Advances in Engineering Software 171 (2022) 103147

Contents lists available at ScienceDirect

Advances in Engineering Software

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

Research paper

An open source *hp*-adaptive discontinuous Galerkin finite element solver for linear elasticity

Thomas Wiltshire, Robert E. Bird, William M. Coombs, Stefano Giani *

Department of Engineering, Lower Mountjoy, South Rd, Durham DH1 3LE, UK

ARTICLE INFO

Keywords:
Open source software
MATLAB
Discontinuous galerkin finite elements
Linear elasticity
hp-Adaptivity
Researcher development

ABSTRACT

Open source codes are a key ingredient to greater research integrity and accountability in computational science and engineering. However, many of these codes have not been developed with modification of the *base* code as their primary consideration. Existing codes may provide an environment for researchers to quickly test out their ideas under different physical conditions in a high level way but they are not always ideal for those interested in the development of numerical methods. The majority of existing open source discontinuous Galerkin finite element codes are written in C++ and there is a significant learning curve for junior researchers to adopt, understand and modify the underlying code/routines. This paper presents an open source *hp*-adaptive discontinuous Galerkin finite element code written in MATLAB that has been explicitly designed to make it easy for users, especially MSc/PhD-level researchers, to understand the method and implement new ideas within the core code. Although the code is focused on solving problems in linear elasticity, it is straightforward to modify it to solve other physical equations.

1. Introduction

Open source codes are essential for the health of research in computational science and engineering as they provide a route-in for researchers to start working with numerical methods without having to implement hundreds (often tens of thousands) of lines of computer code. Open source code also promotes collaboration between researchers and transparent inspection of numerical results - resulting in greater research integrity and accountability [1]. There are already available many open source codes to solve specific engineering problems like for example fractures in materials [2,3] or simulating structures using plates and shells [4,5].

The majority of numerical methods for solving partial differential equations (PDEs) can be divided into two categories; those based on meshes partitioning the physical domain and those not using meshes. Meshless methods [6] were invented to mitigate some of the difficulties associated with resolving the geometry of physical domains with meshes. In common with methods based on meshes like finite element methods (FEMs), meshless methods use shape functions to approximate the solutions but not as interpolants as in FEMs, rather than as approximants. Partition of unity methods is another class of methods not based on standard meshes but on a coverage of the physical domain with overlapping patches [7,8]. The category of methods based on meshes is dominated by FEMs and their variations. One of the most prominent classes of FEMs is multiscale methods [9,10] in which the approximated solution is decomposed into coarse and fine-scale solutions. This is achieved by rewriting the PDE model into a pair of coupled macroscopic and microscopic models. Other classes of methods were invented to overcome some of the aspects of classical FEMs. Isogeometric analysis breaks free from the limitations of polygonal and polyhedral shaped elements and polynomials of integer order shape functions [11]. Extended and enriched FEMs allow adding to the approximation spaces functions incorporating specific knowledge of the solution [12,13]. Cut FEMs eliminate the difficulties to fit complex or implicit surfaces with standard meshes [14,15]. A lot of effort has been made in studying ways to control the error in FEMs [16–18]. A very popular approach is adaptivity as a way to improve the accuracy of computed solutions in many fields adapting meshes without leaving the FEM framework [19–21].

There are of course methods not easily classifiable in these two categories like numerical methods based on Taylor-series [22].

However, the solution of partial differential equations in engineering analysis is dominated by the continuous Galerkin (CG) variant of FEM. There are a huge number of open source codes based on conventional CG

* Corresponding author.
E-mail address: stefano.giani@durham.ac.uk (S. Giani).

https://doi.org/10.1016/j.advengsoft.2022.103147
Received 23 December 2021; Received in revised form 31 March 2022; Accepted 13 May 2022
Available online 13 June 2022

finite elements (for example, see [23–28], amongst many others). An alternative to the conventional CG formulation is the family of discontinuous Galerkin (DG) finite element methods and there are many reasons that researchers may wish to adopt a DG modelling framework, such as:

(i) their ability to easily allow for adaptive mesh refinement, hanging nodes and different polynomial orders in adjacent elements due to the relaxation of the compatibility between elements;
(ii) that, unlike CG elements, DG elements can handle excessive aspect ratios without degradation of their predictive capabilities;
(iii) that, unlike CG methods, degrees of freedom are not shared between elements, making easier to split a mesh on several computing nodes;
(iv) that, unlike CG methods, boundary conditions are imposed weakly, making easier to incorporate complicated boundary conditions into models;
(v) more freedom in choosing shape functions since no continuity across elements is needed.

There are several variants of DG finite element methods, categorised by the inter-element communication technique used to couple adjacent elements. In this paper we adopt a symmetric interior penalty DG (SIPG) approach. Despite their advantages, there are far fewer DG finite element open source codes (compared to CG-based methods) and the majority of these codes have not been developed with understanding and modification of their core code as their primary focus. To the best of the authors' knowledge, the following open-source DG codes are freely available (grouped by underpinning implementation language):

C++ based: hpGEM [29], DUNE (Python binding) [30], DoGPack (Python/MATLAB post-processing possible) [31], NEKTAR++, FreeFEM [32], MFEM [33] and FEniCS (Python front-end available) [34];
Python based: PyFR [35] and Nutils [36].
FORTRAN based: SPEED [37] and SEISOL [38].

It is clear that the majority of these codes are based on C++ for computational efficiency. However, several researchers are interested in the development of *methods* rather than the analysis of large-scale problems. For these researchers, being able to easily access and understand the underlying source code is essential and these C++ implementations may be off-putting or a complete barrier to using these open source codes to develop their ideas. Python, being open-source to its very core, is a credible alternative and several of the C++ codes have Python front ends and/or processing environments, as identified above. However, in the authors' opinion, within mathematical and technical computing, MATLAB is the dominant tool taught to engineers and scientists whilst at university and MATLAB has been specifically designed with these users in mind [39]. The documentation, help and debugging environment offered by MATLAB is unparalleled and this allows researchers (especially junior scientists/engineers) to quickly develop their scientific ideas whilst minimising the syntax barrier.

In this paper we present a MATLAB-based open-source symmetric interior penalty DG finite element code that has been designed to make the method accessible to early career researchers, such as final year undergraduates, MSc and PhD students as well as post-doctoral researchers moving into this area of research. This point is emphasised by the fact that the first author of this manuscript was a final year MEng student during the development of this paper and associated code. Another point of novelty of this paper is the inclusion of error estimate driven *hp*-adaptivity, where the code automatically refines the background mesh in terms of element size (h) or polynomial order (p) of specific elements based on the estimate error in each element. This provides a powerful stress analysis tool which is able to achieve extreme accuracy (errors less than 10^{-10}) with modest computational effort and on resources that are commonly available to early career researchers (standard laptop/desktop PCs). The physical problem focus of the code is linear elasticity, however once the structure of the code is understood, it is straightforward to modify the code to analyse other physical problems, include alternative element formulations, test out different adaptivity routines, etc.

Following this introduction, the layout of the rest of the paper is as follows: Section 2 outlines the considered model problem and key details of the symmetric interior penalty DG finite element method (including the error estimate and adaptivity algorithm), Section 3 explains how to install and test the method, Section 4 details the structure of the program and demonstration problems are show in Section 5. Finally, brief conclusions are included in Section 6. All of the code and associated documentation is available from https://github.com/Robert-Bird/Discontinuous-Galerkin-MATLAB.

2. The method

In this section, we detail the numerical analysis aspects of the implemented method. The model problem and numerical methods implemented in the code are analysed in [40] where in-depth analysis can be found.

2.1. Model problem

The problem implemented in the code is linear elasticity with a variety of boundary conditions: Let Ω be a bounded polygonal domain in $\mathbb{R}^2$ with $\partial\Omega = \Gamma_D \cup \Gamma_N \cup \Gamma_T$, where Γ_D, Γ_N and Γ_T are disjoint sets, and let $\mathbf{u}$ the solution of

$$\begin{aligned} -\nabla\cdot\boldsymbol{\sigma}(\mathbf{u}) &= \mathbf{f} && \text{in} \quad \Omega \\ \mathbf{u} &= \mathbf{g}_D && \text{on} \quad \Gamma_D \\ \boldsymbol{\sigma}(\mathbf{u})\cdot\mathbf{n} &= \mathbf{g}_N && \text{on} \quad \Gamma_N \\ \mathbf{u}\cdot\mathbf{n} &= \mathbf{g}_T\cdot\mathbf{n} && \text{on} \quad \Gamma_T \\ \mathbf{t}(\mathbf{u})\cdot\mathbf{n}_{\parallel} &= 0 && \text{on} \quad \Gamma_T, \end{aligned} \tag{1}$$

where $\mathbf{n} = (n_x, n_y)$ is the unit vector perpendicular to the boundary of Ω and pointing out and $\mathbf{n}_{\parallel}$ is the tangential unit vectors to the boundary, $\mathbf{t}(\mathbf{u})$ is the traction component of the stress, i.e.

$$\mathbf{t}(\mathbf{u}) := \boldsymbol{\sigma}(\mathbf{u})\cdot\mathbf{n}.$$

The functions $\mathbf{f}$, $\mathbf{g}_D$, $\mathbf{g}_N$ and $\mathbf{g}_T$ specify the rhs and the values along the boundaries and they are respectively in $[L^2(\Omega)]^2$, $[H^{1/2}(\Gamma_D)]^2$, $[L^2(\Gamma_N)]^2$ and $[H^{1/2}(\Gamma_T)]^2$. We define the strain tensor for a displacement $\mathbf{v}$ as $\boldsymbol{\epsilon}(\mathbf{v})_{ij} := \frac{1}{2}(\nabla_j\mathbf{v}_i + \nabla_i\mathbf{v}_j)$ and the stress as $\boldsymbol{\sigma}(\mathbf{v}) = \mathbf{D}\boldsymbol{\epsilon}(\mathbf{v})$ where the matrix $\mathbf{D} \in \mathbb{R}^{3\times3\times3\times3}$.

The set Γ_D may be empty. In this case, problem (1) may not have a unique solution if rigid motions are in the kernel [41]. Therefore, the natural choice of space to enforce a unique solution for problem (1) is $\mathbf{u} \in \mathscr{S} := [H^1(\Omega)]^2 \backslash R$, where R is the space containing all rigid motions. In case the prescribed boundary conditions define a problem with rigid motions in the kernel, the well-definiteness of the problem can be enforced using the average boundary condition to remove rigid motions [42]. Such a case happens for example when only Neumann boundary conditions are used, see problem 3 in Section 3.2. The code automatically applies the average boundary condition if needed.

2.2. Symmetric interior penalty discontinuous Galerkin method

In this section we introduce our DG method used in the code to solve problem (1). We assume that the computational domain Ω can be partitioned into a shape regular mesh $\mathscr{T}$ of triangular and affine elements and we denote with K a generic element of $\mathscr{T}$. We allow for irregular meshes with a maximum of one hanging node per edge. Due to our

assumption that the initial mesh is shape regular, allowing at most one hanging node per edge implies that the diameters of the elements in all refined meshes are of bounded variation for any pair of neighbouring elements. We denote $\mathscr{E}(\mathscr{T})$ and $\mathscr{E}^{\text{int}}(\mathscr{T})\subset\mathscr{E}(\mathscr{T})$ the set of all edges of the mesh $\mathscr{T}$ and the subset of all interior edges respectively and by $\mathscr{E}^{\text{BC}}(\mathscr{T})\subset\mathscr{E}(\mathscr{T})$ the subset of all boundary edges. The set $\mathscr{E}^{\text{BC}}(\mathscr{T})$ is partitioned in the three subsets $\mathscr{E}^{\text{D}}(\mathscr{T})$, $\mathscr{E}^{\text{N}}(\mathscr{T})$ and $\mathscr{E}^{\text{T}}(\mathscr{T})$ that are the sets containing the edges forming the three portions of the boundary Γ_D, Γ_N and Γ_T. We define h_K and h_E to be the diameter of the element K and the length of the edge E respectively.

In the implementation, we allow to vary the polynomial degrees across elements under the constraints that the variation is at most equal to one for any pair of neighbouring elements. This is enforced during mesh adaptation, too. For each element K of the mesh $\mathscr{T}$ we associate a polynomial degree $p_K \geq 1$ and we introduce the degree vector $\mathbf{p} = \{p_K : K \in \mathscr{T}\}$ and we define $p_{\min}$ as the minimum of p_K on the mesh $\mathscr{T}$. For any $E \in \mathscr{E}(\mathscr{T})$, we introduce the edge polynomial degree p_E by

$$p_E = \begin{cases} \max(p_K, p_{K'}), & \text{if} \quad E = \partial K^+ \cap \partial K^-,\ E \in \mathscr{E}^{int}(\mathscr{T}), \\ p_K, & \text{if} \quad E = \partial K \cap \partial\Omega,\ E \in \mathscr{E}(\mathscr{T})\backslash\mathscr{E}^{int}(\mathscr{T}), \end{cases} \tag{2}$$

where + and - denote the two elements sharing the face E. Hence, for a given partition $\mathscr{T}$ of Ω and a degree vector $\mathbf{p}$ on $\mathscr{T}$, we define the hp-version DG finite element space by

$$V_{\mathbf{p}}(\mathscr{T}) = \left\{ \mathbf{v} \in \left[L^2(\Omega)\right]^2 \backslash R : \quad \mathbf{v}|_K \in \left[\mathscr{P}_{p_K}(K)\right]^2, \quad K \in \mathscr{T} \right\}, \tag{3}$$

where $\mathscr{P}_{p_K}(K)$ is the space of polynomials of degree at most p_K and R is the set of rigid motions. We also use + and - to denote all quantities related to the elements. Given an edge $E \in \mathscr{E}^{\text{int}}(\mathscr{T})$ shared by two elements K^+ and K^-, we define the jump $[\![\cdot]\!]$ operator and the average $\{\cdot\}$ operator on vectors and tensors as:

$$\begin{aligned} [\![\mathbf{v}]\!]_{ij} &= \mathbf{v}_i^+ \mathbf{n}_{Kj}^+ - \mathbf{v}_i^- \mathbf{n}_{Kj}^+ \\ [\![\boldsymbol{\sigma}]\!]_i &= \boldsymbol{\sigma}_{ij}^+ \mathbf{n}_{Kj}^+ - \boldsymbol{\sigma}_{ij}^- \mathbf{n}_{Kj}^+ \\ \{\boldsymbol{\sigma}(\mathbf{v})\} &= \frac{1}{2}(\boldsymbol{\sigma}(\mathbf{v})^+ + \boldsymbol{\sigma}(\mathbf{v})^-) \end{aligned} \tag{4}$$

where $\mathbf{n}_K^+ = (n_x^+, n_y^+)$ the outward unit normal on the boundary ∂K^+ of an element K. Such definitions are modified on edges along the boundary of the domain, i.e. $E \in \mathscr{E}^{\text{BC}}(\mathscr{T})\subset\mathscr{E}(\mathscr{T})$, along the boundary we set $\{\boldsymbol{\sigma}(\mathbf{v})\} = \boldsymbol{\sigma}(\mathbf{v})$, $[\![\boldsymbol{\sigma}]\!]_i = \boldsymbol{\sigma}_{ij}\mathbf{n}_{Kj}^+$ and $[\![\mathbf{v}]\!]_{ij} = \mathbf{v}_i\mathbf{n}_j^+$. Thus, the DG approximation problem (1) reads as follows: Find $\mathbf{u}_h \in V_{\mathbf{p}}(\mathscr{T})$ such that

$$a^{\text{DG}}(\mathbf{u}_h, \mathbf{v}_h) = l(\mathbf{v}_h) \qquad \forall \mathbf{v}_h \in V_{\mathbf{p}}(\mathscr{T}) \tag{5}$$

where the bilinear form

$$\begin{aligned} a^{\text{DG}}(\mathbf{u},\mathbf{v}) := & \sum_{K\in\mathscr{T}} \int_K \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\epsilon}(\mathbf{v})\, d\mathbf{x} \\ & - \sum_{E\in\mathscr{E}^{\text{int}}(\mathscr{T})\cup\mathscr{E}^{\text{D}}(\mathscr{T})} \int_E \{\boldsymbol{\sigma}(\mathbf{u})\} : [\![\mathbf{v}]\!] + \{\boldsymbol{\sigma}(\mathbf{v})\} : [\![\mathbf{u}]\!]\, ds \\ & + \sum_{E\in\mathscr{E}^{\text{int}}(\mathscr{T})\cup\mathscr{E}^{\text{D}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \int_E [\![\mathbf{u}]\!] : [\![\mathbf{v}]\!]\, ds \\ & - \sum_{E\in\mathscr{E}^{\text{T}}(\mathscr{T})} \int_E (\mathbf{t}(\mathbf{u})\cdot\mathbf{n})(\mathbf{v}\cdot\mathbf{n}) + (\mathbf{t}(\mathbf{v})\cdot\mathbf{n})(\mathbf{u}\cdot\mathbf{n})\, ds \\ & + \sum_{E\in\mathscr{E}^{\text{T}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \int_E (\mathbf{u}\cdot\mathbf{n})(\mathbf{v}\cdot\mathbf{n})\, ds \end{aligned}$$

and the linear form

$$\begin{aligned} l(\mathbf{v}) := & \sum_{K\in\mathscr{T}} \int_K \mathbf{f}\cdot\mathbf{v}\, d\mathbf{x} \\ & - \sum_{E\in\mathscr{E}^{\text{D}}(\mathscr{T})} \int_E \mathbf{g}_D\cdot\boldsymbol{\sigma}(\mathbf{v})\cdot\mathbf{n}\, ds + \sum_{E\in\mathscr{E}^{\text{D}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \int_E \mathbf{g}_D\cdot\mathbf{v}\, ds \\ & + \sum_{E\in\mathscr{E}^{\text{N}}(\mathscr{T})} \int_E \mathbf{g}_N\cdot\mathbf{v}\, ds \\ & - \sum_{E\in\mathscr{E}^{\text{T}}(\mathscr{T})} \int_E (\mathbf{g}_T\cdot\mathbf{n})(\mathbf{t}(\mathbf{v})\cdot\mathbf{n})\, ds + \sum_{E\in\mathscr{E}^{\text{T}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \int_E (\mathbf{g}_T\cdot\mathbf{n})(\mathbf{v}\cdot\mathbf{n})\, ds \end{aligned}$$

where γ is the penalty constant. As proved in [43], the penalty term $\frac{\gamma p_E^2}{h_E}$ adjust for any variation in h or p in the mesh. This means that the penalty constant γ is mesh independent and the only condition that γ must satisfy is to be big enough as stated in [43]. In the authors' experience, $\gamma = 10$ is big enough for most problems. For more complicated PDEs, the definition of the penalty term can be modified in view of [44] to take into account varying coefficients in the second order term of the PDE, see for example [45–50].

The natural norm for problem (5) is the DG norm:

$$\begin{aligned} \|\,|\mathbf{u}|\,\|_{\mathscr{T}} := & \left(\sum_{K\in\mathscr{T}} \| \boldsymbol{\epsilon}(\mathbf{u}) \|_{0,K}^2 + \sum_{E\in\mathscr{E}^{\text{int}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \| [\![\mathbf{u}]\!] \|_{0,E}^2 + \sum_{E\in\mathscr{E}^{\text{D}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \| \mathbf{u} \|_{0,E}^2 \right. \\ & \left. + \sum_{E\in\mathscr{E}^{\text{T}}(\mathscr{T})} \frac{\gamma p_E^2}{h_E} \| \mathbf{u}\cdot\mathbf{n} \|_{0,E}^2 \right)^{1/2}, \end{aligned} \tag{6}$$

where $\|\cdot\|_{0,K}$ and $\|\cdot\|_{0,E}$ are respectively the L^2-norm on an element K and on an edge E.

2.3. Error estimator

This section defines the error estimator implemented in the code. In [40] the reliability and efficiency proofs of the error estimator are presented.

The error estimator is defined as

$$\eta_{\text{err}} = \sqrt{\sum_{K\in\mathscr{T}} (\eta_{R,K}^2 + \eta_{J,K}^2 + \eta_{F,K}^2)}, \tag{7}$$

where the three terms under the sum are defined as

$$\begin{aligned} \eta_{R,K}^2 &:= \frac{h_K^2}{p_K^2} \| \mathbf{f}_h + \nabla\cdot\boldsymbol{\sigma}(\mathbf{u}_h) \|_{0,K}^2, \\ \eta_{J,K}^2 &:= \frac{1}{2} \sum_{E\in\mathscr{E}^{\text{int}}(K)} \frac{\gamma^2 p_E^3}{h_E} \| [\![\mathbf{u}_h]\!] \|_{0,E}^2 + \sum_{E\in\mathscr{E}^{\text{D}}(K)} \frac{\gamma^2 p_E^3}{h_E} \| \mathbf{u}_h - \mathbf{g}_{D,h} \|_{0,E}^2 \\ &\quad + \sum_{E\in\mathscr{E}^{\text{T}}(K)} \frac{\gamma^2 p_E^3}{h_E} \| \mathbf{u}_h\cdot\mathbf{n} - \mathbf{g}_{T,h}\cdot\mathbf{n} \|_{0,E}^2, \\ \eta_{F,K}^2 &:= \frac{1}{2} \sum_{E\in\mathscr{E}^{\text{int}}(K)} \frac{h_E}{p_E} \| [\![\boldsymbol{\sigma}(\mathbf{u}_h)]\!] \|_{0,E}^2 + \sum_{E\in\mathscr{E}^{\text{N}}(K)} \frac{h_E}{p_E} \| \boldsymbol{\sigma}(\mathbf{u}_h)\cdot\mathbf{n} - \mathbf{g}_{N,h} \|_{0,E}^2 \\ &\quad + \sum_{E\in\mathscr{E}^{\text{T}}(K)} \frac{h_E}{p_E} \| \mathbf{t}(\mathbf{u}_h)\cdot\mathbf{n} \|_{0,E}^2 \end{aligned}$$

where $\mathbf{f}_h$ and $\mathbf{g}_{N,h}$ are the L^2 projections of $\mathbf{f}$ and $\mathbf{g}_N$ onto the finite element space and where $\mathbf{g}_{D,h}$ and $\mathbf{g}_{T,h}$ are piecewise polynomial approximated of traces of functions in $H^1(\Omega)$ as in [51].

2.4. Adaptivity

The code supports arbitrary high order triangular elements as

defined in Section 2.2.3 in [52]. The *hp*-adaptive strategy used here was originally proposed in [53] and consequently used in [54]. The elements are chosen for either h or p refinement using Algorithm 1.

The marking strategy in Algorithm 1 uses two threshold values δ_1 and δ_2, with $\delta_2 \geq \delta_1$, to determine what elements to refine in h and what elements in p. All the elements satisfying $\eta_K^2 > \delta_2 \eta_{\max}^2$, where $\eta_{\max}^2 = \max_{K \in \mathcal{T}} \eta_K^2$, are marked for $h-$ refinement and all elements satisfying $\delta_2 \eta_{\max}^2 \geq \eta_K^2 > \delta_1 \eta_{\max}^2$ are marked for $p-$ refinement. The remaining elements are not marked for refinement at all. To only apply h-adaptivity set $\delta_2 = \delta_1 \neq 0$. Similarly, to only apply p-adaptivity, set $\delta_2 = 1$ and $\delta_1 \neq 0$. To uniformly refine in h set $\delta_2 = \delta_1 = 0$ and to uniformly refine in p set $\delta_2 = 1$ and $\delta_1 = 0$.

3. Installation and testing

3.1. Installation

The only MATLAB add on that SIPG Linear Elastic uses is the *Symbolic Math Toolbox*™, which can be downloaded from mathworks.com. To begin the installation, clone or download the repository from GitHub. *Discontinuous-Galerkin-MATLAB* is a MATLAB directory environment that contains main functions, placed in the top directory, that can be run directly, the path_add.m script to add folders to the MATLAB search path, and folders containing routines that perform the analysis. The package has the functionality to support mesh generation with the *MESH2D*, a MATLAB based Delauney mesh generator for two-dimensional geometries [55,56] which can be downloaded from the MathWorks® website. If the user wishes, this package can be integrated into the program by copying the *MESH2D* file into the SIPG Linear Elastic directory. Care should be taken to ensure that the folder containing the package is added to the MATLAB search path, so it is advised that the user edits line 28 of path_add.m as appropriate. The three problems described in Section 5 can be run without *MESH2D*. The code analytical_problem.m needs *MESH2D*. Table 1 describes the subfolders in *SIPG_linear_elastic_v1.0*.

The reference manual can be generated automatically using the M2HTML toolbox [57]:

m2html('mfiles', 'SIPG_dir', 'htmldir','doc_dir', 'recursive','on');

where SIPG_dir is the relative path to the folder *Discontinuous-Galerkin-MATLAB* and doc_dir is the relative path to the folder where the user wants the reference manual to be saved. All functions in the *Discontinuous-Galerkin-MATLAB* code are integrated with the MATLAB help system. The installation consists in running the script path_add.m to add the necessary folders to the MATLAB search path.

3.2. Testing

After installation, the user should call the function *tests.m* in the command window. The function *tests.m* is based on the unit test framework used in Matlab. This is used to check that the program is working as expected on the user's computer by performing a set of four simulations, listed in Table 2, and thereby testing critical routines. By comparing the values from this simulation with a set of expected values stored in *.txt* files, the function checks that all routines are performing nominally.

Apart from the global stiffness matrix calculation *DG_algorithm.m* and the volumetric integral of the body force *force_integration_vol.m*, other routines common to all test problems are the error estimation algorithms *error_calc_ele.m* and the calculation of the L^2 and DG norms. Depending on the boundary conditions defined, different subroutines within the error estimation and the calculation of the DG norm will be executed and tested by *tests.m*.

The primary role of test problem 1 is to verify that the mesh adaptivity algorithms are functioning correctly. To achieve this, the program solves the problem with the displacement solution shown in Table 2 with five different mesh adaptivity methods: uniform h, uniform p, adaptive h, adaptive p and adaptive hp. Test problems 2, 3, and 4 establish that the force surface integral functions associated with each type of boundary condition are performing as expected. When executing a problem with only Neumann boundary conditions, the problem is indeterminate and rigid body motion is possible. To prevent rigid body motion, such that all displacements and rotations are zero, for instance, test problem 3, it is necessary to generate Lagrangian force components to cause the system to act in a deterministic manner. This is performed in the *UV_ave_bc.m* and *rot_ave_BC.m*, which are additional algorithms whose performance is checked by test problem 3.

Table 1
Subfolders of*SIPG_linear_elastic_v1.0*.

adaptivity	Functions to adapt the mesh.
boundary_conditions	Functions to apply average boundary condition.
error_calculation	Functions to compute the a posteriori error estimator.
example_problems	Functions to set up the example problems.
mesh_generation	Functions to generate the meshes.
norms	Functions to compute the norms of the errors of the computed solutions.
plotters	Functions to plot the solutions.
problem_set_up	Functions to set up the example problems.
rhs	Functions to compute the rhs of the linear system
solvers	Functions to solve the linear system.
stiffness_matrix	Functions to compute the DG stiffness matrix of the linear system.
tests	Functions to perform the tests.

4. Program structure

4.1. Mesh data structures and flags

We now consider the format used in the program. The initial mesh of all simulations must be a conforming mesh, i.e. no hanging node must be present. The coordinates of the nodes are stored in the variable coord and element topology information is stored in the variable etpl.mat, discussed below.

The elements used are triangular with an anticlockwise local node numbering convention, and the global node index begins at one. Global nodal coordinates are stored in the variable coord, an [x,y] list with dimensions nodes × 2, where nodes is the number of nodes in the mesh. Nodes are shared between elements if they are not hanging nodes. However, in DG methods, degrees of freedom are not shared between elements. Element topology information is stored in the MATLAB structure array etpl, which initially contains the fields mat, poly and tree; an additional field, ed_recalc, is calculated at the start of each adaptivity loop and is not required as an input. A description of each field, their input format and array dimensions is shown in Table 3.

In the mat field, the element faces are defined by stepping

Algorithm 1. hp-refinement strategy: *For parameters* δ_1, δ_2 *with* $1 \geq \delta_2 \geq \delta_1 \geq 0$.

1) Compute the maximum error $\eta_{\max}^2 = \max_{K \in \mathcal{T}} \eta_K^2$.
2) Identify the set of elements to refine in p: $\mathcal{T}_{p\_ref} = \{K \in \mathcal{T} | \delta_2 \eta_{\max}^2 \geq \eta_K^2 > \delta_1 \eta_{\max}^2\}$.
3) Increase p_K by one for $K \in \mathcal{T}_{p\_ref}$.
4) Identify the set of elements to refine in h: $\mathcal{T}_{h\_ref} = \{K \in \mathcal{T} | \eta_K^2 > \delta_2 \eta_{\max}^2\}$.
5) Identify any elements $K \in \mathcal{T} \cap \mathcal{T}'$, where $\mathcal{T}'$ is the refined mesh, that will have more than one hanging node on a face and add to $\mathcal{T}_{href}$.
6) h refine all elements $K \in \mathcal{T}_{h\_ref}$ to create the new mesh $\mathcal{T}'$.
7) Ensure for every pair of neighbours $K, K' \in \mathcal{T}'$ the variation in polynomial order is not greater than one, otherwise add K', where $p_{K'} < p_K$, to the set $\mathcal{T}'_{p\_ref}$.
8) Increase p_K by one for $K \in \mathcal{T}'_{p\_ref}$.

Table 2
Test problems and main routines used.

Problem Number	Boundary Condition Type	Expression	Main Routines Used
1	Homogenous Dirichlet	$u = \sin(\pi x)\sin(\pi y)$ $v = \sin(\pi x)\sin(\pi y)$	*h_adapt.m, p_adapt.m, hp_adapt.m, DG_algorithm.m, force_integration_vol.m*
2	Inhomogenous Dirichlet	$u = \sin(\pi x)\sin(\pi y)$ $v = \cos(\pi x)\sin(\pi y)$	*DG_algorithm.m, force_integration_vol.m, force_integration_Dirichlet.m*
3	Inhomogenous Neumann	$u = \sin(2\pi x)\sin(2\pi y)$ $v = \cos(2\pi x)\sin(2\pi y)$	*DG_algorithm.m, force_integration_vol.m, UV_ave_bc.m, rot_ave_BC.m, force_integration_Neumann.m*
4	Inhomogenous Dirchlet/ Neumann	$u = x$ $v = y$	*DG_algorithm.m, force_integration_vol.m, force_integration_Neumann.m, force_integration_Dirichlet_Neumann.m*

Table 3
Fields in etpl.

Field	Description	Format	Dimensions
mat	Element topology matrix	[node1,node2,node3]	[nels × 3]
poly	Element polynomial order	[element number,polynomial order]	[nels × 2]
tree	Element tree structure	[active flag, generation number, parent element]	[nels × 3]
ed_recalc	Flags for refined elements	[flag]	[nels]

anticlockwise around the local nodes, indexed by their global node number (row position in coord). Elements in the mesh are indexed according to their row position in etpl.mat, and are assigned a polynomial order in etpl.poly. As for the first generation of elements every element is active, etpl.tree should be initialised by setting all rows as [1,0,0]; storing information about the history of elements using this variable facilitates adaptation of the code to include mesh derefinement, which may be of interest to advanced users.

The final data structure describing the mesh is the element face topology matrix etpl_face, which has the format [element 1,element 2,local face 1, local face 2,normal x,normal y,face type]. In the code it is enforced that element 1 is an element for which the face is a complete face. In case of hanging nodes, element 2 with an hanging node in the middle of the face. This variable stores information about face connectivity for all faces in the mesh, illustrated in Fig. 1.

The local face value ranges between one and three, indexed in an anticlockwise direction around each element; normal x, normal y describe the outward facing normal vector orientation; face type is a flag used to indicate whether the face is internal, and, if it is external, the boundary condition to be imposed. Table 4, below, outlines the flagging convention used in the face type column of etpl_face. If a face is external then the value in the relevant row for element 2 in etpl_face is set to zero, as is the value of local face 2.

The mesh data structure format that SIPG Linear Elastic uses is compatible with the mesh generation algorithm *MESH2D*. Consequently, if additional information is desired by the user about the data structures discussed in this section, this can be found in the *MESH2D* documentation.

4.2. Creation of data structures and flags

The data structures and their respective fields required to solve the SIPG problem are etpl, coord and etpl_face. They are either hardcoded as. txt files, as in the case of problems 1-to-4 in Table 2 or generated with *MESH2D*. There are two stages in creating the data structures if *MESH2D* is used, this is highlighted with the analytical_problem.m example. First, the mesh input data for *MESH2D* is defined, in this case in analytical_problem_generator.m. Second the input data is used by *MESH2D* to create an element topology which is then restructured into coord, etpl and etpl_face; this happens with the routine seed_mesh_square.m.

The input data required for *MESH2D* is, node, edge and BC, and in this example are defined in analytical_problem_generator.m. node is structured as a list of n domain boundary coordinates [x,y], with size [n × 2], and edge is a list of contiguous set of edges that combine to form the boundary of the domain. Each edge in edge is defined by 2 boundary domain coordinate numbers, e.g. [n-1,n], and has the size [n × 2]. Last, BC is defined, it is of size [n×3] with the first two columns the same as edge, the third column is a face type flag which defines the type of boundary condition for each edge in edge.

Once the input data has been defined by the user, seed_mesh_square.m is called. The first operation of which is to call the routine reffne2, of *MESH2D*, which takes node and edge as in inputs and outputs a mesh in the form of an element topology and coordinate matrix, respectively stored in etpl.mat and coord. Additionally, the list conn is also outputted of size [number of external edges × 2], each row is 1 edge and contains 2 global node numbers. It is then extended by the ffnding_the_boundary.m routine to have a third column that contains the face flags for each edge. ffnding_the_boundary.m operates on the basis that the node numbers for each external edge in BC also exist in conn. Hence there exists a set of edges in conn that link between the nodes defining a boundary edge in BC, these edges thus all have the same face flags defined in BC. Now that the element topology of the initial conforming mesh has been created, the fields poly and tree of etpl can be defined as in the previous section. etpl.tree is initialised by setting all rows to [1,0,0] and etpl.poly is initialised by setting the rows to [row number,initial polynomial order]. The next stage to seed_mesh_square.m is creating etpl_face with two steps, both of which are called by etpl_face_square_func.m. First, create etpl_face for all internal faces and second, for all external faces. Both are dependent on etpl.mat and coord created by *MESH2D*, whilst the latter also depends on conn.

The creation of the internal faces is performed by etpl_face_all.m and is described with Algorithm 2. It loops over all the elements in the mesh nels and their local face numbers, 1-to-3, on lines 2 and 3. If the element el and its local face number f have not been stored in etpl_face, controlled by the if statement on line 4, the face counter face_count is increased by 1 and etpl_face_index is updated to indicate the element, el and face f have been seen by the algorithm. Then, el, f and the outward normal n to the face are stored in etpl_face on line 8. Next, a search of etpl.mat is performed to find an neighbour element el_n, this is defined as an element which shares two nodes with el. If el_n exists, then it and the local face number that contains the shared nodes are stored in etpl_face, additionally etpl_face_index is updated to indicate the element, el_n and face f_n have been seen.

Algorithm 2 will find all faces in the domain and store them in etpl_face, however the last column of the etpl_face will not be defined for the external faces. This occurs in etpl_face_ext_ffnd_square.m, described by Algorithm 3, and uses conn to assign the face flag to the external faces of etpl_face.

The last data structure to be created is ed which stores for each element its degrees of freedom and steers the local element stiffness matrices into the global matrix. It is created during the accumulation of

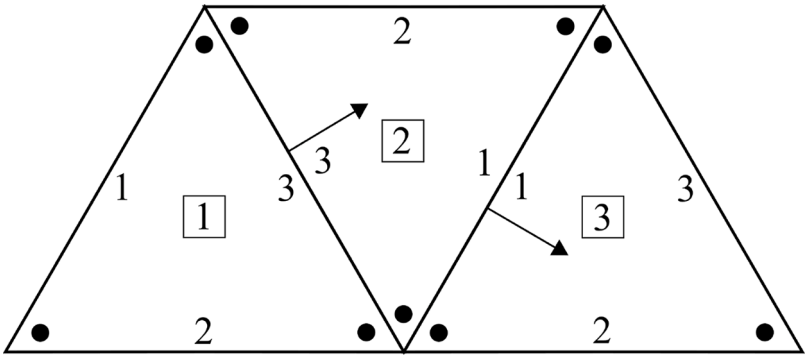

Fig. 1. Example of three elements in a 2D DG Mesh. Arrows indicate the outward normal direction, values in boxes are the element number and values on the edge are element face number (c.f. Fig. 2 of [58]).

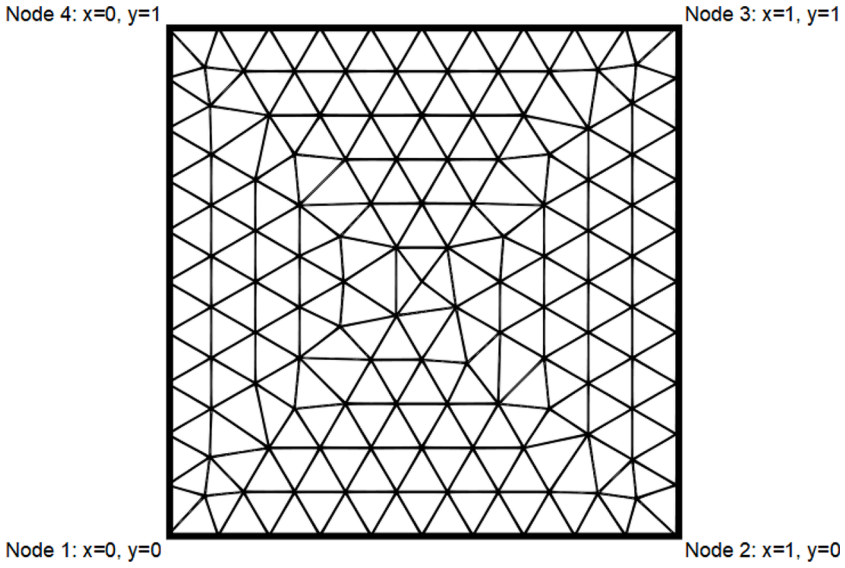

Fig. 2. Unit square domain initial mesh.

Table 4
Face type flags.

Face Type	Flag in etpl˙face
Internal Faces	1
Homogenous Dirichlet BC	2
Nonhomogenous Dirichlet BC	-2
Homogenous Neumann BC	3
Nonhomogenous Neumann BC	-3
Homogenous Mixed Dirichlet/ Neumann BC	4
Nonhomogenous Mixed Dirichlet/Neumann BC	-4

the volumetric part of the stiffness matrix calculation in vol˙int.m and has size [nels × max(NDOF)], where each row corresponds to an element number and max(DOF) is the maximum number of local degrees of freedom[1] that an active element in the mesh has. It's creation is summarised in Algorithm 4, where DOF is degrees of freedom and, NDOF is the number of degrees of freedom.

It is important to highlight three aspects of the DOF storage in ed. First, elements which are not active will have only zeros in their row, as controlled by the if statement. Second, the DOF only increases with active elements, this ensures that no rows and columns with zero values will appear in the global stiffness matrix when the local element matrices are steered into the global matrix. Last, ed is reset to zero after each adaptive step to account for different elements that have become active or changed polynomial order. This therefore means it is possible

[1] Defined by the polynomial order.

1: Make the logic matrix etpl_face_index, of size [nels]×3 and set to 0.
2: **for** Every element, el∈nels **do**
3: **for** Each local face, f∈[1,2,3] **do**
4: **if** etpl_face_index(el,f)=0 **then**
5: face_count+=1.
6: Set etpl_face_index(el,f)=1.
7: find face normal n.
8: set etpl_face(face_count,[1,3,5,6,7]) = [el,f,n(1),n(2),1]).
9: Find neighbour element, el_n, which contains the global nodes corresponding to f of el.
10: **if** el_n exists **then**
11: Find shared face of el and el_n, f_n.
12: Set etpl_face_index(el_n,f_n)=1.
13: set etpl_face(face_count,[2,4]) = [el_n,f_n]).
14: **end if**
15: **end if**
16: **end for**
17: **end for**

Algorithm 2. Internal face creation.

split etpl_face into an internal etpl_face_int and external etpl_face_ext face list.
for Every face f_glo∈etpl_face_ext **do**
    Define element el=etpl_face_ext(f_go,1).
    Define local face f=etpl_face_ext(f_go,3).
    Find the nodes corresponding to f of el.
    Search columns 1 and 2 of conn for the row that contains both nodes.
    Assign etpl_face_ext(f_glo,7)=conn(row,3).
end for
Reform etpl_face=[etpl_face_int;etpl_face_ext];

Algorithm 3. External face creation.

for Every face nel∈[1,nels **do**
    if nel is active **then**
        Define max_ed = max(ed).
        Determine the ndof for nel, NDOF.
        Store DOF in ed, ed(nel,1:NDOF)=(1:NDOF)+max_ed.
    end if
end for

Algorithm 4. ed creation.

for the DOF for an element to change during an adaptivity step.

4.3. Modification of data structures during adaptivity

During the adaptive step the mesh is adapted through changes in the size and polynomial order of elements in the mesh. When the changes occur, the element topology structure etpl, node coordinate list coord and face connectivity and boundary matrix etpl˙face need to be updated. The changes to these datasets occur in mesh˙reffne˙topology˙marking.m, the structure of which is provided by Algorithms 5 and 6.

mesh˙reffne˙topology˙marking.m is governed by the for loop on line 2 of Algorithm 5, it loops over the possible element ages in the mesh from oldest to youngest; a younger element is the result of more refinements compared to an older element. This ensures that during the refinement process a maximum of one hanging node can exist on a face. The second loop on line 3 loops over the elements that are marked for *h*-refinement, within this loop new nodes, elements and faces are created. The first

set nel = size(etpl.mat,1)
for Loop over element ages el_age, oldest to youngest **do**
    for Loop over all elements to be refined, el_ref **do**
        if el_ref is of the age el_age **then**
            Retrieve, or create, midside nodes and store, update coord accordingly.
            Find midpoints of el_ref's vertex nodes.
            Store vertex node numbers in Nn1,Nn2 and Nn3.
            Store node numbers of midsides Nn1-Nn2, Nn2-Nn3 and Nn3-Nn1 in Nn4,Nn5 and Nn6.
            Refine element and assign elements to end of etpl.mat.
            Update activity tree etpl.tree(nel+1:4,:) = [1,etpl.tree(el_ref,2)+1,el_ref] and, etpl.tree(el_ref,1)= 0.
            Assign polynomial of elements etpl.poly(nel+1:4,2)=etpl.poly(el_ref,2).
            Update etpl_face with Algorithm 6.
            Update number of elements, nel=nel+4;.
        end if
    end for
end for

Algorithm 5. Mesh adaptivity Algorithm, etpl.

Retrieve etpl_face corresponding to el_ref and store in etpl_face_current_el, and remove corresponding faces in etpl_face.
Delete the faces in etpl_face that are on el_ref.
Set new_els=(1:4)+nel.
Define local_face_list=[1 2;2 3;3 1].
for Loop through the faces j=1:3 **do**
    Find faces of etpl_face_current_el which have the face number j for el_ref, store in etpl_face_current_el.
    for Loop through rows t of etpl_face_current_el **do**
        if size(etpl_face_current_face,1) == 1 **then**
            Store etpl_face_current_el with new_els(local_face_list(t,:)) in etpl_face.
        else if size(etpl_face_current_face,1) == 2 **then**
            for i1 = 1:2 **do**
                Find the element in new_els(local_face_list(t,:)) that corresponds to row i1, and store in etpl_face.
            end for
        end if
    end for
end for
for For local faces numbers j=1:3 of element nel+4 **do**
    Find face information for element nel+4 and store in etpl_face
end for

Algorithm 6. Mesh adaptivity Algorithm, etpl˙face.

stage of the algorithm updates etpl and coord with homogeneous *h*-refinement on element el˙ref. For homogeneous refinement midside nodes are required, these are either created resulting in updates to coord, or retrieved as they may exist already, on line 5 using the coord˙check˙new.m routine[2]. The node numbers for the vertices are then stored in Nn1,Nn2 and Nn3, the midside nodes Nn4,Nn5 and Nn6, which are combined to create four new elements on line 9

```
etpl.mat(nel+1,:) = [N1 N4 N6];
etpl.mat(nel+2,:) = [N4 N2 N5];
etpl.mat(nel+3,:) = [N6 N5 N3];
etpl.mat(nel+4,:) = [N6 N4 N5];
```

where nel is the number of elements in the mesh. The assignment of the node numbers in these way ensures that the local face numbers of the new elements coincide with el˙ref. Elements nel+1-to-nel+3 use a combination of vertex nodes and midside nodes, nel+4 is a formed from only midside nodes. Next on line 10 new elements are assigned their parent polynomial order, and the active element tree etpl.tree is updated on line 11. This is followed by an updating the face connectivity matrix for all faces corresponding to el˙ref on line 12 with Algorithm 6.

The update to etpl˙face is split into two steps in Algorithm 6, the first step is defined with the for loop which spans lines 4–15, corresponding to the elements nel+1-to-nel+3 as these elements will have faces which interact with element and face information that already exists. The second step occurs with the for loop on lines 16–18 since the faces of element nel+4 are shared only with the newly created elements and hence new face information has to be created. The first section of the algorithm starts by finding the faces that correspond to el˙ref and storing these in etpl˙face˙current˙el. The local faces j=1:3 of el˙ref are then search for in etpl˙face˙current˙el, for a j they stored in etpl˙face˙current˙face. If there is only one row in etpl˙face˙current˙face, then the new elements that share face j with el˙ref are assigned the face information of etpl˙face˙current˙face and stored in etpl˙face. If there are two rows of etpl˙face˙current˙face then the face j of el˙ref has a hanging node, each new element

[2] When using the coord˙check˙new.m routine it is important to highlight that the node check is performed using element topology rather nodal position

on *j* is then assigned the row of etpl˙face˙current˙face. The row which they are assigned is found by finding which row contains an opposite element that it share two nodes with. Last the face information for nel+4 is found and stored in etpl˙face on lines 16–18, this achieved using a similar an algorithm similar to Algorithm 2.

When storing new information in etpl˙face it is important to highlight that all the new element numbers go into the first column, and all new local face numbers are stored in the third column. This ensures that the element and face information of the smaller element on the face is always in the same columns.

4.4. Description of example codes

In this section, we briefly describe the structure of the code analytical˙problem.m. The structures of the other examples example˙problem˙1.m, example˙problem˙2.m and example˙problem˙3.m are similar. The definition of the problem under consideration is done between lines 38 and 39. For each problem, multiple simulations with different choices of parameters can be done sequentially with a single call to *analytical_problem.m*. The loop running all simulations span from line 42 to line 138. Inside the loop, eight steps are executed:

1. Generation of the mesh (line 46): see the description of the data structures in Section 4.1.
2. Entering the adaptivity loop (line 56): this loop applies several adaptive steps until the variable exit is set to 1.
3. Calculating the global stiffness matrix (line 66): this step assembles the global matrix in the linear system.
4. Calculating the global rhs (lines 71–75): each type of contribution to the global rhs is computed on a different line. On line 75, all contribution are assembled.
5. Computing the solution (line 81): the MATLAB implementation of the back-slash is used for the task.
6. Error estimation (lines 87–94): the L^2 and the DG norm of the difference between the true solution and the computed solution are calculated. If the analytical solution is not available then it is set to zero. Also the *a posteriori* error estimator of the computed solution is calculated in this step.
7. Mesh refinement (line 100): the mesh is refined using the specified adaptive algorithm.
8. Postprocessing (lines 110–134): a series of plots are generated.

The definition of the problem is performed setting the variables in Table 5. Examples of such procedures are: analytical˙problem˙generator.m, analytical˙problem˙1˙generator.m, analytical˙problem˙2˙generator.m and analytical˙problem˙3˙generator.m.

In the code, two ways to set up the mesh are used. In analytical˙problem.m, the routine seed˙mesh˙square is used which calls MESH2D. In analytical˙problem˙1˙generator.m, analytical˙problem˙2˙generator.m and analytical˙problem˙3˙generator.m, the routines squaremesh, Lmesh and crackmesh are used. Such routines read the mesh from files.

4.5. How to set up a custom problem

The easiest way to set up a new problem is to make a copy of analytical˙problem.m and modify the code accordingly. The two main steps are:

1. Define the new problem defining all the variable in Table 5.
2. Define the mesh to use either loading it from file or generating using MESH2D, see Section 4.4. The user may also decide to write routines to import meshes saved in existing formats.

Optionally, the postprocessing section of the code may be changed to display the information in the requested format.

Table 5
Set up script variables.

Input	Description	Notes
node	Vertices of the desired domain (used for mesh generation)	Number of vertices by 2 array of the [x,y] coordinates of the vertices of the domain
edge	Defines the connections between the domain vertices, indexed according to their row position in nodes	Number of edges by 2 topology matrix of the connections between the domain vertices
BC	Imposed boundary conditions	See Table 4
av˙bc	Flag to indicate if the average boundary condition is used	See [42]
d˙2	Delta 2 coefficient	Array containing one value for each simulation defining the marking strategy to use, see Section 2.4
d˙1	Delta 1 coefficient	Array containing one value for each simulation defining the marking strategy to use, see Section 2.4
loop˙end	Number of adaptivity loops	1 by number of simulations array defining the no. adaptivity loops for each simulation
sim˙end	Number of separate simulations	-
E	Young's Modulus (Pa)	-
nu	Poisson's ratio	-
u, v	Displacement expression in the x,y directions	Problems with an analytical solution: Symbolic expression for the displacement solution across the domain Problems without known solutions: Symbolic expression for the imposed non-homogenous dirichlet boundary condition
traction	Array containing expressions for the imposed traction in the [x,y] directions	Problems without known solution
fx,fy	Expressions for the imposed body force in the [x,y] directions	Problems without known solution

5. Example problems

In this section, we present three example problems that demonstrate the capabilities of the package. These examples serve as a guide to problem set up by showing how the problems outlined in [40] can be analysed. These are a problem with a smooth solution on a unit square domain, a non-smooth problem on an L-shaped domain, and a crack in a plate problem, performed by executing the .m scripts *example_problem_1.m, example_problem_2.m* and *example_problem_3.m* respectively. These problems do not use the set up scripts for use with *MESH2D* discussed above to generate the mesh.

5.1. Problem with an analytical solution on unit square domain

The first example, *example_problem_1.m*, considers a small strain linear elastic problem on $(x, y) \in \Omega = (0, 1)^2$ where x and y are in meters, see Fig. 2. The problem acts in plane strain with Young's modulus $E_Y = \frac{5}{2}$Pa and a Poisson's ratio $\nu = \frac{1}{4}$. As discussed in [40], the right-hand side **f** of Equation (1) should be selected such that the exact analytical solution is

$$\mathbf{u} = \begin{Bmatrix} \sin(2\pi x)\sin(2\pi y) \\ \sin(2\pi x)\sin(2\pi y) \end{Bmatrix}.$$

For this problem, the boundary condition is homogeneous Dirichlet (zero displacement) along the whole boundary. Fig. 2 shows the mesh used for this example.

In the file analytical˙problem˙1˙generator.m all parameters to run this example are defined. For example, on line 73, the type of boundary conditions are defined. The value 2 corresponds to homogeneous

Dirichlet. The scalar material constants for E, Young's modulus (Pa) and ν, Poisson's ratio are set on line 75. on lines 77 and following lines, we define the number of simulations to perform by setting sim_end=3, and then choose the number of adaptivity loops for each simulation by setting the values in loop_end=[11 4 3], i.e. 11 *hp*-adaptivity loops, 4 *h*-adaptive loops and 3 pure-*h* loops. The penultimate step is to select the d_1 and d_2 parameters for each simulation, achieved by inputting the chosen values into the relevant columns in each array. For this example, the desired δ values are d_2=[0.7, 0.07, 0] and d_1=[0.07, 0.07, 0]. An overview of the variables used in setting up the script is given in Table 5. The program determines the analytical body force from the analytical displacement solution, and then imposes the analytical body force across the domain. These tasks are performed on lines 105 through 123 using the MATLAB symbolic toolbox. This is done in three steps:

1. Differentiating the displacement with respect to area to find the strain components across the domain
2. Multiplying the strain vector by the elastic stiffness matrix D^e to find the stress across the body
3. Differentiating the stress with respect to area to find an expression for the body force

To perform the simulations, the user should enter the command *example_problem_1*. The program will output a table detailing the progress of the mesh generation algorithm, for each adaptivity loop information about the time taken to calculate important parts of the DG algorithm, and the error estimate in the command window. Additionally, for each simulation the program will produce three figures: the first shows a colour coded plot of the boundary conditions applied to the external faces, the second is a log-log plot of the error estimate and the DG and L^2 norm convergence, and the third is a surface plot of the von Mises stress across the domain.

An important difference between SIPG Linear Elastic and the program used in [40] is that the former initialises the simulation with an unstructured mesh, and the latter initialises with a structured mesh. Despite this, comparing the convergence plot for the square problem produced with SIPG Linear Elastic, Fig. 3, to Fig. 1 of [40] it is clear that the magnitude of the error estimator, η_{err}, the error in the DG norm, $\| |\mathbf{u}-\mathbf{u_h}\| |$, are similar for a given number of degrees of freedom, and the convergence patterns also show similarities.

5.2. Non-Smooth problem on an L-Shaped domain

Next, we consider example_problem_2.m, a linear elastic problem on $(x,y) \in \Omega = (-0.5,0.5)^2/([0,0.5]\times[-0.5,0])$ where x and y are in meters, acting in plane strain with the same material constants as in example 1. The exact solution u is chosen such that the problem is singular at $(x,y) = (0,0)$,

$$\mathbf{u} = \begin{Bmatrix} (x^2+y^2)^{2/3} \\ (x^2+y^2)^{2/3} \end{Bmatrix}.$$

The relevant .m script for this problem is analytical_problem_2_generator.m, and the domain is shown in Fig. 4a. The structure of the code in analytical_problem_2_generator.m is similar to the code in analytical_problem_1_generator.m which is discussed in the previous section. Here, following the same process as in [40], we apply the non-homogenous Dirichlet boundary condition defined by **u** to all faces and therefore all external faces in etpl_face have the flag -2 associated with them and the symbolic expressions for u,v are set to equal **u**. We set loop_end=[13 9 3] to perform 13 adaptivity loops for the *hp*-adaptive simulation, 9 for adaptive-*h* refinement, and 3 adaptivity loops for the pure-*h* refinement simulation. The material constants and δ values are the same as the previous example. Again, as this is a problem with a known analytical displacement solution, the analytical body force is calculated using the steps outlined in the previous section and then imposed across the domain.

Calling the poly_plot function found in the *plotters* folder of SIPG Linear Elastic with the values of etpl and coord at the end of the first

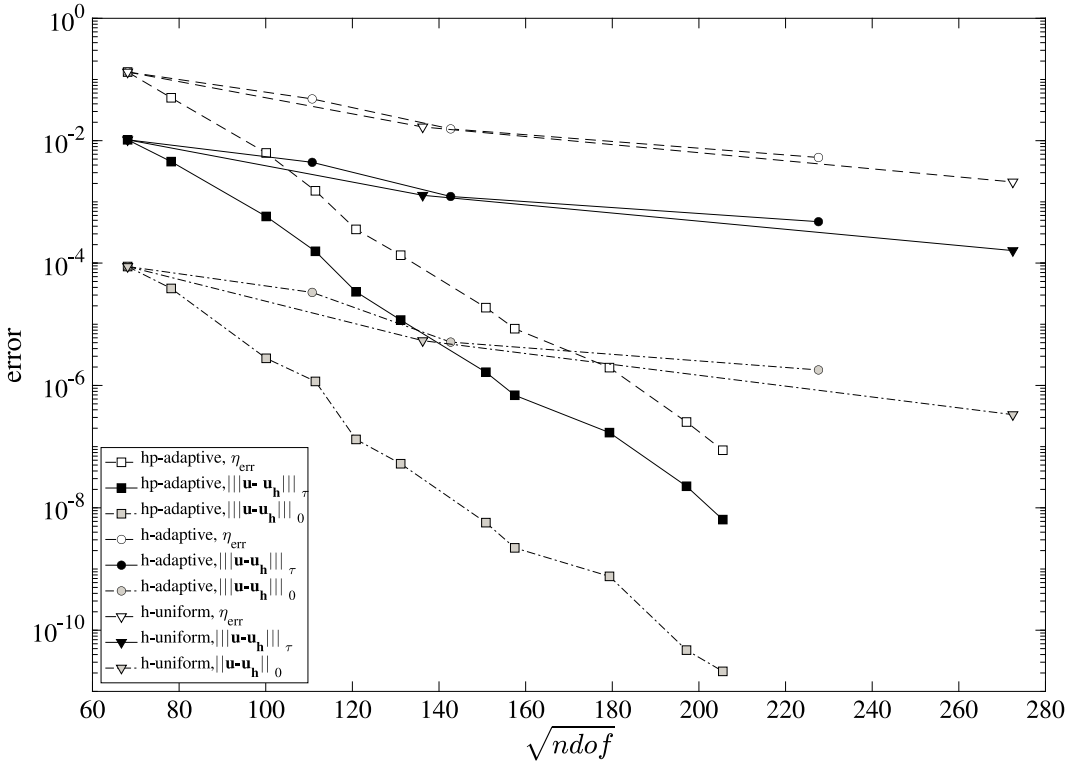

Fig. 3. Log-linear convergence of the error estimator, error in the DG and L^2 norms against mesh refinement using different adaptive strategies for the unit square domain problem.

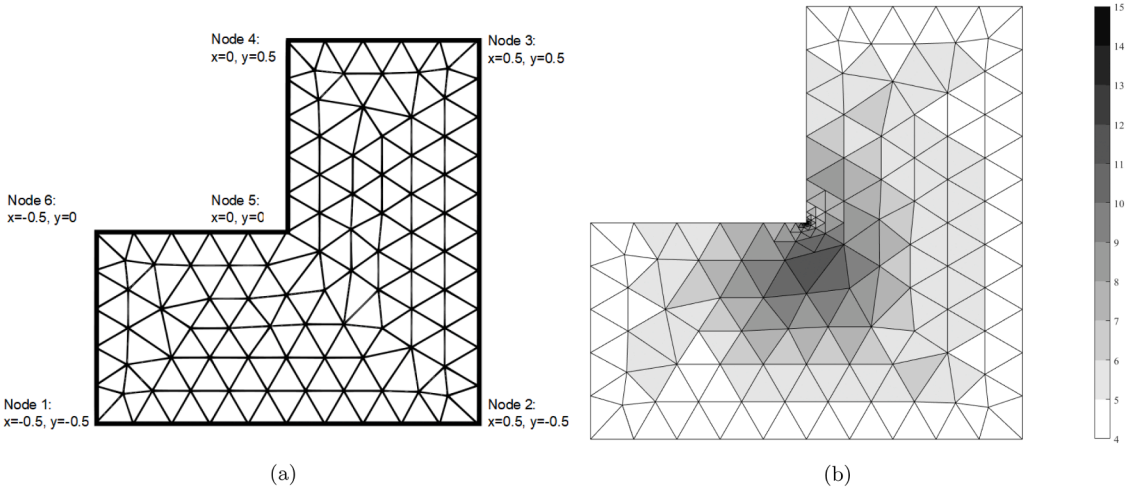

Fig. 4. L shaped domain mesh, before and after *hp* refinement. (a) L shaped domain initial mesh. (b) Afte the 15th *hp* refinement step.

simulation plots a grey-scale colormap of polynomial order distribution across the refined mesh, as shown in Fig. 4b. After the first simulation with *hp* refinement the mesh has been refined in *h* extensively around the point $x = 0, y = 0$. This is because the error estimate for elements near the singularity surpass the threshold set by the δ_2 parameter, and so are refined in *h*; the polynomial order, controlled using the δ_1 parameter, is highest for elements close to the singularity due to the large errors near this location. However, for the bulk of the mesh (where the solution is smooth) the polynomial order remains low, meaning computational effort is used more efficiently to focus on problem areas. The convergence of the error estimator, η_{err}, the error in the DG norm, $\| |\mathbf{u} - \mathbf{u_h}\| |$, and the error in the L^2 norm $\| \mathbf{u} - \mathbf{u_h}\|$ are shown in Fig. 5, below. Despite beginning with an unstructured mesh, the results of the simulation performed with SIPG Linear Elastic are remarkably similar to those reported in [40], highlighting the ability of the program to analyse non-smooth problems effectively.

5.3. Crack in a plate problem

The final example, `example_problem_3.m`, considers a problem with a stronger singularity, a crack in a plate with the same material properties as the previous examples. However, unlike the previous examples there exists no analytical solution for the displacement across the domain. The domain of the problem is described as $(x, y) \in \Omega = ((0, 1.5) \times (-1.5, -1.5)/([0, 0.5] \times 0))$, where x and y are in meters, with the crack tip at $(x, y) = (0.5, 0)$. The boundary of the problem is defined as $\partial\Omega = \Gamma_N \cup \Gamma_D \cup \Gamma_T$, where the subscripts N, D and T denote faces where the

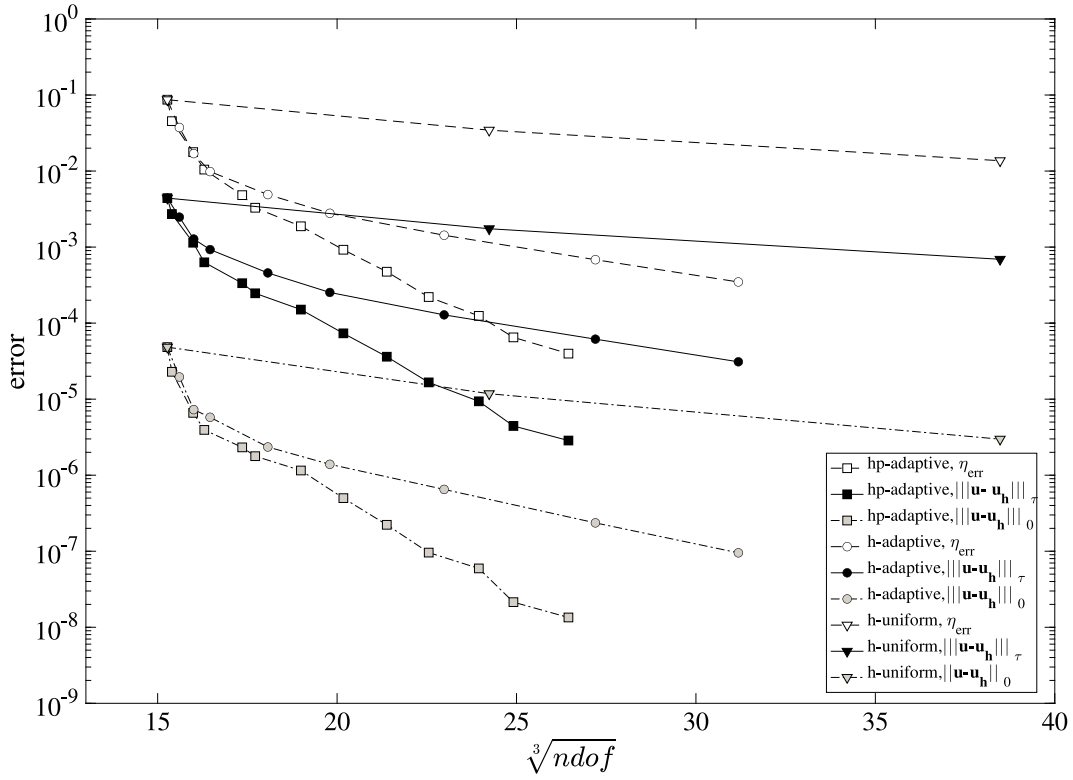

Fig. 5. Log-linear convergence of the error estimator, error in the DG and L^2 norms against mesh refinement using different adaptive strategies for the L-Shaped domain problem.

Neumann, Dirichlet and mixed boundary conditions are applied respectively. Additionally, $\Gamma_N = \Gamma_{N_1} \cup \Gamma_{N_2}$, and

$$\begin{aligned} \boldsymbol{\sigma}(\mathbf{u})\cdot\mathbf{n} &= p_r\cdot\mathbf{n} \quad \text{on } \Gamma_{N_1} = ([0, 0.5] \times -1.5) \\ \mathbf{u} &= \mathbf{0} \quad \text{on } \Gamma_D = ([0, 0.5] \times 1.5) \\ \mathbf{u}\cdot\mathbf{n} &= 0 \quad \text{on } \Gamma_T = (0.5 \times [-1.5, 1.5]) \\ \mathbf{t}(\mathbf{u})\cdot\mathbf{n}_{\|} &= 0 \quad \text{on } \Gamma_T \\ \boldsymbol{\sigma}(\mathbf{u})\cdot\mathbf{n} &= \mathbf{0} \quad \text{on } \Gamma_{N_2} = \partial\Omega\backslash(\Gamma_{N_1} \cup \Gamma_D \cup \Gamma_T) \end{aligned}$$

To set up this problem we consider the choice of variables in the script nonanalytical_problem_3_generator.m. The initial mesh, shown in Fig. 6a, is unstructured and finer than that used in [40].

Here, we wish to impose the following boundary conditions: non-homogenous Neumann between nodes 1 and 2, homogenous mixed between nodes 2 and 3, homogenous Dirichlet between nodes 3 and 4, and homogenous Neumann elsewhere. The etpl_face data for this example, stored in *crack_EtplFace.txt*, therefore provides an insight into how the flagging system in column seven of etpl_face can be used to define a more complex set of boundary conditions. For this example, d_2=[0.3 0.2 0], d_1=[0.07 0.2 0] (NOTE: The first values are different to the value reported in the error estimator paper) and loop_end=[15 15 5].

Unlike the previous examples where the right hand side **f** could be determined from the analytical displacement solution, for problems without an analytical displacement solution across the domain **f** must be specified manually. Hence, inputs for the symbolic expressions for the non-homogenous Dirichlet boundary conditions, u and v, non-homogenous Neumann boundary conditions, traction and body force to be imposed across the domain, fx and fy are required. A final point to note is that a zero input to these variables should be expressed as, for example, traction=[0*X;0*Y]and not[0;0], to conform with the format expected by the symbolic toolbox. Following these guidelines, a zero input is used for u,v,fx and fy, and the traction is specified by setting traction=[0*X; -1].

The convergence of the error estimator for the three adaptive strategies defined by the δ parameters for this problem is shown in Fig. 7. Note that for problems with a non-analytical displacement solution, the DG and L^2 norms will not converge and are therefore not shown in Fig. 7.

The roughly linear convergence of the *hp*-adaptive strategy in Fig. 7 indicates exponential convergence of the error estimator, whereas the rate of convergence of the error estimator for the *h*-adaptive and *h*-uniform strategies decreases as the number of degrees of freedom increases, a result also reported in [40]. This example therefore shows that, by choosing an appropriate adaptive strategy, SIPG Linear Elastic can also be used to analyse non-linear problems with strong singularities.

6. Conclusions

The paper presented an open source MATLAB code to solve linear elasticity using discontinuous Galerkin finite elements with *hp*-adaptivity. MATLAB was chosen as the platform for the project because it is one of the best platforms for prototyping numerical methods.

The code has been consciously designed for developers and researchers to be used as a starting point for their projects. The modularity of the code facilitates modifications like implementing a different operator or changing the definition of the error estimator. Customisation can be done without touching the algorithm responsible for the *hp*-adaptivity.

As shown in literature, *hp*-adaptivity can improve accuracy for a variety of PDE problems but it is still not commonly used due to the difficulties in implementing it correctly. *hp*-adaptivity has been included in the package to allow the users to effortlessly adopt it for their projects. As shown in the examples in Section 5, the code is capable to deliver very high accuracy which is better than what many other packages can deliver and much more than what is requested from a prototype. Given this, the present package can be used from prototyping a new numerical method to producing the results to include in publications spanning through the entire research process.

CRediT authorship contribution statement

Thomas Wiltshire: Software, Validation, Writing – original draft, Visualization. **Robert E. Bird:** Methodology, Software, Validation, Visualization. **William M. Coombs:** Conceptualization, Resources, Visualization, Supervision, Funding acquisition, Project administration. **Stefano Giani:** Conceptualization, Resources, Writing – original draft,

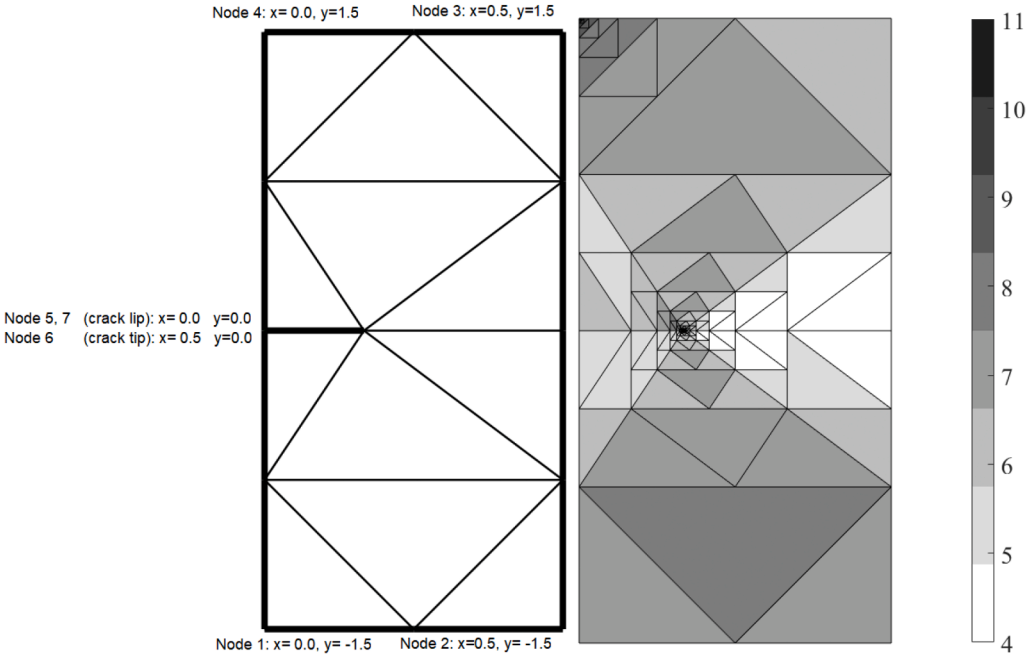

Fig. 6. Crack domain mesh, before and after *hp* refinement. (a) Initial crack mesh. (b) After the 15th *hp* refinement step.

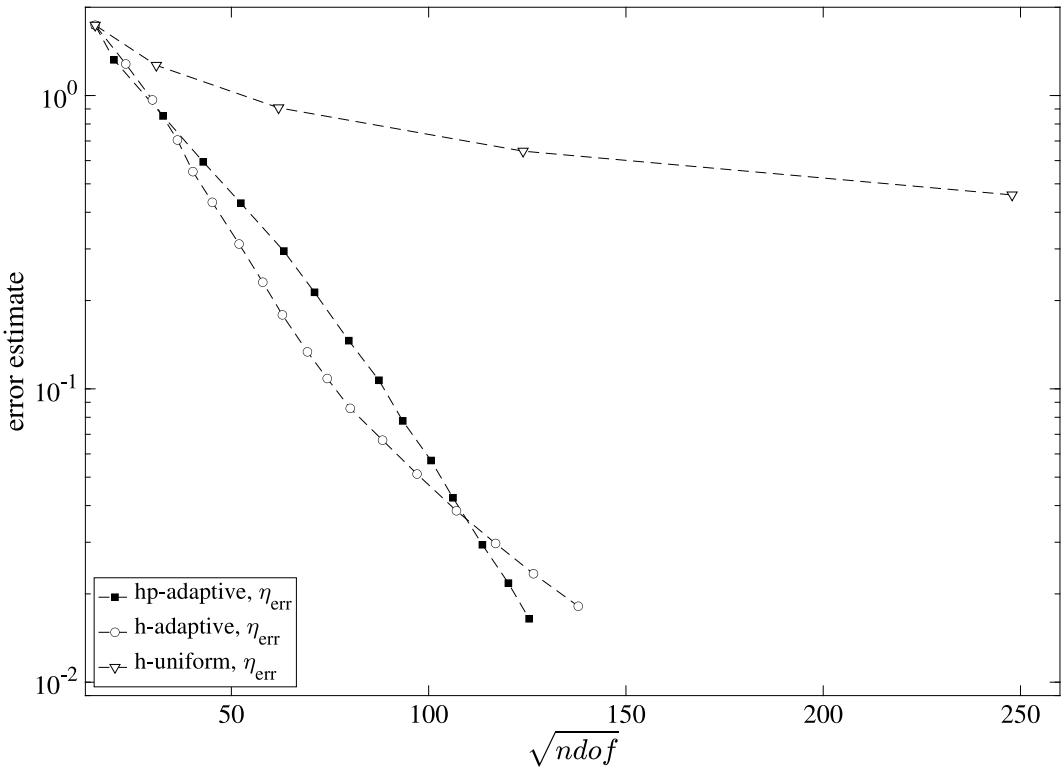

Fig. 7. Log-linear convergence of the error estimator, against mesh refinement using different adaptive strategies for the crack in a plate problem.

Writing – review & editing, Supervision, Project administration.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Robert E. Bird was supported by the Engineering and Physical Sciences Research Council [grant number EP/K502832/1]. Thomas Wiltshire was supported by the Department of Engineering, Durham University. All of the code and associated documentation is available from https://github.com/Robert-Bird/Discontinuous-Galerkin-MATLAB.

References

[1] Romer P.. Jupyter, mathematica, and the future of the research paper. https://paulromer.net/jupyter-mathematica-and-the-future-of-the-research-paper/, [accessed 21/05/2020].
[2] Talebi H, Silani M, Bordas SPA, Kerfriden P, Rabczuk T. A computational library for multiscale modeling of material failure. Comput Mech 2014;53(5):1047–71.
[3] Wu J-Y, Nguyen VP, Nguyen CT, Sutula D, Sinaie S, Bordas SPA. Chapter one - phase-field modeling of fracture. In: Bordas SPA, Balint DS, editors. Advances in Applied Mechanics. vol. 53. Elsevier; 2020. p. 1–183.
[4] Hale JS, Brunetti M, Bordas SPA, Maurini C. Simple and extensible plate and shell finite element models through automatic code generation tools. Comput Struct 2018;209:163–81.
[5] Sutula D, Elouneg A, Sensale M, Chouly F, Chambert J, Lejeune A, et al. An open source pipeline for design of experiments for hyperelastic models of the skin with applications to keloids. J Mech Behav Biomed Mater 2020;112:103999.
[6] Nguyen VP, Rabczuk T, Bordas S, Duflot M. Meshless methods: a review and computer implementation aspects. Math Comput Simul 2008;79(3):763–813.
[7] Bordas S, Menk A. Partition of unity methods. Wiley; 2022.
[8] Melenk J, Babuška I. The partition of unity finite element method: basic theory and applications. Comput Methods Appl Mech Eng 1996;139(1):289–314.
[9] Hughes TJR, Feijóo GR, Mazzei L, Quincy J-B. The variational multiscale method-a paradigm for computational mechanics. Comput Methods Appl Mech Eng 1998;166(1):3–24.
[10] Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Commun Comput Phys 2007;2(3):367–450.
[11] Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 2015;117:89–116.
[12] Agathos K, Ventura G, Chatzi E, Bordas SPA. Well conditioned extended finite elements and vector level sets for three-dimensional crack propagation. In: Bordas SPA, Burman E, Larson MG, Olshanskii MA, editors. Geometrically Unfitted Finite Element Methods and Applications. Springer International Publishing; 2017, ISBN 978-3-319-71431-8. p. 307–29.
[13] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6).
[14] Farina S, Claus S, Hale JS, Skupin A, Bordas SPA. A cut finite element method for spatially resolved energy metabolism models in complex neuro-cell morphologies with minimal remeshing. Adv Model Simul Eng Sci 2021;8(1):5.
[15] Burman E, Claus S, Hansbo P, Larson MG, Massing A. Cutfem: discretizing geometry and partial differential equations: discretizing geometry and partial differential equations. Int J Numer Methods Eng 2015;104(7):472–501.
[16] Allard J, Cotin S, Faure F, Bensoussan P-J, Poyer F, Duriez C, et al. SOFA–An open source framework for medical simulation. Stud Health Technol Inform 2007;125:13–8.
[17] Bui HP, Tomar S, Courtecuisse H, Audette M, Cotin S, Bordas SPA. Controlling the error on target motion through real-time mesh adaptation: applications to deep brain stimulation. Int J Numer Method Biomed Eng 2018;34(5):e2958.
[18] Bui HP, Tomar S, Courtecuisse H, Cotin S, Bordas SPA. Real-time error control for surgical simulation. IEEE Trans Biomed Eng 2018;65(3):596–607.
[19] Duprez M, Bordas SPA, Bucki M, Bui HP, Chouly F, Lleras V, et al. Quantifying discretization errors for soft tissue simulation in computer assisted surgery: a preliminary study. Appl Math Model 2020;77:709–23.
[20] Cangiani A, Georgoulis E, Giani S, Metcalfe S. Hp-adaptive discontinuous galerkin methods for non-stationary convection-diffusion problems. Comput Math Appl 2019;78(9):3090–104.
[21] Giani S. Reliable anisotropic-adaptive discontinuous galerkin method for simplified pn approximations of radiative transfer. J Comput Appl Math 2018;337:225–43.
[22] Jacquemin T, Tomar S, Agathos K, Mohseni-Mofidi S, Bordas SPA. Taylor-series expansion based numerical methods: aprimer, performance benchmarking and new approaches for problems with non-smooth solutions. Arch Comput Methods Eng 2020;27(5).
[23] code_aster: Structures and thermomechanics analysis for studies and research. www.code-aster.org/; [accessed 21/05/2020].
[24] Kratos multi-physics. www.cimne.com/kratos/; [accessed 21/05/2020].
[25] Feap: A finite element analysis programme. projects.ce.berkeley.edu/feap/, [accessed 21/05/2020].
[26] CAST3M. http://www-cast3m.cea.fr/, [accessed 21/05/2020].
[27] CALFEM: a finite element toolbox for matlab. github.com/CALFEM/calfem-matlab, [accessed 21/05/2020].

[28] deal.II - an open source finite element library. www.dealii.org/, [accessed 21/05/2020].
[29] hpGEM: Software library for discontinuous Galerkin methods. hpgem.org/, [accessed 21/05/2020]; https://hpgem.org/.
[30] DUNE: Distributed and unified numerics environment numerics. www.dune-project.org/, [accessed 21/05/2020].
[31] DoGPack: Discontinuous Galerkin package. www.dogpack-code.org/, accessed 21/05/2020].
[32] FreeFEM. freefem.org/, accessed 21/05/2020].
[33] MFEM - finite element discretization library. mfem.org/, accessed 21/05/2020].
[34] FEniCS Project. fenicsproject.org/, accessed 21/05/2020].
[35] PyFR. www.pyfr.org/index.php, accessed 21/05/2020].
[36] Nutlis. www.nutils.org/en/latest/, accessed 21/05/2020].
[37] SPEED. http://speed.mox.polimi.it, accessed 25/02/2022].
[38] SEISOL. https://www.seissol.org, accessed 25/02/2022].
[39] Matlab vs. python: Top reasons to choose matlab. uk.mathworks.com/products/matlab/matlab-vs-python.html, [accessed 21/05/2020].
[40] Bird RE, Coombs WM, Giani S. A posteriori discontinuous galerkin error estimator for linear elasticity. Appl Math Comput 2019;344–345:78–96.
[41] Holzapfel GA. Nonlinear solid mechanics: A Continuum approach for engineering. Wiley; 2000.
[42] Robert B, Will C, Stefano G. A quasi-static discontinuous galerkin configurational force crack propagation method for brittle materials. Int J Numer Methods Eng 2018;113(7):1061–80.
[43] Prudhomme S, Pascal F, Oden J, Romkes A. Review of a priori error estimation for discontinuous Galerkin methods. Tech. Rep. 00–27. The University of Texas at Austin; 2000.
[44] Ern A, Stephansen AF, Zunino P. A discontinuous galerkin method with weighted averages for advection-diffusion equations with locally small and anisotropic diffusivity. IMA J Numer Anal 2009;29(2):235–56.
[45] Bird R, Coombs WM, Giani S. Accurate configuration force evaluation via hp-adaptive discontinuous galerkin finite element analysis. Eng Fract Mech 2019;216:106370.
[46] Giani S. A hp-adaptive discontinuous galerkin method for plasmonic waveguides. J Comput Appl Math 2014;270:12–20.
[47] Giani S. An a posteriori error estimator for hp-adaptive discontinuous galerkin methods for computing band gaps in photonic crystals. J Comput Appl Math 2012;236(18):4810–26.
[48] Giani S, Hall E. An a-posteriori error estimate for hp-adaptive DG methods for elliptic eigenvalue problems on anisotropically refined meshes. Computing 2013;95:319–41.
[49] Giani S. High-order hp-adaptive discontinuous galerkin finite element methods for acoustic problems. Computing 2013;95:215–34.
[50] Giani S, Grubišić L, Ovall JS. Benchmark results for testing adaptive finite element eigenvalue procedures. Appl Numer Math 2012;62(2):121–40.
[51] Houston P, Schötzau D, Wihler TP. Energy norm a posteriori error estimation of hp-adaptive discontinuous galerkin methods for elliptic problems. Math Models Methods Appl Sci 2007;17:33–62.
[52] Solin P, Segeth K, Dolezel I. Higher-order finite element methods. CRC Press; 2003.
[53] Heuer N, Mellado ME, Stephan EP. Hp-adaptive two-level methods for boundary integral equations on curves. Computing 2001;67(4):305–34.
[54] Eibner T, Melenk JM. An adaptive strategy for hp-FEM based on testing for analyticity. Comput Mech 2007;39(5):575–95.
[55] Engwirda D. Locally-optimal delaunay-refinement and optimisation-based mesh generation. School of Mathematics and Statistics, The University of Sydney; 2014. Ph.D. thesis.
[56] Engwirda D. Unstructured mesh methods for the navier-stokes equations. School of Aerospace, Mechanical and Mechatronic Engineering, The University of Sydney; 2005. Master's thesis.
[57] M2HTML. https://github.com/gllmflndn/m2html, accessed 28/03/2022].
[58] Bird R, Coombs W, Giani S. Fast native-matlab stiffness assembly for sipg linear elasticity. Comput Math Appl 2017;74(12):3209–30.