattel, Amer. J. Physiol. Heart Circul. Physiol. 44 (1) (1998) H301.
[40] S. Sherwin, L. Formaggia, J. Peiro, V. Franke, Internat. J. Numer. Methods Fluids 43 (6–7) (2003) 673–700.

[41] J. Alastruey, K. Parker, J. Peiró, S. Sherwin, Commun. Comput. Phys. 4 (2) (2008) 317–336.
[42] D. Moxey, M.D. Green, S.J. Sherwin, J. Peiró, Comput. Methods Appl. Mech. Engrg. 283 (2015) 636–650. http://dx.doi.org/10.1016/j.cma.2014.09.019.
[43] D. Moxey, M. Hazan, S.J. Sherwin, J. Peiró, On the generation of curvilinear meshes through subdivision of isoparametric elements, in: New Challenges in Grid Generation and Adaptivity for Scientific Computing, in: SEMA SIMAI Springer Series, Vol. 5, 2015.
[44] C. Geuzaine, J.-F. Remacle, Int. J. Numer. Methods Eng. 79 (11) (2009) 1309–1331. http://dx.doi.org/10.1002/nme.2579.
[45] L. Torvalds, J. Hamano, GIT: Fast version control system 2014. URL http://git-scm.com.
[46] Trac integrated SCM & project management 2014. URL http://trac.edgewall.org.
[47] CMake 2014. URL http://cmake.org.
[48] Boost C++ libraries 2014. URL http://www.boost.org.
[49] Buildbot 2014. URL http://www.buildbot.net.
[50] J. Cohen, D. Moxey, C. Cantwell, P. Burovskiy, J. Darlington, S.J. Sherwin, Cluster Computing (CLUSTER), 2013 IEEE International Conference on, IEEE, 2013, pp. 1–5.