Research paper

An open source \textit{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

\textbf{ARTICLE INFO}

\textbf{Keywords:}
Open source software
MATLAB
Discontinuous galerkin finite elements
Linear elasticity
\textit{hp}-Adaptivity
Researcher development

\textbf{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 \textit{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
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 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 shown 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/DGcontinuous-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 $\Omega$ be a bounded polygonal domain in $\mathbb{R}^2$ with $\partial\Omega = \Gamma_D \cup \Gamma_N \cup \Gamma_T$, where $\Gamma_D$, $\Gamma_N$ and $\Gamma_T$ are disjoint sets, and let $u$ the solution of

\[
-\nabla \cdot \sigma(u) = f \quad \text{in} \quad \Omega
\]

\[
u = g_0 \quad \text{on} \quad \Gamma_D
\]

\[
\sigma(u) \cdot n = g_N \quad \text{on} \quad \Gamma_N
\]

\[
\mathbf{u} \cdot n = g_T \quad \text{on} \quad \Gamma_T
\]

\[
t(u) \cdot n = 0 \quad \text{on} \quad \Gamma_T,
\]

where $n = (n_x, n_y)$ is the unit vector perpendicular to the boundary of $\Omega$ and pointing out and $n_i$ is the tangential unit vector to the boundary, $t(u)$ is the traction component of the stress, i.e.

\[
t(u) := \sigma(u) \cdot n.
\]

The functions $f$, $g_D$, $g_N$ and $g_T$ specify the rhs and the values along the boundaries and they are respectively in $L^2(\Omega)^2$, $H^1(\Gamma_D)$, $L^2(\Gamma_N)$ and $H^{1/2}(\Gamma_T)^2$, and $H^{1/2}(\Gamma_T)^2$. We define the strain tensor for a displacement $v$ as $\epsilon(v)_ij := \frac{1}{2}(\nabla v_i + \nabla v_j)$ and the stress as $\sigma(v) = D(\epsilon(v))$ where the matrix $D \in \mathbb{R}^{3 \times 3}$.

The set $\Gamma_D$ 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 $\mathbf{u} \in \mathcal{U} := H^1(\Omega)^2 \backslash R$, 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-definedness 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 $\Omega$ can be partitioned into a shape regular mesh $\mathcal{T}$ of triangular and affine elements and we denote with $K$ a generic element of $\mathcal{T}$. 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 $\mathcal{E}(\mathcal{T})$ and $\mathcal{E}^{in}(\mathcal{T}) \subset \mathcal{E}(\mathcal{T})$ the set of all edges of the mesh $\mathcal{T}$ and the subset of all interior edges respectively and by $\mathcal{E}^{bc}(\mathcal{T}) \subset \mathcal{E}(\mathcal{T})$ the subset of all boundary edges. The set $\mathcal{E}^{bc}(\mathcal{T})$ is partitioned in the three subsets $\mathcal{E}^{p}(\mathcal{T}), \mathcal{E}^{n}(\mathcal{T})$ and $\mathcal{E}^{t}(\mathcal{T})$ that are the sets containing the edges forming the three portions of the boundary $\Gamma_{p}, \Gamma_{n}$ and $\Gamma_{t}$. We define $h_{p}$ and $h_{t}$ 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 $\mathcal{T}$ we associate a polynomial degree $p_{K} \geq 1$ and we introduce the degree vector $p = \{p_{K} : K \in \mathcal{T}\}$ and we define $p_{\text{min}}$ as the minimum of $p_{K}$ on the mesh $\mathcal{T}$. For any $E \in \mathcal{E}(\mathcal{T})$, we introduce the edge polynomial degree $p_{E}$ by

$$p_{E} = \begin{cases} \max(p_{K}, p_{K}'), & \text{if } E = \partial K^{+} \cap \partial K^{-}, E \in \mathcal{E}^{in}(\mathcal{T}) \\ p_{E}, & \text{if } E = \partial K^{+} \cap \partial \Omega, E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T}) \end{cases}$$

(2)

where the elements $\Gamma_{p}$ and $\Gamma_{n}$ denote the two elements sharing the face $E$. Hence, for a given partition $\mathcal{T}$ of $\Omega$ and a degree vector $p$, we define the $hp$-version DG finite element space by

$$V_{h,p}(\mathcal{T}) = \left\{ v \in L^{2}(\Omega)^{2} : \forall K \in \mathcal{E}(\mathcal{T}), v_{K} \mid_{\mathcal{T}^{h,p}(K)} = v_{K} \mid_{\mathcal{T}^{h,p}(K)} \right\},$$

(3)

where $\mathcal{T}^{h,p}(K)$ is the space of polynomials of degree at most $p_{K}$ and $R$ is the set of rigid motions. We also use $\cdot$ and $\cdot$ to denote all quantities related to the elements. Given an edge $E \in \mathcal{E}(\mathcal{T})$ shared by two elements $K^{+}$ and $K^{-}$, we define the jump $\left\{ \right\}$ operator and the average $\{ \}$ operator on tensors as:

$$\left\{ v_{*} \right\} = v_{*}^{K^{+}} - v_{*}^{K^{-}}$$

$$\{ \sigma \} = \frac{1}{2} \left( \sigma^{+} + \sigma^{-} \right)$$

(4)

where $n_{K}^{+} = (n_{x}^{+}, n_{y}^{+})$ is the outward unit normal on the boundary $\partial K^{+}$ of an element $K$. Such definitions are modified on edges along the boundary of the domain, i.e. $E \in \mathcal{E}^{bc}(\mathcal{T}) \subset \mathcal{E}(\mathcal{T})$, along the boundary we set $\{ \sigma \} = \sigma_{\Gamma,p}$ and $\left\{ v \right\} = v_{\Gamma,n}$. Thus, the DG approximation problem (1) reads as follows: Find $u_{h} \in V_{h,p}(\mathcal{T})$ such that

$$a^{(h)}(u_{h}, v_{h}) = l(v_{h}) \quad \forall v_{h} \in V_{h,p}(\mathcal{T})$$

(5)

where the bilinear form

$$a^{(h)}(u, v) := \sum_{K \in \mathcal{T}} \int_{K} \sigma(u) : e(v) \, dx$$

- $\sum_{E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T})} \int_{E} \left\{ \sigma(u) \right\} : \{ v \} + \{ \sigma(v) \} : \{ u \} \, ds$

+ $\sum_{E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T})} \int_{E} [\{ u \}] : [\{ v \}] \, ds$

- $\sum_{E \in \mathcal{E}^{in}(\mathcal{T})} \int_{E} (t(u) \cdot n)(v \cdot n) + (t(v) \cdot n)(u \cdot n) \, ds$

+ $\sum_{E \in \mathcal{E}^{in}(\mathcal{T})} \frac{p_{E}^{2}}{h_{E}} \int_{E} (u \cdot n)(v \cdot n) \, ds$

and the linear form

$$l(v) := \sum_{K \in \mathcal{T}} \int_{K} f \cdot v \, dx$$

- $\sum_{E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T})} \int_{E} g_{\sigma}(\sigma(v) \cdot n) \, ds + \sum_{E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T})} \frac{p_{E}^{2}}{h_{E}} \int_{E} g_{\sigma}(\sigma(v) \cdot n) \, ds$

+ $\sum_{E \in \mathcal{E}(\mathcal{T}) \setminus \mathcal{E}^{in}(\mathcal{T})} \int_{E} g_{\sigma}(\sigma(v) \cdot n) \, ds$

where $\gamma$ is the penalty constant. As proved in [43], the penalty term $\frac{p_{E}^{2}}{h_{E}}$ adjust for any variation in $h$ or $p$ in the mesh. This means that the penalty constant $\gamma$ is mesh independent and the only condition that $\gamma$ 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:

$$\| u \|_{h,p} := \left( \sum_{E \in \mathcal{E}(\mathcal{T})} \| e(u) \|_{E}^{2} + \sum_{E \in \mathcal{E}(\mathcal{T})} \frac{p_{E}^{2}}{h_{E}} \| u \|_{E}^{2} + \sum_{E \in \mathcal{E}(\mathcal{T})} \| u \|_{E}^{2} \right)^{1/2}$$

(6)

where $\| \cdot \|_{E}$ 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

$$\eta_{h} = \sqrt{\sum_{K \in \mathcal{T}} \left( \eta_{h,k}^{2} + \eta_{h,k}^{2} + \eta_{h,k}^{2} \right)}$$

(7)

where the three terms under the sum are defined as

$$\eta_{h,k}^{2} := \frac{h_{E}}{p_{E}^{2}} \| f_{h} - \nabla \cdot \sigma(u_{h}) \|_{E}^{2}$$

$$\eta_{h,k}^{2} := \frac{1}{7} \sum_{E \in \mathcal{E}(\mathcal{T})} \frac{p_{E}^{2}}{h_{E}} \| u_{h} - g_{h} \|_{0,E}^{2} + \sum_{E \in \mathcal{E}(\mathcal{T})} \frac{p_{E}^{2}}{h_{E}} \| u_{h} - g_{h} \|_{0,E}^{2}$$

$$\eta_{h,k}^{2} := \frac{1}{7} \sum_{E \in \mathcal{E}(\mathcal{T})} \frac{h_{E}}{p_{E}^{2}} \| \sigma(u_{h}) \cdot n - g_{h} \|_{E}^{2} + \sum_{E \in \mathcal{E}(\mathcal{T})} \frac{h_{E}}{p_{E}^{2}} \| \sigma(u_{h}) \cdot n - g_{h} \|_{E}^{2}$$

where $f_{h}$ and $g_{h}$ are the $L^{2}$ projections of $f$ and $g_{b}$ onto the finite element space and where $g_{h}$ and $g_{b}$ 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 chosen for either $h$ or $p$ refinement using Algorithm 1.

The marking strategy in Algorithm 1 uses two threshold values $\delta_1$ and $\delta_2$, with $\delta_2 \geq \delta_1$, to determine what elements to refine in $h$ and what elements in $p$. All the elements satisfying $\eta_k^2 > \delta_1 \eta_{\text{max}}^2$, where $\eta_{\text{max}} = \max_{K \in \mathcal{T}} \eta_k^2$, are marked for $h$ - refinement and all elements satisfying $\delta_2 \eta_{\text{max}}^2 \geq \eta_k^2 > \delta_1 \eta_{\text{max}}^2$ 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.

| Subfolders of SIPG_linear_elastic_v1.0. |
|----------------------------------------|
| **adaptness** | Functions to adapt the mesh. |
| **boundary_conditions** | Functions to apply average boundary condition. |
| **error_calculation** | Functions to compute the 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. |

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.

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 $[\text{x,y}]$ list with dimensions $\text{nodes} \times 2$, 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...
Table 2
Test problems and main routines used.

| Problem Number | Boundary Condition Type       | Expression                          | Main Routines Used                                      |
|----------------|--------------------------------|-------------------------------------|--------------------------------------------------------|
| 1              | Homogenous Dirichlet           | \( u = \sin(\pi x)\sin(\pi y) \)  | \( h_{\text{adapt}}.m, p_{\text{adapt}}.m, hp_{\text{adapt}}.m, DG_{\text{algorithm}}.m, \text{force}_{\text{integration}}.vol.m \) |
| 2              | Inhomogenous Dirichlet         | \( u = \sin(\pi x)\sin(\pi y) \)  | \( \text{DG}_{\text{algorithm}}.m, \text{force}_{\text{integration}}.vol.m, \text{force}_{\text{integration}}.\text{Dirichlet}.m \) |
| 3              | Inhomogenous Neumann           | \( u = \cos(\pi x)\sin(\pi y) \)  | \( \text{DG}_{\text{algorithm}}.m, \text{force}_{\text{integration}}.vol.m, \text{UV}_{\text{ave}}.bc.m, \text{rot}_{\text{ave}}.BC.m, \text{force}_{\text{integration}}.\text{Neumann}.m \) |
| 4              | Inhomogenous Dirichlet/Neumann | \( u = x \)                        | \( \text{DG}_{\text{algorithm}}.m, \text{force}_{\text{integration}}.vol.m, \text{force}_{\text{integration}}.\text{Neumann}.m, \text{force}_{\text{integration}}.\text{Dirichlet}.\text{Neumann}.m \) |

Table 3
Fields in etpl.

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

The last data structure to be created is \( \text{ed}_{\text{recalc}} \), which stores information about the history of elements using this variable. The final data structure describing the mesh is the element face to edge connection 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 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 \([\text{element 1, element 2, local face 1, local face 2, normal x, normal y, face type}]\). 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 hard coded 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 \([n \times 2]\), 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 \([n \times 2]\). Last, BC is defined, it is of size \([n \times 3]\) 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 refine2, 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 \([\text{number of external edges} \times 2]\), each row is 1 edge and contains 2 global node numbers. It is then extended by the finding\_the\_boundary.m routine to have a third column that contains the face flags for each edge. finding\_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 \([\text{row number}, \text{initial polynomial order}]\). 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 find\_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 stores the local element stiffness matrices into the global matrix. It is created during the accumulation of
the volumetric part of the stiffness matrix calculation in vol_int.m and has size \([\text{nels} \times \text{max(DOF)}]\), where each row corresponds to an element number and \text{max(DOF)} is the maximum number of local degrees of freedom that an active element in the mesh has. Its creation is summarised in Algorithm 4, where \text{DOF} is degrees of freedom and \text{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.
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 \( \text{etpl} \), node coordinate list \( \text{coord} \) and face connectivity and boundary matrix \( \text{etpl\_face} \) need to be updated. The changes to these datasets occur in \text{mesh\_refine\_topology\_marking.m}\footnote{When using the \text{coord\_check\_new.m} routine it is important to highlight that the node check is performed using element topology rather nodal position}, the structure of which is provided by Algorithms 5 and 6.

\text{mesh\_refine\_topology\_marking.m} is governed by the \text{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 stage of the algorithm updates \text{etpl} and \text{coord} with homogeneous \( h \)-refinement on element \text{el\_ref}. For homogeneous refinement midside nodes are required, these are either created resulting in updates to \text{coord}, or retrieved as they may exist already, on line 5 using the \text{coord\_check\_new.m} routine.

\begin{verbatim}
1: set nel = size(etpl.mat,1)
2: for Loop over element ages el\_age, oldest to youngest do
3:   for Loop over all elements to be refined, el\_ref do
4:     if el\_ref is of the age el\_age then
5:       Retrieve, or create, midside nodes and store, update coord accordingly.
6:     end if
7:   end for
8: end for

Algorithm 5. Mesh adaptivity Algorithm, etpl.
\end{verbatim}

\begin{verbatim}
10: Update activity tree \text{etpl\_tree}(nel+1:4,:) = [1,etpl.tree(\text{el\_ref},2)+1,\text{el\_ref}] and, etpl.tree(\text{el\_ref},1)=0.
11: Assign polynomial of elements \text{etpl\_poly}(nel+1:4,:) = etpl.poly(\text{el\_ref,2}).
12: Update \text{etpl\_face} with Algorithm 6.
13: Update number of elements, nel=nel+4;
14: end if
15: end for
16: end for

Algorithm 5. Mesh adaptivity Algorithm, etpl.
\end{verbatim}

stage of the algorithm updates \text{etpl} and \text{coord} with homogeneous \( h \)-refinement on element \text{el\_ref}. For homogeneous refinement midside nodes are required, these are either created resulting in updates to \text{coord}, or retrieved as they may exist already, on line 5 using the \text{coord\_check\_new.m} routine. The node numbers for the vertices are then stored in \text{Nn1}, \text{Nn2} and \text{Nn3}, the midside nodes \text{Nn4}, \text{Nn5} and \text{Nn6}, which are combined to create four new elements on line 9

\begin{verbatim}
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,:)=\[N6 N4 N5\];
\end{verbatim}

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

The update to \text{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 \( \text{el\_ref}+1 \)-to-\( \text{el\_ref}+3 \) 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 \( \text{el\_ref}+4 \) are shared only with the newly created elements and hence new face information has to be created.

\begin{verbatim}
etpl.mat(nel+1:5)=[N1 N4 N6];
etpl.mat(nel+2:5)=[N2 N5 N3];
etpl.mat(nel+3:5)=[N5 N6 N3];
etpl.mat(nel+4:5)=[N6 N4 N5];
\end{verbatim}

1: Retrieve \text{etpl\_face} corresponding to \( \text{el\_ref} \) and store in \text{etpl\_face\_current\_el} and remove corresponding faces in \text{etpl\_face}.

2: Delete the faces in \text{etpl\_face} that are on \text{el\_ref}.
3: Set new.el=\( (1:4)+\text{nel} \).
4: Define local face list=[1;2;3;\ 1;2;3;\ 1;2;3];
5: for Loop through the faces \( j=1:3 \) do
6:  Find faces of \text{etpl\_face\_current\_el} which have the face number \( j \) for \text{el\_ref}, store in \text{etpl\_face\_current\_el}.
7:  for Loop through rows \( t \) of \text{etpl\_face\_current\_el} do
8:   if size(\text{etpl\_face\_current\_face},1)==1 then
9:     Store \text{etpl\_face\_current\_el} with new els(\text{local face list}(t,:)) in \text{etpl\_face}.
10:  else if size(\text{etpl\_face\_current\_face},1)==2 then
11:    for i=1:2 do
12:      Find the element in new.els(\text{local face list}(t,:)) that corresponds to row \( i \), and store in \text{etpl\_face}.
13:    end for
14:  end if
15: end for
16: end for
17: for For local faces numbers \( j=1:3 \) of element \( \text{nel}+4 \) do
18:  Find face information for element \( \text{nel}+4 \) and store in \text{etpl\_face}.
19: end for

Algorithm 6. Mesh adaptivity Algorithm, etpl\_face.
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 1 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 crack mesh 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

| 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 |
| av_bc | Imposed boundary conditions | See Table 4 |
| d'1   | Delta 1 coefficient | Array containing one value for each simulation defining the marking strategy to use, see Section 2.4 |
| d'2   | Delta 2 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 analytical solution: Symbolic expression for the displacement solution across the domain Problems without known solutions: Symbolic expression for the imposed non-homogeneous 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 \(E_y = 3\) Pa and a Poisson’s ratio \(\nu = \frac{1}{3}\). As discussed in [40], the right-hand side \(f\) of Equation (1) should be selected such that the exact analytical solution is

\[
\mathbf{u} = \left\{ \sin(2\pi x)\sin(2\pi y), \sin(2\pi x)\sin(2\pi y) \right\}.
\]

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
Advances in Engineering Software 171 (2022) 103147

T. Wilshire et al.

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 d’ values are d’1=[0.7, 0.07, 0] and d’2=[0.7, 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 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 domain, the second is a log-log plot of the error estimate and the analytical_p problem_2_generator.m, and the domain is shown in Fig. 4a. The structure of the code in example__problem_2.m is similar to the code in example__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,x,y are set to equal u. We set loop_end=[13 9 3] to perform 13 adaptivity loops for the hp-adaptive simulation, 9 for adapt-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 polyplot function found in the plotters folder of SIPG Linear Elastic with the values of etpl and coord at the end of the first

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

Next, we consider example_problem_2.m, a linear elastic problem on \((x,y) \in \Omega = (-0.5, 0.5)^2 / (0.05 \times [-0.5, 0])\) 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 \((x,y) = (0,0)\),

\[
\mathbf{u} = \begin{cases} 
(x^2 + y^2)^{3/2} \\
(x^2 + y^2)^{5/2} 
\end{cases}
\]

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 \(u,x,y\) are set to equal \(u\). We set loop_end=[13 9 3] to perform 13 adaptivity loops for the hp-adaptive simulation, 9 for adapt-h refinement, and 3 adaptivity loops for the pure-h refinement simulation. The material constants and \(\delta\) 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.

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.
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 \( \delta_2 \) parameter, and so are refined in \( h \); the polynomial order, controlled using the \( \delta_1 \) 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, \( \eta_{\text{err}} \), the error in the DG norm, \( \| \mathbf{u} - \mathbf{u}_h \|_\text{DG} \), and the error in the \( L^2 \) norm \( \| \mathbf{u} - \mathbf{u}_h \|_2 \) 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_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 \( (x, y) \in \Omega = ((0, 1.5) \times (-1.5, 1.5)) \setminus (0.5 \times 0) \), 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
Neumann, Dirichlet and mixed boundary conditions are applied respectively. Additionally, \( \Gamma_N = \Gamma_{N_1} \cup \Gamma_{N_2} \), and

\[
\begin{align*}
\sigma(u) \cdot n &= p \cdot n & \text{on } \Gamma_{N_i} &= (0, 0.5) \times [-1.5, 1.5] \\
u &= 0 & \text{on } \Gamma_D &= (0, 0.5) \times [1.5] \\
u \cdot n &= 0 & \text{on } \Gamma_T &= (0.5 \times [-1.5, 1.5]) \\
t(u) \cdot n &= 0 & \text{on } \Gamma_T \\
\sigma(u) \cdot n &= 0 & \text{on } \Gamma_{N_i} &= \Omega \setminus (\Gamma_{N_1} \cup \Gamma_D \cup \Gamma_T)
\end{align*}
\]

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, \( t \) and \( b \), and traction and body force to be imposed across the domain, \( f_x \) and \( f_y \) 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 \( u, v, f_x \) and \( f_y \), and the traction is specified by setting `traction=[0*X;0*Y];`.

The convergence of the error estimator for the three adaptive strategies defined by the \( \delta \) 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 of delivering 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,
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://palromer.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 - partition of unity methods. Wiley; 2022.
[4] 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.
[5] Wu J-Y, Nguyen VP, Nguyen CT, Sutula D, Sinaie S, Bordas SPA. Chapter one - partition of unity methods. Wiley; 2022.
[6] Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 2015;117:89–116.
[7] 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, Buehler M, editors. Advances in Geometrically Unfitted Finite Element Methods and Applications. Springer International Publishing; 2017. ISBN 978-3-319-71431-8. p. 307–29.
[8] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6).
[9] 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):1.
[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, Buehler M, editors. Advances in 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):1.
[15] Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Commun Comput Phys 2007;2(3):367–450.
[16] Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 2015;117:89–116.
[17] 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, Buehler M, editors. Advances in Geometrically Unfitted Finite Element Methods and Applications. Springer International Publishing; 2017. ISBN 978-3-319-71431-8. p. 307–29.
[18] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6).
[19] 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):1.
[20] Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Commun Comput Phys 2007;2(3):367–450.
[21] Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 2015;117:89–116.
[22] 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, Buehler M, editors. Advances in Geometrically Unfitted Finite Element Methods and Applications. Springer International Publishing; 2017. ISBN 978-3-319-71431-8. p. 307–29.
[23] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6).
[24] 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):1.
[25] Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Commun Comput Phys 2007;2(3):367–450.
[26] Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simul 2015;117:89–116.
[27] 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, Buehler M, editors. Advances in Geometrically Unfitted Finite Element Methods and Applications. Springer International Publishing; 2017. ISBN 978-3-319-71431-8. p. 307–29.
[28] Bordas S, Nguyen PV, Dunant C, Guidoum A, Nguyen-Dang H. An extended finite element library. Int J Numer Methods Eng 2007;71(6).
[29] 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):1.
[30] Ren W, Vanden-Eijnden E. Heterogeneous multiscale methods: a review. Commun Comput Phys 2007;2(3):367–450.
[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] Nutils. 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, Pascual F, Oden J, Romkon 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;216:106576.
[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, Griebel M, Ovall JS. Benchmark results for testing adaptive finite element eigenvalue procedures. Appl Numer Math 2012;62(2):121–40.
[51] Houston P, Schotzau 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, Stephen 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.