Durham University

Durham Research Online

Deposited in DRO:

17 August 2017

Version of attached file:

Accepted Version

Peer-review status of attached file:

Peer-reviewed

Citation for published item:

Bird, R.E. and Coombs, W.M. and Giani, S. (2017) 'Fast native-MATLAB stiffness assembly for SIPG linear elasticity.', Computers and mathematics with applications., 74 (12). pp. 3209-3230.

Further information on publisher's website:

https://doi.org/10.1016/j.camwa.2017.08.022

Publisher's copyright statement:

© 2017 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/ by/4.0/).

Additional information:

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in DRO
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full DRO policy for further details.

Durham University Library, Stockton Road, Durham DH1 3LY, United Kingdom
Tel : +44 (0)191 334 3042 | Fax : +44 (0)191 334 2971
https://dro.dur.ac.uk

Fast native-MATLAB stiffness assembly for SIPG linear elasticity

R.E. Bird[a,*], W.M. Coombs[a], S. Giani[a]

[a] *Durham University, School of Engineering and Computing Sciences, South Rd, Durham, DH1 3LE, UK*

Abstract

When written in MATLAB the finite element method (FEM) can be implemented quickly and with significantly fewer lines, when compared to compiled code. MATLAB is an attractive environment for generating bespoke routines for scientific computation as it contains a library of easily accessible inbuilt functions, effective debugging tools and a simple syntax for generating scripts. However, there is a general view that MATLAB is too inefficient for the analysis of large problems. Here this preconception is challenged by detailing a vectorised and blocked algorithm for the global stiffness matrix computation of the symmetric interior penally discontinuous Galerkin (SIPG) FEM. The major difference between the computation of the global stiffness matrix for SIPG and conventional continuous Galerkin approximations is the requirement to evaluate inter-element face terms, this significantly increases the computational effort. This paper focuses on the face integrals as they dominate the computation time and have not been addressed in the existing literature. Unlike existing optimised finite element algorithms available in the literature the paper makes use of only native MATLAB functionality and is compatible with Octave GNU. The algorithm is primarily described for 2D analysis for meshes with homogeneous element type and polynomial order. The same structure is also applied to, and results presented for, a 3D analysis. For problem sizes of 10^6 degrees of freedom

*Corresponding author
Email address: robert.e.bird@durham.ac.uk (R.E. Bird)

(DOF), both 2D and 3D computations of the local stiffness matrices were approximately 30 times faster when compared to conventional matrix formulation algorithms. Additionally, when computing the complete global stiffness matrix for problems with 10^6 DOF, both the 2D and 3D codes achieved runtimes of less than 30 s.
Keywords: efficient, MATLAB, stiffness matrix, symmetric interior penalty, discontinuous Galerkin, linear elasticity.

---

1. Introduction

Finite element analysis (FEA) is commonly used as a technique for solving partial differential equations by engineers, mathematicians and scientists. The MATLAB environment, with its library of functions and debugging procedures, allows bespoke FEA routines to be generated quickly with few lines. Examples include Coombs *et al.* [1], Sigmund [2] and others. However, an unoptimised MATLAB script will often run significantly slower than unoptimised compiled code [3]. This paper demonstrates how the advantage of using only native MATLAB to generate FEA routines is not necessarily penalised with slow run times when written in an optimised form for the symmetric interior penalty Galerkin (SIPG) method.

Significant progress on optimising FEA routines in MATLAB was achieved by Dabrowksi *et al.* [3] in 2008. The authors presented MILAMIN, an open source optimised non-native MATLAB implementation of continuous Galerkin (CG) FEA code that is capable of setting up, solving, and post processing 2D unstructured mesh problems with 10^6 degrees of freedom (DOF) in under a minute. One common method to compute the global stiffness matrix is to compute each local element matrix in turn through a series of small matrix multiplications. When creating the MILAMIN algorithm the authors recognised that there were two significant bottlenecks with this method. Firstly, two nested `for` loops are required to generate all the element stiffness matrices in a mesh. The outer loop, to loop through all the elements and the inner loop, to loop

through all the Gauss points. As MATLAB loops are inherently slow and the iteration number of the element loop is big when calculating the stiffness matrix for a large mesh (excess of 10^6 DOF) this was recognised as the first bottleneck. The second bottleneck was recognised as the time required to transfer data between the RAM and the CPU cache; this time was significantly larger than the calculation in the CPU itself - even for large calculations [3]. Matrix calculations in MATLAB are performed by the Linear Algebra Package (LAPACK) which calls the Basic Linear Algebra Subprogramme (BLAS) package. In conventional FEA codes the BLAS package is called for every Gauss point for each element individually making the total transfer time significant.

Dabrowski *et al.* [3] removed both bottlenecks by designing an algorithm where an entry in a local stiffness matrix could be computed for all elements simultaneously. Their size was consequently reduced. As an entry is calculated for all elements simultaneously the number of BLAS calls is proportional to the number of entries in the local element stiffness matrix, rather than the number of elements in the mesh. The number of BLAS calls is therefore in general smaller and no longer dependent on the size of the problem, the data transfer time is subsequently minimised removing the second bottle neck. The MILAMIN routine was further improved by maximising cache reuse, a technique known as blocking. This work has since been extended by introducing parallel vectorised stiffness matrix calculations in [4].

More recently Rahman and Valdman [5] produced a fast MATLAB script for a volumetric integral of elements with linear nodal shape functions. The focus was to start with a non-vectorised code with a standard finite element structure and then improve its computational speed through vectorisation. One of the key characteristics was to preserve the code's original structure, this ensured that the readability was not lost which is often the case in code optimisation. Lack of readability in optimised codes was also highlighted by Dabrowksi *et al.* [3]. Additionally, Anjam and Valdman [6] produced a vectorised MATLAB script for Raviart-Thomas elements used in discretizations of H(div) spaces and Nédélec elements in discretizations of H(curl) space. Andreassen *et al.* [7] provided a

comparison and discussion of computational performance between different vector computational languages to assemble a FE global stiffness matrix. Cuvelier *el al.* [8, 9] presented a more general approach to vectorise routines for multiple vector languages.

In a FEA code once all the local element stiffness matrices have been calculated they are assembled together to form a sparse global stiffness matrix. In native MATLAB this is achieved using the command `sparse` which generates a sparse matrix from triplets of data: row position, column position and the associated value. The native performance is slow, Dabrowski *et al.* [3] used `sparse2` a non-native MATLAB command. Other sparse matrix commands for MATLAB have also been created, investigated, improved and discussed in [10], who also provide their own improvement and GPU implementation.

In this paper the SIPG method for linear elasticity is implemented. Discontinuous Galerkin (DG) methods were first introduced by Reed *et al.* [11] for solving the neutron transport equation. Richter [12] prompted an extension of the original DG method to elliptical problems including linear convective-diffusion terms. However, the discontinuous approximation was only applied for the convective terms, with mixed methods for the second-order elliptic operators. Bassi and Rebay [13] introduced the complete discontinuous approximations for both the convective and second-order elliptical operators.

One arising characteristic of DG methods is that the degrees of freedom are element specific, allowing simple communication at the element interfaces. Specifically, hp-refinement is simplified due to its capability to incorporate hanging nodes at the element interfaces. These qualities make the DG method very suitable for efficient adaptive refinement to achieve high fidelity simulations [14]. The penalty for allowing this flexibility is that the number of terms to be integrated in the week form and degrees of freedom is higher for the same number and type of elements when compared to the CG method. The additional integrals are face connectivity stiffness terms which couple the unshared degrees of freedom between elements. This increases the number of calculations required to produce the global stiffness matrix, $\boldsymbol{K}$ [15], the need for efficient production

of the $\boldsymbol{K}$ matrix is therefore necessary even for relatively small problems.

This paper extends the algorithm presented by Dabrowski *et al.* [3] to include optimised integration of the face terms for SIPG, [16], for linear elastic problems in a vectorised blocked form. In this paper all the algorithms are designed for native MATLAB functionality only, a clean departure from the majority of the optimised MATLAB algorithms available in literature [3, 5, 4]. The only other known vectorised, non-blocked, MATLAB code on DG methods is by Frank *et al.* [17]. The authors in [17] consider the time dependent diffusion equation as their model problem, cast within a local DG formulation in 2D. Here we design a block vectorised code in native MATLAB, which exploits the symmetry in SIPG, to model linearly elastic problem in 2D and 3D.

The paper begins with a brief overview of the SIPG formulation for linear elasticity followed by a reformulation into a matrix form that can be computed in a vectorised algorithm in Section 2. The vectorised algorithm for computing SIPG face stiffness terms is presented and discussed in Section 3, the volume integral is omitted as it is thoroughly covered in [3]. This is followed by a discussion on: generating the local face stiffness matrices, efficiently generating global variables, Gauss quadrature on faces and sparse storage of the local stiffness matrices into the global stiffness matrix. In Section 3 the Linear2D_DG.m script is explained, with the full code available at [18]. Timing results, validation, and discussions are presented in Section 4, followed by a conclusion in Section 5.

2. Optimising the DG method

2.1. SIPG weak form for linearly elastic problems

Here we consider the following model problem on a bounded Lipschitz polygonal/polyhedral domain Ω in $\mathbb{R}^d$, $d \in \{2, 3\}$, with the boundary $\partial\Omega_N \cup \partial\Omega_D = \partial\Omega$, where $\partial\Omega_D$ and $\partial\Omega_N$ are the portions of the boundary where homogeneous Dirichlet and Neumann boundary conditions respectively applied. The strong form of the problem, for small strain hyperelasticity, is defined as

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(\boldsymbol{u}) = 0 \text{ in } \Omega, \quad \boldsymbol{\sigma}(\boldsymbol{u}) \cdot \boldsymbol{n} = \boldsymbol{g}_N \text{ on } \partial\Omega_N, \text{ and } \boldsymbol{u} = 0 \text{ on } \partial\Omega_D. \tag{1}$$

$\boldsymbol{g}_D$ and $\boldsymbol{g}_N$ are data in $[L^2(\Omega)]^d$, they are respectively the applied Dirichlet and Neumann boundary conditions. The Cauchy stress tensor is defined as $\boldsymbol{\sigma} = \partial\hat{\psi}(\boldsymbol{\varepsilon})/\partial\boldsymbol{\varepsilon}(\boldsymbol{u})$, where $\hat{\psi}$ is the free energy function for hyperelasticity, $\boldsymbol{\varepsilon}$ is small strain, $\boldsymbol{u}$ is displacement and $\boldsymbol{n}$ is the normal unit vector to the boundary. The Cauchy stress tensor can also be described $\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\varepsilon}(\boldsymbol{u})$ where $\boldsymbol{D}$ is a material stiffness tensor relating stress and strain.

This paper provides only a description of the 2D optimised code, therefore a description of the 3D element spaces and respective mesh is omitted. The polygonal finite element mesh $\mathcal{T}$ is homogeneous in element type and is in general unstructured. Two element types are defined here, the triangle and quadrilateral, however since only one element type is present in a mesh both types are referred to as K. The polygonal mesh $\mathcal{T}$ is comprised of elements K which are either the image of the reference triangle or quadrilateral under an affine elemental mapping $F_K : \widehat{K} \to K$. The homogeneous discontinuous Galerkin finite element space for triangle elements is defined as $W_{\underline{p}}(\mathcal{T}) = \{\boldsymbol{w} \in [L^2(\Omega)]^d : \forall K \in \mathcal{T}, \boldsymbol{w}|_K \in \mathcal{P}_p(K)\}$ and for quadrilateral elements as $W_{\underline{p}}(\mathcal{T}) = \{\boldsymbol{w} \in [L^2(\Omega)]^d : \forall K \in \mathcal{T}, \boldsymbol{w}|_K \in Q_p(K)\}$. Where $P_p(K)$ is the space of polynomials on K of degree less than or equal to 1 and $Q_p(K)$ is the space of polynomials on K less or equal to p in each dimension.

We denote by $\mathcal{F}(K)$ the set of the three elemental faces for the triangle, or as the set of the four elemental faces for the quadrilateral, of an element K. If the intersection $F = \partial K^+ \cap \partial K^-$ of two elements $K^+, K^- \in \mathcal{T}$ is a segment, we call F an interior face of $\mathcal{T}$. The set of all interior faces is denoted by $\mathcal{F}_I(\mathcal{T})$. Analogously, if the intersection $F = \partial K \cap \partial\Omega$ of an element $K \in \mathcal{T}$ and $\partial\Omega$ is a segment, we call F a boundary face of $\mathcal{T}$.

The SIPG method for the approximation of the model problem (1) is now introduced in the bilinear form where the homogeneous Dirichlet boundary conditions on $\partial\Omega_D$ are applied strongly. Find the displacement $\boldsymbol{u}_h \in W_{\underline{p}}(\mathcal{T})$ such

that $\boldsymbol{a}(\boldsymbol{u}_h, \boldsymbol{w}) = \boldsymbol{l}(\boldsymbol{w})$ for all $\boldsymbol{w} \in W_{\underline{p}}(\mathcal{T})$, where

$$\begin{aligned}\boldsymbol{a}_K(\boldsymbol{u}_h, \boldsymbol{w}) = &\sum_{K\in\mathcal{T}} (\boldsymbol{\sigma}(\boldsymbol{u}_h), \boldsymbol{\varepsilon}(\boldsymbol{w}))_K - \sum_{F\in\mathcal{F}_I(\mathcal{T})} \langle\{\boldsymbol{\sigma}(\boldsymbol{u}_h)\}, [\![\boldsymbol{w}]\!]\rangle_F \\ &- \sum_{F\in\cup\mathcal{F}_I(\mathcal{T})} \langle[\![\boldsymbol{u}_h]\!], \{\boldsymbol{\sigma}(\boldsymbol{w})\}\rangle_F + \sum_{F\in\mathcal{F}_I(\mathcal{T})} \beta\langle p_F^2 h_F^{-1} [\![\boldsymbol{u}_h]\!], [\![\boldsymbol{w}]\!]\rangle_F,\end{aligned} \tag{2}$$

and

$$\boldsymbol{l}(\boldsymbol{w}) = \sum_{F\in\mathcal{F}_N(\mathcal{T})} \langle \boldsymbol{g}_N, \boldsymbol{w}\rangle_F. \tag{3}$$

β is a penalty term for linear elastic SIPG defined in [19], h_f is this size of an element face, and

$$\{\boldsymbol{v}\} = \boldsymbol{v}\Big|_{F^+} \cdot \boldsymbol{n}^+ - \boldsymbol{v}\Big|_{F^-} \cdot \boldsymbol{n}^+, \tag{4}$$

$$[\![\boldsymbol{v}]\!] = \frac{1}{2}\left(\boldsymbol{v}\Big|_{F^+} + \boldsymbol{v}\Big|_{F^-}\right) \tag{5}$$

where the element faces of K^+ and K^- on an intersection $F \in \mathcal{F}_I(T)$ are respectively referred to as F^+ and F^-. Additionally for convenience $(\cdot,\cdot)$ and $\langle\cdot,\cdot\rangle$ are used, where $(a,b)_\Omega = \int_\Omega ab$ and $\langle a,b\rangle_{\partial\Omega} = \int_{\partial\Omega} ab$.

2.2. Matrix form of the SIPG method

Now that the weak form of the problem has been described it is possible to express the stress, strain and displacements in (2) as function of nodal displacements, shape functions and their derivatives, and material stiffness. Once expressed, each term in the bilinear form can be reformulated as a set of matrix multiplications which can be used to compute the stiffness matrix for SIPG. The first step to achieving the matrix formulation is decomposing the element displacements $\boldsymbol{u}_h$ into a matrix of element shape functions $\boldsymbol{N}_n$ and their corresponding coefficients $\boldsymbol{u}_n$ such that $\boldsymbol{u}_h = \boldsymbol{N}_n\boldsymbol{u}_n$ where

$$\boldsymbol{N}_n = \begin{bmatrix} N_1 & 0 & N_2 & 0 & \dots & N_{\text{nen}} & 0 \\ 0 & N_1 & 0 & N_2 & \dots & 0 & N_{\text{nen}} \end{bmatrix}, \tag{6}$$

$\boldsymbol{u}_n = [u_1, v_1, \dots, u_{\text{nen}}, v_{\text{nen}}]^T$, nen is the number of element nodes, and, u and v are respectively the displacements in the x and y directions of the Cartesian coordinate system. Similarly the test function can be represented as $\boldsymbol{w} = \boldsymbol{N}_n\boldsymbol{w}_n$.

As the small strain tensor is a function of $\boldsymbol{u}_h$, the strain can also be expressed as a set of matrix multiplications $\boldsymbol{\varepsilon} = \boldsymbol{L}\boldsymbol{N}_n\boldsymbol{u}_n$ with the additional term

$$\boldsymbol{L} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x}, \end{bmatrix} \tag{7}$$

as the small strain partial differential matrix operator [15]. From hyperelasticity the Cauchy stress is simply expressed as $\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\varepsilon} = \boldsymbol{D}\boldsymbol{L}\boldsymbol{N}_n\boldsymbol{u}_n$, where $\boldsymbol{D}$ is the plane stress or strain stiffness matrix. Substituting the matrix forms of the stress, strain and displacement into (2), and setting

$$\boldsymbol{B}_n = \boldsymbol{L}\boldsymbol{N}_n = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \dots & \frac{\partial N_{\text{nen}}}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & \dots & 0 & \frac{\partial N_{\text{nen}}}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \dots & \frac{\partial N_{\text{nen}}}{\partial y} & \frac{\partial N_{\text{nen}}}{\partial x} \end{bmatrix}, \tag{8}$$

gives,

$$\begin{aligned} \boldsymbol{a}_K(\boldsymbol{u}_h, \boldsymbol{w}) = &\sum_{K\in\mathcal{T}} (\boldsymbol{D}\boldsymbol{B}_n\boldsymbol{u}_n, \boldsymbol{B}_n\boldsymbol{w}_n)_K - \sum_{F\in\mathcal{F}_I(\mathcal{T})} \langle \{\boldsymbol{D}\boldsymbol{B}_n\boldsymbol{u}_n\}, [\![\boldsymbol{N}_n\boldsymbol{w}_n]\!]\rangle_F \\ &- \sum_{F\in\cup\mathcal{F}_I(\mathcal{T})} \langle [\![\boldsymbol{N}_n\boldsymbol{u}_n]\!], \{\boldsymbol{D}\boldsymbol{B}_n\boldsymbol{w}_n\}\rangle_F + \sum_{F\in\mathcal{F}_I(\mathcal{T})} \beta\langle p_F^2 h_F^{-1} [\![\boldsymbol{N}_n\boldsymbol{u}_n]\!], [\![\boldsymbol{N}_n\boldsymbol{w}_n]\!]\rangle_F. \end{aligned} \tag{9}$$

The test function term in the Neumann boundary condition (3) is also expressed as a matrix multiplication

$$\boldsymbol{l}(\boldsymbol{w}) = \sum_{F\in\mathcal{F}_N(\mathcal{T})} \langle \boldsymbol{g}_N, \boldsymbol{N}_n\boldsymbol{w}_n\rangle_F. \tag{10}$$

Each term in the bilinear form can now be reformulated into a set of matrix multiplications by setting (10) equal to (9), multiplying out the brackets and dividing by $\boldsymbol{w}_n$ to give

$$\begin{aligned} \sum_{F\in\mathcal{F}_N(\mathcal{T})} \int_F \boldsymbol{N}^T\boldsymbol{g}_N = &\sum_{K\in\mathcal{T}} \int_K \boldsymbol{B}_n^T\boldsymbol{D}\boldsymbol{B}_n\boldsymbol{u}_n - \sum_{F\in\mathcal{F}_I(\mathcal{T})} \int_F (\boldsymbol{C}_1 + \boldsymbol{C}_2 + \boldsymbol{C}_3 + \boldsymbol{C}_4 \\ &+ \boldsymbol{D}_1 + \boldsymbol{D}_2 + \boldsymbol{D}_3 + \boldsymbol{D}_4 + \boldsymbol{E}_1 + \boldsymbol{E}_2 + \boldsymbol{E}_3 + \boldsymbol{E}_4) \\ = &\sum_{K\in\mathcal{T}} \boldsymbol{K}_{\boldsymbol{CG}}\boldsymbol{u}_n + \sum_{F\in\mathcal{F}_I(\mathcal{T})} \boldsymbol{K}_{\boldsymbol{LF}}\boldsymbol{u}_n \\ = &(\boldsymbol{K}_{\boldsymbol{K}} + \boldsymbol{K}_{\boldsymbol{F}})\boldsymbol{U}_n = \boldsymbol{K}\boldsymbol{U}_n \end{aligned} \tag{11}$$

where $(\boldsymbol{K_K}+\boldsymbol{K_F})$ is the global stiffness matrix, $\boldsymbol{K}$, comprised of the global element stiffness matrix and the face stiffness matrix respectively. $\boldsymbol{U}_n$ is vector containing all the nodal displacements for all $K \in \mathcal{T}$ such that with respect to the mesh topology $\boldsymbol{U_n} = \sum_{K\in\mathcal{T}} \boldsymbol{u}_n(K)$. The remaining terms in (11) in their full form are

$$\boldsymbol{C}_1 = \quad \boldsymbol{B}^{+T}_n \boldsymbol{D}\boldsymbol{n}^{+T}\boldsymbol{N}^+_n\boldsymbol{u}^+_n/2 \qquad = \boldsymbol{M_{C1}}\boldsymbol{u}^+_n, \tag{12}$$

$$\boldsymbol{C}_2 = -\ \boldsymbol{B}^{+T}_n \boldsymbol{D}\boldsymbol{n}^{+T}\boldsymbol{N}^-_n\boldsymbol{u}^-_n/2 \qquad = \boldsymbol{M_{C2}}\boldsymbol{u}^-_n, \tag{13}$$

$$\boldsymbol{C}_3 = \quad \boldsymbol{B}^{-T}_n \boldsymbol{D}\boldsymbol{n}^{+T}\boldsymbol{N}^+_n\boldsymbol{u}^+_n/2 \qquad = \boldsymbol{M_{C3}}\boldsymbol{u}^+_n, \tag{14}$$

$$\boldsymbol{C}_4 = -\ \boldsymbol{B}^{-T}_n \boldsymbol{D}\boldsymbol{n}^{+T}\boldsymbol{N}^-_n\boldsymbol{u}^-_n/2 \qquad = \boldsymbol{M_{C4}}\boldsymbol{u}^-_n, \tag{15}$$

$$\boldsymbol{D}_1 = \quad \boldsymbol{N}^{+T}_n \boldsymbol{n}^+\boldsymbol{D}\boldsymbol{B}^+_n\boldsymbol{u}^+_n/2 \qquad = \boldsymbol{M_{D1}}\boldsymbol{u}^+_n, \tag{16}$$

$$\boldsymbol{D}_2 = \quad \boldsymbol{N}^{+T}_n \boldsymbol{n}^+\boldsymbol{D}\boldsymbol{B}^-_n\boldsymbol{u}^-_n/2 \qquad = \boldsymbol{M_{D2}}\boldsymbol{u}^-_n, \tag{17}$$

$$\boldsymbol{D}_3 = -\ \boldsymbol{N}^{-T}_n \boldsymbol{n}^+\boldsymbol{D}\boldsymbol{B}^+_n\boldsymbol{u}^+_n/2 \qquad = \boldsymbol{M_{D3}}\boldsymbol{u}^+_n, \tag{18}$$

$$\boldsymbol{D}_4 = -\ \boldsymbol{N}^{-T}_n \boldsymbol{n}^+\boldsymbol{D}\boldsymbol{B}^-_n\boldsymbol{u}^-_n/2 \qquad = \boldsymbol{M_{D4}}\boldsymbol{u}^-_n, \tag{19}$$

$$\boldsymbol{E}_1 = \quad \beta\frac{p^2}{h_F}\boldsymbol{N}^{+T}_n\boldsymbol{N}^+_n\boldsymbol{u}^+_n \qquad = \boldsymbol{M_{E1}}\boldsymbol{u}^+_n, \tag{20}$$

$$\boldsymbol{E}_2 = -\ \beta\frac{p^2}{h_F}\boldsymbol{N}^{+T}_n\boldsymbol{N}^-_n\boldsymbol{u}^-_n \qquad = \boldsymbol{M_{E2}}\boldsymbol{u}^-_n, \tag{21}$$

$$\boldsymbol{E}_3 = -\ \beta\frac{p^2}{h_F}\boldsymbol{N}^{-T}_n\boldsymbol{N}^+_n\boldsymbol{u}^+_n \qquad = \boldsymbol{M_{E3}}\boldsymbol{u}^+_n, \tag{22}$$

$$\boldsymbol{E}_4 = \quad \beta\frac{p^2}{h_F}\boldsymbol{N}^{-T}_n\boldsymbol{N}^-_n\boldsymbol{u}^-_n \qquad = \boldsymbol{M_{E4}}\boldsymbol{u}^-_n. \tag{23}$$

The superscripts $+$ and $-$ in equations (12) to (23) correspond to variables existing respectively in K^+ and K^-. The variable $\boldsymbol{n}^+$ is a matrix of normal components to F^+, its form for the SIPG linear elastic 2D problem is

$$\boldsymbol{n}^{+T} = \begin{bmatrix} n_x & 0 & n_y \\ 0 & n_y & n_x \end{bmatrix}. \tag{24}$$

Last the set M is defined as $M = \{\boldsymbol{M_{C1}}, \boldsymbol{M_{C2}}, \boldsymbol{M_{C3}}, \boldsymbol{M_{C4}}, \boldsymbol{M_{D1}}, \boldsymbol{M_{D2}}, \boldsymbol{M_{D3}}, \boldsymbol{M_{D4}}, \boldsymbol{M_{E1}}, \boldsymbol{M_{E2}}, \boldsymbol{M_{E3}}, \boldsymbol{M_{E4}}\}$ which is the set of partial stiffness matrices which when integrated over a single face F, and assembled together with respect to the element topology of K^+ and K^-, produce the local SIPG face stiffness matrix $\boldsymbol{K_{LF}}$ for the face F.

2.3. Vectorising the SIPG face integration

In Section 2.2 the matrix formulation of the SIPG method was expressed in (11). Traditionally from here the SIPG local face stiffness matrices are produced by computing the local stiffness for each element, K, and face, F, individually. However the size of the loops in such an algorithm is proportional to the size of the problem, i.e. the number of elements and faces. As loops in MATLAB are significantly slower than compiled code an algorithm with this structure is unacceptable to use for large problems [3]. The approach used in this paper to speed up the computation of the global stiffness is to reformulate each partial stiffness matrix in (11) so that each matrix can be computed in a vectorised blocked algorithm.

A vectorised calculation is where multiple results for a scalar equation are calculated simultaneously. This is achieved by providing the inputs to a scalar equation as vectors and only performing entry-wise operations in the code. The current form of the partial stiffness matrices in (11) can not integrated in a vectorised algorithm since the result of each partial stiffness matrix can only be found through matrix operations. To calculate the integral of a partial stiffness matrix in a vectorised algorithm the matrices in M need to multiplied out to give a resultant single matrix with entries comprising of only scalar equations. This allows an entry in a matrix to be integrated for all faces simultaneously. This removes the necessity to have a loop, that loops over all faces. The result is the size of the `for` loops in the algorithm are no longer dependent on the size of the problem. However, the Gauss point integration loop still exists, additionally two more loops are added to loop over the local nodal element combinations. These three loops are not dependent on the size of the problem and in general, expect for small problems, smaller in comparison to the number of elements in the mesh. Therefore the speed up provided by having these loops to allow vectorisation for large problems is significantly more than the loss of speed inherent with MATLAB loops.

The method for reformulating each partial stiffness matrix in equations (12) to (23) is the same, here the integral matrix term $\boldsymbol{M_{C2}}$ from (13) is used as an

example

$$\sum_{F\in\mathcal{F}_I}\left(\int_F \boldsymbol{M_{C2}}\right)\boldsymbol{u}^- = -\frac{1}{2}\sum_{F\in\mathcal{F}_I}\left(\int_F \boldsymbol{B}^{+T}\boldsymbol{D}\boldsymbol{n}^{+T}\boldsymbol{N}^-\right)\boldsymbol{u}^-. \tag{25}$$

The vector $\boldsymbol{u}^-$ is omitted in the integral as it is the solution to the linear elastic problem and so is unknown.

To reformulate $\boldsymbol{M_{C2}}$ the shape functions and the derivatives, $\boldsymbol{N}^-$ and $\boldsymbol{B}^+$, are expanded into their full form so $\boldsymbol{M_{C2}}$ becomes,

$$\boldsymbol{M_{C2}} = \begin{bmatrix} \frac{\partial N_1^+}{\partial x} & 0 & \frac{\partial N_1^+}{\partial y} \\ 0 & \frac{\partial N_1^+}{\partial y} & \frac{\partial N_1^+}{\partial x} \\ \vdots & \vdots & \vdots \\ \frac{\partial N_{nen}^+}{\partial x} & 0 & \frac{\partial N_{nen}^+}{\partial y} \\ 0 & \frac{\partial N_{nen}^+}{\partial y} & \frac{\partial N_{nen}^+}{\partial x} \end{bmatrix} \boldsymbol{D}\boldsymbol{n}^+ \begin{bmatrix} N_1^- & 0 & \dots & N_{nen}^- & 0 \\ 0 & N_1^- & \dots & 0 & N_{nen}^- \end{bmatrix}. \tag{26}$$

The form of $\boldsymbol{B}^+$ and $\boldsymbol{N}^-$ are repeated down the rows and along the columns respectively so $\boldsymbol{M_{C2}}$ can therefore be rewritten in the condensed form

$$\boldsymbol{M_{C2}} = \sum_{i=1}^{\text{nen}}\sum_{j=1}^{\text{nen}} \begin{bmatrix} \frac{\partial N_i^+}{\partial x} & 0 & \frac{\partial N_i^+}{\partial y} \\ 0 & \frac{\partial N_i^+}{\partial y} & \frac{\partial N_i^+}{\partial x} \end{bmatrix} \boldsymbol{D}\boldsymbol{n}^+ \begin{bmatrix} N_j^- & 0 \\ 0 & N_j^- \end{bmatrix}, \tag{27}$$

where i and j are respectively the local finite element nodes numbers for elements K^+ and K^- who's shape functions pre-and-post multiplied $\boldsymbol{M_{C2}}$. The material stiffness matrix $\boldsymbol{D}$ is either acting in plane strain or stress and so is represented as

$$\boldsymbol{D} = \begin{bmatrix} A & B & 0 \\ B & A & 0 \\ 0 & 0 & C \end{bmatrix}. \tag{28}$$

When multiplied out (27) becomes

$$\boldsymbol{M}^{\boldsymbol{r}}_{\boldsymbol{C2}} = \sum_{i=1}^{\text{nen}}\sum_{j=1}^{\text{nen}} \begin{bmatrix} N_j^-(A\frac{\partial N_i^+}{\partial x}n_x^+ + C\frac{\partial N_i^+}{\partial y}n_y^+) & N_j^-(B\frac{\partial N_i^+}{\partial x}n_y^+ + C\frac{\partial N_i^+}{\partial y}n_x^+) \\ N_j^-(B\frac{\partial N_i^+}{\partial y}n_x^+ + C\frac{\partial N_i^+}{\partial x}n_y^+) & N_j^-(A\frac{\partial N_i^+}{\partial y}n_y^+ + C\frac{\partial N_i^+}{\partial x}n_x^+) \end{bmatrix}. \tag{29}$$

$\boldsymbol{M}^{\boldsymbol{r}}_{\boldsymbol{C2}} \equiv \boldsymbol{M_{C2}}$, however for the sake of clarity the reduced 2-by-2 matrix form of $\boldsymbol{M_{C2}}$ is redefined. An equivalent matrix exists for all the partial stiffness

matrices in M. The new set M^r is now defined and contains the equivalent 2-by-2 matrix forms of matrices in the set M, denoted with the superscript r, such that $M^r = \{\boldsymbol{M^r_{C1}}, \boldsymbol{M^r_{C2}}, \boldsymbol{M^r_{C3}}, \boldsymbol{M^r_{C4}}, \boldsymbol{M^r_{D1}}, \boldsymbol{M^r_{D2}}, \boldsymbol{M^r_{D3}}, \boldsymbol{M^r_{D4}}, \boldsymbol{M^r_{E1}}, \boldsymbol{M^r_{E2}}, \boldsymbol{M^r_{E3}}, \boldsymbol{M^r_{E4}}\}$. All the entries in $\boldsymbol{M_{C2}}$ are now represented by 4 scalar equations which are looped over the indices (i,j).

3. Code Assembly

The complete code layout is summarised by Algorithm 1, and correlates to lines of Linear2D_DG.m, the optimised SIPG .m script provided by [18]. The algorithm contains three stages:

1. Area integral: lines 1-5. MATLAB code: lines 22-64.
2. SIPG face integral: lines 6-11. MATLAB code: lines 93-310.
3. Sparse storage: lines 12. MATLAB code: lines 311-350.

In stage 2 the SIPG face integral computes the local face stiffness matrix $\boldsymbol{K_{LF}}$ for all faces in the mesh. Stage 2 dominates the number of lines in the code due to having 12 terms to evaluate rather than just one like in stage 1 (11). In stage 3 the local face stiffness matrices are assembled into a sparse global stiffness matrix completing the algorithm. The optimised SIPG area integral is not discussed as it is identical to the optimised CG area integral in [3] expect that the elements do share degrees of freedom. This paper focuses on the novel implementation of the blocked vectorised integration of the partial stiffness matrices in M to produce the global face stiffness matrix $\boldsymbol{K_F}$. When considering the fast vectorised computation of the SIPG face terms in (11) there are six main aspects to address: Optimising the CPU cache reuse (blocking), the structure of the vectorised algorithm, memory allocation, reducing the number of BLAS operations, generating variables for the blocked algorithm, and the reference Gauss point locations. In the following sections each of these points will be addressed in turn.

Section 2.3 demonstrated that the matrices in M could be represented by a repeated 2-by-2 matrix of scalar equations looped over the nodal indices (i,j).

Algorithm 1 Complete Code layout.

```
for Area block do
    for Area Gauss blocks do
        Area integral
    end for
end for
for SIPG face block do
    for Gauss point loop do
        SIPG face integral set-up: Figure 3
        SIPG face integral: Figure 1
    end for
end for
Sparse storage: Figure 5
```

Arranging the partial stiffness matrices in M into their equivalent form in M^r means that each partial stiffness matrix is constructed from entries, each of which are scalar equations that are applicable to all finite elements in the mesh. Reformulating the matrices in M into the form in M^r allows the integral of each entry in the partial stiffness to be computed for all faces simultaneously in a vectorised algorithm; the schematic for such an algorithm in MATLAB is represented in Figure 1. The following subsections use the matrix $\boldsymbol{M^r_{C2}}$ as an example.

3.1. Blocking

When the CPU performs a BLAS operation the best performance is achieved when all the data required for the operation resides in the lowest level of cache, as this is fastest accessed. However when the data size is too large to reside entirely in the cache, sections are stored on higher levels of CPU cache or the RAM, both of which are slower to access. The technique for maximising the vector size, with the condition that the data for a BLAS operation resides in the cache memory, is called blocking. A vector integral calculation for an entry

in a partial stiffness matrix can exceed the CPU cache size. In the case where the cache memory is exceeded the set of faces $\mathcal{F}(\mathcal{T})$ is split into blocks of faces, defined as SIPG face blocks. The vector calculation for a matrix entry is now performed for each block in turn so the cache reuse is maximised, reducing the overall run time. This process is dictated by the `for` loop on line 2, Figure 1, which runs through all the SIPG face blocks in the mesh.

3.2. Structure

The structure of the algorithm which generates the SIPG local face stiffness matrix is described in Figure 1. It is characterised by four `for` loops appearing on lines 2, 3, 6, and 11. The first loop, loops through all the SIPG face blocks. The second loop is the Gauss point loop which numerically integrates the partial stiffness matrices in M^r to generate the local face stiffness matrix (11) for all faces in the SIPG face block. The final two loops go through all nodal combinations (i, j) in the matrices of the set M^r which when integrated form the local face stiffness matrices in (11).

3.3. Reducing the number of BLAS operations

It is possible to take advantage of the structure of the matrices in the set M^r to reduce the number of BLAS operations. As an example, the entry (1,1) of $\boldsymbol{M^r_{C2}}$ can be split into two components; a component which varies with row number `i` represented in Figure 1 as `C2t_11` on line 9, and one with column number `j`. All entries of $\boldsymbol{M^r_{C1}}$, $\boldsymbol{M^r_{C2}}$, $\boldsymbol{M^r_{C3}}$, and $\boldsymbol{M^r_{C4}}$ can be split into two components in the same way. The component which is a function of `i` requires more BLAS operations and is calculated outside the inner node loop (line 9), Figure 1. The `i` component is then multiplied with the `j` component and added to `glob_pn` on line 14. This reduces the number of BLAS calls, the computation time, and time associated with calling the routine.

An equivalent $\boldsymbol{M^r_{D2}}$ matrix can be constructed for $\boldsymbol{M_{D2}}$, (11). Unlike $\boldsymbol{M^r_{C2}}$, the components of entries in $\boldsymbol{M^r_{D2}}$ which are a function of column number require more BLAS operations than those which are a function of row number. As

an example, the entry (1,2) of $\boldsymbol{M^r_{D2}}$ can be split into two components; a component which varies with column number `j2` represented in Figure 1 as `D2t_12` on line 10, and one with row number `i2`. The column index `j2` is defined on (line 7) and the row index `i2` is defined on (line 12), Figure 1. The multiplication of the column and row dependent variables, as with $\boldsymbol{M^r_{C2}}$, still occurs in the inner loop (line 14), Figure 1. Equivalently all entries of $\boldsymbol{M^r_{D1}}$, $\boldsymbol{M^r_{D2}}$, $\boldsymbol{M^r_{D3}}$, and $\boldsymbol{M^r_{D4}}$ can be split into two components.

The number of BLAS calls can be reduced further by utilising the symmetry of the global stiffness matrix so that only the upper triangular components of the local stiffness matrices need to be calculated. The `for` loop nodal indices (lines 6 and 11 of Figure 1) are therefore restricted to this part of the matrix. However, in order to keep the size of the node index loops the same between $\boldsymbol{M^r_{C2}}$ and $\boldsymbol{M^r_{D2}}$, `j2` and `i2` of $\boldsymbol{M^r_{D2}}$ are looped through in reverse order (lines 7 and 12 of Figure 1). Lastly, the MATLAB indices `j2` and `i2` refer to variables in $\boldsymbol{M^r_{D1}}$, $\boldsymbol{M^r_{D2}}$, $\boldsymbol{M^r_{D3}}$ and $\boldsymbol{M^r_{D4}}$, and the, `i` and `j`, indices refer to variables in $\boldsymbol{M^r_{C1}}$, $\boldsymbol{M^r_{C2}}$, $\boldsymbol{M^r_{C3}}$, $\boldsymbol{M^r_{C4}}$, $\boldsymbol{M^r_{E1}}$, $\boldsymbol{M^r_{E2}}$, $\boldsymbol{M^r_{E3}}$ and $\boldsymbol{M^r_{E4}}$.

The loop indices, `i` and `j`, correspond to the node number of elements in the local matrix. The assembly of all matrices in M for a face F results in a symmetric matrix, therefore only the upper triangular entries of each matrix in M need to be computed, reducing the number of BLAS calls. For a nodal combination (`i`,`j`) the partial stiffness matrices in M^r will provide a two-by-two matrix for the degrees of freedom that exist at these nodes. When considering nodes on the leading diagonal, i.e. when `i==j`, only the upper triangular components of the matrices in M^r are required. An `if` statement is present (line 15 of Figure 1) so all the entries in a matrix of M^r are computed only if `i<j`, and the lower triangular components are omitted if `i==j`.

To complete the global stiffness matrix formulation, the transpose of the global matrix is added to itself. To avoid doubling values on the leading diagonals of the local matrix, diagonal terms of the $\boldsymbol{M^r}$ matrix are divided by 2 when `i==j` by `half(i,j)`. `half(i,j)` is a simple script added which returns a value 0.5 if `i==j` and 1 otherwise.

3.4. Memory allocation

Memory for variables which increase in size during the nodal loops are pre-allocated, this prevents reallocation of the variables on the RAM which reduces the run time. The partial stiffness matrices in M^r can be split into four sets. Each set can be summed together to form

$$\boldsymbol{G_s} = \sum_{F \in \mathcal{F}_I} \int_F \boldsymbol{M_{Cs}} + \boldsymbol{M_{Ds}} + \boldsymbol{M_{Es}} \quad \text{where} \quad s = 1 \ldots 4. \tag{30}$$

For the faces F all the components of a set, for example $s = 1$,

$$\boldsymbol{G_1} = \sum_{F \in \mathcal{F}_I} \int_F \boldsymbol{M_{C1}} + \boldsymbol{M_{D1}} + \boldsymbol{M_{E1}} \tag{31}$$

reside in the same location in the global stiffness matrix, therefore only one storage variable needs to be preallocated for $\boldsymbol{M_{C1}}$ $\boldsymbol{M_{D1}}$ and $\boldsymbol{M_{E1}}$. In Figure 1 the storage variable `glob_2` is defined for $\boldsymbol{G_2}$ on line 1, Figure 1, for all $F \in \mathcal{F}(\mathcal{T})$. Performance improvements where found when a second temporary storage variable was used during the local matrix calculation, `glob_pn` defined on line 4, which corresponded to all faces in the current SIPG face block for the set $\boldsymbol{G_2}$. Once integration is completed for the current SIPG face block, `glob_pn` is stored into `glob_2` on line 21 of Figure 1.

The variable `glob_pn` is a three dimensional array; the first dimension corresponds to the face numbers in the SIPG block, the second and third dimensions respectively correspond to the degrees of freedom of the finite element that pre-and-post-multiplied the partial stiffness matrix. As an example the local degrees of freedom of the entry (1,1) of $\boldsymbol{M^r_{C2}}$ are a function of node numbers `i` and `j`. The degrees of freedom are provided by the variables `Ai` and `Aj`, they are used to steer the entry (1,1) into the appropriate second and third dimension of `glob_pn` (line 14 of Figure 1). Equivalently entry (1,2) of $\boldsymbol{M^r_{D2}}$ is a function of the node numbers `i2` and `j2`. Here the degree of freedom numbers `Bi` and `Bj+1` store entry (1,2) into the appropriate position in `glob_pn` (line 16 of Figure 1).

3.5. Generating variables for blocked algorithm

When integrating an entry of a partial stiffness matrix simultaneously for multiple SIPG faces, the shape function and their derivatives for the scalar equation for that entry must be in vector form. To compute a local SIPG face stiffness matrix, $\boldsymbol{K_{LF}}$, information is required from both the K^+ and K^- elements sharing a face. During mesh generation the face connectivity for all faces in the mesh $\mathcal{F}(\mathcal{T})$ are stored in the face connectivity matrix `etpl_face` where a column correlates to face number in $\mathcal{F}(\mathcal{T})$, Figure 2b.

The script represented in Figure 3 runs on line 5 of Figure 1. The true representation of first two `for` loops, the block and Gauss point integration loop (lines 1 and 4 of Figure 3), is in Figure 1, however these loops are shown in Figure 3 for clarity. The third loop corresponds to the local element face number `fn`. This is required as local shape function values, `Nr`, and their local derivatives, `dNr`, are unique to a local element face.

To compute global shape function derivative terms, `dNx_p` and `dNy_p`, for multiple elements simultaneously, only one local face can be considered at time. Therefore manipulation of `etpl_face` is required to only consider SIPG faces in the current block and current local face number. `etpl_face` contains the face information for all faces in $\mathcal{F}(\mathcal{T})$. However as discussed in Section 3.1 the faces are split into SIPG face blocks which are considered one at a time during the vectorised computation. The information for the faces in the current SIPG face block is selected from `etpl_face` and stored in `etpl_face_block` by the index `block_index`. Additionally the shape functions and their derivatives can only be computed for one local face number, `fn`, at time governed by the face loop on line 5 of Figure 3. Therefore the face information for elements K^+ with the current face number `fn` is stored in `etpl_face_block_fn` on line 7.

The rows of `dNx_p`, and `dNy_p`, correspond to the same ordering of elements in `etpl_fac_block(:,1)`. The columns correspond to a shape function number which is selected by `i`, `j`, `i2` or `j2` in Figure 1. Their computation occurs in several large matrix operations. First the Jacobian components `Jxp` and `Jyp` are

computed on lines 10-11, through

$$\underbrace{\begin{bmatrix} \frac{\partial x_1}{\partial \xi} & \frac{\partial x_2}{\partial \xi} & \dots & \frac{\partial x_{nfb}}{\partial \xi} \\ \frac{\partial x_1}{\partial \eta} & \frac{\partial x_2}{\partial \eta} & \dots & \frac{\partial x_{nfb}}{\partial \eta} \end{bmatrix}}_{\texttt{Jxp}} = \underbrace{\begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \dots & \frac{\partial N_{nen}}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \dots & \frac{\partial N_{nen}}{\partial \eta} \end{bmatrix}}_{\texttt{dNr(indx\_dNr(fn,gp),:)}} \underbrace{\begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^{nfb} \\ x_2^1 & x_2^2 & \cdots & x_2^{nfb} \\ \vdots & \vdots & \dots & \vdots \\ x_{nen}^1 & x_{nen}^2 & \cdots & x_{nen}^{nfb} \end{bmatrix}}_{\texttt{coord\_xp}}, \quad (32)$$

where the subscript $_{nfb}$ corresponds to the number of DG faces in the block. The Jacobian determinant and its inverse are computed in an explicit manner on lines 12-14. The global shape function derivatives for the current face, `fn`, are calculated on lines 15-16, using

$$\underbrace{\begin{bmatrix} \frac{\partial N_1}{\partial x_1} & \frac{\partial N_2}{\partial x_1} & \dots & \frac{\partial N_{nen}}{\partial x_1} \\ \frac{\partial N_1}{\partial x_2} & \frac{\partial N_2}{\partial x_2} & \dots & \frac{\partial N_{nen}}{\partial x_2} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial N_1}{\partial x_{nfb}} & \frac{\partial N_2}{\partial x_{nfb}} & \dots & \frac{\partial N_{nen}}{\partial x_{nfb}} \end{bmatrix}}_{\texttt{dNx\_p(index\_p,:)}} = \underbrace{\begin{bmatrix} \frac{\partial \xi}{\partial x_1} & \frac{\partial \eta}{\partial x_1} \\ \frac{\partial \xi}{\partial x_2} & \frac{\partial \eta}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial \xi}{\partial x_{nfb}} & \frac{\partial \eta}{\partial x_{nfb}} \end{bmatrix}}_{\texttt{invJxp}} \underbrace{\begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \dots & \frac{\partial N_{nen}}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \dots & \frac{\partial N_{nen}}{\partial \eta} \end{bmatrix}}_{\texttt{dNr(indx\_dNr(fn,gp),:)}}. \quad (33)$$

The result is stored into `dNx_p` and `dNy_p` with `index_p`, this ensures the element ordering remains consistent with `etpl_face_block`.

The shape functions are only dependent on their local position and therefore local value `Nr`. The values are stored into the matrix `Np`, again with `index_p` to ensure consistent element ordering with `etpl_face_block`.

The algorithm in Figure 3 is only applicable to K^+ elements but with a few simple changes can be for K^- elements: line 6 change `etpl_face_block(:,3)` to `etpl_face_block(:,4)`, line 8 change `etpl_face_block_fn(:,1)` to `etpl_face_block_fn(:,2)`, line 15 and 16 change `dNx_p` and `dNy_p` to `dNx_n` and `dNy_n` and lastly line 17 change `Np` to `Nn`. Gauss points along a face for K^- elements are considered in reverse order so that they align with K^+ Gauss points in the global domain; MATLAB code: lines 153-154.

3.6. Reference Gauss point locations

For each local face, `fn`, the Gauss point locations in the reference frame are hard coded into the algorithm. Their positions on a face are used to generate local shape functions and their derivatives. Each face in the reference frame is numbered as shown for a triangle element, Figure 4a, and quadrilateral element, Figure 4b. The Gauss points on F^+ are numbered clockwise whilst on F^- they are anticlockwise. This ensures that the Gauss points for two connected elements align in the global domain.

The face integrals are performed with respect to the reference local face coordinate $\zeta \in [-1, 1]$. To determine the shape functions and shape function derivative values from ζ, it is necessary to convert from the reference line domain to reference element domain, coordinates $\xi \in [0, 1]$ and $\eta \in [0, 1]$, with

$$\xi = \frac{(\zeta + 1)(\xi_a - \xi_b)}{2} + \xi_b \tag{34}$$

and

$$\eta = \frac{(\zeta + 1)(\eta_a - \eta_b)}{2} + \eta_b. \tag{35}$$

Here, a refers to the most clockwise vertex existing at the end of the face, and b the previous vertex. As an example on the triangular element, Figure 4a, face 2, $a = \mathrm{A}$ and $b = \mathrm{B}$ but for face 1 would be, $a = \mathrm{B}$ and $b = \mathrm{C}$. Using the values of ξ and η, mapped from ζ, the shape functions and the reference shape functions derivatives can be determined for each Gauss point location on each face. The face calculations use standard Gauss quadrature weights and locations.

3.7. Sparse Storage

The summation of all local face stiffness matrices forms a global stiffness matrix, $\boldsymbol{K_F}$ in (11). The global numbering for the degrees of freedom along the rows and columns of the local face matrices correspond to their row and column position in the global face stiffness matrix $\boldsymbol{K_F}$.

In Figure 1 `glob_2` stores all components of $\boldsymbol{G_2}$ from (31). Equivalent storage variables exist for the remaining subscripts see Table 3. To store `glob_2` into a

Partial stiffness matrices	Face term	row	column
$\sum_{F\in\mathcal{F}_I} \int_F \boldsymbol{M_{C1}} + \boldsymbol{M_{D1}} + \boldsymbol{M_{E1}} \rightarrow$	glob_1	pos_i	pos_j
$\sum_{F\in\mathcal{F}_I} \int_F \boldsymbol{M_{C2}} + \boldsymbol{M_{D2}} + \boldsymbol{M_{E2}} \rightarrow$	glob_2	pos_i	neg_j
$\sum_{F\in\mathcal{F}_I} \int_F \boldsymbol{M_{C3}} + \boldsymbol{M_{D3}} + \boldsymbol{M_{E3}} \rightarrow$	glob_3	neg_i	pos_j
$\sum_{F\in\mathcal{F}_I} \int_F \boldsymbol{M_{C4}} + \boldsymbol{M_{D4}} + \boldsymbol{M_{E4}} \rightarrow$	glob_4	neg_i	neg_j

Table 3: Storage variables and their associated row and column degree of freedom numbers. The row and column degrees of freedom, `i` and `j`, have a prefix `pos` and `neg`. `pos` and `neg` correspond to pre-or-post multiplication of $\boldsymbol{N}^+$, or $\boldsymbol{B}^+$, and $\boldsymbol{N}^-$, or $\boldsymbol{B}^-$.

global stiffness matrix it is first rearranged into a vector form with the MATLAB function `reshape`, line 4 of Figure 5. The new data structure of `glob_2` is described in Table 4. When steering `glob_1`, `glob_2`, `glob_3` or `glob_4` into global

pos_i	neg_j	glob_2_rs
1	1	glob_2(1,1,1)
1	2	glob_2(1,1,2)
⋮	⋮	⋮
1	ndof	glob_2(1,1,ndof)
⋮	⋮	⋮
ndof	ndof	glob_2(1,ndof,ndof)
ndof+1	ndof+1	glob_2(2,1,1)
ndof+1	ndof+2	glob_2(2,1,2)
⋮	⋮	⋮
tndof	tndof	glob_2(nf,ndof,ndof)

Table 4: Reshaping of `glob_2` into a vector form `glob_2_rs` for MATLAB function `sparse`.

face matrix $\boldsymbol{K_F}$ (line 13) the row numbers correspond to the finite elements' degrees of freedom, that pre-multiplied the partial stiffness matrices, of `glob_1`, `glob_2`, `glob_3` or `glob_4`. The column numbers represent the degrees of freedom

Element type	# area Gauss points	# face Gauss points
Constant strain triangle	1	2
Bi-linear quadrilateral	4	2
Bi-quadratic quadrilateral	9	3

Table 5: The number of Gauss points required for the area and face integral for different element types.

of finite elements that post-multiplied.

After all the stiffness matrices are stored into the global sparse matrix, the sparse matrix is transposed and summated (line 14), completing the global stiffness matrix.

4. Blocking and numerical analysis

This section demonstrates the efficiency gain obtained when using vectorised blocked scripts to generate all the SIPG local face stiffness matrices (11). All computations were performed in a native MATLAB environment using double precision float accuracy, the backward slash operator '\' is used to solve any linear system of equations. The .m file was run from a terminal using MATLAB rather than from the MATLAB GUI. All meshes were structured and homogeneous in element type, they were constructed from either, four noded bi-linear quadrilateral elements with linear basis functions in each direction, eight noded bi-quadratic quadrilateral elements with quadratic basis functions in each direction, or three noded constant strain triangular elements. For all elements the degrees of freedom existed on the nodes. The number of Gauss points for the area and face integral is displayed in Table 5.

Timing experiments on computers are susceptible to a lack of precision and accuracy, this is caused by both the computer performing background tasks and components fluctuating in temperature. When testing a range of SIPG face block sizes, the order of the block sizes was randomised and tested, this process was repeated.

Component	Computer 1	Computer 2
Family	AMD A10-series	Haswell
Frequency	3.8 GHz AMD A10-5800K	3.60 GHz Intel Core i7-4790
No. cores	4 (no multithread)	8 (no multithread)
L2 cache	2×2Mb	4×256 Kb
RAM	8 GB	16 GB
OS	Ubuntu 14.04.1	Ubuntu 15.04
MATLAB vers.	R2014a	R2014b

Table 6: Computer specifications for blocking experiments.

All blocking experiments were performed on both computers specified in Table 6. Numerical analysis verification and speed tests, Section 4.4, were performed only using Computer 1.

4.1. Variables of vectorised multiplication

MATLAB incorporates LAPACK, which calls BLAS, to perform its mathematical computations, it is a library of numerical linear algebra routines written in Fortran [21]. Arrays in Fortran are stored in column-major order form, this section investigates the importance of the orientation of variables in MATLAB when using performing large vector calculations.

A script was written to investigate the speed differences when performing vector calculations in different array orientations in MATLAB, Figure 6. Column-major operations occurred on line 7, and row-major operations on line 19. The `for` loops on lines 1 and 13, loop though a logarithmically distributed range of vector sizes from $10 \rightarrow 10^6$. The loops for `i` and `j` represent the nodal loops in the face integration Algorithm represented in Figure 1. The results are shown in Figure 7. Element-wise multiplication of arrays in column-major form are consistently and significantly faster than arrays in row-major form. The memory addresses of variables in the same column vary less than along the same row. Therefore the find and read time for a variable along a column is

faster. All calculations, if possible, were therefore made to occur in this format.

4.2. Optimum block size

There is an optimum size of vector for an element-wise vector calculation which achieves the most floating point operations per second (flops). Managing element-wise vector operations of lengths larger than the optimum size into smaller sizes, of optimum length, is a technique known as blocking. The vectorised SIPG code described in this paper is designed to blocked. If the blocking algorithm for the SIPG code is effective a peak in performance corresponding to the optimum vector calculation length is expected. This section investigates whether the code has an optimum vector length, what the length is, and the number of flops achieved for this length.

To determine the performance of the optimised SIPG code, the time to calculate the linear elastic SIPG stiffness matrix $\boldsymbol{K}$ was tested for different block sizes. The block size was logarithmically distributed, $10 \rightarrow 10^6$, and $\boldsymbol{K}$ consisted of 10^6 degrees of freedom. The performance in terms of Mflops is presented in Figure 8 for lines 1-5 and 6-10 of Algorithm 1, the area and face integrals. The test was preformed on a 2D domain, Ω, where $x, y \in [0, 1]$. The mesh distribution within Ω is structured with each element having the same area. The Young's Modulus was set to 10 Pa and Poisson's ratio had a value of 0.2. The tested computer architectures, OS, and MATLAB version used, is shown in Table 6.

Computer 1 and 2 have a respective theoretical peak performance of 3.04×10^9 and 5.7×10^9 double precision floating point operations per second (flops). These peaks correlate to the fastest computation times achieved, shown in Table 7, and thus as their time is smallest are the optimum block sizes to perform the 2D SIPG area and face integrals.

Figure 8 shows computer 1's fastest performance to generate the area and face integral was ≈ 500 Mflops for a block size of $\approx 3 \times 10^4$ achieving a $\approx 16\%$ to-

	Fastest area integral (s)	Fastest face integral (s)
Computer 1	0.25	1.04
Computer 2	0.11	0.46
Computer 2 (Octave)	0.23	1.75

Table 7: Fastest computation times of the area and face integral for computer 1 and 2, corresponding to the peak values in Figure 8.

tal efficiency of the theoretical peak performance. Whereas computer 2 achieved a higher ≈ 1000 Mflops for both the area and face integral corresponding to an efficiency of 17.5%. As an optimum block size was achieved for the both the area and face integral Figure 8 demonstrates a correct implementation of a vectorised blocked algorithm to compute the SIPG global stiffness matrix with comparison to [3]. The algorithm worked correctly on both computer architectures.

In comparison to an unoptimised code. Computer 1 took respectively 10.2 s and 21.2 s to compute the area and face integrals, whereas Computer 2 took 5.9 s and 17.91 s. Comparing the speed of the optimised code in Table 7 to the unoptimised code, computer 1 achieved a speed increase of 51 times for the area integral and 20 times for the face integral with a total speed increase of 24 times. The total speed increase from pure vectorisation is 13.7 times with blocking being 1.8 times faster than pure vectorisation. Computer 2 achieved a speed increase of 54 times for the area integral and 39 times for the face integral with a total speed increase of 41 times. The total speed increase from pure vectorisation is 23 times blocking being 1.8 times faster than pure vectorisation.

It is noted that for both computer architectures in Figure 8 that the block Mflop performance is still decreasing when the block size exceeds that of the number of area integrals and face integrals. This suggest that for larger problems the performance gain from block is going to be more substantial.

At the peak performance the cache reuse is maximised. After the peak the proportion of data lying outside the lowest level of CPU becomes larger and so the performance decreases. The performance is still decreasing as the block size

exceeds the number of area and face integrals, marked by the vertical lines on Figure 8. This indicates that for larger problems the advantage of blocking over purely vectorised code is going to become larger but also indicates the cache reuse is being maximised.

Computer 1's cache is larger than computer 2, as larger variables can reside in the lowest level cache the optimum block size for peak performance is therefore also larger. This can also be seen in Figure 7.

A speed run on Computer 2 using Octave version 3.8.2 was also performed to verify that the code was effective in both MATLAB and Octave. Figure 8 demonstrated that a peak in performance was achieved for both the area and face integral. Similar to the tests performed in MATLAB, once the peak was reached the performance continued to decrease and only stabilised once the block size exceed the number of elements and faces. The optimal performance, in comparison to MATLAB, was also slower with the area and face integrals corresponding to a loss in performance of $\approx$ 2.1 and $\approx$ 3.82 times. Despite being slower, the speed up for the vectorised blocked when compared to an unoptimised code for the area and face integral was $\approx$ 113 and $\approx$ 41 times, much greater than that achieved with MATLAB.

The `sparse` formulation time was $\approx$ 15s for computer 1 and $\approx$ 9 s for computer 2. A small investigation into whether blocking arrays whilst using native MATLAB function `sparse` had any influence on the storage time, but it was found that only with computers with limited ram, (4Gb), yielded any performance improvement. As highlight in [3, 10] using non-native MATLAB variations of the `sparse` command can significantly increase performance.

4.3. Algorithm validation

To validate the correct implementation of FEA code for a linear set of equations, an eigenvalue convergence test can be performed. The test involves a 2D domain, Ω, where $x, y \in [0, 1]$. The mesh distribution within Ω is structured with each element having the same area. Homogeneous Dirichlet boundary conditions are applied on $\partial\Omega \equiv \partial\Omega_D$. The stiffness material matrix (28) is unusually

defined as $A = 1$, and, B and $C = 0$. This uncouples the degrees of freedom so that the stiffness matrix contains two 2D Poisson problems added together with a doubled mass coefficient. The smallest eigenvalue of the problem is π^2. Last the SIPG penalty term is set as $\beta = 10$, see (2).

An undamped dynamic system of equations for linear elasticity modelled using SIPG is

$$(\boldsymbol{K} - \lambda_1^2 \boldsymbol{A})\boldsymbol{U}_n = 0, \tag{36}$$

where $\boldsymbol{K}$ is the global stiffness matrix, $\boldsymbol{A}$ is the mass matrix, [15], and λ_1^2 is the first natural frequency squared. The computeted convergence rates were close to the analytical convergence rates for all elements, as shown in Figure 9, [14].

4.4. Hole in plate verification

To demonstrate that the optimised 2D DG algorithm converges to correct solutions for linearly elastic problems, as well as to demonstate the perfomance gains using an optimised SIPG code, a plane stress analysis of an infinite plate with a hole at its centre subjected to a uniaxial tensile stress is now considered, [22]. Here the performance of an optimised area integral as present by [3] is also analysed in conjunction with the optimised SIPG face integral presented here.

The solution to the infinite problem is provided by [23]. The infinite problem is truncated at the boundary by using the stress solution as Neumann boundary conditions on $\partial\Omega$. The reduced problem setup is provided by Figure 10. The analytical stress solution is

$$\sigma_{xx} = \sigma_\infty \left[1 - \frac{a^2}{r^2}\left(\frac{3}{2}\cos(2\theta) + \cos(4\theta)\right) + \frac{3a^4}{2r^4}\cos(4\theta)\right], \tag{37}$$

$$\sigma_{yy} = \sigma_\infty \left[-\frac{a^2}{r^2}\left(\frac{1}{2}\cos(2\theta) - \cos(4\theta)\right) - \frac{3a^4}{2r^4}\cos(4\theta)\right], \tag{38}$$

and,

$$\sigma_{xy} = \sigma_\infty \left[-\frac{a^2}{r^2}\left(\frac{1}{2}\sin(2\theta) + \sin(4\theta)\right) + \frac{3a^4}{2r^4}\sin(4\theta)\right], \tag{39}$$

where θ and r are polar coordinates and a is the hole radius see Figure 10.

A schematic of the problem is shown in Figure 10, with sides of length $l = 10$ m and hole radius $a = 1$ m. The material is modelled in plane stress with a Young's modulus of 10^3 Pa, Poisson's ratio of 0.2, with an applied far field stress of $\sigma_\infty = 10^2$.

For this problem all three element types are used: The constant strain triangle, the bi-linear linear quadrilateral, and the bi-quadratic quadrilateral. An example of their respective meshes is shown in Figures 11a and 11b. For the constant strain triangle element and bi-linear quadrilateral element the meshes have the same number of nodes along the radius and the circumference. The bi-quadratic quadrilateral element has the same number of element vertex nodes along the circumference and radius. Along the circumference the nodes are equally spaced in terms of θ. Along the radius, r, a scaling factor, sf, is applied to prevent distorted elements. The $||\boldsymbol{u} - \boldsymbol{u}_h||_{L^2(\Omega)}$ error between the analytical solution for the displacement $\boldsymbol{u}$, [23], and the computed displacement $\boldsymbol{u}_h$ is calculated from

$$||\boldsymbol{u} - \boldsymbol{u}_h||_{L^2(\Omega)} = \sqrt{\sum_{K \in \Omega} \int_K |\boldsymbol{u} - \boldsymbol{u}_h|^2}. \tag{40}$$

This error is used in Figures 12a and 12b to validate the convergence rates for different element types, [14], as well as to compare performance gains between the optimised and non-optimised codes.

Convergences rates of 2.1, 2.0 and 4.1 were achieved for the constant strain triangle, and for the linear and quadratic quadrilaterals using the optimised SIPG code which are very similar to their analytical counterparts 2, 2 and 4, [14]. This demonstrates correct implementation of the optimised code for multiple elements types for a linear elastic problem.

For the speed investigation computer 1 was used, the block size of 3×10^4 was used for all computations. For all elements the performance gain of the optimised code against the non-optimised code improves with problem size, Figure 12a. The initial poor performance improvement was because at a low element number the number of BLAS calls was similar for both codes. The highest performance gain of ≈ 90 was achieved by the order 1 triangle elements,

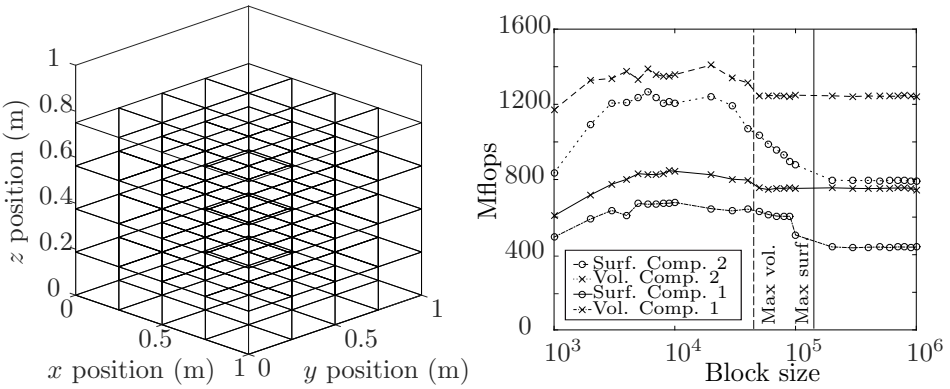
the lowest performance gain of ≈ 9 was achieved by the order bi-quadratic quadrilateral elements. For both quadrilateral elements the rate of performance gain with problem decreases. For Triangular elements the rate of performance gain remains constant.

The ratio between the number of matrix calculation BLAS calls between the optimised and non optimised codes is similar to that of the performance gain for large problems. For the quadrilateral element, an error of $||\boldsymbol{u} - \boldsymbol{u}_h||_{L^2(\Omega)} \approx 4 \times 10^{-2}$ has a BLAS call ratio ≈ 10 and a performance gain of ≈ 10 for both the area and face integral. The same can be said for the linear quadrilateral when considering an error of $||\boldsymbol{u} - \boldsymbol{u}_h||_{L^2(\Omega)} \approx 4 \times 10^{-2}$. The SIPG algorithm has a performance gain for the area and face integral of ≈ 36 and ≈ 27 with a corresponding BLAS call ratio of ≈ 36 and ≈ 27.

However for the triangular element this correlation breaks down. For a $||\boldsymbol{u} - \boldsymbol{u}_h||_{L^2(\Omega)} \approx 3 \times 10^{-2}$ the BLAS call ratios for the area and surface are 24.3, and, 12 respectively. This is far below the performance gain in Figure 12a. This is likely due to the optimised triangle BLAS call number of approximately 500 where as for the quadrilateral optimised, and all unoptimised, codes BLAS calls exceed 5000. There is no correlation between the computational time and BLAS call number, this would indicate that the speed of the optimised for the triangle codes is longer dictated by the BLAS overhead, whereas the optimised quadrilateral codes are.

4.5. 3D verification

Here a unit sided cube exist in a reference 3D Cartesian coordinate system where $[x, y, z] \in \mathbb{R}^3$. The cube is modelled using the SIPG method described in Section 2.1. Roller boundary conditions exist on all faces except where $z = 1$; here a displacement of $dw = -0.25$ m is applied in the z direction. The material has a Young's modulus of 1 Pa and Poisson's ratio of 0.2.



(a) (b)

Figure 13: (a) Compression of a unit cube with roller boundary conditions. (b) Mflops performance of lines 1-5 and 6-10 of Algorithm 1 for different block sizes for a 3D SIPG problem with $\approx 10^6$ degrees of freedom. Optimum performance times of 4.1 s and 29.6 s were achieved for the volume and surface integral respectively.

The displacements u, v, w correspond to the directions of x, y, z. The analytical solution to the constant stress problem is $u, v = 0$ and $w = z \times dw$. Any mesh discretisation would achieve the correct solution to machine precision, here a homogeneous $5 \times 5 \times 5$ mesh of tri-linear hexahedral elements are used, the result is shown in Figure 13a. The problem run is of a unit cube consisting of a homogeneous distribution of linear hexahedral elements. The volume and surface integrals require 8 and 4 gauss points respectively.

As shown in Figure 13b, it is also possible to block vectorised 3D SIPG code for both volume and surface integrals. For the both computers there is a peak in performance at a block size of $\approx 10^4$. The peak corresponds to the cache reuse being maximised. After the peak, the drop in performance corresponding to not all data, required for a BLAS operation, residing in the cache. Similar to Section 4.2 the performance is still still decreasing once the block size becomes larger than the number of volume and surface integrals, highlighting the importance of blocking for larger problems.

The fastest runtime achieved for the volume and surface integral by computer 1 was respectively 4.1 s and 9.2, computer 2 achieved a run time of 1.4 s and

3.9 s. The total runtime to generate the global stiffness matrix was 29.3 s and 20.2 s for computer 1 and 2. Profiling reviled the MATLAB function `sparse`, necessary to generate the global stiffness matrix, took the majority of the time 18 s and 13 s for computer 1 and 2.

5. Conclusion

This paper for the first time has presented an efficient blocked vectorised algorithm for producing SIPG face stiffness terms in a native MATLAB environment for linear elasticity for a range of elements in both 2D and 3D. Optimisation was achieved by: (i) maximising the CPU cache reuse by changing the vector size for the BLAS operations; (ii) storing vectors in a column-major form; (iii) ensuring all matrix calculations were as large as possible and (iv) reducing the number of calculations by only considering symmetric terms.

The block length optimisation results demonstrate a clear optimal block length, which is consistent between all integral types and problem types. The peaks coincide with a maximisation of the cache reuse. Additionally a number of different verification techniques have been used to demonstrate correct implementation of both linear systems in both 2D and 3D.

The optimal block length for the hardware used in the study was found to be at $\approx 3 \times 10^4$ corresponding to a total CPU usage of $\approx 16\%$, similar to results found in literature. All codes were able to compute the global stiffness matrix for a 10^6 degrees of freedom system in under 30 s.

In the linear elastic 2D performance gain study, in Section 4.4, it was shown that the gain continues to increase with problem size. It was also shown that the performance gains were dependent on element type, with triangular elements achieving gains excess of 50 times. There was also a correlation between gains and the ratio of BLAS calls for the quadrilateral code. This suggests that optimised quadrilateral code still is still subject to a bottle neck from the BLAS call overhead. This was not the case for the triangular code which had significantly fewer BLAS calls.

The script could be optimised further by using the MATLAB's parallel function `parfor` and incorporating GPUs into the calculation. The final scripts are designed to be a black box, taking in element topology and outputting the global stiffness matrix for a SIPG problem.

Appendix A. List of variables

Name	Dimensions	Description
`Ai`	1	The degree of freedom row positioning of entry (1,1) of partial stiffness matrices $\boldsymbol{M^r_{C1}}$, $\boldsymbol{M^r_{C2}}$, $\boldsymbol{M^r_{C3}}$, $\boldsymbol{M^r_{C4}}$, $\boldsymbol{M^r_{E1}}$, $\boldsymbol{M^r_{E2}}$, $\boldsymbol{M^r_{E3}}$, and $\boldsymbol{M^r_{C4}}$, for current `i`.
`Aj`	1	The degree of freedom column positioning of entry (1,1) of partial stiffness matrices $\boldsymbol{M^r_{C1}}$, $\boldsymbol{M^r_{C2}}$, $\boldsymbol{M^r_{C3}}$, $\boldsymbol{M^r_{C4}}$, $\boldsymbol{M^r_{E1}}$, $\boldsymbol{M^r_{E2}}$, $\boldsymbol{M^r_{E3}}$, and $\boldsymbol{M^r_{C4}}$, for current `j`.
`Bi`	1	The degree of freedom row positioning of entry (1,1) of partial stiffness matrices $\boldsymbol{M^r_{D1}}$, $\boldsymbol{M^r_{D2}}$, $\boldsymbol{M^r_{D3}}$, and $\boldsymbol{M^r_{D4}}$, for current `i2`.
`Bj`	1	The degree of freedom column positioning of entry (1,1) of partial stiffness matrices $\boldsymbol{M^r_{D1}}$, $\boldsymbol{M^r_{D2}}$, $\boldsymbol{M^r_{D3}}$, and $\boldsymbol{M^r_{D4}}$, for current `j2`.
`bl_index`	[1,`nel_block`]	An index for selecting rows of `etpl_face` and `glob_2` which are in the current SIPG face block loop.
`Block_n`	1	Current SIPG face block number.
`C2t_11`	[`nel_block`,1]	Vector of entry (1,1) of $\boldsymbol{M^r_{C2}}$, which vary with `i`, for all faces in the current SIPG face block loop.
`coord`	`Nnodes`,2	Coordinates of all nodes in the mesh.

`coord_xp`	[sum(`index_p`),1]	Nodal x-coordinates of all elements in the current SIPG face block with local face number `fn`. Their order is dictated by `etpl_face(:,1)`.
`coord_yp`	[sum(`index_p`),1]	Nodal y-coordinates of all elements in the current SIPG face block with local face number `fn`. Their order is dictated by `etpl_face(:,1)`.
`D2t_12`	[`nel_block`,1]	Vector of entry (1,2) of $\boldsymbol{M}^{r}_{\boldsymbol{D2}}$, which vary with `j2`, for all faces in the current SIPG face block loop.
`det`	[sum(`index_p`),1]	Jacobian determinant for all K^+ elements in the current SIPG face block loop with local face number `fn`.
`dNr`	[`nf`*`ngp`*2,`nen`]	Reference shape function derivatives for all Gauss points for all local element faces.
`dNx_p`	[`nel_block`,`ndof`]	Global shape function derivatives, with respect to x, for all F^+ faces in the current loop.
`dNy_p`	[`nel_block`,`ndof`]	Global shape function derivatives, with respect to y, for all F^+ faces in the current loop.
`ed`	[`nels`,`ndof`]	Steering matrix matrix $\forall K \in \mathcal{T}$. Row number corresponds to element, column number to the global degree of freedom.
`ed_p`	[`tot_f`,`ndof`]	Steering matrix of local stiffness matrices to global, for degrees of freedom of K^+ elements with order `etpl_face(:,1)`.
`ed_n`	[`tot_f`,`ndof`]	Steering matrix of local stiffness matrices to global, for degrees of freedom of K^- elements with order `etpl_face(:,2)` into global stiffness matrix.
`etpl`	[`nels`,`nen`]	Element topology matrix of all elements in the mesh.
`etpl_face`	[`tot_f`,7]	Description of the SIPG faces in the mesh.

`etpl_face_block`	[sum(`index_p`),7]	Columns of `etpl_face` for SIPG faces in the current block number.
`etpl_face_block_fn`	[`nel_block`,7]	Columns of `etpl_face_block` for SIPG faces with local positive face number K^+.
`fn`	1	Current face number
`glob_pn`	[`bl_num_f`,`ndof`,`ndof`]	Temporary storage variables for all SIPG faces matrices pre-and-post multiplied by a $^+$ and $^-$ element respectively.
`glob_2`	[`tot_f`,`ndof`,`ndof`]	Storage variable for $\boldsymbol{G}_2$.
`glob_2_rs`	[`tot_f*nndof`,1]	`glob_2_rs` reshaped into a vector form
`gp`	1	Current Gauss point number in the loop
`i`	1	Row number for partial stiffness matrices: $\boldsymbol{M}^r_{\boldsymbol{C1}}$, $\boldsymbol{M}^r_{\boldsymbol{C2}}$, $\boldsymbol{M}^r_{\boldsymbol{C3}}$, $\boldsymbol{M}^r_{\boldsymbol{C4}}$, $\boldsymbol{M}^r_{\boldsymbol{E1}}$, $\boldsymbol{M}^r_{\boldsymbol{E2}}$, $\boldsymbol{M}^r_{\boldsymbol{E3}}$, and $\boldsymbol{M}^r_{\boldsymbol{C4}}$.
`i2`	1	Row number for partial stiffness matrices: $\boldsymbol{M}^r_{\boldsymbol{D1}}$, $\boldsymbol{M}^r_{\boldsymbol{D2}}$, $\boldsymbol{M}^r_{\boldsymbol{D3}}$, and $\boldsymbol{M}^r_{\boldsymbol{D4}}$.
`index_p`	`nel_block`,1	Logical variable to select columns of `etpl_face_block` with the face number `fn`. 1 indicates same `fn`, 0 otherwise.
`indx_dNr`	[1,2]	Index to select rows of `dNr` for a specific `fn` and `gp`.
`indx_Nr`	1	Index to select row of `Nr` for a specific `fn` and `gp`.
`int_W`	[`nel_block`,1]	Gauss face integral weight and Jacobian determinant for all SIPG faces in the current block loop.
`invJxp`	[sum(`index_p`),2]	Row 1 of inverse Jacobian matrix for all K^+ elements in the current SIPG face block loop with local face number `fn`.
`invJyp`	[sum(`index_p`),2]	Row 2 of inverse Jacobian matrix for all K^+ elements in the current SIPG face block loop with local face number `fn`.

j	1	Column number for partial stiffness matrices: $\boldsymbol{M^r_{C1}}$, $\boldsymbol{M^r_{C2}}$, $\boldsymbol{M^r_{C3}}$, $\boldsymbol{M^r_{C4}}$, $\boldsymbol{M^r_{E1}}$, $\boldsymbol{M^r_{E2}}$, $\boldsymbol{M^r_{E3}}$, and $\boldsymbol{M^r_{C4}}$.
j2	1	Column number for partial stiffness matrices: $\boldsymbol{M^r_{D1}}$, $\boldsymbol{M^r_{D2}}$, $\boldsymbol{M^r_{D3}}$, and $\boldsymbol{M^r_{D4}}$.
jxp	[2,sum(index_p)]	Column 1 of Jacobian matrix for all K^+ elements in the current SIPG face block loop with local face number fn.
jyp	[2,sum(index_p)]	Column 2 of Jacobian matrix for all K^+ elements in the current SIPG face block loop with local face number fn.
K	[max(ed(:)),max(ed(:))]	Global stiffness matrix.
ndof	1	Total number of degrees of freedom for one element.
nen	1	Number of nodes for one element.
ngp	1	Number of face Gauss points.
neg_i	tot_f*nndof	Row degrees of freedom for all local stiffness matrices pre-multiplied by a K^- element, corresponding to global storage vectors glob_3_rs and glob_3_rs.
neg_j	tot_f*nndof	Column degrees of freedom for all local stiffness matrices post-multiplied by a K^- element, corresponding to global storage vectors glob_2_rs and glob_3_rs.
nndof	1	Total number of entries in local element matrix (ndof*ndof).
Nr	[nf*ngp,el_nodes]	Local face shape functions.
Np	[nel_block,1]	Shape functions for all K^+ element faces in the current block.
num_blocks	1	Number of SIPG face blocks.

num_faces	1	Number of SIPG faces in a block.
nx	[nel_block,1]	Normal x component to interior faces in block.
ny	[nel_block,1]	Normal y component to interior faces in block.
pen	1	SIPG penalty values for linear elasticity.
pos_el	[sum(index_p),1]	List K^+ elements in SIPG face block loop with face number fn.
pos_i	tot_f*nndof	Row degrees of freedom for all local stiffness matrices pre-multiplied by a K^+ element, corresponding to global storage vectors glob_1_rs and glob_2_rs.
pos_j	tot_f*nndof	Column degrees of freedom for all local stiffness matrices post-multiplied by a K^+ element, corresponding to global storage vectors glob_1_rs and glob_3_rs.
nel_block	1	Total number of faces in the block.
nnodes	1	Total number of nodes in the mesh.
nels	1	Total number of elements in the mesh.
tot_f	1	Total number of interior faces.
tndof	1	Total number of degrees of freedom in mesh.

[1] W. Coombs, M, R. Crouch, S, C. Augarde, E, 70-line 3D finite deformation elastoplastic finite-element code, Proc. Numerical Methods in Geotechnical Engineering (NUMGE), Trondheim, Norway (2010) 151–156.

[2] O. Sigmund, A 99 line topology optimization code written in MATLAB, Structural and Multidisciplinary Optimization 21 (2001) 120–127.

[3] M. Dabrowski, M. Krotkiewski, D. Schmid, MILAMIN: MATLAB-based finite element method solver for large problems, Geochemistry, Geophysics, Geosystems 9 (2008).

[4] M. Krotkiewski, M. Dabrowski, Parallel symmetric sparse matrix–vector product on scalar multi-core cpus, Parallel Computing 36 (2010) 181–198.

[5] T. Rahman, J. Valdman, Fast MATLAB assembly of FEM stiffness-and mass matrices in 2d and 3d: nodal elements, Applied Mathematics and Computation 219 (2013) 7151–7158.

[6] I. Anjam, J. Valdman, Fast MATLAB assembly of FEM matrices in 2D and 3D: Edge elements, Applied Mathematics and Computation 267 (2015) 252–263.

[7] E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, O. Sigmund, Efficient topology optimization in MATLAB using 88 lines of code, Structural and Multidisciplinary Optimization 43 (2011) 1–16.

[8] F. Cuvelier, C. Japhet, G. Scarella, An efficient way to perform the assembly of finite element matrices in MATLAB and octave (2013).

[9] F. Cuvelier, C. Japhet, G. Scarella, An efficient way to assemble finite element matrices in vector languages, BIT Numerical Mathematics (2015) 1–32.

[10] S. Engblom, D. Lukarski, Fast MATLAB compatible sparse assembly on multicore computers, Parallel Computing 56 (2014) 1–17.

[11] W. H. Reed, T. Hill, Triangular mesh methods for the neutron transport equation, Los Alamos Report LA-UR-73-479 (1973).

[12] G. R. Richter, The discontinuous Galerkin method with diffusion, Mathematics of computation 58 (1992) 631–643.

[13] F. Bassi, S. Rebay, A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier–stokes equations, Journal of computational physics 131 (1997) 267–279.

[14] S. Giani, E. J. Hall, An a posteriori error estimator for hp-adaptive discontinuous Galerkin methods for elliptic eigenvalue problems, Mathematical Models and Methods in Applied Sciences 22 (2012) 1250030.

[15] N. S. Ottosen, H. Petersson, Introduction to the finite element method, Prentice Hall Internationa,, 1992.

[16] D. Arnold, F. Brezzi, B. Cockburn, L. Marini, Unified analysis of discontinuous Galerkin methods for elliptic problems, SIAM Journal on Numerical Analysis 39 (2002) 1749–1779.

[17] F. Frank, B. Reuter, V. Aizinger, P. Knabner, Festung: A MATLAB/GNU octave toolbox for the discontinuous Galerkin method, part i: Diffusion operator, Computers & Mathematics with Applications 70 (2015) 11–46.

[18] R. Bird, W. Coombs, S. Giani, Vectorised SIPG matrix formulation for MATLAB and Octave, https://github.com/robertbirddurham/SIPG_MATLAB_OPTIMISED.git, Accessed: 2017-03-28.

[19] P. Hansbo, M. G. Larson, Energy norm a posteriori error estimates for discontinuous Galerkin approximations of the linear elasticity problem, Computer Methods in Applied Mechanics and Engineering 200 (2011) 3026–3030.

[20] T. Weinzierl, A framework for parallel PDE solvers on multiscale adaptive Cartesian grids, Verlag Dr. Hut, 2009.

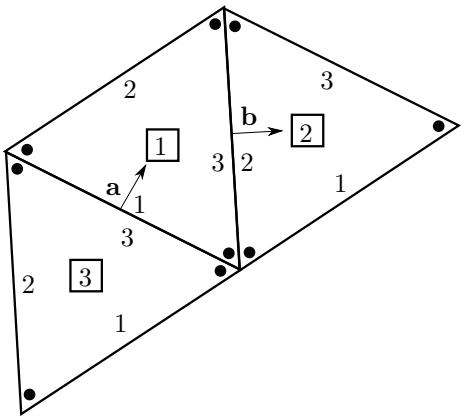
[21] C. Moler, MATLAB Incorporates LAPACK, http://uk.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html, Accessed: 2015-09-25.

[22] Z. Stowell, Elbridge, Stress and strain concentration at a circular hole an infinite plate (1950).

[23] G. Kirsch, Die theorie der elastizität und die bedürfnisse der festigkeitslehre, Springer, 1898.

```
glob_2=zeros(tot_num_faces,ndof,ndof);
for Block_n = 1:num_blocks % MATLAB code: lines 93-310.
    for gp = 1:ngp % MATLAB code: lines 125-299.
        glob_pn=zeros(bl_num_f,ndof,ndof);
        % Variable generation for current block Figure 3.
        for i = 1:nen % MATLAB code: lines 175-297.
            j2=(nen+1)-i;
            Ai=(i-1)*2; Bj=(j2-1)*2;

            C2t_11=((A.*dNx_p(:,i).*nx)+(C.*dNy_p(:,i).*ny)).*int_W;
            % Components of the remaining entries which vary in 'i' of:
            % M_C1, M_C2, M_C3, and M_C4
            % are also computed here.

            D2t_12=((B.*dNy_p(:,j2).*nx)+(C.* dNx_p(:,j2).*ny)).*int_w;
            % Components of the remaining entries which vary in 'j2' of:
            % M_D1, M_D2, M_D3, and M_D4
            % are also computed here.

            for j = i:nen
                i2=(nen+1)-j;
                Aj=(j-1)*2; Bi=(i2-1)*2;

                glob_pn(:,Ai,Aj)=Nn(:,j).*C2t_11.*half(i,j)+glob_pn(:,Ai,Aj);
                % Computations of all components of:
                % glob_pp, glob_pn, glob_np and glob_nn occur here.

                if i<j
                    glob_pn(:,Bi,Bj+1)=Np(i2).*D2t_12+glob_pn(:,Bi,Bj+1);
                end
            end
        end
    end
    glob_2(bl_index,:,:)=glob_pn;
end
```

Figure 1: Vectorised calculation schemetic of entry (1,1) in $\boldsymbol{M_{C2}}$ and (1,2) in $\boldsymbol{M_{D2}}$.



(a)

SIPG Face	**a**	**b**
K^+	3	3
K^-	1	2
F^+	3	3
F^-	1	2
n_x	$1/\sqrt{2}$	1
n_y	$1/\sqrt{2}$	0
h	$\sqrt{2}$	1

(b)

Figure 2: (a) Example of 3 elements in a 2D DG mesh. Arrows indicate the outward normal direction, values in a box the element number and values on the element edge the local face number. (b) The transpose of the matrix `etpl_face` for DG faces **a** and **b** on Figure 2a. n_x and n_y are the outward normal components, and h is the length of the face, (K^+) is the positive element number with face (f_e^+) and (K^-) is the negative element number with face (f_e^-).

```
1    for Block_n = 1:num_blocks % MATLAB code: lines 93-310.
2        etpl_face %defined in Figure 2b
3        etpl_face_block = etpl_face(:,bl_index)
4        for gp = 1:ngp % MATLAB code: lines 125-299.
5            for fn = 1:num_faces % MATLAB code: lines 127-170.
6                index_p = etpl_face_block(:,3)==fn;
7                etpl_face_block_fn = etpl_face_block(:,index_p);

                 K+ % Elements for current SIPG face block and local face number
8                pos_el = etpl_face_block_fn(:,1);

                 % Vector of x (coord_xp) and y (coord_yp)
                 % coordinates for pos_el
9                [coord_xp,coord_yp] = coord(etpl(pos_el,:),:);

                 % Jacobian calculation for pos_el (32)
10               Jxp = dNr(indx_dNr(fn,gp),:)*coord_xp(index_p,:);
11               Jyp = dNr(indx_dNr(fn,gp),:)*coord_yp(index_p,:);

                 % Vectorised determinant and calculation [3]
12               det = (Jxp(1,:).*Jyp(2,:))-(Jxp(2,:).*Jyp(1,:));

                 % Vectorised inverse Jacobian calculation
13               invJxp = [ det.*(Jyp(2,:))',-det.*(Jyp(1,:))'];
14               invJyp = [-det.*(Jxp(2,:))', det.*(Jxp(1,:))'];

                 % Global shape function derivative calculation (33)
15               dNx_p(index_p,:) = invJx_p*dNr(indx_dNr(fn,gp),:);
16               dNy_p(index_p,:) = invJy_p*dNr(indx_dNr(fn,gp),:);

                 % Shape functions storage
17               Np(index_p,:) = repmat(Nr(indx_Nr(fn,gp),:)...
                                                    ,sum(index_p),1);
18               % Calc. for dNx_n, dNy_n and Nn. MATLAB code: lines 152-168
19           end
20       end

         % Vectorised integral weight calculation
21       int_W=2.*pen./etpl_face_block(:,end);
22   end
```

Figure 3: An algorithm for generating variables for multiple DG faces simulataneously. The element topogly matrix, `etpl`, is arranged so the rows correspond to an element number and the columns a node number. The coordinate matrix `coord` is arranged so that the rows correspond to a node number, the first column the x-coordinate and the second column the y-coordinate.

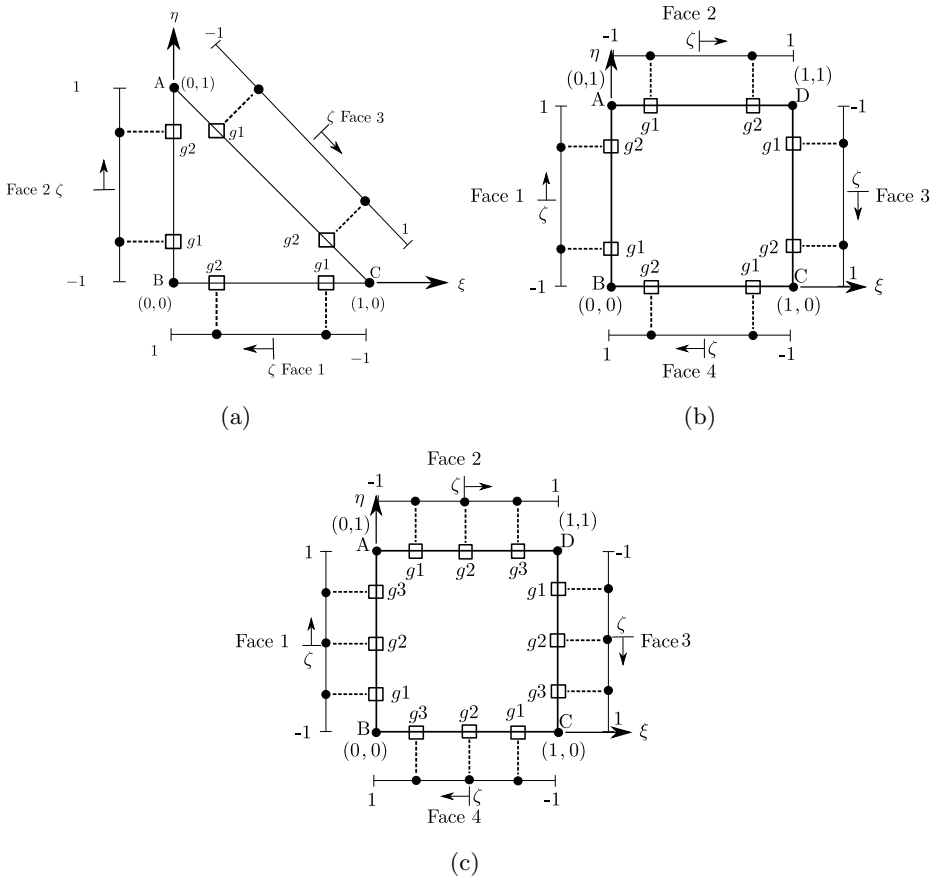


Figure 4: K^+ gauss point ordering and face ordering for both the constant strain triangular element (a) and bi-linear quadrilateral element (b), and bi-quadratic quadrilateral element (c). ξ and η are the coordinates in the reference element domain, ζ is the coordinate in the reference line domain and $g\#$ is the a gauss point number specific to a face number. Note for a face on a negative element the positions of $g1$ and $g2$ will be reversed.

```
1    tot_f=size(etpl_face,2); %total number of DG faces
2    ndof=(nen*2);        % number of degrees of freedom
3    nndof=(nen*2)^2;     % number of degrees of freedom (ndof) squared

% reshaping the global storage matrix into a vector, Table 4.
4    glob_2_rs = reshape(reshape(glob_2,tot_f,nndof)',tot_f*(nndof),1);

5    ed_p=ed(etpl_face(1,:),:); %steering matrix for + element dof
6    ed_n=ed(etpl_face(2,:),:); %steering matrix for - element dof

% steering vectors pos_i, pos_j, neg_i and neg_j, Table 4
7    pos_i = reshape(repmat(ed_p,1,ndof)',1,tot_f*nndof);
8    ed_pve = reshape(ed_p',1,ndof*tot_f);
9    pos_j = reshape(repmat(ed_pve,ndof,1),1,tot_f*nndof);
10   neg_i = reshape(repmat(ed_n,1,ndof)',1,tot_f*nndof);
11   ed_nve = reshape(ed_n',1,ndof*tot_f);
12   neg_j = reshape(repmat(ed_nve,ndof,1),1,tot_f*nndof);

% Global stiffness matrix sparse storage
13   k = k - sparse(pos_i,neg_j,glob_2_rs);
14   k=k+k'; % Completing the global stiffness matrix formulation
```

Figure 5: Segment of Matlab script for storing `glob_2` into a sparse matrix, where `ed` is a matrix describing the global degree of freedom numbering for each element. MATLAB code: lines 311-350

```
for s=ceil(logspace(1,6))
    a = rand(s,1);
    column_major = zeros(s,6,6);
    tic
    for i = 1:6
        for j = 1:6
            % Column major vector calculation
            column_major(:,i,j)=column_major(:,i,j)+a.*a;
        end
    end
    toc
    clear column_major a
end
for s=ceil(logspace(1,6))
    row_major = zeros(6,s,6);
    b = rand(1,s);
    tic
    for i = 1:6
        for j = 1:6
            % Row major vector calculation
            row_major(i,:,j)=row_major(i,:,j)+b.*b;
        end
    end
    toc
    clear row_major b
end
```

Figure 6: A MATLAB script to investigate how the orientation of vector entry-wise multiplications is affected by array orientation.

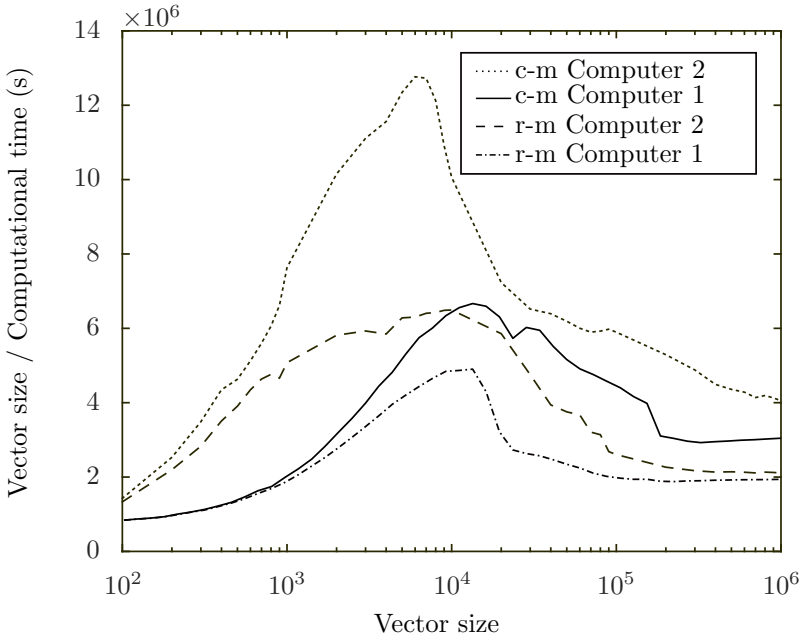


Figure 7: A computiational speed comparison or performing calculations with vectors stored in a row-major or column-major form. The r-m corresponds to calculations occuring in row orientated vectors and c-m corresponds to calculations operating in column orientated vector form.

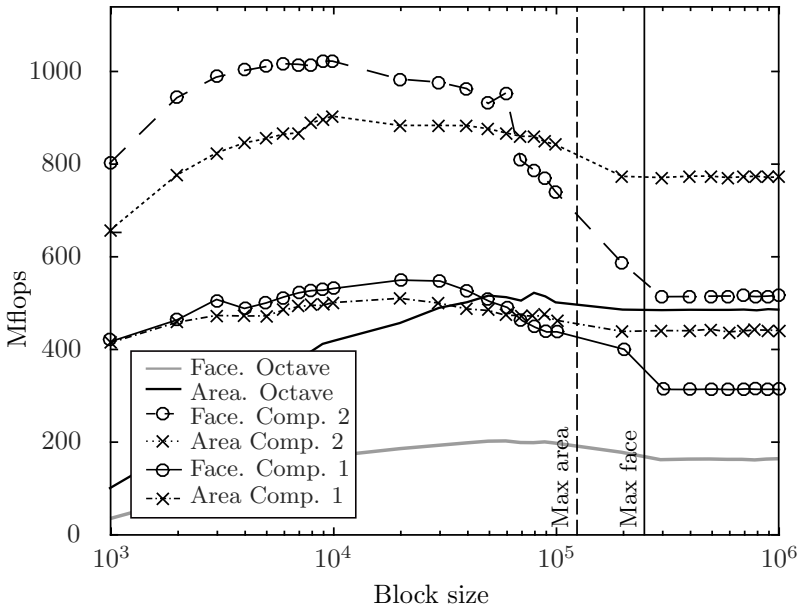


Figure 8: Flops performance of lines 1-5 and 6-10 of Algorithm 1 for different vector block sizes for a 2D SIPG problem with $\approx 10^6$ degrees of freedom using MATLAB. Additionally Mflops performance in Octave using Computer 2. also Times for peak Mflops performance shown in Table 7.

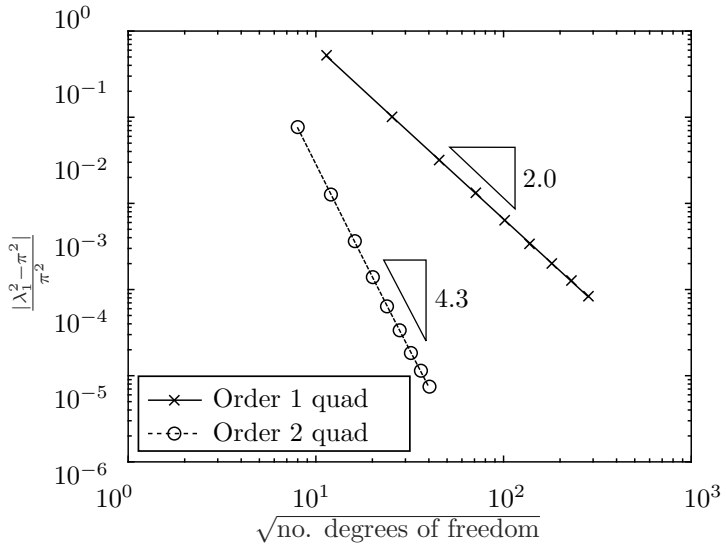


Figure 9: Convergence rates of 2.0 and 4.3 respectively achieved for shape functions with a polynomial order of 1 and 2 using the optimised code for a linear set of equations.

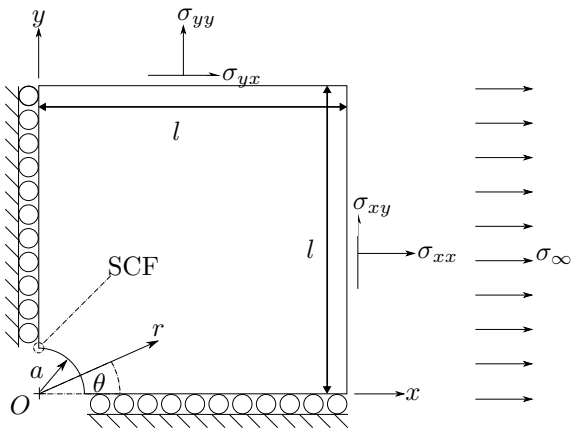


Figure 10: Schematic of computational experiment. Given that the problem is symmetric, roller boundary conditions exist at $x = 0$ and $y = 0$. a is the hole radius, σ_{xx}, σ_{yy} and σ_{xy} are the plane and shear stress, and, σ_∞ is the uniform far field stress of the infinite plate. r and θ are polar coordinates.

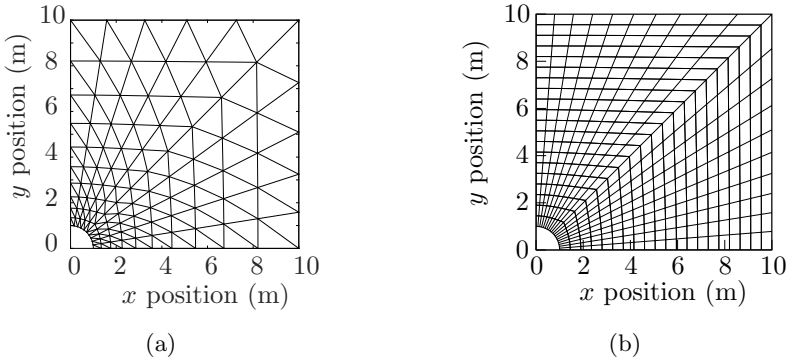


Figure 11: The element mesh distribution for the problem in Figure 10 with triangle elements (a) and quadrilateral elements (b).

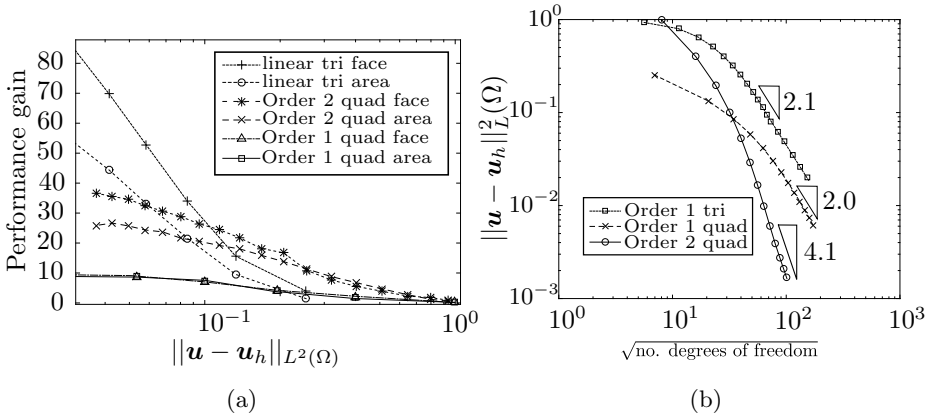


Figure 12: (a): A performance comparison between optimised and non-optimised scripts against error for the hole in an infinite plate problem. (b): L2 convergence rates for linear triangle, and, linear and quadratic quadrilateral.