福昕PDF编辑器

• 永久 • 轻巧 • 自由

点击升级会员

点击批量购买

永久使用
无限制使用次数

极速轻巧
超低资源占用，告别卡顿慢

自由编辑
享受Word一样的编辑自由

扫一扫，关注公众号

arXiv:1804.02338v1 [cs.NA] 6 Apr 2018

AUTOMATIC SYMBOLIC COMPUTATION FOR DISCONTINUOUS GALERKIN FINITE ELEMENT METHODS

PAUL HOUSTON * AND NATHAN SIME †

Abstract. The implementation of discontinuous Galerkin finite element methods (DGFEMs) represents a very challenging computational task, particularly for systems of coupled nonlinear PDEs, including multiphysics problems, whose parameters may consist of power series or functionals of the solution variables. Thereby, the exploitation of symbolic algebra to express a given DGFEM approximation of a PDE problem within a high level language, whose syntax closely resembles the mathematical definition, is an invaluable tool. Indeed, this then facilitates the automatic assembly of the resulting system of (nonlinear) equations, as well as the computation of Fréchet derivative(s) of the DGFEM scheme, needed, for example, within a Newton-type solver. However, even exploiting symbolic algebra, the discretisation of coupled systems of PDEs can still be extremely verbose and hard to debug. Thereby, in this article we develop a further layer of abstraction by designing a class structure for the automatic computation of DGFEM formulations. This work has been implemented within the FEniCS package, based on exploiting the Unified Form Language. Numerical examples are presented which highlight the simplicity of implementation of DGFEMs for the numerical approximation of a range of PDE problems.

Key words. Symbolic computation, finite element methods, discontinuous Galerkin methods

AMS subject classifications. 65N30

1. Introduction.

The finite element method (FEM) represents an indispensable computational tool for the accurate, efficient, and rigorous numerical approximation of continuum models arising within a wide range of scientific and engineering application areas. Key reasons for the success of FEMs include their applicability to very general classes of partial differential equations (PDEs), simple treatment of complicated computational geometries and enforcement of boundary conditions, ease of adaptivity including both local mesh subdivision (h–refinement) and local polynomial enrichment (p–refinement), and, from a mathematical point of view, the availability of tools for their rigorous error analysis. However, when compared with their finite difference counterparts, FEMs are typically regarded as being *complicated* to implement. Indeed, assembly of the underlying matrix stemming from the FEM discretisation of a given (linear, for example) PDE problem typically involves mapping each element present in the computational mesh, defined in the global coordinate system, to a given reference or canonical element, where both the local FEM basis and corresponding quadrature, needed to approximate the underlying integral, is defined. In this manner, local elemental stiffness matrices and load vectors may be computed; these entries are then inserted into the global matrix and right-hand side vector, respectively, according to the elementwise local-to-global degree of freedom mapping, subject to the enforcement of inter-element continuity constraints and the imposition of boundary conditions. This general assembly strategy is almost universally employed within both open source and commercial software, e.g., FreeFem++ [23], DUNE [8], deal.II [4] and OpenFOAM [40], to name just a few. Thereby, in principle, assuming that a user can define both the element stiffness matrix and load vector, then a numerical approximation may be readily determined. However, most open source software packages do not

* School of Mathematical Sciences, University of Nottingham, University Park, Nottingham, NG7 2RD, UK, email: Paul.Houston@nottingham.ac.uk.

†Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, UK, email: njcs4@cam.ac.uk.

provide easy-to-use user interfaces; indeed, typically the user must first understand the low level language in which the code is written, e.g., C++, Fortran, etc, and develop an understanding of the underlying datastructures and function/subroutine calls defined within their chosen package in order to be able to develop code specific to their own application. This can, of course, be a rather time-consuming exercise, and often requires one to use multiple software libraries, for example, when certain features are not available within the chosen package which the user needs to utilise.

In the nonlinear setting, assuming a Newton-type iteration is exploited to solve the underlying system of nonlinear equations stemming from the given FEM employed, the general strategy of assembly of the resulting linearised equations is similar to the linear case; in this setting the load vector represents the residual of the numerical scheme. However, in this case, the Fréchet derivative of the FEM must now be computed; we point out that, in the context of the numerical detection of bifurcation points, the number of derivatives that must be computed just to form the so-called *extended system* needed to accurately compute the bifurcation point, even before a Newton iteration is implemented, is dependent on the codimension of the singularity being sought, cf. [10, 11]. In general, the evaluation and implementation of both the FEM residual and, moreover, the Fréchet derivative(s) of the scheme is a difficult and time consuming task, which is inevitably prone to human error. This is particularly the case when exploiting FEMs for the numerical approximation of systems of coupled nonlinear PDEs, including multiphysics problems, whose parameters may consist of power series or functionals of the solution variables, cf. [28].

Thus far, our discussion has primarily focussed on the application of conforming FEMs, whereby, excluding Neumann, or weakly imposed boundary conditions, only element contributions need to be evaluated. The exploitation of more general FEMs, and in particular discontinuous Galerkin FEMs (DGFEMs) requires the implementation of inter-element flux terms, which involves combinations of both inner– and outer–traces of the FEM solution, relative to a given orientation of the element face. In recent years, DGFEMs have become an increasingly popular class of FEMs, most notably due to their local conservation properties, inherent numerical stability for convection–dominated diffusion problems, limited interelement communication, which is restricted only to neighbouring elements, and has important advantages for the implementation of boundary conditions and the parallel efficiency of the method, and finally the ease in which so–called hanging nodes can be treated, and the efficient implementation of *hp*–adaptivity. Indeed, tremendous progress has been made on both the analytical and computational aspects of DGFEMs; for a review of some of the key developments in the subject, we refer to the recent monographs [13, 14, 24, 35]. For a historical review of DGFEMs, we refer to the articles [3, 12], and the references cited therein.

The addition of inter-element face terms within DGFEMs further complicates the implementation of such schemes within standard FEM packages. Assuming for a moment that the underlying PDE problem is linear, we note that for a given interior face, shared by two neighbouring elements, there are, in general, four local face matrices, which stem from the different combinations of traces of the FEM solution from the two elements whose boundaries form the current face. Once these local face–wise matrices have been assembled, they can then be inserted into the global FEM matrix in an analogous manner to the treatment of the element stiffness matrix. On boundary faces, contributions to the load vector must also be computed. Again, in the nonlinear setting, the task of computing both the residual vector and Fréchet derivative(s) of

the DGFEM scheme is a very technical and time consuming task.

The purpose of this article is to discuss the use of symbolic algebra to facilitate the assembly of FEM matrix problems, and in particular those arising from the application of DGFEMs, for the numerical approximation of general nonlinear systems of PDEs. The general approach is to develop a high level language syntax, which closely corresponds to the mathematical formulation of the underlying FEM. Thereby, through this layer of abstraction, the user needs to only specify the FEM residual in a concise and easy to read manner, whereby the evaluation of the Fréchet derivative(s) of the scheme are automatically computed symbolically and the resulting low level C++/Fortran code for element stiffness and face matrices and load/residual vectors are automatically generated by the so-called *form compiler*. In this way, any existing open source FEM package may be utilised, subject to the implementation of a suitable interface which directly calls the automatically generated snippets of code when assembling element and face matrices and load/residual vectors. Most importantly, we stress that once the user has selected a particular FEM package which is appropriate for their purposes, the interface to that software platform which links to the automatically generated code only needs to be written and debugged *once*; it may then subsequently be exploited to solve a potentially huge variety of PDE problems, with a plethora of FEM schemes. At this point it is pertinent to mention that some FEM packages do indeed include such a symbolic interface at the heart of their design; most notably we mention the excellent FEniCS package, cf. [1, 31], for example. The Unified Form Language (UFL) component of FEniCS provides an easy to use python interface which allows for the automatic FEM numerical approximation of systems of PDEs in a user-friendly manner. Other form compilers include SyFi [2], and Manycore Form Compiler [32], for example.

However, even exploiting a package such as UFL, as powerful as it is, the definition of DGFEMs in this framework is still rather verbose, particularly for nonlinear systems of coupled PDEs. With this in mind, we present a further layer of abstraction for the automatic computation of DGFEM formulations employing symbolic algebra. Indeed, in this article DGFEM utility functions for general PDE operators are developed, which significantly simplifies the specification of the DGFEM discretisation of a given problem. This work was originally inspired by the need to numerically model the formation of a hydrogen plasma in a microwave power assisted chemical vapour deposition reactor employed for the manufacture of synthetic diamond; this includes constituent equations for the background gas mass average velocity, gas temperature, electromagnetic field energy and plasma density, cf. [28, 36].

In [36] we originally developed easy-to-use DGFEM utility functions as part of our inhouse software package AptoPy [36], for application with our own FEM package AptoFEM [26]. AptoPy is written in Python and exploits the open source symbolic algebra package SymPy [37]. We stress that these choices are entirely user-dependent; indeed, in the past we have employed the symbolic algebra packages REDUCE [22], Mathematica, and Maple. In addition to AptoFEM, ENTWIFE has also been employed as the low level FEM package. The DGFEM utility functions written in AptoPy have been ported to UFL for use within the FEniCS package; full open source codes are available from `https://bitbucket.org/nate-sime/dolfin_dg`. With this in mind, for simplicity of presentation, throughout this article, we only show snippets of UFL code in order to highlight how the DGFEM utility functions may be exploited in practice; the corresponding AptoPy syntax is quite similar, cf. [36].

The outline of this article is as follows. In Section 2 we briefly outline the DGFEM

discretisation of general systems of nonlinear conservation laws. Then, in Section 3 we propose a computational framework for the automatic generation of DGFEM schemes within a simple unified setting. On the basis of this work, in Section 4 we provide some examples to illustrate the flexibility and ease within which systems of PDEs may be numerically approximated using the software developed in this article. Finally, in Section 5 we provide a summary of the work undertaken in this article, as well as outlining potential future developments.

2. Discontinuous Galerkin Finite Element Methods. As a representative PDE example, in this section we outline the DGFEM discretisation for the following system of conservation laws:

$$\nabla \cdot (\mathcal{F}^c(\mathbf{u}) - \mathcal{F}^v(\mathbf{u}, \nabla \mathbf{u})) = \mathbf{f} \quad \text{in } \Omega, \tag{2.1}$$

where Ω is an open bounded domain in $\mathbb{R}^d$, $d \geq 1$, with boundary Γ. Here, $\mathbf{u} = (u_1, \ldots, u_m)^\top$, $m \geq 1$, $\mathcal{F}^c(\mathbf{u}) = (\mathbf{f}_1^c(\mathbf{u}), \ldots, \mathbf{f}_d^c(\mathbf{u}))$ and $\mathcal{F}^v(\mathbf{u}, \nabla \mathbf{u}) = (\mathbf{f}_1^v(\mathbf{u}, \nabla \mathbf{u}), \ldots, \mathbf{f}_d^v(\mathbf{u}, \nabla \mathbf{u}))$ represent the convective and diffusive fluxes, respectively, which are assumed to be continuously differentiable, and $\mathbf{f}$ is a given source function. For simplicity of presentation, we assume that (2.1) may be supplemented with the boundary conditions:

$$\mathbf{u} = \mathbf{g}_D \quad \text{on } \Gamma_D, \tag{2.2}$$

$$\mathcal{F}^v(\mathbf{u}, \nabla \mathbf{u}) \cdot \mathbf{n} = \mathbf{g}_N \quad \text{on } \Gamma_N, \tag{2.3}$$

where $\Gamma = \Gamma_D \cup \Gamma_N$, and Γ_D and Γ_N are two disjoint subsets, with Γ_D nonempty and relatively open in Γ. We stress that more general boundary conditions can also be considered; for example, in the case when (2.1) represents the compressible Euler or Navier-Stokes equations, we refer to, for example, [17] and [20], respectively.

For the purposes of discretisation, we rewrite (2.1) in the following equivalent form:

$$\nabla \cdot (\mathcal{F}^c(\mathbf{u}) - G(\mathbf{u}) \nabla \mathbf{u}) \equiv \frac{\partial}{\partial x_k} \left(\mathbf{f}_k^c(\mathbf{u}) - G_{kl}(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x_l} \right) = \mathbf{f} \quad \text{in } \Omega.$$

Here, the matrices

$$G_{kl}(\mathbf{u}) = \partial \mathbf{f}_k^v(\mathbf{u}, \nabla \mathbf{u}) / \partial u_{x_l}, \quad k, l = 1, \ldots, d, \tag{2.4}$$

are the homogeneity tensors defined by $\mathbf{f}_k^v(\mathbf{u}, \nabla \mathbf{u}) = G_{kl}(\mathbf{u}) \partial \mathbf{u} / \partial x_l$, $k = 1, \ldots, d$. We write the homogeneity tensor product

$$(G(\mathbf{u}) \nabla \mathbf{u})_{ik} = \sum_{j=1}^{m} \sum_{l=1}^{d} (G_{kl}(\mathbf{u}))_{ij} \frac{\partial \mathbf{u}_j}{\partial x_l} \tag{2.5}$$

such that $\mathcal{F}^v(\mathbf{u}, \nabla \mathbf{u}) = G(\mathbf{u}) \nabla \mathbf{u}$.

For simplicity of presentation, we now proceed to discretise (2.1)–(2.3) based on employing the symmetric interior penalty (SIPG) formulation presented in [20]; however, we stress that other DGFEMs can easily be included within this general setting, cf. below. To this end, we partition Ω into a mesh $\mathcal{T}_h = \{\kappa\}$ consisting of non-overlapping open element domains κ, such that $\bar{\Omega} = \cup_{\kappa \in \mathcal{T}_h} \bar{\kappa}$. For each $\kappa \in \mathcal{T}_h$, we denote by $\mathbf{n}_\kappa$ the unit outward normal vector to the boundary $\partial \kappa$. We assume that each $\kappa \in \mathcal{T}_h$ is an image of a fixed reference element $\hat{\kappa}$, that is, $\kappa = \sigma_\kappa(\hat{\kappa})$ for all

$\kappa \in \mathcal{T}_h$. On the reference element $\hat{\kappa}$ we define spaces of polynomials of degree $\ell \geq 0$ as follows:

$$\mathcal{Q}_\ell = \text{span}\left\{\hat{\mathbf{x}}^\alpha : 0 \leq \alpha_i \leq \ell,\ 0 \leq i \leq d\right\}, \qquad \mathcal{P}_\ell = \text{span}\left\{\hat{\mathbf{x}}^\alpha : 0 \leq |\alpha| \leq \ell\right\}.$$

Thereby, with this notation, we define the following DGFEM finite element space

$$\begin{aligned}\mathbf{V}_\ell^m(\mathcal{T}_h) = \{\mathbf{v} \in [L_2(\Omega)]^m : \mathbf{v}|_\kappa \circ \sigma_\kappa \in [\mathcal{Q}_\ell]^m \ \text{ if } \hat{\kappa} = \sigma_\kappa^{-1}(\kappa) \text{ is the unit hypercube,} \\ \text{and } \mathbf{v}|_\kappa \circ \sigma_\kappa \in [\mathcal{P}_\ell]^m \ \text{ if } \hat{\kappa} = \sigma_\kappa^{-1}(\kappa) \text{ is the unit simplex; } \kappa \in \mathcal{T}_h\}.\end{aligned}$$

An *interior face* of $\mathcal{T}_h$ is defined as the (non-empty) $(d-1)$–dimensional interior of $\partial\kappa^+ \cap \partial\kappa^-$, where κ^+ and κ^- are two adjacent elements of $\mathcal{T}_h$, not necessarily matching. A *boundary face* f of $\mathcal{T}_h$ is defined as a (non-empty) $(d-1)$–dimensional facet of κ, $\kappa \in \mathcal{T}_h$, where κ is a boundary element of $\mathcal{T}_h$, such that $f \subset \partial\kappa \cap \Gamma$. We denote by $\Gamma_{\mathcal{I}}$ the union of all interior faces of $\mathcal{T}_h$. Let κ^+ and κ^- be two adjacent elements of $\mathcal{T}_h$, and $\mathbf{x}$ an arbitrary point on the interior face $f = \partial\kappa^+ \cap \partial\kappa^-$. Furthermore, let $\mathbf{v}$ and $\underline{\tau}$ be vector- and matrix-valued functions, respectively, that are smooth inside each element $\kappa^\pm$. By $(\mathbf{v}^\pm, \underline{\tau}^\pm)$, we denote the traces of $(\mathbf{v}, \underline{\tau})$ on f taken from within the interior of $\kappa^\pm$, respectively. Then, the averages of $\mathbf{v}$ and $\underline{\tau}$ at $\mathbf{x} \in f$ are given by $\{\!\{\mathbf{v}\}\!\} = (\mathbf{v}^+ + \mathbf{v}^-)/2$ and $\{\!\{\underline{\tau}\}\!\} = (\underline{\tau}^+ + \underline{\tau}^-)/2$, respectively. Similarly, the jump of $\mathbf{v}$ at $\mathbf{x} \in f$ is given by $\underline{[\![\mathbf{v}]\!]} = \mathbf{v}^+ \otimes \mathbf{n}_{\kappa^+} + \mathbf{v}^- \otimes \mathbf{n}_{\kappa^-}$, where we denote by $\mathbf{n}_{\kappa^\pm}$ the unit outward normal vector of $\kappa^\pm$, respectively. On $f \subset \Gamma$, we set $\{\!\{\mathbf{v}\}\!\} = \mathbf{v}$, $\{\!\{\underline{\tau}\}\!\} = \underline{\tau}$ and $\underline{[\![\mathbf{v}]\!]} = \mathbf{v} \otimes \mathbf{n}$, where $\mathbf{n}$ denotes the unit outward normal vector to Γ.

The interior penalty DGFEM discretisation of (2.1)–(2.3) is given by: find $\mathbf{u}_h \in \mathbf{V}_\ell^m(\mathcal{T}_h)$ such that

$$\begin{aligned}&\mathcal{N}(\mathbf{u}_h, \mathbf{v}_h) \\ &\equiv -\int_\Omega \mathbf{f} \cdot \mathbf{v}_h \,\mathrm{d}\mathbf{x} - \int_\Omega \mathcal{F}^c(\mathbf{u}_h) : \nabla_h \mathbf{v}_h \,\mathrm{d}\mathbf{x} + \sum_{\kappa \in \mathcal{T}_h} \int_{\partial\kappa \setminus \Gamma} \mathcal{H}(\mathbf{u}_h^+, \mathbf{u}_h^-, \mathbf{n}^+) \cdot \mathbf{v}_h^+ \,\mathrm{d}s \\ &\quad + \int_\Omega \mathcal{F}^v(\mathbf{u}_h, \nabla_h \mathbf{u}_h) : \nabla_h \mathbf{v}_h \,\mathrm{d}\mathbf{x} - \int_{\Gamma_{\mathcal{I}}} \{\!\{\mathcal{F}^v(\mathbf{u}_h, \nabla_h \mathbf{u}_h)\}\!\} : \underline{[\![\mathbf{v}_h]\!]} \,\mathrm{d}s \\ &\quad - \int_{\Gamma_{\mathcal{I}}} \{\!\{G^\top(\mathbf{u}_h) \nabla_h \mathbf{v}_h\}\!\} : \underline{[\![\mathbf{u}_h]\!]} \,\mathrm{d}s + \int_{\Gamma_{\mathcal{I}}} \underline{\delta}(\mathbf{u}_h) : \underline{[\![\mathbf{v}_h]\!]} \,\mathrm{d}s + \mathcal{N}_\Gamma(\mathbf{u}_h, \mathbf{v}_h) = 0 \quad (2.6)\end{aligned}$$

for all $\mathbf{v}_h$ in $\mathbf{V}_\ell^m(\mathcal{T}_h)$. The subscript h on the operator ∇_h is used to denote the discrete counterpart of ∇, defined elementwise. Here, $\mathcal{H}(\cdot,\cdot,\cdot)$ denotes the (convective) numerical flux function; this may be chosen to be any two–point monotone Lipschitz function which is both consistent and conservative; typical choices include the (local) Lax–Friedrichs flux, the HLLE flux, the Roe flux, and the Vijayasundaram flux, cf. [30, 39].

The penalisation term $\underline{\delta}(\cdot)$ arising in the DGFEM scheme (2.6) is given by

$$\underline{\delta}(\mathbf{u}_h) = \ C_{\text{IP}} \tfrac{\ell^2}{\mathtt{h}} \{\!\{G(\mathbf{u}_h)\}\!\} \underline{[\![\mathbf{u}_h]\!]},$$

where C_{IP} is a (sufficiently large) positive constant, cf. [15]. Moreover, $\mathtt{h} \in L_\infty(\Gamma_{\mathcal{I}} \cup \Gamma)$ is defined as $\mathtt{h}(\mathbf{x}) = \min\{m_{\kappa^+}, m_{\kappa^-}\}/m_f$, if $\mathbf{x}$ is in the interior of $f = \partial\kappa^+ \cap \partial\kappa^-$ for two neighbouring elements in the mesh $\mathcal{T}_h$, and $\mathtt{h}(\mathbf{x}) = m_\kappa/m_f$, if $\mathbf{x}$ is in the interior of $f = \partial\kappa \cap \Gamma$. Here, for a given (open) bounded set $\omega \subset \mathbb{R}^s$, $s \geq 1$, we write m_ω to denote the s–dimensional measure of ω.

Finally, we define the boundary terms present in the form $\mathcal{N}_\Gamma(\cdot,\cdot)$ by

$$\begin{aligned}\mathcal{N}_\Gamma(\mathbf{u}_h,\mathbf{v}_h) = &\int_\Gamma \mathcal{H}_\Gamma(\mathbf{u}_h^+,\mathbf{u}_\Gamma(\mathbf{u}_h^+),\mathbf{n})\cdot\mathbf{v}_h^+\,\mathrm{d}s + \int_{\Gamma_D}\underline{\delta}_\Gamma(\mathbf{u}_h^+):\mathbf{v}_h\otimes\mathbf{n}\,\mathrm{d}s\\ &-\int_{\Gamma_N}\mathbf{g}_N\cdot\mathbf{v}_h\,\mathrm{d}s - \int_{\Gamma_D}\mathbf{n}\cdot\mathcal{F}_\Gamma^v(\mathbf{u}_h^+,\nabla_h\mathbf{u}_h^+)\,\mathbf{v}_h^+\,\mathrm{d}s\\ &-\int_{\Gamma_D}\left(G_\Gamma^\top(\mathbf{u}_h^+)\nabla_h\mathbf{v}_h^+\right):\left(\mathbf{u}_h^+-\mathbf{u}_\Gamma(\mathbf{u}_h^+)\right)\otimes\mathbf{n}\,\mathrm{d}s,\end{aligned}$$

where $\underline{\delta}_\Gamma(\mathbf{u}_h) = C_{\mathrm{IP}}\frac{\ell^2}{\mathrm{h}}G_\Gamma(\mathbf{u}_h^+)\left(\mathbf{u}_h-\mathbf{u}_\Gamma(\mathbf{u}_h)\right)\otimes\mathbf{n}$. Here, the viscous boundary flux $\mathcal{F}_\Gamma^v$ and the corresponding homogeneity tensor G_Γ are defined by

$$\mathcal{F}_\Gamma^v(\mathbf{u}_h,\nabla\mathbf{u}_h) = \mathcal{F}^v(\mathbf{u}_\Gamma(\mathbf{u}_h),\nabla\mathbf{u}_h) = G_\Gamma(\mathbf{u}_h)\nabla\mathbf{u}_h = G(\mathbf{u}_\Gamma(\mathbf{u}_h))\nabla\mathbf{u}_h.$$

The convective boundary flux $\mathcal{H}_\Gamma$ is defined by

$$\mathcal{H}_\Gamma(\mathbf{u}_h^+,\mathbf{u}_\Gamma(\mathbf{u}_h^+),\mathbf{n}) = \mathbf{n}\cdot\mathcal{F}^c(\mathbf{u}_\Gamma(\mathbf{u}_h^+)).$$

Finally, the boundary function $\mathbf{u}_\Gamma(\mathbf{u})$ is given according to the type of boundary condition imposed; in the current setting $\mathbf{u}_\Gamma(\mathbf{u}) = \mathbf{g}_D$ on Γ_D and $\mathbf{u}_\Gamma(\mathbf{u}) = \mathbf{u}$ on Γ_N. For further details regarding the imposition of more general boundary conditions, we refer to [20], for example.

3. Computational Framework for DGFEMs. In this section we present the general computational framework for the automatic generation of DGFEM (semi-) linear forms in a concise and easy to use manner. We begin by outlining the treatment of both convective and viscous components arising in conservation laws. In this setting, the DGFEM formulations can be constructed in a consistent manner. We exploit this by designing utility functions to automatically generate these symbolic DGFEM formulations. We then proceed by proposing a hierarchical framework for computing DGFEM formulations of PDE operators. This hierarchy takes advantage of the DGFEM utility functions, providing a modular framework for a suite of DGFEM formulations for various operators arising in PDE problems of engineering interest.

3.1. Automatic Treatment of Convective Terms. In this section we discuss the automatic symbolic representation of the DGFEM discretisation of the term involving the convective flux function $\mathcal{F}^c(\cdot)$. To this end, we recall that the convective numerical flux function $\mathcal{H}(\cdot,\cdot,\cdot)$, cf. (2.6), must be defined on the element interfaces. Defining a callable function `F_c` which specifies $\mathcal{F}^c(\cdot)$ we construct the abstraction of the numerical flux function as shown in Listing 1. The methods `interior` and `exterior` will be called to construct the flux function on interior and exterior faces, respectively. The method `setup` is provided for the inheriting class to initialise any members prior to calls to `interior` and `exterior`. This design allows for the flux function to be different on the two types of faces present in the mesh.

For simplicity, here we consider two implementations of the `ConvectiveFlux` class, based on employing the local Lax-Friedrichs and HLLE fluxes, though we stress that other fluxes, such as the Vijayasundaram or Roe flux, for example, may also be employed. Thereby, on the boundary $\partial\kappa$ of an element $\kappa\in\mathcal{T}_h$, we define the local-

```
class ConvectiveFlux:

    def __init__(self):
        pass

    def setup(self):
        pass

    def interior(self, F_c, u_p, u_m, n):
        pass

    def exterior(self, F_c, u_p, u_m, n):
        pass
```

Listing 1: The abstraction of the numerical flux function $\mathcal{H}(\cdot,\cdot,\cdot)$.

Lax Friedrichs flux $\mathcal{H}_{\mathrm{LF}}$ and HLLE flux $\mathcal{H}_{\mathrm{HLLE}}$ by

$$\mathcal{H}_{\mathrm{LF}}\left(\mathbf{u}_h^+,\mathbf{u}_h^-,\mathbf{n}_\kappa\right)\big|_{\partial\kappa} = \tfrac{1}{2}\left(\mathcal{F}^c\left(\mathbf{u}_h^+\right)\cdot\mathbf{n}_\kappa + \mathcal{F}^c\left(\mathbf{u}_h^-\right)\cdot\mathbf{n}_\kappa + \alpha\left(\mathbf{u}_h^+ - \mathbf{u}_h^-\right)\right), \quad (3.1a)$$

$$\mathcal{H}_{\mathrm{HLLE}}\left(\mathbf{u}_h^+,\mathbf{u}_h^-,\mathbf{n}_\kappa\right)\big|_{\partial\kappa} = \tfrac{1}{\lambda^+-\lambda^-}\lambda^+\mathcal{F}^c(\mathbf{u}_h^+)\cdot\mathbf{n}_\kappa - \lambda^-\mathcal{F}^c(\mathbf{u}_h^-)\cdot\mathbf{n}_\kappa - \lambda^+\lambda^-(\mathbf{u}_h^+ - \mathbf{u}_h^-), \quad (3.1b)$$

respectively. Here, α is the local dissipation parameter, which is selected to be the maximum of the (absolute value) of the eigenvalues of the Jacobi matrix

$$B(\mathbf{u},\mathbf{n}_\kappa) = \sum_{i=1}^{d}\frac{\partial\mathbf{f}_i^c}{\partial\mathbf{u}}\mathbf{n}_{\kappa,i},$$

evaluated on the element face. More precisely, we have that

$$\alpha|_{\partial\kappa} = \max_{\mathbf{w}=\mathbf{u}_h^+,\mathbf{u}_h^-}\left\{\left|\lambda_{\max}\left(B\left(\mathbf{w},\mathbf{n}_\kappa\right)\right)\right|\right\}$$

, where $\lambda_{\max}$ denotes the largest eigenvalue (in modulus) of $B\left(\cdot,\mathbf{n}_\kappa\right)$. Additionally,

$$\lambda^+ = \max\left(\max_{\mathbf{w}=\mathbf{u}_h^+,\mathbf{u}_h^-}\left\{\lambda_{\max}\left(B\left(\mathbf{w},\mathbf{n}_\kappa\right)\right)\right\},0\right),$$

$$\lambda^- = \min\left(\min_{\mathbf{w}=\mathbf{u}_h^+,\mathbf{u}_h^-}\left\{\lambda_{\min}\left(B\left(\mathbf{w},\mathbf{n}_\kappa\right)\right)\right\},0\right),$$

where $\lambda_{\min}$ denotes the smallest eigenvalue (in modulus) of $B\left(\cdot,\mathbf{n}_\kappa\right)$.

Given `F_c`, which specifies $\mathcal{F}^c\left(\cdot\right)$, the numerical flux functions $\mathcal{H}_{\mathrm{LF}}(\cdot,\cdot,\cdot)$ and $\mathcal{H}_{\mathrm{HLLE}}(\cdot,\cdot,\cdot)$ can be automatically generated in order to yield the discretisation of the convective term present in the underlying PDE problem; we note that the constructors of both classes `LocalLaxFriedrichs` and `HLLE` require the symbolic representation of the eigenvalues of B. As an example, we consider the application of the DGFEM employing the local Lax-Friedrichs flux for the numerical approximation of the linear advection equation

$$\nabla\cdot(\mathbf{b}u) = f, \quad (3.2)$$

```
def F_c(u): return b*u
H = LocalLaxFriedrichs(lambda u, n: dot(b, n))
H.setup(F_c, u('+'), u('-'), n('+'))
conv_interior = H.interior(F_c, u('+'), u('-'), n('+'))*(v('+') - v('-'))*dS
```

Listing 2: Example of the automatic calculation and symbolic representation of the local Lax-Friedrichs flux $\mathcal{H}_{\mathrm{LF}}(u^+, u^-, \mathbf{n}_\kappa)$ for the linear advection equation shown in (3.2).

```
from sympy import *

dim = 2
u = Matrix([Symbol("u%d" % d, real=True) for d in range(dim)])
n = Matrix([Symbol("n%d" % d, real=True) for d in range(dim)])

F_c = u*u.T

B = zeros(dim, dim)
for d in range(dim):
  B += F_c[:, d].jacobian(u)*n[d]

print(B.eigenvals().keys())
```

Listing 3: Automatic symbolic algebra computation of flux Jacobian eigenvalues required by the local Lax-Friedrichs and HLLE fluxes applied to the incompressible Navier- Stokes convective flux component $\mathcal{F}^c(\mathbf{u}) = \mathbf{u} \otimes \mathbf{u}$.

where $\mathbf{b} = (b_1, b_2, \ldots, b_d)^\top$, $\mathcal{F}^c(u) = \mathbf{b}u$, and f is some given forcing function. Here, the dissipation parameter can be shown to be $\alpha|_{\partial\kappa} = |\mathbf{b} \cdot \mathbf{n}_\kappa|$. The UFL code required to generate the numerical flux function, together with the corresponding element boundary term arising in the DGFEM scheme for this problem is given in Listing 2. We note that the use of the class `HLLE` defining the HLLE numerical flux follows in an analogous manner.

In many cases an analytical expression for the eigenvalues of the Jacobi matrix $B(\mathbf{u}, \mathbf{n}_\kappa)$, $\kappa \in \mathcal{T}_h$, may be difficult to evaluate; thereby, packages such as SymPy [37] may be used to compute them symbolically. To this end, $B(\mathbf{u}, \mathbf{n}_\kappa)$ can be assembled using symbolic differentiation; the eigenvalues of this matrix may then be computed symbolically by exploiting the Berkowitz algorithm [7]. In Listing 3 we show an example of this method applied to the convective component of the incompressible Navier-Stokes equations, i.e., $\mathcal{F}^c(\mathbf{u}) = \mathbf{u} \otimes \mathbf{u}$.

3.2. Automatic Treatment of Viscous Terms. In this section we develop utility functions which automatically generate the DGFEM discretisation of second–order PDE operators. To this end, we recall from Section 2 that the viscous component of the underlying PDE is given by

$$-\nabla \cdot \mathcal{F}^v(\mathbf{u}, \nabla\mathbf{u}) = \mathbf{0}; \tag{3.3}$$

```
class DGFemSIPG(DGFemViscousTerm):

  def interior_residual(self, dInt):
    G = self.G
    F_v, u, v, grad_v = self.F_v, self.U, self.V, self.grad_v_vec
    sig, n = self.sig, self.n

    residual = \
      - inner(tensor_jump(u, n), avg(hyper_tensor_T_product(G, grad_v)))*dInt
          \
      - inner(ufl_adhere_transpose(avg(F_v(U))), tensor_jump(v, n))*dInt \
      + inner(sig('+')*hyper_tensor_product(g_avg(G), tensor_jump(u, n)),
          tensor_jump(v, n))*dInt
    return residual

  def exterior_residual(self, u_gamma, dExt):
    G = self._make_boundary_G(self.G, u_gamma)
    F_v, u, v, grad_u, grad_v = self.F_v, self.U, self.V, grad(self.U), self.
        grad_v_vec
    n = self.n

    residual = \
      - inner(dg_outer(u - u_gamma, n), hyper_tensor_T_product(G, grad_v)) *
          dExt \
      - inner(hyper_tensor_product(G, grad_u), dg_outer(v, n)) * dExt \
      + inner(self.sig*hyper_tensor_product(G, dg_outer(u - u_gamma, n)),
          dg_outer(v, n)) * dExt
    return residual
```

Listing 4: The class `DGFemViscousTerm` provides the abstract interface for interior and exterior residual formulation using symbolic algebra. The class `DGFemSIPG` uses UFL to automatically formulate the interior and exterior terms of the SIPG formulation of the viscous component of (2.6).

here, we have set the right-hand in (3.3) to zero, for simplicity, in order to concentrate on the discretisation of the viscous term. The semilinear DGFEM formulation of (3.3), shown in equation (2.6), is encapsulated by implementations of the abstract class

$$\texttt{DGFemViscousTerm(F\_v, u, v, gamma, G, n)},$$

where

$$\texttt{F\_v(u, grad\_u)} = \mathcal{F}^v(\mathbf{u}, \nabla\mathbf{u}) \tag{3.4}$$

is a callable function, which defines the viscous flux function, `u` and `v` denote the trial and test functions, respectively, $\texttt{gamma} \equiv \gamma = C_{\text{IP}}\ell^2/\mathrm{h}$ is the DGFEM penalisation coefficient, $\texttt{G} \equiv G(\mathbf{u})$ is the homogeneity tensor, and $\texttt{n} \equiv \mathbf{n}$ is the face normal.

Recalling the definition of the homogeneity tensor (2.4) and the homogeneity tensor product (2.5), $G(\mathbf{u})$ is automatically computed using the function `homogeneity_tensor (F_v, u)` and the function `hyper_tensor_product(G, tau)` computes the homogeneity

```
def F_v(u, grad_u): return (u + 1)*grad(u)
G = homogeneity_tensor(F_v, u)
vt = DGFemSIPG(F_v, u, v, sig, G, n)
residual = dot((u + 1)*grad(u), grad(v))*dx \
         + vt.interior_residual(dS) \
         + vt.exterior_residual(g_D, ds_D) \
         + vt.neumann_residual(g_N, ds_N) \
         - f*v*dx
```

Listing 5: Example UFL code for the DGFEM discretisation of the quasi-linear PDE (3.5) using the `DGFemSIPG` utility class.

tensor product. These functions may then be employed for the generation of the corresponding DGFEM formulation. On the basis of these two functions, we introduce the abstract class `DGFemViscousTerm`; this offers the following three methods for handling each of the boundary components present in the DGFEM scheme:

1. `DGFemViscousTerm.interior_residual(dS)` automatically generates terms associated with the interior boundaries $\Gamma_{\mathcal{I}}$ present in (2.6);
2. `DGFemViscousTerm.exterior_residual(u_gamma, ds_i)` generates the terms associated with exterior boundary component `ds_i` present in (2.7) with boundary condition $\mathbf{u}_\Gamma(\mathbf{u}) =$ `u_gamma`;
3. Finally, `DGFemViscousTerm.neumann_residual(gN, ds_i)` generates any terms arising from Neumann boundary conditions with flux specification $\mathcal{F}^v(\mathbf{u}, \nabla\mathbf{u})\cdot\mathbf{n} = \mathbf{g}_N$.

We demonstrate an example of the implementation of the `DGFemViscousTerm` with the SIPG method in Listing 4; for the sake of brevity, we have removed input and error checking. Further implementations such as the non–symmetric interior penalty (NIPG) and Baumann–Oden schemes are available in the classes `DGFemNIPG` and `DGFemBO`, respectively. The DGFEM formulation proposed by Bassi & Rebay [6] is challenging in the symbolic framework due to the requirement of solving element-wise problems for the local lifting operator. Such operations are not easily formulated in the UFL for use with DOLFIN, for example; the implementation of DGFEMs defined based on employing local lifting operators will be considered as part of our programme of future research. For a more detailed discussion of the formulation of lifting operators in DOLFIN, we refer to [34, Chapter 5].

An example of the UFL code required to generate the DGFEM semilinear residual discretisation of the quasi-linear second-order PDE problem

$$-\nabla\cdot((u+1)\nabla u) = f \text{ in } \Omega, \tag{3.5a}$$

$$u = g_D \text{ on } \Gamma_D, \tag{3.5b}$$

$$\nabla u\cdot\mathbf{n} = g_N \text{ on } \Gamma_N \tag{3.5c}$$

is shown in Listing 5.

3.3. Automatic Generation of DGFEM Formulations. Even with the utility functions outlined in Sections 3.1 and 3.2, the specification of the DGFEM scheme can be very verbose; this is particularly the case when discretising systems comprising many PDE variables with multiple boundary conditions. In this section we propose a hierarchical scheme for the management of DGFEM formulations of PDE operators

```
bcs = [DGDirichletBC(bc_1, ds_1), DGDirichletBC(bc_2, ds_2), ...]
vt = DGFemViscousTerm(F_v, u, v, delta)
ext = sum(vt.exterior_residual(bc.get_function(), bc.get_boundary()) for bc
    in bcs)
```

Listing 6: Example of the automatic generation of the exterior residual terms of a given `DGFemViscousTerm` for a list of Dirichlet boundary conditions.

of increasing complexity.

3.3.1. Boundary Conditions. Firstly, we outline a simple framework for managing the boundary conditions enforced within a given PDE problem. We define the `DGBC` (discontinuous Galerkin boundary condition) abstract class from which the classes `DGDirichletBC` and `DGNeumannBC` inherit. These implementations simply serve to store the boundary condition and the boundary component over which the condition should be enforced. For example, applying a Dirichlet boundary condition as required by the quasi-linear PDE problem stated in (3.5) simply requires instantiation of `DGDirichletBC(ds_D, g_D)`. Similarly, for the imposition of the Neumann boundary condition we construct `DGNeumannBC(ds_N, g_N)`; note that Robin conditions can be implemented in an analogous manner. By generating a list of boundary conditions in this manner, we may easily formulate their imposition within the DGFEM scheme. As an example of a series of Dirichlet boundary conditions being automatically generated using `DGFemViscousTerm`, we refer to Listing 6.

3.3.2. Abstract DGFEM Formulation. We encapsulate the abstraction of a DGFEM scheme in the class `DGFemFormulation` which prescribes one abstract method `generate_fem_formulation`. The `DGFemFormulation` constructor requires the mesh $\mathcal{T}_h$, the function space $\mathbf{V}_\ell^m$ and the list of boundary conditions. This class will serve as the base class for the DGFEM formulation of all derived PDE operators. In this work we describe two direct children of `DGFemFormulation`: `HyperbolicOperator` and `EllipticOperator`. We stress that the flexibility of this design permits DGFEM formulations of other PDE operators, cf. Sections 4.6, 4.7, and 4.8 below.

The class `HyperbolicOperator` inherits `DGFemFormulation` and its purpose is to generate DGFEM formulations of the PDE operator $\nabla \cdot \mathcal{F}^c(\cdot)$. The class implementation is shown in Listing 7; here, `generate_fem_formulation` is overridden to construct the DGFEM formulation of the provided convective flux operator on the interior and exterior faces, as well as on the elements in the mesh. The numerical flux function provided in the constructor `H`, which implements `ConvectiveFlux`, is used for the DGFEM formulation. To highlight to modularity of this design, consider an extension of the `HyperbolicOperator` for the time-dependent Burgers' equation

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(1/2\, u^2) = 0. \tag{3.6}$$

Setting $t = y$, we may recast (3.6) in the following equivalent form

$$\nabla \cdot \mathcal{F}^c(u) \equiv \nabla \cdot \begin{pmatrix} \frac{1}{2}u^2 \\ u \end{pmatrix} = 0; \tag{3.7}$$

the implementation of the class `SpacetimeBurgersOperator` (using `LocalLaxFriedrichs` by default) is depicted in Listing 8. Here, we note that the derived class must simply

```
class HyperbolicOperator(DGFemFormulation):

    def __init__(self, mesh, V, bcs,
                 F_c=lambda u: u,
                 H=LocalLaxFriedrichs(lambda u, n: inner(u, n))):
        DGFemFormulation.__init__(self, mesh, V, bcs)
        self.F_c = F_c
        self.H = H

    def generate_fem_formulation(self, u, v, dx=None, dS=None):
        if dx is None:
            dx = Measure('dx', domain=self.mesh)
        if dS is None:
            dS = Measure('dS', domain=self.mesh)
        n = FacetNormal(self.mesh)

        residual = -inner(self.F_c(u), grad(v))*dx

        self.H.setup(self.F_c, u('+'), u('-'), n('+'))
        residual += inner(self.H.interior(self.F_c, u('+'), u('-'), n('+')),
            (v('+') - v('-')))*dS

        for bc in self.dirichlet_bcs:
            gD = bc.get_function()
            dSD = bc.get_boundary()

            self.H.setup(self.F_c, u, gD, n)
            residual += inner(self.H.exterior(self.F_c, u, gD, n), v)*dSD

        for bc in self.neumann_bcs:
            dSN = bc.get_boundary()

            residual += inner(dot(self.F_c(u), n), v)*dSN

        return residual
```

Listing 7: The `HyperbolicOperator` class.

specify the form of the convective flux $\mathcal{F}^c(u)$ and the flux function $\mathcal{H}(\cdot,\cdot,\cdot)$; indeed, once these are defined, the automatic generation of the DGFEM formulation is then handled by the parent `DGFemFormulation`.

Secondly, we introduce the class `EllipticOperator`, which inherits `DGFemFormulation`, and requires the specification of $\mathcal{F}^v(\cdot,\nabla\cdot)$ at instantiation. The overridden method `generate_fem_formulation` is then written to automatically generate the interior and exterior boundary, as well as the element, integration terms, implementing all of the concepts of the utility functions for elliptic operators in `DGFemViscousTerm`. The outline of the `EllipticOperator` class, which generates the SIPG formulation by default, is given in Listing 9. To highlight the modularity of `EllipticOperator` in

```
class SpacetimeBurgersOperator(HyperbolicOperator):

    def __init__(self, mesh, V, bcs, flux=None):

        def F_c(u):
            return as_vector((u**2/2, u))

        if flux is None:
            flux = LocalLaxFriedrichs(lambda u, n: u*n[0] + n[1])

        HyperbolicOperator.__init__(self, mesh, V, bcs, F_c, flux)
```

Listing 8: The `SpacetimeBurgersOperator` class.

Listing 10 we show the implementation of the class `PoissonOperator` which specifies $\mathcal{F}^v(u, \nabla u) = \mathcal{K}\nabla u$, where $\mathcal{K}$ is the diffusion coefficient. An example of the automatic generation of the DGFEM formulation of the quasi-linear elliptic PDE problem (3.5) is given in Listing 11.

3.3.3. Hierarchy of PDE Operators. A natural extension of this framework is a hierarchy of automatically generated DGFEM operators. As shown in Listing 8, the `SpacetimeBurgersOperator` inherits from `HyperbolicOperator`. Consider now, for example, the compressible Navier Stokes equations. Here the inheritance chain may begin with the `HyperbolicOperator`, from which a `CompressibleEulerOperator` would inherit, and in turn a `CompressibleNavierStokesOperator` would inherit `CompressibleEulerOperator` and additionally `EllipticOperator` for the viscosity terms. Further implementations of each member of this class hierarchy need not only be undertaken by inheritance. Operators deriving from models of large physical systems may more appropriately aggregate sub-operators as necessary. The derived classes must simply manage the function spaces and boundary conditions amongst the aggregated `DGFemFormulation` members. This framework significantly reduces the code required for subsequent development of DGFEM formulations of PDE operators of increasing complexity. By ensuring that each layer of the hierarchy is correctly verified and fully tested means that DGFEM formulations may be debugged in a very straightforward manner.

4. Examples. In this section we present a series of examples of increasing complexity to highlight the ease in which each PDE problem may be discretised using a DGFEM formulation, based on employing the class hierarchy proposed and implemented within this article. We stress that the verbosity and complexity of specifying each DGFEM discretisation is vastly reduced within this modular framework. To this end, we consider the discretisation of a simple scalar advection-diffusion equation, the compressible Euler equations, the compressible Navier-Stokes equations posed in both conserved and entropy variables, the indefinite Maxwell problem, and a hyperelasticity problem. In each case, for simplicity of presentation, we employ the SIPG DGFEM discretisation of the second-order PDE operator and a local Lax-Friedrichs flux for the numerical approximation of the convective terms. For brevity, in some of the examples given below only code snippets will be shown, however, full versions of the corresponding python scripts are available from `https://bitbucket.org/nate-sime/dolfin_dg`.

```
class EllipticOperator(DGFemFormulation):

    def __init__(self, mesh, fspace, bcs, F_v, C_IP=10.0):
        DGFemFormulation.__init__(self, mesh, fspace, bcs)
        self.F_v = F_v
        self.C_IP = C_IP

    def generate_fem_formulation(self, u, v, dx=None, dS=None, vt=None):
        if dx is None:
            dx = Measure('dx', domain=self.mesh)
        if dS is None:
            dS = Measure('dS', domain=self.mesh)

        h = CellVolume(self.mesh)/FacetArea(self.mesh)
        n = FacetNormal(self.mesh)
        sigma = self.C_IP*Constant(max(self.fspace.ufl_element().degree()**2,
             1))/h
        G = homogeneity_tensor(self.F_v, u)

        if vt is None:
            vt = DGFemSIPG(self.F_v, u, v, sigma, G, n)

        if inspect.isclass(vt):
            vt = vt(self.F_v, u, v, sigma, G, n)

        assert(isinstance(vt, DGFemViscousTerm))

        residual = inner(self.F_v(u, grad(u)), grad(v))*dx
        residual += vt.interior_residual(dS)

        for dbc in self.dirichlet_bcs:
            residual += vt.exterior_residual(dbc.get_function(), dbc.
                get_boundary())

        for dbc in self.neumann_bcs:
            residual += vt.neumann_residual(dbc.get_function(), dbc.
                get_boundary())

        return residual
```

Listing 9: The `EllipticOperator` class.

4.1. Example 1: Advection-diffusion problem. In this first example, we highlight the use of the classes `HyperbolicOperator` and `EllipticOperator` given in Listings 7 & 9, respectively. To this end, given $\Omega = (0,1)^2$, with boundary Γ, consider

```
class PoissonOperator(EllipticOperator):

    def __init__(self, mesh, fspace, bcs, kappa=1):
        def F_v(u, grad_u):
            return kappa*grad_u

        EllipticOperator.__init__(self, mesh, fspace, bcs, F_v)
```

Listing 10: The `PoissonOperator` class need only inherit the `EllipticOperator` and define its own viscous flux, `F_v`.

```
po = PoissonOperator(mesh, V, DGDirichletBC(ds, gD), kappa=u+1)
residual = po.generate_fem_formulation(u, v) - f*v*dx
```

Listing 11: Example of implementing the `PoissonOperator` utility class.

the problem: find u such that

$$-\nabla\cdot(\mathcal{K}\nabla u)+\nabla\cdot\left(\mathbf{b}u^2\right)=f \qquad \text{in } \Omega, \tag{4.1}$$

$$u=g_D \qquad \text{on } \Gamma, \tag{4.2}$$

where $\mathcal{K}>0$ denotes the diffusion coefficient and $\mathbf{b}=(b_1,b_2)^\top$. Thereby, setting $m=1$ and $d=2$, the PDE problem (4.1), (4.2) can be written in the general form (2.1)–(2.3), where $\mathcal{F}^c(\mathbf{u})=\mathbf{b}u^2$, $\mathcal{F}^v(\mathbf{u},\nabla\mathbf{u})=\mathcal{K}\nabla u$, and $\Gamma_N=\emptyset$. Moreover, it can easily be shown that the dissipation parameter arising in the local Lax-Friedrichs flux, is given by $\alpha|_{\partial\kappa}=\max_{w=u^+,u^-}2\,|w\mathbf{b}\cdot\mathbf{n}_\kappa|$, while the homogeneity tensor G, cf. (2.4), has entries $G_{11}=G_{22}=\mathcal{K}$, $G_{12}=G_{21}=0$.

The specification of the numerical fluxes $\mathcal{F}^c$ and $\mathcal{F}^v$, when $\mathcal{K}=1+u$, along with $\alpha(u,\mathbf{n})$ used with the local Lax-Friedrichs flux function within UFL syntax is provided in the code listing presented in Listing 12. Exploiting the software concepts outlined in Section 3 for the automatic computation of DGFEM formulations, the UFL code to construct and solve the resulting DGFEM approximation of (4.1), (4.2) is presented in Listing 13, where we have specified that $\mathbf{b}=(1,1)^\top$, $f=-4\mathrm{e}^{2(x-y)}-2\mathrm{e}^{x-y}$, and $g_D=e^{x-y}$; thereby, the analytical solution to (4.1), (4.2) is given by $u=e^{x-y}$. We note that on the final line of the code listing given in Listing 13, the call to DOLFIN's `solve` function automatically computes the Gâteaux derivative of the DGFEM semilinear form $\mathcal{N}(\cdot,\cdot)$, cf. (2.6), employed for the numerical approximation of (4.1), (4.2) and utilises a Newton solver to evaluate the DGFEM solution; for further details, we refer to [2]. Finally, in Figure 4.1 we show the asymptotic behaviour of the underlying DGFEM on a sequence of uniformly refined triangular meshes with polynomial orders $\ell=1,2,3,4$; as we expect, we observe that the $L_2(\Omega)$ and $H^1(\Omega)$ norms of the error tend to zero at the respective rates of $\mathcal{O}(h^{\ell+1})$ and $\mathcal{O}(h^\ell)$ as the mesh size h tends to zero. The complete code employed for this numerical example is provided in the file `advection_diffusion.py`.

4.2. Example 2: Compressible Euler Equations. In this second example, we consider the DGFEM discretisation of the compressible Euler equations: find

```
    def F_c(u):
        return b*u**2

    H = LocalLaxFriedrichs(lambda u, n: 2*u*dot(b, n))

    def F_v(u, grad_u):
        return (u + 1)*grad_u
```

Listing 12: Example 1: UFL representation of $\mathcal{F}^c$ and $\mathcal{F}^v$ of (4.1).

```
mesh = UnitSquareMesh(32, 32)

V = FunctionSpace(mesh, 'DG', 1)
u, v = Function(V), TestFunction(V)

gD = Expression('exp(x[0] - x[1])', element=V.ufl_element())
f = Expression('-4*exp(2*(x[0] - x[1])) - 2*exp(x[0] - x[1])',
               element=V.ufl_element())
b = Constant((1, 1))

ho = HyperbolicOperator(mesh, V, DGDirichletBC(ds, gD), F_c, H)
eo = EllipticOperator(mesh, V, DGDirichletBC(ds, gD), F_v)

residual = ho.generate_fem_formulation(u, v) \
          + eo.generate_fem_formulation(u, v) \
          - f*v*dx

solve(residual == 0, u)
```

Listing 13: Example 1: Automatic DGFEM formulation for the numerical approximation of (4.1), (4.2), using the definitions of the fluxes in Listing 12.

$\mathbf{u} = (\rho, \rho\mathbf{u}_{\mathrm{v}}, \rho E)^\top : \Omega \to \mathbb{R}^+ \times \mathbb{R}^d \times \mathbb{R}^+$ such that

$$\nabla \cdot \mathcal{F}^c(\mathbf{u}) = \mathbf{0} \quad \text{in } \Omega, \tag{4.3}$$

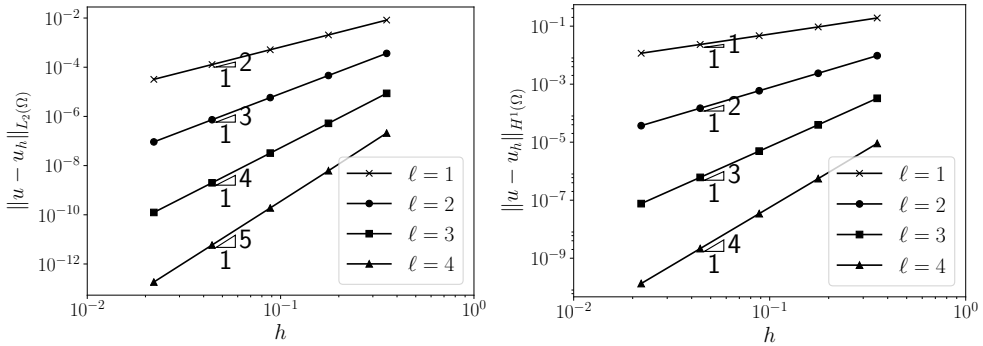
where the convective flux is given by

$$\mathcal{F}^c(\mathbf{u}) = \begin{pmatrix} \rho\mathbf{u}_{\mathrm{v}} \\ \rho\mathbf{u}_{\mathrm{v}} \otimes \mathbf{u}_{\mathrm{v}} + p\underline{\mathbf{I}} \\ (\rho E + p)\,\mathbf{u}_{\mathrm{v}} \end{pmatrix}. \tag{4.4}$$

Here, ρ, $\mathbf{u}_{\mathrm{v}}$, p, and E denote the density, velocity vector, pressure, and specific total energy, respectively. The equation of state of an ideal gas is given by

$$p = (\gamma - 1)\rho\left(E - \frac{1}{2}|\mathbf{u}_{\mathrm{v}}|^2\right),$$

where $\gamma = c_p/c_v$ is the ratio of specific heat capacities at constant pressure, c_p, and constant volume, c_v; for dry air, $\gamma = 1.4$. Finally, $\underline{\mathbf{I}}$ denotes the $d \times d$ identity matrix.



(a) (b)

Fig. 4.1: Example 1: Convergence of the DGFEM with h–refinement: (a) $\|u - u_h\|_{L_2(\Omega)}$; (b) $\|u - u_h\|_{H^1(\Omega)}$.

```
def F_c(U):
  rho, u1, u2, E = U[0], U[1]/U[0], U[2]/U[0], U[3]/U[0]
  p = (gamma - 1.0)*rho*(E - 0.5*(u1**2 + u2**2))
  H = E + p/rho
  return as_matrix([[rho*u1,        rho*u2       ],
                    [rho*u1**2 + p, rho*u1*u2    ],
                    [rho*u1*u2,     rho*u2**2 + p],
                    [rho*H*u1,      rho*H*u2     ]])

def alpha(U, n):
    rho, u1, u2, E = U[0], U[1]/U[0], U[2]/U[0], U[3]/U[0]
    p = (gamma - 1.0)*rho*(E - 0.5*(u1**2 + u2**2))
    u = as_vector([u1, u2])
    c = sqrt(gamma*p/rho)
    lambdas = [dot(u, n) - c, dot(u, n), dot(u, n) + c]
    return lambdas
```

Listing 14: Example 2: Specification of the convective flux and dissipation parameter α for the two-dimensional compressible Euler equations in UFL.

The automatic construction of the DGFEM formulation for the numerical approximation of (4.3) is encapsulated within the class `CompressibleEulerOperator`. We note that this class simply inherits the `HyperbolicOperator` class, with the specification of the Euler flux and dissipation parameter required for the definition of the Lax Friedrichs flux, cf. Listing 14 for the case when $d = 2$. We note that in this setting $\lambda_{\max} = \max\{|\mathbf{u}_v|, |\mathbf{u}_v| \pm c\}$, where $c = \sqrt{\gamma p/\rho}$ is the speed of sound.

As an illustration of the exploitation of `CompressibleEulerOperator`, we consider the DGFEM approximation of Ringleb's flow problem for which an analytical solution may be obtained using the hodograph method, cf. [9]. The DGFEM discretisation of this test case has also been studied in [18]. This problem represents a transonic flow in a curved channel domain, where the flow is mainly subsonic, with a small supersonic

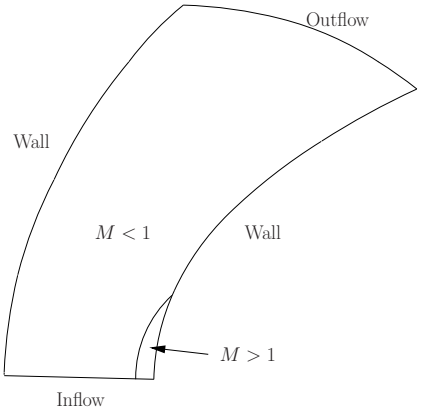


Fig. 4.2: Example 2: Computational domain for Ringleb's flow.

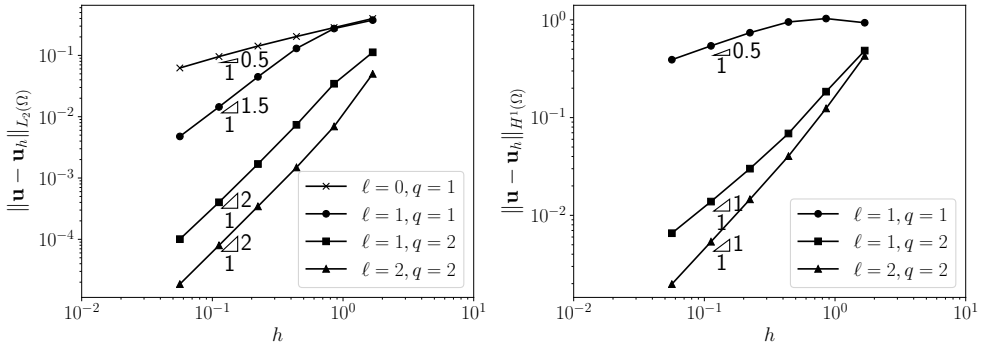


Fig. 4.3: Example 2: Convergence of the DGFEM with h–refinement for Ringleb's flow: (a) $\|\mathbf{u} - \mathbf{u}_h\|_{L_2(\Omega)}$; (b) $\|\mathbf{u} - \mathbf{u}_h\|_{H^1(\Omega)}$.

region near the right-hand-side wall; cf. Figure 4.2. Here, inflow/outflow boundary conditions are imposed on the top and bottom boundaries of Ω, while a solid wall condition is specified on the left- and right-hand side boundaries. Following [21] the latter boundary conditions are enforced based on employing a symmetry technique through the specification of the boundary function $\mathbf{u}_\Gamma$. More precisely, we treat the walls as part of the Dirichlet boundary, whereby we set

$$\mathbf{u}_\Gamma(\mathbf{u}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 - 2n_1^2 & -2n_1 n_2 & 0 \\ 0 & -2n_1 n_2 & 1 - 2n_2^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{u};$$

thereby, here $\Gamma = \Gamma_D$ and $\Gamma_N = \emptyset$.

Functions employed for the specification of the analytical solution $\mathbf{u}$ of Ringleb's flow, together with routines for the construction of the computational mesh are provided within the file `ringleb.py`. The curved boundaries on the walls of the domain

```
gD = RinglebAnalyticalSoln(element=V.ufl_element(), domain=mesh)
u_vec = project(gD, V)
n = FacetNormal(mesh)

slip_proj = as_matrix(((1,            0,            0, 0),
                       (0,  1-2*n[0]**2, -2*n[0]*n[1], 0),
                       (0, -2*n[0]*n[1],  1-2*n[1]**2, 0),
                       (0,            0,            0, 1)))
slip_bc = slip_proj * u_vec

bcs = [DGDirichletBC(ds(0), gD), DGDirichletBC(ds(1), slip_bc)]
ceo = CompressibleEulerOperator(mesh, V, bcs)
residual = ceo.generate_fem_formulation(u_vec, v_vec)

solve(residual == 0, u_vec)
```

Listing 15: Example 2: Code snippet for the automatic generation of the DGFEM formulation for the numerical approximation of Ringleb's flow.

must be treated in a careful manner to ensure optimal convergence of the underlying DGFEM discretisation. Indeed, our computational experience suggests that a curved polynomial description of the boundary of order $q \geq \ell + 1$ is necessary to ensure that the $L_2(\Omega)$ and $H^1(\Omega)$ norms of the error tend to zero at the optimal rates of $\mathcal{O}(h^{\ell+1})$ and $\mathcal{O}(h^{\ell})$ as the mesh size h tends to zero, for fixed ℓ, cf. Figure 4.3. The complete python code for this numerical example is provided `ringleb_example.py`; a snippet of this code is depicted in Listing 15 in order to highlight the simplicity of the specification of the DGFEM for this example.

4.3. Example 3a: Compressible Navier-Stokes Equations. In this example we consider the DGFEM discretisation of the compressible Navier-Stokes equations: find $\mathbf{u} = (\rho, \rho\mathbf{u}_{\mathrm{v}}, \rho E)^{\top} : \Omega \to \mathbb{R}^{+} \times \mathbb{R}^{d} \times \mathbb{R}^{+}$ such that

$$\nabla \cdot \left(\mathcal{F}^{c}\left(\mathbf{u}\right) - \mathcal{F}^{v}\left(\mathbf{u}, \nabla\mathbf{u}\right)\right) = \mathbf{f} \quad \text{in } \Omega, \tag{4.5}$$

where $\mathcal{F}^{c}$ is defined as in (4.4) and the viscous flux is given by

$$\mathcal{F}^{v}(\mathbf{u}, \nabla\mathbf{u}) = \begin{pmatrix} \mathbf{0} \\ \tau \\ \tau\mathbf{u}_{\mathrm{v}} + \mathcal{K}\nabla T \end{pmatrix}, \tag{4.6}$$

where T denotes the temperature and $\mathcal{K}$ is the thermal conductivity coefficient. The stress tensor is defined by

$$\tau = \mu\left(\nabla\mathbf{u}_{\mathrm{v}} + \nabla\mathbf{u}_{\mathrm{v}}^{\top} - \tfrac{2}{3}\left(\nabla \cdot \mathbf{u}_{\mathrm{v}}\right)\underline{\mathbf{I}}\right),$$

where μ is the dynamic viscosity coefficient. Finally, we note that $\mathcal{K}T = \frac{\mu\gamma}{\mathrm{Pr}}E - \frac{1}{2}\mathbf{u}_{\mathrm{v}}^{2}$, where $\mathrm{Pr} = 0.72$ is the Prandtl number.

Given the UFL code in Listing 14, together with the specification of the viscous flux in Listing 16, in the case when $d = 2$, we have implemented the class `CompressibleNavierStokesOperator` class, which inherits both the `CompressibleEuler-Operator` and `EllipticOperator` classes. Indeed, the `CompressibleEulerOperator`

```
def F_v(U, grad_U):
    rho, rhou, rhoE = conserved_variables(U)
    u = rhou/rho

    grad_rho = grad_U[0, :]
    grad_rhou = as_matrix([[grad_U[j,:] for j in range(1, dim + 1)]])[0]
    grad_rhoE = grad_U[-1,:]
    # Quotient rule to find grad(u) and grad(E)
    grad_u = as_matrix([[(grad_rhou[j,:]*rho - rhou[j]*grad_rho)/rho**2 for j
         in range(dim)]])[0]
    grad_E = (grad_rhoE*rho - rhoE*grad_rho)/rho**2

    tau = mu*(grad_u + grad_u.T - 2.0/3.0*(tr(grad_u))*Identity(dim))
    K_grad_T = mu*gamma/Pr*(grad_E - dot(u, grad_u))

    r = as_matrix([[0.0,                                0.0
                                    ],
                   [tau[0,0],                           tau[0,1]
                                                ],
                   [tau[1,0],                           tau[1,1]
                                                ],
                   [dot(tau[0,:], u) + K_grad_T[0], (dot(tau[1,:], u)) +
                       K_grad_T[1]]])
    return r
```

Listing 16: Example 3a: Specification of the viscous flux for the two-dimensional compressible Navier-Stokes equations in UFL.

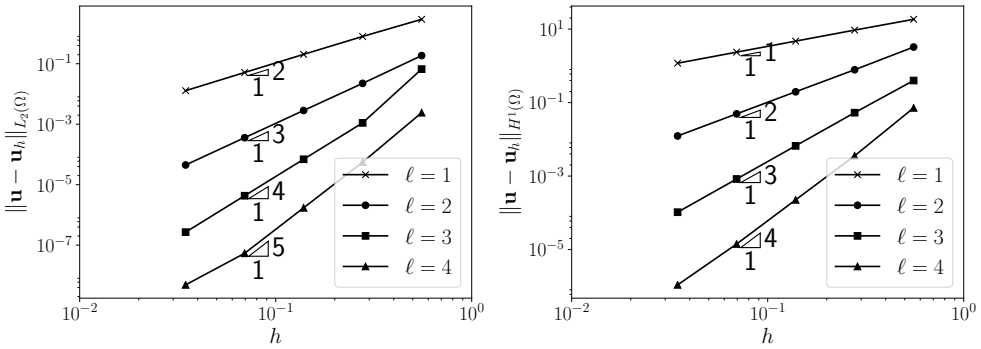
component is treated in an identical manner as in the previous section, while the `EllipticOperator` component is employed with the UFL specification of the viscous flux. On the basis of these classes, the homogeneity tensor and the resulting symbolic DGFEM formulation can then be automatically generated. Moreover, the Gâteaux derivative of the DGFEM formulation is automatically computed by UFL for use within the Newton solver managed in DOLFIN by invoking the call to `solve`. As a simple test, we consider the example outlined in [20]; namely, we set $\Omega = (0, \pi)^2$ and select $\mathbf{f}$ so that the analytical solution to (4.3) is given by

$$\begin{pmatrix} \rho \\ \rho \mathbf{u}_{\mathrm{v},1} \\ \rho \mathbf{u}_{\mathrm{v},2} \\ \rho E \end{pmatrix} = \begin{pmatrix} \sin(2(x+y)) + 4 \\ 1/5 \sin(2(x+y)) + 4 \\ 1/5 \sin(2(x+y)) + 4 \\ (\sin(2(x+y)) + 4)^2 \end{pmatrix}, \tag{4.7}$$

where $\mathbf{u}_{\mathrm{v}} = (\mathbf{u}_{\mathrm{v},1}, \mathbf{u}_{\mathrm{v},2})^{\top}$. Furthermore, we set $\mu = 1$, $\Gamma = \Gamma_D$ and $\Gamma_N = \emptyset$. The snippet of UFL code required to solve this problem is given in Listing 17; the complete code is provided in `compressible_navierstokes_square.py`. The orders of convergence of $\|\mathbf{u} - \mathbf{u}_h\|_{L_2(\Omega)}$ and $\|\mathbf{u} - \mathbf{u}_h\|_{H^1(\Omega)}$ are reported in Figure 4.4 as the mesh is uniformly refined for $\ell = 1, 2, 3, 4$; as in [20] we observe that $\|\mathbf{u} - \mathbf{u}_h\|_{L_2(\Omega)} = \mathcal{O}(h^{\ell+1})$ and $\|\mathbf{u} - \mathbf{u}_h\|_{H^1(\Omega)} = \mathcal{O}(h^{\ell})$ as h tends to zero, for each fixed polynomial order ℓ.

```
gD = Expression(('sin(2*(x[0]+x[1])) + 4',
                 '0.2*sin(2*(x[0]+x[1])) + 4',
                 '0.2*sin(2*(x[0]+x[1])) + 4',
                 'pow((sin(2*(x[0]+x[1])) + 4), 2)'),
                element=V.ufl_element())

f = Expression((...),
               element=V.ufl_element())

u, v = interpolate(gD, V), TestFunction(V)

cnso = CompressibleNavierStokesOperator(mesh, V, DGDirichletBC(ds, gD))
residual = cnso.generate_fem_formulation(u, v) - inner(f, v)*dx

solve(residual == 0, u)
```

Listing 17: Example 3a: Code snippet for the automatic generation of the DGFEM formulation for the numerical approximation of the compressible Navier-Stokes equations.



(a) (b)

Fig. 4.4: Example 3a: Convergence of the DGFEM with h–refinement for the compressible Navier-Stokes equations: (a) $\|\mathbf{u}-\mathbf{u}_h\|_{L_2(\Omega)}$; (b) $\|\mathbf{u}-\mathbf{u}_h\|_{H^1(\Omega)}$.

4.4. Example 3b: Compressible Flow Around a NACA0012 Airfoil. To demonstrate the application of the automatic formulation of DGFEMs applied to exterior flow problems, in this section we consider laminar flow around a NACA0012 airfoil whose geometry is defined by

$$y = \pm 5t\left(0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1036x^4\right), \tag{4.8}$$

where the thickness fraction $t = 0.12$. Here, we set the angle of attack $\alpha = 2°$, Reynolds number $\mathrm{Re} = 5 \times 10^3$, inlet flow Mach number $M = 0.5$, and impose an adiabatic no slip condition on the airfoil; cf. [20, 19]. We note that the adiabatic no slip condition imposes a zero heat flux condition $\mathcal{K}\nabla T \cdot \mathbf{n} = 0$ on the airfoil surface boundary. We have implemented this by defining a new boundary condition in

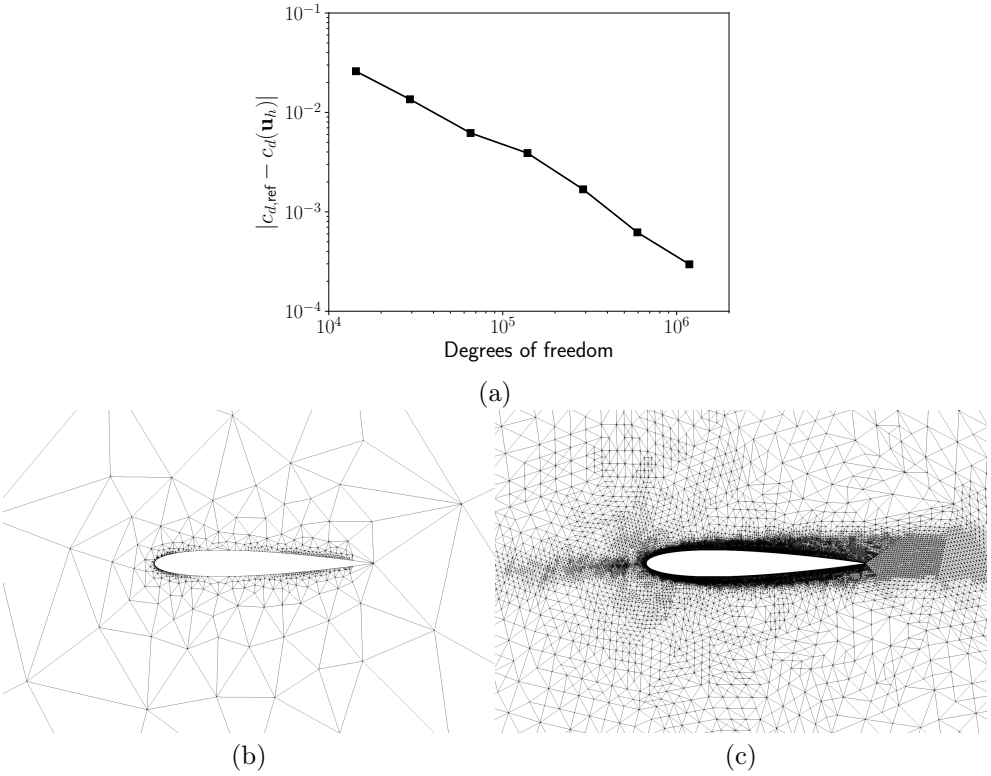


Fig. 4.5: Example 3b: Convergence of the DGFEM with DWR h–refinement for the compressible Navier-Stokes equations with $\ell = 1$: (a) $|C_{d,\text{ref}} - C_d(\mathbf{u}_h)|$; (b) Initial mesh; (c) Final mesh.

the class `DGAdiabaticWallBC` inheriting `DGBC`. This boundary condition is then recognised by the `CompressibleNavierStokesOperator` implementation which automatically generates the DGFEM formulation accordingly.

In this example, we shall undertake goal-oriented adaptive mesh refinement based on employing a dual weighted residual (DWR) *a posteriori* error estimator; for details, we refer to [19]. In particular, we select the quantity of interest to be the drag coefficient

$$c_d(\mathbf{u}) = \int_{\Gamma_{\text{airfoil}}} \frac{1}{C_\infty} (p\mathbf{n} - \tau\mathbf{n}) \cdot \psi_{\text{drag}} \, \mathrm{d}s, \tag{4.9}$$

where Γ_{airfoil} denotes the surface of the airfoil, $\psi_{\text{drag}} = (\cos(\alpha), \sin(\alpha))^\top$, $C_\infty = \frac{1}{2}\rho_\infty \mathbf{u}^2_{\mathbf{v},\infty} l_{\text{ref}}$, ρ_∞ and $\mathbf{u}_{\mathbf{v},\infty}$ denote the far field density and velocity, respectively, and l_{ref} is the reference length scale. The symbolic representation of the underlying dual problem is automatically formulated using the `dwr` component of `dolfin_dg`. The error in the approximate drag coefficient on each of the meshes employed is evaluated relative to a reference drag coefficient, $c_{d,\text{ref}}$ computed from a very fine mesh problem. Setting $\ell = 1$, the convergence of the error in the computed drag coefficient with respect to the number of degrees of freedom in the underlying finite element space, together with the initial and final computational meshes are shown in Figure 4.5. The

```
def U_to_V(U, gamma):
    rho, u1, u2, E = U[0], U[1]/U[0], U[2]/U[0], U[3]/U[0]
    i = E - 0.5*(u1**2 + u2**2)
    U1, U2, U3, U4 = U
    s = ln((gamma-1)*rho*i/(U1**gamma))
    V1 = 1/(rho*i)*(-U4 + rho*i*(gamma + 1 - s))
    V2 = 1/(rho*i)*U2
    V3 = 1/(rho*i)*U3
    V4 = 1/(rho*i)*(-U1)
    return as_vector([V1, V2, V3, V4])

def V_to_U(V, gamma):
    V1, V2, V3, V4 = V
    U = as_vector([-V4, V2, V3, 1 - 0.5*(V2**2 + V3**2)/V4])
    s = gamma - V1 + (V2**2 + V3**2)/(2*V4)
    rhoi = ((gamma - 1)/((-V4)**gamma))**(1.0/(gamma-1))*exp(-s/(gamma-1))
    U = U*rhoi
    return U
```

Listing 18: Example 3c: Transformations (4.10) and (4.11) in UFL representation.

code to generate these results is available in the file `dg_naca0012_2d.py`.

4.5. Example 3c: Entropy Variable Formulation of the Compressible Navier-Stokes Equations. In this section, we consider the DGFEM approximation of the compressible Navier-Stokes equations written in terms of so–called (symmetrisation) entropy variables; for further details, we refer, for example, to [5, 29]. For simplicity of presentation we only consider the two–dimensional case, though the extension to $d = 3$ follows in an analogous manner. Thereby, we introduce the change of variable $\mathbf{u} \mapsto \mathbf{V}(\mathbf{u})$ given by

$$\mathbf{V} = \frac{1}{\rho e}\begin{pmatrix} -\rho E + \rho e(\gamma + 1 - s) \\ \rho \mathbf{u}_{\mathrm{v},1} \\ \rho \mathbf{u}_{\mathrm{v},2} \\ -\rho \end{pmatrix}, \; s = \ln\left(\frac{(\gamma-1)\rho e}{\rho^\gamma}\right), \; e = E - \frac{1}{2}|\mathbf{u}_{\mathrm{v}}|^2, \tag{4.10}$$

and its inverse

$$\begin{aligned} \mathbf{u} &= \rho e \begin{pmatrix} -V_4 \\ V_2 \\ V_3 \\ 1 - \frac{1}{2}(V_2^2 + V_3^2)/V_4 \end{pmatrix}, \; \rho e = \left(\frac{\gamma - 1}{(-V_4)^\gamma}\right)^{\frac{1}{\gamma-1}} \exp\left(\frac{-s}{\gamma - 1}\right), \\ s &= \gamma - V_1 + \frac{1}{2}\frac{(V_2^2 + V_3^2)}{V_4}. \end{aligned} \tag{4.11}$$

Here, e is the internal energy density and s is a nondimensional entropy. The implementation of this mapping and its inverse in the UFL symbolic algebra framework is depicted in Listing 18. On the basis of this transformation, the symmetrised formulation of the compressible Navier-Stokes equations is given by: find $\mathbf{V} = (V_1, V_2, V_3, V_4)^\top$:

```
def F_c(V):
    U = V_to_U(V, gamma)
    rho, u1, u2, E = U[0], U[1]/U[0], U[2]/U[0], U[3]/U[0]
    p = (gamma - 1.0)*rho*(E - 0.5*(u1**2 + u2**2))
    H = E + p/rho
    res = as_matrix([[rho*u1,        rho*u2       ],
                     [rho*u1**2 + p, rho*u1*u2    ],
                     [rho*u1*u2,     rho*u2**2 + p],
                     [rho*H*u1,      rho*H*u2     ]])
    return res

def F_v(V, grad_V):
    V = variable(V)
    U = V_to_U(V, gamma)
    dudv = diff(U, V)
    grad_U = dot(dudv, grad_V)

    ...

    r = as_matrix([[0.0,0.0],
                   [tau[0,0],tau[0,1]],
                   [tau[1,0],tau[1,1]],
                   [dot(tau[0,:], u) + K_grad_T[0], (dot(tau[1,:], u)) +
                       K_grad_T[1]]])
    return r
```

Listing 19: Example 3c: Convective and viscous fluxes of the entropy formulation of the compressible Navier-Stokes equations. For the sake of brevity we do not repeat the code for the viscous flux tensor which is provided in Listing 16.

$\Omega \to \mathbb{R}^- \times \mathbb{R}^2 \times \mathbb{R}^-$ such that

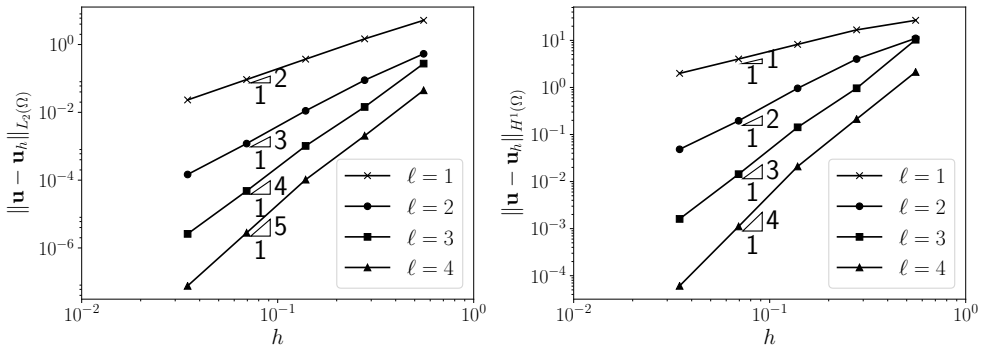
$$\nabla \cdot (\mathcal{F}^c_{\mathbf{V}}(\mathbf{V}) - \mathcal{F}^v_{\mathbf{V}}(\mathbf{V}, \nabla\mathbf{V})) = \mathbf{f} \quad \text{in } \Omega. \tag{4.12}$$

where $\mathcal{F}^c_{\mathbf{V}}(\mathbf{V}) = \mathcal{F}^c(\mathbf{u}(\mathbf{V}))$ and $\mathcal{F}^v_{\mathbf{V}}(\mathbf{V}, \nabla\mathbf{V}) = \mathcal{F}^v(\mathbf{u}(\mathbf{V}), \nabla\mathbf{u}(\mathbf{V}))$.

On the basis of the compressible Euler and compressible Navier-Stokes examples presented in the previous two sections, the DGFEM formulation of problem (4.12) is straightforward in the symbolic algebra framework of the UFL. We use the convective and viscous fluxes shown in Listings 14 and 16 and implement the transformations (4.10) and (4.11) noting that $\nabla\mathbf{u}(\mathbf{V}) = \frac{\partial \mathbf{u}(\mathbf{V})}{\partial \mathbf{V}} \nabla\mathbf{V}$. Indeed, the UFL formulation of the convective and viscous fluxes, $\mathcal{F}^c_{\mathbf{V}}$ and $\mathcal{F}^v_{\mathbf{V}}$, respectively, are presented in Listing 19. These constructs are then exploited within the DGFEM utility functions in the same manner as in the examples presented in Listings 15 and 17. As a simple test, we consider the numerical approximation of the compressible Navier-Stokes example presented in Section 4.3, based on employing the above entropy variable formulation. To this end, the underlying test problem is first transformed according to the change of variable $\mathbf{u} \mapsto \mathbf{V}(\mathbf{u})$, cf. (4.10), whereby the DGFEM is computed; we then transform the numerical solution according to the inverse mapping (4.11) in order to evaluate the error in the underlying approximation. As in the previous example, we again ob-

```
# Dirichlet conditions and error suite
gamma = 1.4
gD = Expression((...),
                element=V.ufl_element())

f = Expression((...),
               element=V.ufl_element())

V_vec, V_test = interpolate(gD, V), TestFunction(V)

bo = CompressibleNavierStokesOperatorEntropyFormulation(mesh, V,
    DGDirichletBC(ds, gD))
residual = bo.generate_fem_formulation(V_vec, V_test) - inner(f, V_test)*
    dx

solve(residual == 0, V_vec)
```

Listing 20: Example 3c: Code snippet for the automatic generation of the DGFEM formulation for the numerical approximation of the entropy formulation of the compressible Navier-Stokes equations.



(a) (b)

Fig. 4.6: Example 3c: Convergence of the DGFEM with h–refinement for the compressible Navier-Stokes equations in entropy formulation: (a) $\|\mathbf{u}-\mathbf{u}_h\|_{L_2(\Omega)}$; (b) $\|\mathbf{u}-\mathbf{u}_h\|_{H^1(\Omega)}$.

serve optimal convergence of the underlying DGFEM, cf. Figure 4.6; the code for this example is provided in Listing 20, cf. `compressible_navierstokes_entropy_square.py`.

4.6. Example 4: Maxwell Operator. In this penultimate example we demonstrate the extensibility of the symbolic DGFEM framework developed in this article by considering the automatic discretisation of the Maxwell problem: find $\mathbf{u}$ such that

$$\nabla \times \mathcal{F}^m(\mathbf{u}, \nabla \times \mathbf{u}) - k^2\mathbf{u} = \mathbf{f} \quad \text{in } \Omega, \tag{4.13}$$

$$\mathbf{n} \times \mathbf{u} = \mathbf{n} \times \mathbf{g}_D \quad \text{on } \Gamma, \tag{4.14}$$

```
def F_m(u, curl_u):
    return curl_u
```

Listing 21: Example 4: UFL representation of $\mathcal{F}^m(\mathbf{u}, \nabla \times \mathbf{u})$ of (4.13).

```
mesh = RectangleMesh(Point(-1., -1.), Point(1., 1.), 32, 32)

V = VectorFunctionSpace(mesh, "DG", 1)
u, v = Function(V), TestFunction(V)

k = Constant(2.0)
gD = Expression(("sin(k*x[1])", "sin(k*x[0])"), k=k, element=V.ufl_element())

mo = MaxwellOperator(mesh, V, [DGDirichletBC(ds, gD)], F_m)
residual = mo.generate_fem_formulation(u, v) - k**2*dot(u, v)*dx

solve(residual == 0, u)
```

Listing 22: Example 4: Automatic DGFEM formulation for the numerical approximation of (4.13), (4.14).

where, for simplicity, we set $\mathcal{F}^m(\mathbf{u}, \nabla \times \mathbf{u}) = \nabla \times \mathbf{u}$ and $k > 0$ is the wave number; for solvability, we assume that k^2 is not a Maxwell eigenvalue. The class `MaxwellOperator` implements the symmetric interior penalty discretisation of (4.13), (4.14) outlined in [27]. Thereby, given the ufl representation of $\mathcal{F}^m(\mathbf{u}, \nabla \times \mathbf{u})$ depicted in Listing 21, the corresponding code needed to compute the DGFEM approximation of (4.13), (4.14) can be written in the very compact form shown in Listing 22; see `maxwell.py`. As a simple test case, here we have set $\Omega = (-1, 1)^2$, $k = 2$, $\mathbf{g}_D = (\sin(kx), \sin(ky))^\top$ and $\mathbf{f} = \mathbf{0}$ such that the analytical solution is given by $\mathbf{u} = \mathbf{g}_D$. Numerical experiments demonstrating the performance of the resulting scheme on a sequence of uniformly refined triangular meshes are presented in Figure 4.7.

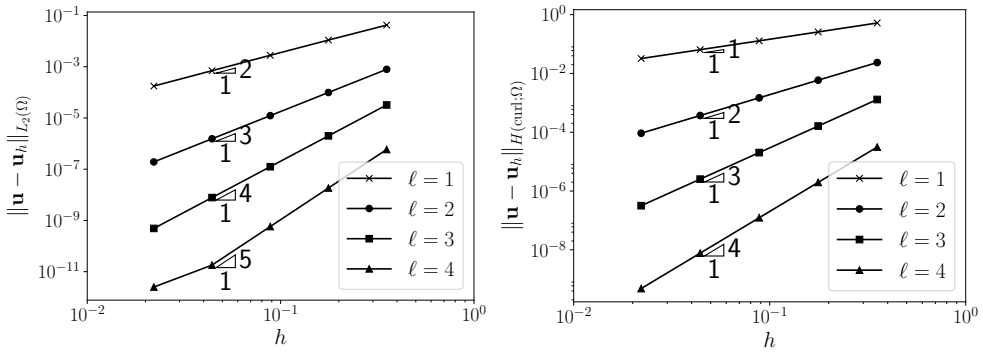
4.7. Example 5a: Hyperelasticity. Our final two examples highlight the flexibility of the DGFEM framework proposed in this article for the discretisation of hyperelasticity problems; in this setting DGFEM schemes offer computational benefits in the nearly–incompressible regime, cf. [16, 38]. Given a domain Ω_0 defining an elastic body's reference configuration with boundary $\partial\Omega_0$ and outward pointing unit normal vector $\mathbf{N}$, we seek the displacement vector at all points in the domain $\mathbf{u}(\mathbf{X}) : \Omega_0 \to \mathbb{R}^3$ which determines the mapping from the reference $\mathbf{X} \in \Omega_0$ to the deformed $\mathbf{x} \in \Omega$ configuration, such that $\mathbf{x} = \mathbf{X} + \mathbf{u}$. The constitutive model demands mass balance, so we seek $\mathbf{u}$ such that

$$-\nabla_{\mathbf{X}} \cdot \mathbf{P} = \mathbf{B} \quad \text{in } \Omega_0, \tag{4.15a}$$

$$\mathbf{u} = \mathbf{u_0} \quad \text{on } \partial\Omega_{0,D}, \tag{4.15b}$$

$$\mathbf{P} \cdot \mathbf{N} = \mathbf{T} \quad \text{on } \partial\Omega_{0,N}, \tag{4.15c}$$

where $(\nabla_{\mathbf{X}})_i := \partial/\partial X_i$ is the gradient in the reference domain, $\mathbf{P} = \partial\Psi(\mathbf{F})/\partial\mathbf{F}$ is the first Piola-Kirchoff stress tensor, $\Psi(\cdot)$ is the strain-energy function, $\mathbf{B}$ is a body force, $\mathbf{F} = \mathbf{I} + \nabla_{\mathbf{X}}\mathbf{u}$ is the strain tensor, $\mathbf{T}$ is a traction force, and $\partial\Omega_{0,D}$ and $\partial\Omega_{0,N}$



(a) (b)

Fig. 4.7: Example 4: Convergence of the DGFEM with h–refinement: (a) $\|\mathbf{u} - \mathbf{u}_h\|_{L_2(\Omega)}$; (b) $\|\mathbf{u} - \mathbf{u}_h\|_{H(\mathrm{curl};\Omega)}$.

```
def F_v(u, grad_u):
    F = variable(Identity(3) + grad_u)
    psi = (mu/2)*(tr(F.T*F) - 3) - mu*ln(det(F)) + (lmbda/2)*(ln(det(F)))**2
    return diff(psi, F)
```

Listing 23: Example 5a: UFL representation of the neo-Hookean hyperelasticity equation (4.15).

denote the Dirichlet and Neumann boundaries, respectively. Here, we examine a neo-Hookean hyperelasticity model where the strain energy density function is given by

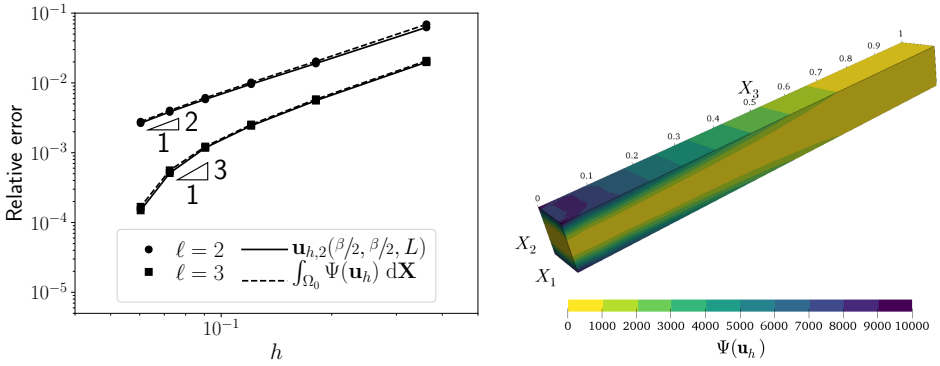
$$\Psi(\mathbf{F}) = \frac{\mu}{2}\left(\mathrm{tr}(\mathbf{F}^\top\mathbf{F}) - 3\right) - \mu\ln(\det\mathbf{F}) + \frac{\lambda}{2}\left(\ln(\det\mathbf{F})\right)^2,$$

where $\mu = E/\left(2\left(1+\nu\right)\right)$ and $\lambda = E\nu/\left(\left(1+\nu\right)\left(1-2\nu\right)\right)$ are the first and second Lamé parameters, respectively, given in terms of Young's modulus E and Poisson ratio ν.

Clearly we observe that equation (4.15) can be rewritten with $\mathcal{F}^v(\mathbf{u}, \nabla_{\mathbf{X}}\mathbf{u}) \equiv \mathbf{P}$, cf. Listing 23; thereby, the DGFEM framework outlined previously may be directly applied. We highlight that the flexibility of this approach naturally allows us to consider other potential strain energy models, such as Saint Venant–Kirchhoff, Ogden, and Mooney–Rivlin [25].

To test the DGFEM solver we repeat the numerical example 4.1 presented in [33]. To this end, we simulate the small strain deformation of the beam $\Omega_0 = (0,\beta)^2\times(0,L)$, where $\beta = 0.1$, $L = 1$, with the material parameters $E = 200\times10^9$ and $\nu = 0.3$. There is no applied body force, i.e., $\mathbf{B} = \mathbf{0}$, the beam is clamped on the near face, $\partial\Omega_{0,D} = (0,\beta)^2\times\{0\}$, $\mathbf{u}|_{\partial\Omega_{0,D}} = \mathbf{0}$, an external distributed force is applied to the far face

$$\mathbf{T} = \begin{cases} \left(0, 10^4/\beta^2, 0\right)^\top & \text{if } X_3 = L, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$



(a) (b)

Fig. 4.8: Example 5a: Mesh refinement study of the cantilever simulation outlined in Section 4.7: (a) Convergence of the relative error in the functionals of the DGFEM approximation; (b) The computed internal strain energy.

and $\partial\Omega_{0,N} = \partial\Omega_0 \setminus \partial\Omega_{0,D}$. The system has a known analytical solution for the tip deflection $\mathbf{u}_2(\beta/2, \beta/2, L) = 2 \times 10^{-3}$ and the internal strain energy $\int_{\Omega_0} \Psi(\mathbf{u})\, \mathrm{d}\mathbf{X} = 10$. Convergence of the relative errors of the DGFEM approximation of this problem are shown in Figure 4.8. The code to generate these results is available in the file `dg_cantilever.py`.

4.8. Example 5b: Near–Incompressible Hyperelastic Buckling. To demonstrate the capability of the DGFEM framework applied to larger three–dimensional problems we draw inspiration from a numerical example of an elastic body buckled in a compressive state, cf. Section 6.7 in [38]. Thereby, we set $\Omega_0 = (0, 1/2)^2 \times (0, 2)$, $E = 1 \times 10^8$, $\nu = 0.46$, $\mathbf{B} = \mathbf{T} = \mathbf{0}$ and prescribe $\mathbf{u}|_{X_3=0} = \mathbf{0}$ and $\mathbf{u}|_{X_3=2} = (0, 0.07, -0.5)^\top$. Using the proposed DGFEM framework the resulting deformed mesh configuration is shown in Figure 4.9. Here, we have employed both uniform mesh refinement, as well as adaptive refinement of the computational mesh based on employing a DWR *a posteriori* error indicator, where we select the internal strain energy $\int_{\Omega_0} \Psi(\mathbf{u})\, \mathrm{d}\mathbf{X}$ to be the quantity of interest. As expected, the adaptive refinement algorithm selects regions to enrich the computational mesh near the edges of the Dirichlet boundary, where large changes in the stress occur, as well as regions under compression, which lead to large changes in the internal strain energy. The implementation of this example is provided in the file `dg_compression.py`.

5. Concluding remarks. In this article we have exploited the use of symbolic algebra for the automatic computation of DGFEMs for the numerical approximation of general systems of nonlinear PDEs. In particular, we have proposed and implemented a class structure in order to allow for the specification of a given DGFEM in a clear and concise manner. While the examples we have presented have primarily focused on flow problems, we stress that the generality of this approach allows for the treatment of a wide range of PDEs stemming from diverse application areas. Indeed, in the final two examples, we have included the DGFEM approximation of the indefinite Maxwell problem and a hyperelasticity problem, respectively. The exploitation of the software developed within this article allows the user to build up the DGFEM

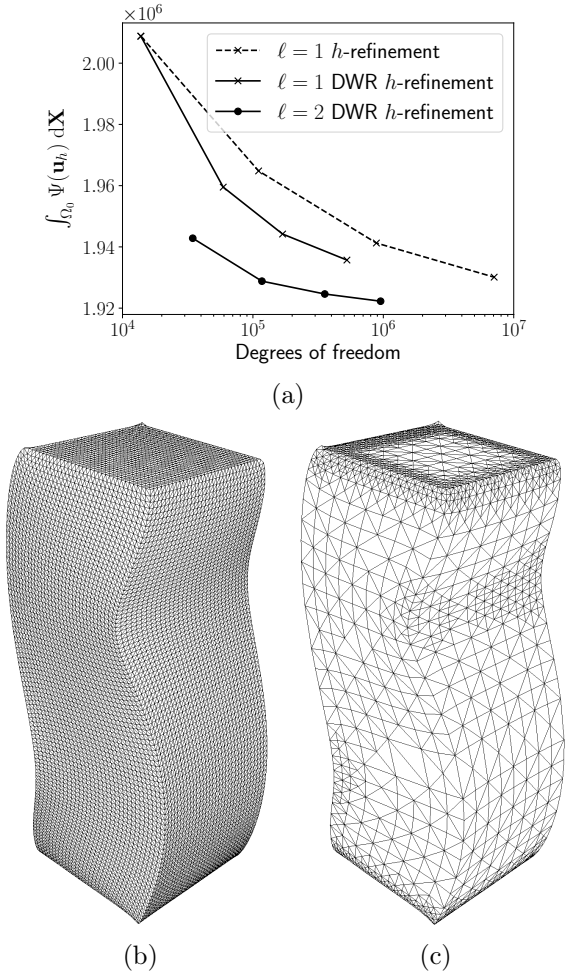


Fig. 4.9: Example 5b: Mesh refinement study of the compressible neo-Hookean elasticity model: (a) Convergence of the quantity of interest, $\int_{\Omega_0} \Psi(\mathbf{u}_h)\ \mathrm{d}\mathbf{X}$; (b) Deformed configuration generated by employing $\ell = 1$ and uniform h–refinement comprising 7 077 888 degrees of freedom; (c) Deformed configuration generated using adaptive DWR h–refinement with $\ell = 2$ giving rise to 953 220 degrees of freedom.

discretisation of systems of multi-physics PDEs in a simple and concise manner; as an example, we have employed the software here for the DGFEM approximation of microwave power assisted chemical vapour deposition reactor employed for the manufacture of synthetic diamond; see [28, 36]. Moreover, the code can easily be extended to other problems, and indeed other DGFEM schemes, so that users can tailor the software to their own applications.

REFERENCES

[1] M.S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, and G.N. Wells, *The FEniCS project version 1.5*, Arch. Numer. Software **3**

(2015), no. 100, 9–23.
[2] M.S. Alnæs and K.-A. Mardal, *SyFi and SFC: Symbolic finite elements and form compilation*, Automated Solution of Differential Equations by the Finite Element Method (A. Logg, K.-A. Mardal, and G.N. Wells, eds.), Lecture Notes in Computational Science and Engineering, vol. 84, Springer, 2012, pp. 269–278.
[3] D.N. Arnold, F. Brezzi, B. Cockburn, and L.D. Marini, *Unified analysis of discontinuous Galerkin methods for elliptic problems*, SIAM J. Numer. Anal. **39** (2001), 1749–1779.
[4] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general purpose object oriented finite element library*, ACM T. Math. Software **33** (2007), no. 4, 24/1–24/27.
[5] T.J. Barth, *Numerical methods for gasdynamic systems on unstructured meshes*, An Introduction to Recent Developments in Theory and Numerics for Conservation Laws (D. Kröner, M. Ohlberger, and C. Rohde, eds.), Lecture Notes in Computational Science and Engineering, vol. 5, Springer, 1999, pp. 195–285.
[6] Francesco Bassi, Stefano Rebay, G. Mariotti, S. Pedinotti, and M. Savini, *A high-order accurate discontinuous finite element method for inviscid and viscous turbomachinery flows*, 2nd European Conference on Turbomachinery Fluid Dynamics and Thermodynamics, Antwerpen, Belgium, March 5–7, 1997 (R. Decuypere and G. Dibelius, eds.), Technologisch Instituut, 1997, pp. 99–108.
[7] S.J. Berkowitz, *On computing the determinant in small parallel time using a small number of processors*, Information Processing Letters **18** (1984), no. 3, 147–150.
[8] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander, *The distributed and unified numerics environment, version 2.4*, Arch. Numer. Software **4** (2016), no. 100, 13–29.
[9] G. Chiocchia, *Exact solutions to transonic and supersonic flows*, AGARD (1985), AR–211.
[10] K.A. Cliffe, A. Spence, and S.J. Tavener, *The numerical analysis of bifurcation problems with application to fluid mechanics*, Acta Numerica (A. Iserles, ed.), vol. 9, Cambridge University Press, 2000, p. 39–131.
[11] K.A. Cliffe and S.J. Tavener, *Implementation of extended systems using symbolic algebra*, Continuation Methods in Fluid Dynamics, Notes on Numerical Fluid Mechanics, 2000, pp. 81–92.
[12] B. Cockburn, *An introduction to the discontinuous Galerkin method for convection-dominated problems*, Advanced numerical approximation of nonlinear hyperbolic equations (Cetraro, 1997), Springer, Berlin, 1998, pp. 151–268. MR 1 728 854
[13] B. Cockburn, G.E. Karniadakis, and C.-W. Shu (eds.), *Discontinuous Galerkin methods*, Springer-Verlag, Berlin, 2000, Theory, computation and applications, Papers from the 1st International Symposium held in Newport, RI, May 24–26, 1999.
[14] D.A. Di Pietro and A. Ern, *Mathematical aspects of discontinuous Galerkin methods*, Mathématiques & Applications (Berlin) [Mathematics & Applications], vol. 69, Springer, Heidelberg, 2012.
[15] E.H. Georgoulis, E. Hall, and P. Houston, *Discontinuous Galerkin methods on hp–anisotropic meshes II: A posteriori error analysis and adaptivity*, Appl. Numer. Math. **59(9)** (2009), 2179–2194.
[16] P. Hansbo and M. G. Larson, *Discontinuous Galerkin methods for incompressible and nearly incompressible elasticity by Nitsche's method*, Comput. Methods Appl. Mech. Engrg. **191** (2002), no. 17, 1895 – 1908.
[17] R. Hartmann and P. Houston, *Adaptive discontinuous Galerkin finite element methods for nonlinear hyperbolic conservation laws*, SIAM J. Sci. Comput. **24** (2002), 979–1004.
[18] R. Hartmann and P. Houston, *Adaptive discontinuous Galerkin finite element methods for the compressible Euler equations*, J. Comput. Phys. **183** (2002), no. 2, 508–532.
[19] ———, *Symmetric interior penalty DG methods for the compressible Navier-Stokes equations I: Method formulation*, International Journal of Numerical Analysis and Modeling **3** (2005), no. 1, 1–20.
[20] R. Hartmann and P. Houston, *An optimal order interior penalty discontinuous Galerkin discretization of the compressible Navier–Stokes equations*, J. Comput. Phys. **227** (2008), no. 22, 9670–9685.
[21] ———, *Error estimation and adaptive mesh refinement for aerodynamic flows*, VKI LS 2010-01: 36th CFD/ADIGMA Course on *hp*-Adaptive and *hp*-Multigrid Methods, Oct. 26-30, 2009 (H. Deconinck, ed.), Von Karman Institute for Fluid Dynamics, Belgium, 2010.
[22] A.C. Hearn, *Reduce user's manual version 3.8*, 2004.
[23] F. Hecht, *New development in FreeFem++*, J. Numer. Math. **20** (2012), no. 3-4, 251–265.
[24] J.S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods*, Texts in Applied

Mathematics, vol. 54, Springer, New York, 2008, Algorithms, analysis, and applications.
[25] G. A. Holzapfel, *Nonlinear solid mechanics: A continuum approach for engineering*, Wiley, 2000.
[26] P. Houston, *AptoFEM finite element analysis software*, 2015.
[27] P. Houston, I. Perugia, A. Schneebeli, and D. Schötzau, *Interior penalty method for the indefinite time-harmonic Maxwell equations*, Numer. Math. **100** (2005), no. 3, 485–518.
[28] P. Houston and N. Sime, *Numerical modelling of MPA-CVD reactors with the discontinuous Galerkin finite element method*, J. Phys. D: Appl. Phys. **50** (2017), no. 29, 295202.
[29] T.J.R. Hughes, L.P. Franca, and M. Mallet, *A new finite element formulation for computational fluid dynamics: I. symmetric forms of the compressible Euler and Navier-Stokes equations and the second law of thermodynamics*, J. Comput. Phys. **54** (1986), 223–234.
[30] D. Kröner, *Numerical schemes for conservation laws*, Wiley-Teubner, 1997.
[31] A. Logg, K.A. Mardal, and G. Wells, *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book (Lecture Notes in Computational Science and Engineering)*, Springer, 2012.
[32] G.R. Markall, D.A. Ham, and P.H.J. Kelly, *Towards generating optimised finite element solvers for GPUs from high-level specifications*, Proc. Comp. Sci. **1** (2010), no. 1, 1815 – 1823.
[33] L. Noels and R. Radovitzky, *A general discontinuous Galerkin method for finite hyperelasticity. formulation and numerical applications*, Internat. J. Numer. Methods Engrg. **68** (2006), no. 1, 64–97.
[34] K. Ølgaard, *Automated computational modelling for complicated partial differential equations*, Ph.D. thesis, Technische Universiteit Delft, 2013.
[35] B. Rivière, *Discontinuous Galerkin methods for solving elliptic and parabolic equations*, Frontiers in Applied Mathematics, vol. 35, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2008, Theory and implementation.
[36] N. Sime, *Numerical modelling of chemical vapour deposition reactors*, Ph.D. thesis, University of Nottingham, 2016.
[37] SymPy Development Team, *Sympy: Python library for symbolic mathematics*, 2014.
[38] A. Ten Eyck and A. Lew, *Discontinuous Galerkin methods for non-linear elasticity*, Internat. J. Numer. Methods Engrg. **67** (2006), no. 9, 1204–1243.
[39] E.F. Toro, *Riemann solvers and numerical methods for fluid dynamics*, Springer, 1997.
[40] H.G. Weller and G. Tabor, *A tensorial approach to computational continuum mechanics using object-oriented techniques*, Comput. Phys. **12** (1998), no. 6, 620–631.