Computers and Mathematics with Applications 78 (2019) 3007–3026

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

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

Rapid non-linear finite element analysis of continuous and discontinuous Galerkin methods in MATLAB

S. O'Sullivan, R.E. Bird, W.M. Coombs, S. Giani *

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

ARTICLE INFO

Article history:
Available online 12 April 2019

Keywords:
Elasto-plasticity
Finite element analysis
Discontinuous Galerkin
MATLAB code vectorisation

ABSTRACT

MATLAB is adept at the development of concise Finite Element (FE) routines, however it is commonly perceived to be too inefficient for high fidelity analysis. This paper aims to challenge this preconception by presenting two optimised FE codes for both continuous Galerkin (CG) and discontinuous Galerkin (DG) methods. Although this has previously been achieved for linear-elastic problems, no such optimised MATLAB script currently exists, which includes the effects of material non-linearity. To incorporate these elasto-plastic effects, the externally applied load is split into a discrete number of loadsteps. Equilibrium is determined at each loadstep between the externally applied load and the arising internal forces using the Newton–Raphson method. The optimisation of the scripts is primarily achieved using vectorised blocking algorithms, which minimise RAM-to-cache overheads and maximise cache reuse.

The optimised codes yielded maximum speed gains of $\times 25.7$ and $\times 10.1$ when compared to the corresponding unoptimised scripts, for CG and DG respectively. It was identified that with increasing refinement of the mesh, the solver time begins to dominate the overall simulation time. This bottleneck has a greater disadvantage on the DG code, predominantly due the asymmetric nature of the global stiffness matrix. The implementation of an efficient solver would see further improvement to the overall run times, particularly for large problems.

1. Introduction

Finite Element Analysis (FEA) has become a frequently used computational technique in modern industry. The method has had a substantial precedent within the engineering community since the initial concept was first established in the 1940s, originating in the field of structural mechanics. Its development has been greatly aided over the years by the exponential increase of computational power and memory storage, allowing high fidelity problems to be solved rapidly.

Finite element codes can be developed in a number of computing languages including C, C++ and Fortran, however these compiled codes can require thousands of lines of code to be written. Conversely, high-level computational environments, such as MATLAB, allow equivalent routines to be written in a fraction of the time and with far fewer lines. Nonetheless, MATLAB is often overlooked for finite element analysis (FEA) due to its computational inefficiencies, particularly when executing `for`-loops. To overcome this, a number of optimisation techniques can be implemented in order to reduce run-times. Without these optimisation routines, MATLAB simply cannot compete, in terms of computational speed, with alternative languages for FEA.

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

https://doi.org/10.1016/j.camwa.2019.03.012

MATLAB utilises the Linear Algebra PACKage (LAPACK), written in Fortran, which performs mathematical operations by calling Basic Linear Algebra Subroutines (BLAS) [1]. The poor performance associated with MATLAB `for`-loops is primarily due to the accumulation of RAM-to-cache overheads,[1] which arise each time data is sent to and from cache.[2] To negate this, data can be sent in the form of a large vector, the BLAS calculations performed *in cache* before the data is transferred back to RAM. This significantly reduces the overheads associated with the calculation compared to non-vectorised BLAS calls. Optimum performance is achieved when the size of the vectors, referred to as 'blocks', sent to BLAS match the CPU cache memory size such that high speed cache reuse is maximised. Should the size of the block exceed the cache memory, LAPACK executes data-management algorithms and sub-optimal performance is realised [1].

In 2008, Dabrowski et al. [2] published an open source MATLAB code capable of solving a 2D linear elastic problem with 10^6 degrees of freedom in under one minute (MILAMIN). The algorithms, however, include a number of non-native MATLAB functions, which are written in C. Whilst the use of these non-native functions can improve the overall performance, the code becomes less transportable as the MATLAB software must have the SuiteSparse [3] package installed. Consequently, Bird et al. [4] conducted an investigation into extending the work of Dabrowski et al. to present a highly optimised linear elastic using only native MATLAB functions. Other work in this area includes the extension of the Dabrowski et al. [2] techniques to include parallelisation [5]. More recently, vectorisation techniques have been applied to different element formulations [6,7], comparisons made between different algorithms [8] and general routines devised which are applicable to multiple vector languages [9]. Although the focus of this paper is on non-linear solid mechanic, it is worth noting that the MILAMIN framework [2] has recently been extended to simulate the non-linear deformation of incompressible Newtonian and non-Newtonian fluids [10].

Bird et al. [4] also extended the algorithm to discontinuous Galerkin methods (DG) for linear elastic problems. The DG method introduces an alternative way to assemble and solve the system in question, whereby discontinuities between elements are permitted by averaging the jumps of variables at element interfaces [4]. In contrast to the standard continuous Galerkin (CG) methods, the degrees of freedom at the nodes are not shared by neighbouring elements. To maintain stability and ensure a realistic system response, the local stiffness calculations include a penalising term, which prohibits complete separation of elements faces. DG methods aid in localised p- and h-type refinement of a problem as no intermediate mesh is required to negate hanging nodes [11]. However the calculation of the element stiffness matrices requires additional integral terms and, therefore, is more computationally expensive than CG methods. This further highlights the need for an efficient calculation of the stiffness terms. Moreover, due to the element specific degrees of freedom, the global stiffness matrix is larger than that of a corresponding CG mesh, resulting in longer linear system solve times.

This paper extends the work of [2] and [4] to present two optimised elasto-plastic codes using CG and DG methods. Elasto-plasticity, also referred to as material non-linearity, describes a situation whereby the stress is no longer linearly dependent on the strain. The introduction of elasto-plasticity further improves the FEA approximations, which otherwise do not take into account the non-linear material properties. The first code formulated in this paper builds upon the work of Coombs et al. [12], who present a concise 70-line non-optimised MATLAB script to solve material and geometrically non-linear problems. This code acts as a framework upon which several optimisation techniques are implemented. By employing similar techniques to those used in the optimised CG code and extending the code formulated by Bird et al. [4], a second optimised code is presented for the incomplete interior penalty discontinuous Galerkin (IIPG) method for elasto-plastic problems.

This paper first provides the underlying theory behind DG finite element methods and elasto-plasticity in finite element analysis, before providing a breakdown of the key sections of each algorithm where these concepts are applied. The results obtained from the code are verified against a problem with an analytical solution, before providing performance testing to determine the overall speed gains achieved through optimisation. As the focus of this paper is on the efficient implementation of elasto-plasticity for CG and DG finite element methods, the equations are presented in matrix–vector format to facilitate implementation. We restrict the paper to two-dimensional analysis, although the algorithms can equally be applied to one and three dimensional non-linear problems.

2. Governing equations for CG and DG

2.1. Formulation of the weak form for linear-elasticity

The strong form equation is the governing partial differential equation which requires equilibrium to be satisfied at every point within the domain. For the linear-elastic problem, the strong form can be written as

$$-[L]^T\{\sigma\} = \{f\} \qquad \text{in } \Omega, \tag{1}$$

[1] These overheads exist in other languages, such as C/C++and Fortran, however they appear to not be as severe as in MATLAB where they can significantly degrade an algorithm's performance.

[2] See Dabrowski et al. [2] for a detailed discussion on the overheads associated with RAM to CPU cache transfer and the benefits of blocking for cache reuse.

subject to the following boundary conditions

$$[\sigma]\{n\} = \{g_N\} \quad \text{on} \quad \partial\Omega_N \qquad \text{and} \qquad \{u\} = \{g_D\} \quad \text{on} \quad \partial\Omega_D, \tag{2}$$

where $\Omega \subset \mathbb{R}^n$ is the domain, with boundary $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$, $[L]$ is the standard differential operator, $\{\sigma\}$ and $[\sigma]$ are the Cauchy stress in vector and matrix form, respectively, $\{f\}$ is vector of the body forces, $\{n\}$ is the unit outward normal to the boundary and $\{g_N\}$ and $\{g_D\}$ are the prescribed Neumann (traction) and Dirichlet (displacement) boundary conditions. To implement the governing equations in finite elements, the weak form of the equation must be derived, such that equilibrium is satisfied in a weighted average sense across all elements in the domain. Also, meshes and finite element spaces have to be defined. The finite element mesh is denoted by $\mathcal{T}_h$ and it is assumed to be shape-regular and to be constituted by quadrilateral elements. Single elements are denoted by E and faces by F. The set of all interior faces of the mesh $\mathcal{T}_h$ is denoted by $\mathcal{E}_h$. The CG and DG finite element spaces defined on $\mathcal{T}_h$ are denoted by $\mathcal{V}_p^{\mathrm{CG}}(\mathcal{T}_h)$ and $\mathcal{V}_p^{\mathrm{DG}}(\mathcal{T}_h)$, respectively, where p is the order of the elements. In the CG case we assume that $\mathcal{V}_p^{\mathrm{CG}}(\mathcal{T}_h) \subset [H_0^1(\Omega)]^n$ and in the DG case $\mathcal{V}_p^{\mathrm{DG}}(\mathcal{T}_h) \subset [L^2(\Omega)]^n$.

As the focus of this paper is on algorithmic and performance issues associated with the MATLAB implementation of non-linear solids mechanics, details of the adopted CG and DG formulations are not given in this section. Instead the interested reader is referred to the Appendix for details of the adopted CG and DG methods.

2.2. Extension to elasto-plasticity

In contrast to the linear elastic problem, for elasto-plasticity the stress in the domain is no longer linearly dependent on the nodal displacements and the stiffness of the material varies through the physical domain. The fundamental concept of the finite element method is that the determined solution must leave the system in a state of equilibrium. That is, the sum of the applied external forces is equal and opposite to the sum of the element internal forces. This underlying principal is referred to as the non-linear incremental equation of equilibrium

$$\{f^{\mathrm{oobf}}\} = \{f^{\mathrm{ext}}\} - \{f^{\mathrm{int}}\} = \{0\}, \tag{3}$$

where $\{f^{\mathrm{oobf}}\}$ is the residual out-of-balance force vector and $\{f^{\mathrm{ext}}\}$ is the sum of externally applied loads, including body forces and tractions. $\{f^{\mathrm{int}}\}$ denotes the internal forces at each node summed from the element internal force vectors $\{f^{\mathrm{e}}\}$. Consequently, it is essential to formulate a set of equations to calculate the internal forces in each element.

3. Elasto-plasticity

This section will outline the underlying concepts of elasto-plasticity which are implemented in the finite element codes.

3.1. Loadsteps

Material non-linearity provides a situation whereby the relationship between the stress and strain varies as the material yields. That is $\{\sigma\} \neq [D^e]\{\varepsilon\}$, where $[D^e]$ is the elastic stiffness matrix, $\{\varepsilon\} = [B]\{u\}$ is the strain and $[B]$ is the strain–displacement matrix. It is necessary to discretise the applied load into a number of loadsteps and solve the non-linear equation of equilibrium (3) for the unknown displacements, $\{d\}$, at each load level.

3.2. Newton–Raphson convergence of out-of-balance forces

By combining the equilibrium concept with the Newton–Raphson method, it is possible to iterate to a solution for each loadstep in a number of steps (where $n+1$ denotes the current loadstep number and k the Newton–Raphson iteration number):

1. The process will begin at a previously converged state, where $\{f^{\mathrm{oobf}}\} = \{0\}$ (point (i) in Fig. 1). The external force vector $\{f^{\mathrm{ext}}\}$ will be updated to the subsequent loadstep, and thus $\{f^{\mathrm{oobf}}\} \neq 0$.
2. Calculate the incremental nodal displacements $\{\delta d_{k+1}\}$. This is obtained by $\{\delta d_{k+1}\} = [K]^{-1}\{f_k^{\mathrm{oobf}}\}$, where $[K]$ is the global stiffness matrix, which acts as a tangent.
3. Recalculate the internal force vector $\{f^{\mathrm{int}}\}$ from the updated displacements $\{d_{n+1}^{k+1}\}$.
4. Recalculate the residual out-of-balance force vector $\{f_{k+1}^{\mathrm{oobf}}\}$ using Eq. (3). This gives point (ii) in Fig. 1.
5. Repeat steps (2)–(4) until the normalised residual out-of-balance force converges to a given tolerance, that is

$$\frac{\|\{f^{\mathrm{oobf}}\}\|}{\|\{f^{\mathrm{ext}}\}\|} \leqslant \text{tolerance}. \tag{4}$$

in this paper the tolerance is set to 1×10^{-9}.

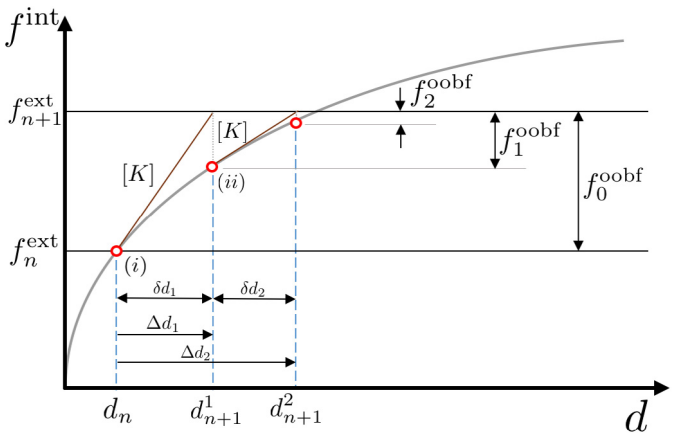

Fig. 1. Newton–Raphson convergence process.

3.3. Associated flow perfect plasticity

Here we restrict the material to linear elastic, perfectly plastic behaviour with associated plastic flow and a von Mises yield surface (Prandtl–Reuss constitutive model). This is the simplest, general, form of elasto-plasticity, however the algorithms presented in this paper are applicable to more sophisticated forms of non-linear material behaviour.

In 1913, R.E. von Mises [13] postulated an analytical yield surface f, used to define the admissibility of a given stress state. The yield function can be expressed as

$$f = \rho^2 - \rho_y^2 = 0, \tag{5}$$

where ρ is the deviatoric stress, and ρ_y is a limiting yield stress. The deviatoric stress ρ is defined as

$$\rho = \sqrt{2J_2}, \tag{6}$$

where J_2 is the second stress invariant determined by

$$J_2 = \frac{1}{2}\text{tr}([s][s]) \qquad \text{and} \qquad [s] = [\sigma] - \frac{\text{tr}([\sigma])}{3}[I], \tag{7}$$

where $[s]$ is the traceless deviatoric stress matrix, $[I]$ is a 3×3 identity matrix, and $\text{tr}(\cdot)$ is the trace operator. The von Mises yield surface can most easily be visualised in the principal stress space as an open-ended cylinder with its major axis aligned with the hydrostatic axis (where $\sigma_1 = \sigma_2 = \sigma_3$) with a radius of ρ_y. It should be noted that, in this paper, the yield surface remains at a fixed radius, ρ_y, due to the perfect plasticity assumption.

For associated flow plasticity theory we assumed that the outward normal to the yield surface provides the flow direction controlling the evolution of plastic strains, that is

$$\{\dot{\varepsilon}^{\text{p}}\} = \dot{\gamma}\left\{\frac{\partial f}{\partial \sigma}\right\} \tag{8}$$

where $\dot{\gamma}$ is the scalar plastic multiplier rate (or consistency parameter). This plastic multiplier must satisfy the Kuhn–Tucker–Karush consistency conditions

$$f \leq 0, \qquad \dot{\gamma} \geq 0 \qquad \text{and} \qquad f\dot{\gamma} = 0. \tag{9}$$

These conditions enforce that the material must either be on the yield surface undergoing elasto-plastic deformation ($f = 0$ and $\dot{\gamma} \geq 0$) or inside the yield surface with purely elastic behaviour ($f \leq 0$ and $\dot{\gamma} = 0$). The above plasticity equations are given in rate form, however in order for them to be implemented within a finite element algorithm, or even used as an stress–strain relationship to predict material behaviour at a single point, they must be integrated into an incremental relationship between stress and strain. This stress integration is the focus of the next section.

3.4. Stress integration

The key question that stress integration algorithms are required to answer is: *given a current stress state,* $\{\sigma_n\}$*, which is subjected to a strain increment,* $\{\Delta\varepsilon\}$*, what is the updated stress state,* $\{\sigma_{n+1}\}$*?* Algorithms to answer this question for

elasto-plastic constitutive models are normally split into three categories, namely: (i) explicit, (ii) implicit and (iii) exact stress integration. Several papers explore and contrast different stress integration approaches and an interested reader is refereed to the works of [14–16], amongst others. Explicit stress integration methods (see for example the initial work of Ilyushin [17] which was later applied to Prandtl–Reuss by Mendelson [18] and generalised by Nayak and Zienkewicz [19]) have advantages in terms of their simplicity but they do not enforce the consistency conditions at the updated stress state [20]. Implicit stress integration algorithms are normally formulated in terms of a prediction step followed by a correction for stress stated that induce elasto-plastic behaviour. These approaches were initially formulated by Wilkins [21] and have been developed over the last 50 years [22–26], including some analytical (closed-form) techniques [27–29] and general approaches that can be applied to arbitrary yield envelopes [30–32]. For textbook accounts of implicit methods see [33,34], amongst others. The key advantage of implicit methods over the earlier explicit approaches, despite their additional computational complexity, is that they rigorously enforce the consistency conditions at the new stress state. They also allow for larger stress increments to the applied to the constitutive model. However, in this paper we adopt an exact stress integration approach for von Mises elasto-plasticity of Wei et al. [35] for reasons outlined below.

The primary advantage of exact stress integration methods [15,35–39] is that they remove any errors associated with the stress integration process. However, exact stress integration methods are generally considered to be too expensive for routine engineering stress analysis [39]. In this paper we challenge this criticism as, due to their closed-form nature, exact stress integration routines are far more amenable to vectorisation as compared to iterative approaches (such as implicit stress updating). See Wei et al. [35] for details of the stress integration algorithm and consistent tangent, and Coombs et al. [12] for an example conventional (non vectorised) implementation. Section 4.7 provides details on the efficient implementation of the exact stress update algorithm.

An additional advantage of adopting an exact (or implicit) stress update algorithm is that it allows determination of the algorithmic consistent tangent [40,41] which facilitates optimum convergence of the global non-linear problem (3). The benefits of using the consistent tangent operator have recently been demonstrated by Duretz et al. [42]. This tangent is the linearisation of the stress updated algorithm with respect to changes in the trial elastic strain state and replaces $[D]$ in the global stiffness matrix, $[K]$, assembled using (A.2). The algorithmic consistent tangent derived by Wei et al. [35] can be expressed as

$$[D] = \eta_1\{s_n\}\{s_n\}^T + \eta_2\{s_n\}\{\Delta\epsilon\}^T + \eta_3\{\Delta\epsilon\}\{s_n\}^T + \eta_4\{\Delta\epsilon\}\{\Delta\epsilon\}^T + \eta_5[I], \tag{10}$$

where η_i are variables that depend on the stress state and the deviatoric component of the applied strain increment, $\{\Delta\epsilon\}$. $\{s_n\}$ is the vector form of the deviatoric stress, $[s]$ see (7), evaluated at the previously converged (or initial) stress state. As $\eta_2 \neq \eta_3$, the algorithmic consistent tangent is non-symmetric and the numerical implications of this will be explored in Sections 4.7 and 4.8.

4. Elasto-plastic optimised CG algorithm

Algorithm 1 outlines the code structure used for the optimised elasto-plastic MATLAB algorithm, highlighting the key sections of code, which are further explained in the subsections that follow. However, before looking at specific aspects of the code it is helpful to explain the main speed-up mechanism — blocking.

Standard finite element implementations loop over each Gauss point for each finite element and determine the stiffness contribution of each Gauss point to their parent element and then assemble the element stiffness matrix into a global stiffness matrix. The critical speed up mechanism for the algorithm shown in Algorithm 1 is blocking. Elements are grouped into blocks, where the optimum block size is dependent on the physical architecture of the machine used to perform the assembly calculations, and then vectorised stiffness calculations are performed for all the elements in the block for a specific Gauss point location. The stiffness contributions are then assembled into a global stiffness matrix.

4.1. Mesh set-up and block size determination

Line 1 of Algorithm 1 denotes where the problem set-up and script initialisation occur. The mesh set-up is achieved by calling a MATLAB function, which returns the element coordinates `coord`, the element topology `etpl` and boundary conditions `bc`. Furthermore, the material properties of Young's modulus and Poisson's ratio are returned, which are used to calculate the elastic stiffness and compliance matrix. These matrices are used several times later in the code, and hence it is favourable only to declare these variables once in the code. Additionally, it is here that the material yield stress is defined.

The number of elements per block `nelblo` is determined from a user input, referring to the size of the vectors sent to BLAS, which is dependent on CPU cache size. Finally, memory allocation is declared for two vectors used to store all of the element stiffness matrices and element internal force vectors, `K_all` and `felem_all` respectively, until the global stiffness matrix is assembled. The allocations are defined as `K_all=zeros(nels,nDoF,nDoF)` and `felem_all = zeros(nels,nDoF)`, where `nels` is the number of elements in the domain and `nDoF` is the total number of degrees of freedom for an element.

Algorithm 1 Optimised CG elastoplatic code.

1: Mesh set-up and block size determination
2: **for** Loadstep loop **do**
3: Update the external force vector $\{f^{\text{ext}}\}$
4: Calculate initial residual out-of-balance force $\{f^{\text{oobf}}\}$
5: **while** $\frac{||\{f^{\text{oobf}}\}||}{||\{f^{\text{ext}}\}||} \geqslant$ tolerance **do**
6: Solve equations for displacement increment
7: **for** Element block loop **do**
8: Element block memory allocation
9: **for** Gauss point loop **do**
10: Gauss point initialisation
11: Strain increment calculation
12: Call to **Constitutive model** (von Mises elasto-plasticity)
13: Local stiffness calculation $[k^e]$
14: Local internal force calculation $\{f^e\}$
15: **end for**
16: Assignment to temporary block storage
17: **end for**
18: Global stiffness matrix and internal force vector assembly
19: Update out-of-balance residual force $\{f^{\text{oobf}}\}$
20: **end while**
21: Set reference variables equal to converged values
22: **end for**

```
for lstp=0:no_lstps
   fext=(lstp/no_lstps)*fext0;
   oobf=react+fext-fint; oobfnorm=2*NRtol;
   NRit=0;
   while ((NRit<NRitmax)&&(oobfnorm>NRtol))
      NRit = NRit + 1;
      ...
   end
end
```

Fig. 2. Code fragment corresponding to lines 3–5 of Algorithm 1.

4.2. Out-of-balance residual calculation

Lines 3 and 4 of Algorithm 1 are responsible for updating the external force vector `fext` and calculating the residual out-of-balance force `oobf`, respectively. Fig. 2 shows the corresponding code fragment. It can be observed that the external force vector is updated upon each load step `lstp`, as an appropriate proportion of the total external force applied `fext0`, where `no_lstps` is the total number loadsteps.

The residual out-of-balance force is calculated as the difference between the internal force vector `fint` and the sum of the external and reaction forces (`fext` and `react` respectively). It should be noted that the L2 norm variable `oobfnorm` is assigned a value of twice the Newton–Raphson tolerance `NRtol`, such that the **`while`**`()` loop is entered on the first iteration of each loadstep.

Furthermore, an iteration counter `NRit` is included to ensure the number of iterations does not exceed the user defined maximum number iterations `NRitmax`, should the solution fail to converge.

4.3. Linear solution for displacement increment

Line 6 of Algorithm 1 is where the system solver function is called. Using the out-of-balance force array `oobf` and the global stiffness matrix from the previous iteration `K`, the solver returns the incremental displacements and reaction forces, subject to the system boundary conditions. The solver uses the native MATLAB function `mldivide()`, which is capable of solving a large system of linear equations.

4.4. Element block memory allocation

Line 8 of Algorithm 1 denotes where the memory allocations for the current block take place. Memory is assigned to the following variables: the Jacobians `Jx`, `Jy`, inverse Jacobians `invJx`, `invJy`, the block storage local stiffness matrices `K_block`, the block storage element internal force vector `felem` and the updated stress vector `sig_new`. The variables `K_block` and `felem` act as temporary storage for all elements considered within the block before they are stored in `K_all` and `felem_all`,

Table 1
Size of memory allocated to the variables in each block.

Variable	Rows	Columns	Third dimension
Jx, Jy	nelblo	nD	–
invJx, invJy	nelblo	nD	–
K_block	nelblo	nDoF	nDoF
felem	nelblo	nDoF	–
sig_new	nelblo	3	–

```
deps = zeros(nelblo, 3);
for i = 1:nen
 deps(:,1)=deps(:,1)+dNx(:,i).*du(etpl(indx_coord,i));
 deps(:,2)=deps(:,2)+dNy(:,i).*dv(etpl(indx_coord,i));
 deps(:,3)=deps(:,3)+dNy(:,i).*du(etpl(indx_coord,i))...
                    +dNx(:,i).*dv(etpl(indx_coord,i));
end
```

Fig. 3. Code fragment corresponding to line 11 of Algorithm 1.

respectively. Table 1 presents the size of each variable to which memory is assigned in this section, where `nD` is the number of degrees of freedom at each node.

At the start of each loop, `K_block` and `felem` are required to be set to zero, so as to eliminate the possibility of incorrect addition during the Gauss point loop.

4.5. Gauss point initialisation

Line 10 of Algorithm 1 denotes where the derivatives of the shape functions, with respect to the global coordinates, are determined `dNx`, `dNy`. The method used is consistent with that of [2] and [4] for the volume and surface terms and, therefore, will not be discussed.

4.6. Strain increment calculation

Line 11 of Algorithm 1 denotes where the strain increment is calculated at the Gauss point locations. To achieve maximum speed gains, a set of expressions must be derived such that the strain increment $\{\delta\varepsilon\}$ can be simultaneously calculated for the current Gauss point of every element in the block. For plane-strain/stress problems, the strain increment expressions can be written as

$$\begin{Bmatrix} \delta\varepsilon_{xx} \\ \delta\varepsilon_{yy} \\ \delta\gamma_{xy} \end{Bmatrix} = \sum_{i=1}^{n_{en}} \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix} \begin{Bmatrix} \delta u_i \\ \delta v_i \end{Bmatrix}, \tag{11}$$

where δu and δv are the incremental displacements at the nodes as returned from the solver. $\frac{\partial N}{\partial x}$ and $\frac{\partial N}{\partial y}$ are the shape function derivatives with respect to the global coordinates x and y respectively, and n_{en} is the number of element nodes. By multiplying out (11), expressions are obtained for each strain direction as outlined in the code fragment shown in Fig. 3, where `deps` is the strain increment and `du`, `dv` are the incremental displacements in x and y respectively.

4.7. Constitutive model

Line 12 of Algorithm 1 is where the constitutive model is called. This function is used to assess the stress state of each Gauss point, returning an updated $[D]$ matrix. It should be noted that this constitutive model is consistent with the function implemented by Coombs et al. [12], however it has been rewritten in a vectorised block form to reduce the number of calls to BLAS. This requires the separation of all the Gauss points in the current block into those acting elastically and those undergoing elasto-plastic deformation. This is done by implementing the von Mises yield surface introduced in Section 3.3, shown in the code fragment in Fig. 4.[3] Here, `epsEtr` is the trial strain state of the Gauss point,

[3] Note that in the algorithm shown in Fig. 4 the stress and strain quantities are expressed in general 9-component form with the following format $\{\bar{\sigma}\} = \{\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yx}, \sigma_{yz}, \sigma_{zy}, \sigma_{zx}, \sigma_{xz}\}$, where the bar denotes a 9-component form. This is for a number of reasons: (i) it facilitates the calculation of the trace of a matrix in vector form such that both shear components are correctly accounted for which allows for vectorisation of the calculation of ρ and (ii) it allows the algorithm to the applied to 1D, 2D (plane stress, plane strain and axi-symmetry) and 3D analysis by only changing the supplied increment in strain.

```
bm1=[1 1 1 0 0 0 0 0 0]';
epsEtr=epsEn+deps;
sig_tr=De*(epsEtr);
s_tr=sig_tr-bm1*sum(sig_tr(1:3,:))/3;
f=sum(s_tr.*s_tr,1)-fc^2;
elast_indx = find(f<0);
plast_indx = find(f>=0);
if plast_indx
   ...
end
```

Fig. 4. Code fragment corresponding to line 12 of Algorithm 1. `bm1` is a vector of ones and zeros, used to calculate the trace in vector form. `fc` is the user specified yield stress of the material.

calculated by adding the strain increment `deps` to the previously converged strain state `epsEn`. By multiplying this trial strain state by the elastic stiffness matrix `De`, the trial stress state `sig_tr` is obtained for every element in the current block.

An expression for the yield function can be derived by substituting (7) into (6), and subsequently substituting the resultant equation into (5). This gives

$$f = \mathrm{tr}([s][s]) - \rho_y^2. \tag{12}$$

Within the code, this is implemented by first calculating the trace of the deviatoric stress matrix in vector form, which is assigned to the variable `s_tr`. The squaring of `s_tr` is calculated using `sum(s_tr.*s_tr,1)`, such that a single scalar quantity of ρ is obtained for each Gauss point in the block. Consequently, the vector of the yield function `f` is calculated. Two logical statements are made using this vector: `elast_indx = find(f<0)` and `plast_indx = find(f>=0)`. These statements return vectors containing the locations of the elastic and plastic Gauss points within the `f` vector where `f<0` and `f>=0` respectively. That is, `elast_indx` will contain an index for the elements that contain an admissible stress state for the Gauss point in question, which are therefore elastic. Conversely, `plast_indx` denotes the elements for which the stress state of the Gauss point lies on or outside the yield surface, which will undergo elasto-plastic deformation.

The variable `plast_indx` is also used as a logical indicator, as shown in Fig. 4. The script section bounded by '`if plast_indx`' will only be entered if there exists one or more plastic Gauss point in the current block. Should this be the case, this section of code will be executed in order to calculate the $[D]$ matrix for all elasto-plastic Gauss points. See [35] or [12] for details of the determination of $[D]$, the algorithmic consistent tangent.

The values of [D] are calculated and subsequently output within a MATLAB structure `D.a`...`D.i`, where $a \ldots i$ denote a location in $[D]$ such that

$$[D] = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}. \tag{13}$$

For each plastic Gauss point within the block, the values $a \ldots i$ are unique and thus $[D]$ is potentially non-symmetric (see (10) in Section 3.4). The values are steered into the overall `D.a` vector using `plast_indx`, for example `D.a(plast_indx,:) =` (expression for plastic a). For all elastic Gauss points, the values $a \ldots i$ are assigned consistent values, such that the matrix takes the form of the elastic stiffness matrix $[D^e]$ e.g. `D.a(elast_indx,:) =` (expression for elastic a). Thus, the total length of each array contained within the `D` structure is the block length `nelblo`.

Additionally, the updated stress and strain states are calculated and outputted from the function, `epsE` and `sig_new` respectively. Again, these variables are plastic dependent and, therefore, also require assembling using `elast_indx` and `plast_indx`.

4.8. Local stiffness calculation

Line 13 of Algorithm 1 references where the local stiffness matrices $[k^e]$ are calculated for all elements in the block. For elasto-plasticity, the $[D]$ matrices are potentially non-symmetric and thus the resultant stiffness matrices are also non-symmetric. Consequently, all terms in the stiffness matrix need to be calculated, degrading the overall performance (both in terms of stiffness assembly and linear solution). In order to negate this problem, the $[D]$ matrices for the plastic Gauss points are forced to be symmetric. By taking the average of the off-diagonal terms, a symmetrical matrix is obtained

$$[D_{AS}] = \begin{bmatrix} a & \alpha & \beta \\ \alpha & e & \gamma \\ \beta & \gamma & i \end{bmatrix}, \tag{14}$$

```
for ii=1:nen
   mult_ii = (ii-1)*nD;
   felem(:,1+mult_ii)=felem(:,1+mult_ii)...
               + (dNx(:,ii).*sig_new(indx_coord,1)...
               + dNy(:,ii).*sig_new(indx_coord,3)).*w;
   felem(:,2+mult_ii)=felem(:,2+mult_ii)...
               + (dNx(:,ii).*sig_new(indx_coord,3)...
               + dNy(:,ii).*sig_new(indx_coord,2)).*w;
end
```

Fig. 5. Code fragment corresponding to line 14 of Algorithm 1. Here, `felem` is the element internal force vector, `dNx`, `dNy` are the shape function derivatives, `sig_new` are the non-linear stress states and `w` denotes the integral weight points.

where $[D_{AS}]$ denotes the average symmetric form of $[D]$. With reference to (13)

$$\alpha = \frac{1}{2}(b+d), \quad \beta = \frac{1}{2}(c+g), \quad \gamma = \frac{1}{2}(f+h), \tag{15}$$

where a, e and i retain their original values. Although this results in sub-optimal convergence through a small error in the global stiffness tangent, the speed gained through the assembly of the local stiffness matrices results in an overall performance improvement. It should also be noted that the performance of the solver is enhanced when solving a symmetric matrix.

From here, a similar technique is used to those evident in [4] and [2], whereby the upper triangle of the stiffness matrices is calculated explicitly by looping over the element nodes. Each term within the stiffness matrix is a function of the values from the $[D]$ matrix, the shape function derivatives `dNx` and an integration weighting `w`. This Jacobian determinant `detJ` is included within `w`. The key difference is that $[D]$, in general, can be different at each Gauss point in each element.

4.9. Local internal force calculation

Line 14 of Algorithm 1 denotes where the local element internal force vector is calculated. The expression for this is obtained by summarising (A.6) as

$$\begin{Bmatrix} f^e_{x_j} \\ f^e_{y_j} \end{Bmatrix} = \sum_{i=1}^{n_{gp}} \sum_{j=1}^{n_{en}} \begin{bmatrix} \frac{\partial N_j}{\partial x} & 0 & \frac{\partial N_j}{\partial y} \\ 0 & \frac{\partial N_j}{\partial y} & \frac{\partial N_j}{\partial x} \end{bmatrix} \begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix}_i \det([J])w_i. \tag{16}$$

By multiplying out this equation, expressions are derived for each element internal force vector entry. The implementation of this is achieved by looping over the element nodes, as shown in Fig. 5.

4.10. Assignment to temporary block storage

Line 16 of Algorithm 1 denotes where the local stiffness and internal force vectors for the current block are assigned to temporary variables, `K_all` and `felem_all` respectively. These variables remain unused until the calculations for all blocks are complete. The variables are subsequently used to assemble the global stiffness matrix and internal force vector.

4.11. Global stiffness matrix and internal force vector assembly

Line 18 of Algorithm 1 refers to where the global stiffness matrix `K` and global internal force vector `fint` are assembled. The global stiffness matrix is assembled using the native MATLAB function `sparse()`, a memory efficient way to store large matrices by only storing non-zero terms and their corresponding matrix locations.

The assembly of the global stiffness matrix is therefore achieved by executing

```
K=sparse(ed_i,ed_j,K_all,tndof,tndof),
```

where `ed_i`, `ed_j` are the steering vectors for rows and columns respectively.[4] The overall size of the matrix is defined by `tndof`, the total number of degrees of freedom for the system. The assembly of `fint` is achieved by executing

```
fint=accumarray(ed,felem_all),
```

where `ed` is a vector denoting the degrees of freedom at each node. The MATLAB function `accumarray()` is used to construct an array with accumulation. This is necessary here, where the internal forces at nodes can contain contributions from more than one element.

[4] Note that `ed_i`, `ed_j` are only calculated once (on the first Newton iteration of the first load step) and then stored for repeated calls to `sparse`. The code allows the `sparse` function to sum the repeated row/index entries in the global stiffness matrix rather than pre-computing reduced fill-in of the global stiffness matrix.

Algorithm 2 DG stiffness and internal force calculation.

1: **for** Face block loop **do**
2: Face block initialisation
3: **for** Surface Gauss point loop **do**
4: **for** Loop over element faces **do**
5: Determine global shape functions for each face
6: **end for**
7: Strain increment calculation for positive and negative Gauss points
8: Calls to **Constitutive model** for positive and negative Gauss points
9: Local stiffness calculation $[k^e]$
10: Local internal force calculation $\{f^e\}$
11: **end for**
12: Assignment to temporary block storage
13: **end for**

```
for i = 1:nen
    deps_p(:,1)=deps_p(:,1)+dNx_p(:,i).*du_p(:,i);
    deps_n(:,1)=deps_n(:,1)+dNx_n(:,i).*du_n(:,i);
    deps_p(:,2)=...
end
```

Fig. 6. Code fragment showing the first component of strain increment calculation for positive and negative Gauss points.

4.12. Update out-of-balance residual force

Line 19 of Algorithm 1 is where the out-of-balance residual force is recalculated using the newly calculated internal force vector. The L2 norm of the subsequent vector is obtained using the native MATLAB `norm()` function. This value is then compared to the Newton–Raphson tolerance and, if smaller, the loadstep has converged and the **`while`**-loop is exited.

4.13. Set reference variables equal to converged values

Line 21 of Algorithm 1 denotes where the reference variables are set to the converged values from the completed loadstep. The reference variables required for the next loadstep are the total elastic strain vector `epsE` and the total displacement `uvw`, which are assigned to `epsEn` and `uvwold` respectively.

5. Elasto-plastic optimised algorithm in DG

The elasto-plastic optimised DG code takes the same form as the CG code, however it requires one extra loop to determine the stiffness and internal force contributions from the element faces. The DG algorithm can be expressed by simply including the additional pseudo code presented in Algorithm 2 between lines 17 and 18 of Algorithm 1. The stiffness and internal force calculations discussed in the CG code can be directly implemented into the DG code, as this is identical to the area integral required for term (Q) in (A.2). It should be noted that lines 1 to 6 are unchanged from the DG linear-elastic code presented by Bird et al. [4] and, therefore, will not be discussed.

5.1. Strain increment calculation

The DG strain increment calculation for the DG face terms is performed on Line 7 of Algorithm 2. As in the CG code, the strain increment due to increased externally applied load must be calculated. However, the DG code differs such that the strain increments must be obtained at the current Gauss point for both the $^+$ and $^-$ elements. Fig. 6 shows the implementation of this within the code for the first component of strain $\delta\varepsilon_{xx}$ only. Here, `deps_p` and `deps_n` denote the positive and negative strain increments respectively, while `dNx_p`, `dNy_p` and `dNx_n`, `dNy_n` are the positive and negative shape function derivatives respectively. Additionally, the incremental displacements of the positive and negative elements are denoted by `du_p` and `du_n` respectively.

5.2. Constitutive model for positive and negative Gauss points

Line 8 of Algorithm 2 references where the constitutive model is called for both the positive and negative elements surfaces. A consistent constitutive model should be used for all 3 calls throughout the DG code (1 call for the area calculations and 2 calls for surface calculations), varying only in the strain states applied as an input to the function. That is, for the positive element face ∂E^+, the strain state from the previously converged loadstep `epsEn_surfp` and the

Algorithm 3 Local internal force calculation for DG.

1: **for** Loop over element nodes **do**
2: Calculate penalty contribution terms
3: **end for**
4: **for** Loop over element nodes **do**
5: Calculate the degrees of freedom for current node
6: Calculate the surface contribution terms
7: Assemble local internal force using penalty and surface contribution terms
8: **end for**

positive strain increment `deps_p` are sent to the constitutive model. This is also the case for the negative element face ∂E^-, using `epsEn_surfn` and `deps_n` instead.

Each Gauss point is determined to be elastic or plastic using the same von Mises criteria discussed in Section 4.7 and the appropriate $[D]$ matrices are returned. Again, the values of these matrices are returned in MATLAB structure form, where `Dp.a...i` and `Dn.a...i` denote the positive and negative $[D]$ matrices respectively. It should be noted that the values $a \ldots i$ take the same asymmetrical form as shown in (13). Also returned are the updated non-linear stress states `sig_p` and `sig_n` and the strain states `epsE_surfp` and `epsE_surfn`, for the positive and negative surfaces respectively.

5.3. Local stiffness calculation

Line 9 of Algorithm 2 is where the terms $R_{1\ldots4}$ and $S_{1\ldots4}$ of (A.2) are calculated, where each term is a `[ndof]×[ndof]` matrix. As previously discussed, the (Q) term has already been calculated in the area integral. By multiplying out the equations in (A.4), it was found that there exists a set of common algebraic expressions for entries of the $R_{1\ldots4}$. These repeated terms are a function of the shape function derivatives, the outward facing normal vectors and components of the $[D]$ matrices calculated in the constitutive model.

By executing two nested **`for`**-loops over the element nodes, the stiffness terms for each node's degrees of freedom can be calculated. The repeated terms that arise from multiplying out the $R_{1\ldots4}$ expressions vary only in columns and, therefore, lie outside the second nested **`for`**-loop. Consequently, these terms are only calculated `nen` number of times, reducing the number of calls to BLAS and, thus, reducing the overhead time. However, due to the omission of the symmetric DG term, the entirety of each stiffness matrix must be determined. This differs from the optimised linear elastic code, whereby only the upper triangle is calculated and subsequently transposed.

Within the nested **`for`**-loop, the $R_{1\ldots4}$ stiffness terms are calculated and placed into the correct location within the matrix, using steering terms `mult_i` and `mult_j`. The stabilising terms $S_{1\ldots4}$ are also calculated here, which are derived from the derivatives of the shape functions, nodal displacements and integration weightings. These stabilising terms are subtracted from the $R_{1\ldots4}$ terms, in accordance with (A.2), to obtain the overall stiffness matrix contribution for the Gauss point in question. On each Gauss point loop, the stiffness matrix contributions from the current Gauss point are summed with those from the previous loops.

5.4. Local internal force calculation

Algorithm 3 describes the code structure used to calculate the internal force contributions, line 10 of Algorithm 2, from terms $R_{1\ldots4}$ and $S_{1\ldots4}$, utilising two sequential **`for`**-loops. The initial loop calculates the internal force components, which arise from the penalty terms $S_{1\ldots4}$ and are functions of the shape functions and the node displacements for both the positive and negative elements. The second loop is used to calculate the internal force components arising from the surface integral terms $R_{1\ldots4}$ and are functions of the outward facing normal vectors and the non-linear stress components of the positive and negative elements. The overall element internal force components at each element degree of freedom can subsequently be determined by combining the contributions from the area integral (calculated at line 14 of Algorithm 1), the surface integral and the penalty terms.

It should be noted that the first **`for`**-loop cannot be nested within the second, as the penalty term is a cumulative variable, which requires information from every element node before it can be used to calculate the internal forces at each degree of freedom.

5.5. Assignment to temporary storage block

This occurs on Line 12 of Algorithm 2 and is equivalent to Line 16 of Algorithm 1 in CG, with the inclusion of two temporary variables for the positive and negative element internal forces, `felem_p_all` and `felem_n_all` respectively.

6. Numerical results and discussion

This section presents the analysis of a two-dimensional elasto-plasticity problem with a known analytical solution. The section initially focuses on determination of the optimum block size for the hardware/software used and then investigates the speed-up of the optimised algorithm.

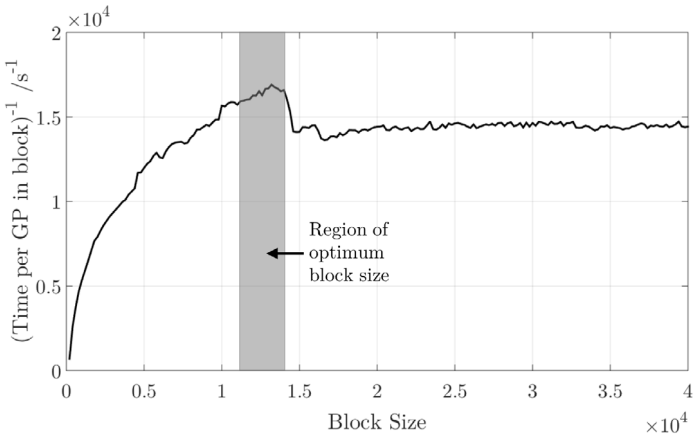

Fig. 7. Performance variation of constitutive model with varying block size.

6.1. Determining the optimum block size

Optimum performance is reached by maximising the cache reuse and, therefore, is CPU dependent. Performance testing was carried out to determine the optimum block size for the computer in use. All simulations were run within the Windows 7 OS on version MATLAB R2015b. The CPU used an Intel Xeon E5-1620 V2@3.70 GHz processor with 64GB of RAM.

To determine the optimum block size, a `for`-loop was set up around the vectorised constitutive model (the bottleneck in the vectorisation of the non-linear finite element algorithm), which increased the block size by 200 on each loop. All variables were cleared at the beginning of each loop and the stress, strain and incremental strain vectors were subsequently declared. These variables were consistent for each block size, to ensure the tested trial stress state was deemed inadmissible, such that the plastic section of code was executed. The variables were repeated to match the block size, using the MATLAB `repmat()` function.

Each loop was timed and subsequently divided by the block size to determine the average calculation time per Gauss point in the block. By plotting the inverse of these times, the peak speed can be observed, as shown in Fig. 7. It should be noted that each loop was executed 40 times, from which the mean time was calculated, to reduce the effect of speed variation due to CPU background tasks. Here, the optimum block size was observed to be $\approx$ 13000 and, henceforth, this was the block size used for all simulations[5]. The grey region in Fig. 7 is the region where cache-reuse is maximise and the overhead associated with data transfer is relatively insignificant. For block sizes $>$ 15000, the routine requires more memory than available in the CPU cache and this inhibits cache reuse (see Dabrowski et al. [2] for a discussion on this point). Note that the block size is not influenced by the number of Gauss points used to integrate the finite element as the Gauss point loop contains the blocking algorithm.

6.2. Result validation against an analytical solution

The results obtained for both CG and DG were validated using a problem with an analytical solution. The problem analysed was the stretching of a double-notched plate, initially presented by Nagtegaal et al. [43] for small strain plasticity. The plate assumes a Young's modulus $E = 206.9$ GPa, a Poisson's ratio $\nu = 0.2$ and a yield stress $\rho_y = 0.45$ GPa.

The height and width of the specimen were defined to be 30 mm and 15 mm respectively, with a 2 mm unit linking ligament at mid height, as shown in Fig. 8(ii). For this implementation, a plane-strain assumption was made, such that the total out-of-plane strains were assumed to be zero. Due to the symmetrical nature of the problem, only one quarter of the specimen was discretised using 75 bi-linear four-noded quad elements integrated using 2×2 Gauss quadrature (four points per element). For this geometry, Nagtegaal et al. [43] postulates a small strain analytical load due of $F_{\text{lim}} \approx 2.673$kN. Fig. 8(i) shows the force–displacement graph for the first 6 refinements of the mesh, run using the optimised CG script. Each refinement halves the element size in both directions, hence quadrupling the number of elements. In each case, a displacement of 2 mm was applied to the top edge of the plate over 20 equal loadsteps.

It is clear from the responses that the numerical solution from the CG code converges towards the correct analytical solution. It is also evident that the same response is achieved in DG, as shown in Fig. 9, which compares the error after each refinement. The degrees of freedom for each of the meshes is given in Table 2.

[5] Note that this is the optimum block size for this specific architecture and the optimum block size will vary depending on the hardware/software used. However, 13000 can provide a useful initial investigation point for determining the optimum block size for other machines.

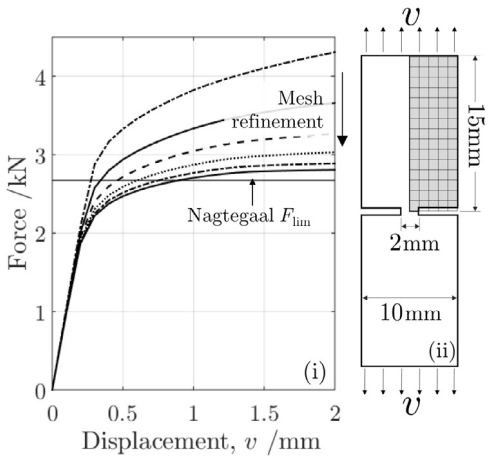

Fig. 8. Nagtegaal et al. [43] plate problem (i) load–displacement curve for mesh refinement (ii) specimen dimensions.

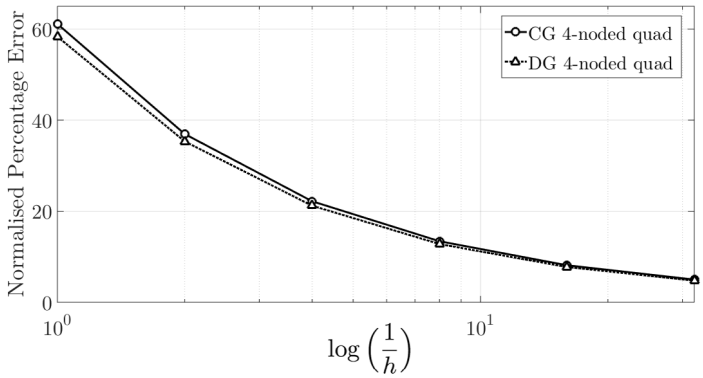

Fig. 9. Percentage error of solution with decreasing element size h.

Table 2
Degrees of freedom for the CG and DG meshes.

Refinement	CG	DG
1	192	600
2	682	2,400
3	2,562	9,600
4	9,922	38,400
5	39,042	153,600
6	154,882	614,400

6.3. Average-symmetric vs non-symmetric [D] matrix

This section assesses the effect of applying the average-symmetric [D] matrix in the CG optimised code. The effect of the matrix symmetricity is analysed for both the Newton–Raphson convergence rate and the system solver performance. Fig. 10 presents two convergence graphs for loadstep 5 of the fourth refinement. It is evident from both graphs that using a non-symmetric [D] matrix results in a quicker convergence rate, such that it requires two fewer iterations. It can also be seen from Fig. 10(ii) that the logarithmic gradient of convergence of the non-symmetric matrix is ≈ 2 or greater for each iteration. This suggests correct implementation, as the Newton–Raphson method should asymptotically approach quadratic convergence if the correct tangent is used.

Conversely, the average symmetric matrix converges sub-optimally. This is clearly apparent in Fig. 10(ii) whereby the logarithmic gradient of convergence is < 2 for all iterations. This is due to the error in the global stiffness tangent acquired when forcing the [D] matrix to be symmetric. Although this results in the need for additional iterations per loadstep, dramatic speed gains are achieved in both the assembly of the local stiffness matrix and the solver time. This is shown in Fig. 11, which shows the time to assemble the local stiffness matrices, the solver time and the overall time for the first six refinements. The speed gains are clearly evident for all three measured sections of code, with maximum

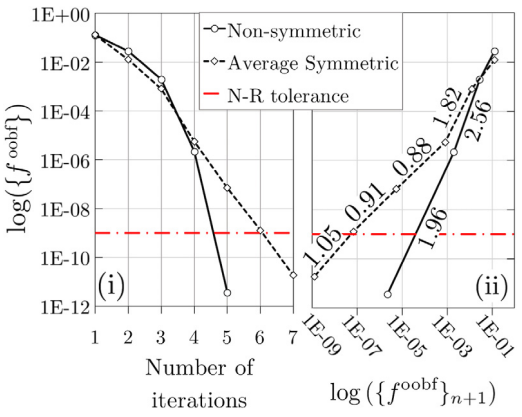

Fig. 10. N–R convergence of loadstep 5 of the fourth refinement for CG optimised code. Gradients are displayed in (ii).

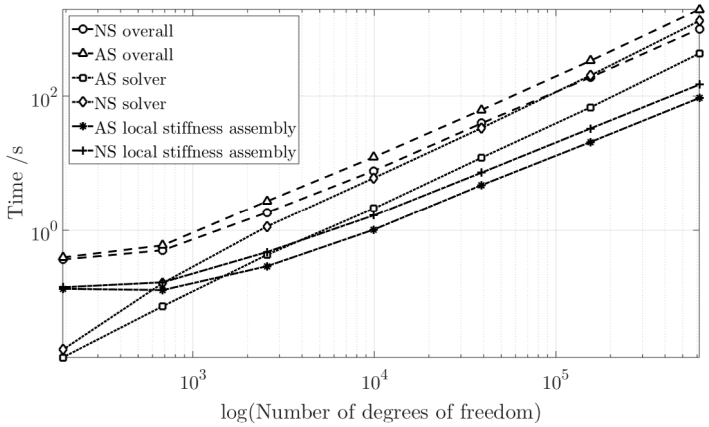

Fig. 11. Time taken for local stiffness assembly, solver and overall simulation for the DG analysis. AS denotes average-symmetric, whilst NS denotes non-symmetric.

speed gains obtained for the sixth refinement. Here, applying the average-symmetric matrix method yielded speed gains of $\times 1.58$, $\times 3.05$ and $\times 1.94$ for the local stiffness assembly, solver time and overall simulation respectively.

6.4. Identification of performance bottlenecks

It is possible to identify performance bottlenecks by analysing the time spent in various sections of the code. Fig. 12 shows a time breakdown of the DG optimised code, where the solver time, surface integral time, volume integral time and sparse assembly are plotted as a percentage of the overall simulation time. The graph clearly highlights the dominance of the solver time for large problems. For the sixth refinement, the solver time accounts for 76% of the total simulation time, acting as a performance bottleneck. This outcome was also observed within the optimised CG code, however the asymmetrical nature of the DG global stiffness matrix results in greater performance degradation.

6.5. Overall performance improvements

The overall performance improvements were analysed by comparing the elasto-plastic CG and DG optimised codes to their corresponding non-optimised MATLAB scripts for the notched plate problem. The simulation times for the first six refinements can be observed in Fig. 13 and Table 3 for both CG and DG optimised and non-optimised scripts.

For the CG code, the maximum speed gain of $\times 25.7$ is achieved for refinement 4. Beyond this, the solver time begins to dominate and the speed enhancement is marginally reduced, with a speed gain of $\times 22.0$ for refinement 6. Maximum speed gains of $\times 10.1$ are obtained for refinements 3 and 4 of the DG code, beyond which the solver time dominates heavily. By excluding the solver time, set-up time and sparse assembly time, a true comparison can be made between

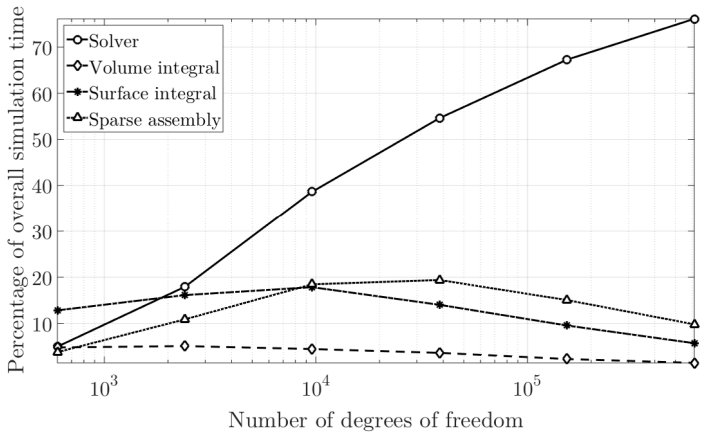

Fig. 12. Percentage of time spent in each section of the DG code.

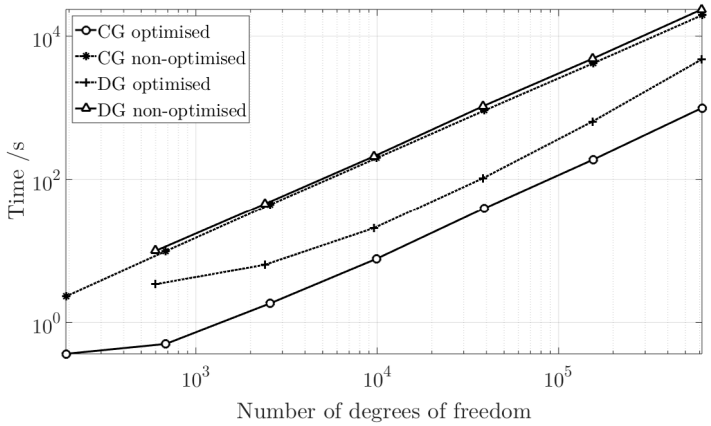

Fig. 13. Simulation times for CG and DG codes to complete refinements 1 to 6, for optimised and non-optimised scripts.

Table 3
Simulation times for optimised and non-optimised codes.

	CG		DG	
Refinement	Optimised time (s)	Non-optimised time (s)	Optimised time (s)	Non-optimised time (s)
1	0.371	2.31	3.43	9.99
2	0.502	9.88	6.38	45.5
3	1.85	44.4	20.8	211
4	7.75	199	104	1060
5	39.6	919	644	4830
6	189	4170	4720	23500

the optimised sections of code and the corresponding non-optimised sections. Table 4 provides a summary of this data, showing maximum speed gains of $\times 52$ and $\times 47$ for CG and DG respectively.

The total number of iterations and the run time per Newton–Raphson iteration are given in Table 5. The run time per iteration is based on the total time minus the solver time, set-up time and sparse assembly time, that is the run times given Table 4. The maximum speed gain per iteration for the CG method is $\times 55$ which is greater than the speed gain based on Table 4. This is due to more iterations being required in the optimised symmetric CG algorithm which has sub-optimal convergence, as shown in Section 6.3. However, in the authors' opinion quantifying the speed gains in terms of the total simulation time is more appropriate for non-linear algorithms if they require different numbers of iterations to find convergence.

Table 4
Simulation times for optimised and non-optimised codes, disregarding the solver time, setup time and sparse assembly time.

Refinement	CG		DG	
	Opt time −(Solve + Setup + Sparse) (s)	Non-opt time −(Solve + Setup + Sparse) (s)	Opt time −(Solve + Setup + Sparse) (s)	Non-opt time −(Solve + Setup + Sparse) (s)
1	0.322	2.27	3.08	9.61
2	0.343	9.64	4.52	43.4
3	0.945	43.0	8.74	199
4	3.54	192	25.2	971
5	17.2	876	94.2	4260
6	74.9	3930	403	19100

Table 5
Total number of Newton iterations and run times per iteration (in brackets, s)

Refin.	CG		DG	
	Opt.	Non-opt.	Opt.	Non-opt.
1	61 (5.3×10^{-3})	59 (3.8×10^{-2})	61 (5.0×10^{-2})	61 (1.6×10^{-1})
2	73 (4.7×10^{-3})	65 (1.5×10^{-1})	69 (6.6×10^{-2})	69 (6.3×10^{-1})
3	80 (1.2×10^{-2})	74 (5.8×10^{-1})	79 (1.1×10^{-1})	79 ($2.5 \times 10^{+0}$)
4	89 (4.0×10^{-2})	84 ($2.3 \times 10^{+0}$)	96 (2.6×10^{-1})	96 ($1.0 \times 10^{+1}$)
5	103 (1.7×10^{-1})	97 ($9.0 \times 10^{+0}$)	105 (9.0×10^{-1})	105 ($4.1 \times 10^{+1}$)
6	115 (6.5×10^{-1})	109 ($3.6 \times 10^{+1}$)	118 ($3.4 \times 10^{+0}$)	118 ($1.6 \times 10^{+2}$)

It is worth noting that modern BLAS libraries support multithreading and in order to investigate the impact of multithreading, MATLAB was initiated with a single thread[6] and the run times with a single thread compared with the default four threads for the optimised CG code. The runtime change moving from four threads to a single thread was insignificant apart from the 6th refinement where the runtime increased by 6.7%. However, the majority of this increase in runtime was in the solver, with an increase of 14.1%, and in the constitutive model, with an increase in runtime of 5.8%. The runtime change of the rest of the algorithm combined was negligible confirming that the majority of the stiffness matrix calculations are memory bandwidth bounded rather than compute bounded.

7. Conclusions

This paper presents two optimised codes capable of setting up and solving elasto-plastic problems in 2D, for CG and DG. Firstly, the optimised CG code was developed, building upon the work of Coombs et al. [12] who present a 70-line MATLAB script to solve for material and geometrically non-linear problems. In this code, the major performance bottleneck is observed from the large number of calls made to BLAS, resulting in the build up of overheads. The accumulation of these overheads predominantly occur in the constitutive model, where a high number of mathematical operations are performed on every Gauss point within the domain. By implementing blocking algorithms, initially proposed by Dabrowski et al. [2], the number of calls to BLAS are significantly reduced and, therefore, so are the overheads. This results in speed gains of $\approx$100 times in the constitutive model alone.

It was also found that the $[D]$ matrix is often non-symmetric for plastic Gauss points. This has a negative consequence on the formulation of the local stiffness matrices $[k^e]$, such that all the matrix terms must be calculated. To negate this, the code adopts a technique which forces the plastic $[D]$ matrices to be symmetric by averaging the off-diagonal terms. Consequently, only the upper triangle of the local stiffness matrices is required to be calculated, significantly reducing the number of calculations required. Once the global stiffness matrix $[K]$ is assembled, the complete matrix is determined by summing this matrix with the transpose of itself. Moreover, the symmetricity of global stiffness matrix results in improved solver performance. The penalty for using average-symmetric $[D]$ matrices comes through sub-optimal Newton–Raphson convergence at each loadstep, due to the small error in the global stiffness tangent. However the effect of this is outweighed by the speed gains achieved elsewhere in the code, resulting in a maximum overall simulation speed gain of $\times$1.94 when compared to the non-symmetric optimised code. This further breaks down to speed gains of $\times$1.58 and $\times$3.05 for the local stiffness formulation and solver time respectively.

Comparison of the optimised CG code to the equivalent non-optimised code by Coombs et al. [12] yields a maximum overall simulation speed gain of $\times$25.7 (and $\times$52 if the solver, setup and sparse time are disregarded). To ensure the correct implementation of the methods, the results were validated against a problem with an analytical solution.

[6] This can be done via the command line using `matlab -singleCompThread`.

The optimised DG code takes an identical structure to the CG code, with the inclusion of a surface integral loop. The surface loop used is similar to that developed by Bird et al. [4], who presents an optimised SIPG code for linear elasticity. However, due to difficulty in deriving the interior forces for the SIPG method, this paper adopts a IIPG approach. Furthermore, this loop requires two additional calls to the constitutive model to assess the stress state of surface Gauss points on the positive and negative elements. From this, appropriate $[D]$ matrices are returned. Unlike the CG code, the DG script does not force the $[D]$ matrices to be symmetric as the IIPG method will result in a non-symmetric stiffness matrix regardless. For this reason, the speed gains achieved are not as substantial as for CG, as all terms in the stiffness matrices must be calculated. Nonetheless a maximum speed gain $\times 10.1$ was observed in comparison to the equivalent non-optimised code.

It was also identified for both codes that with increasing refinement of the mesh, the solver time begins to dominate the overall simulation. This bottleneck has a greater disadvantage on the DG code due to the global stiffness matrix $[K]$ being non-symmetric alongside being much larger as a result of unshared degrees of freedom at the nodes. For example, for a DG analysis with $\approx 6 \times 10^5$ degrees of freedom, the solver time accounted for 76% of the overall simulation time. Although the current speed gains achieved are significant, the bottleneck now resides in the linear solver rather than the stiffness assembly as for standard MATLAB finite element implementations.

The presented computational efficiency codes could be used in solving more challenging problems. Recently Bird et al. presented a DG code for configurational force crack propagation for brittle materials [44]. The DG code presented in this paper could be used together with the configurational force method in [44] to simulate crack propagation in materials with plastic behaviour. The efficiency of the CG and DG codes could be further increased introducing adaptivity to the problem. An *hp* a posteriori error estimator for linear elasticity in the DG setting is presented in [45]. Due to the similarities between linear elasticity and the linear system arising from the Constitutive model in Algorithm 1, it plausible to foreseen the possibility to apply the a posteriori error estimator from [45] also in the elasto-plastic context.

In addition to the above points, it should be noted that the developed code is only applicable to the analysis of material where yielding is governed by a von Mises yield surface (for example metals or undrained soils). However, it is straightforward to replace the von Mises constitutive model with an alternative depending on the material analysed. The advantage of the adopted closed-form von Mises algorithm is that it allows straightforward vectorisation which may be problematic for other constitutive algorithms that require iterations to update the stress state and stiffness of the material. This is an interesting area for future research.

Acknowledgement

This work was supported by the Engineering and Physical Sciences Research Council, UK grant number [EP/M507854/1].

Appendix. Finite element formulation

A.1. IIPG approximation: linear elasticity

This paper adopts a reduction of the symmetric interior penalty discontinuous Galerkin (SIPG) method. This reduction refers to the removal of the symmetrisation term, due to difficulty formulating an expression for the internal forces for problems involving elasto-plasticity. This leaves the IIPG form [46], which proves far simpler for implementation of elasto-plasticity, as adopted by [47,48]. The IIPG approximation to the linear elastic problem can be written as: Find $\{u_h\} \in \mathcal{V}_p^{\mathrm{DG}}(\mathcal{T}_h)$ such that for all $\{v_h\} \in \mathcal{V}_p^{\mathrm{DG}}(\mathcal{T}_h)$ we have

$$((\{v_h\}^T\{f\})) = ((\{v_h\}^T[L]^T[D][L]\{u_h\})) - \langle\langle [\![\{v_h\}]\!]^T[D]\{\!\{[L]^T\{u_h\}\}\!\}\rangle\rangle + \alpha^r(u_h, v_h) \quad \text{in } \Omega. \tag{A.1}$$

Here $((\cdot))$ and $\langle\langle\cdot\rangle\rangle$ denote an integral over all element $E \in \mathcal{T}_h$, and all portions of element boundaries in the interior of the mesh, i.e. $\partial E \cap \mathcal{E}_h \neq \emptyset$ for all $E \in \mathcal{T}_h$, respectively. $\{\!\{\cdot\}\!\}$ and $[\![\cdot]\!]$ are the average and jump operators respectively, as defined in [4]. Additionally, $[D]$ is the Hookean stiffness matrix and α^r is a stabilising term as defined in [4]. The shape function matrix $[N]$ is used to interpolate to a given point within an element and can be used for the approximations $\{u_h\} = [N]\{u\}$ and $\{v_h\} = [N]\{v\}$, where $\{u\}$ is a vector containing the displacements of the point's parent element. The boundary conditions are imposed strongly, therefore no integral face terms have to be computed along the boundary of the domain. It should be noted that for CG only the area integral is considered.

Multiplying out the $[\![\cdot]\!]$ and $\{\!\{\cdot\}\!\}$ terms, substituting the shape function equations for $\{u_h\}$ and $\{v_h\}$ and subsequently eliminating the test function $\{v_h\}$, (A.1) can be written that

$$\begin{aligned} \underset{\forall E\in\mathcal{T}_h}{\mathbf{A}}(P) &= \underset{\forall E\in\mathcal{T}_h}{\mathbf{A}}(Q) - \underset{\forall F\in\mathcal{E}_h}{\mathbf{A}}\langle R_{1\ldots4} - S_{1\ldots4}\rangle \\ &\equiv [K]\{d\}, \end{aligned} \tag{A.2}$$

where $\mathcal{A}$ is the standard finite element assembly operator, $\{d\}$ contains the displacements for all nodes in the finite element mesh, $[K]$ is the global stiffness matrix, $(\cdot)$ denotes an integral over E, $\langle\cdot\rangle$ denotes an integral over $\partial E \cap \mathcal{E}_h$,

$$\begin{aligned} P &= [N]^T\{f\}, \\ Q &= [B]^T[D][B]\{u\}, \end{aligned} \tag{A.3}$$

with $[B] := [L][N]$ and where

$$\begin{aligned} R_1 &= [N^+]^T[n^+][D^+][B^+]\{u^+\}/2, \\ R_2 &= [N^+]^T[n^+][D^-][B^-]\{u^-\}/2, \\ R_3 &= -[N^-]^T[n^+][D^+][B^+]\{u^+\}/2, \\ R_4 &= -[N^-]^T[n^+][D^-][B^-]\{u^-\}/2, \\ S_1 &= \eta_\sigma[N^+]^T[N^+]\{u^+\}, \\ S_2 &= -\eta_\sigma[N^+]^T[N^-]\{u^+\}, \\ S_3 &= -\eta_\sigma[N^-]^T[N^+]\{u^-\} \qquad \text{and} \\ S_4 &= \eta_\sigma[N^-]^T[N^-]\{u^-\}. \end{aligned} \tag{A.4}$$

Here, the $^+$ and $^-$ denote element properties corresponding to E^+ and E^- respectively, which are defined as neighbouring elements, connected by a face F. The penalty term $\eta_\sigma = \beta_F E p^2 h_F^{-1}$, where E is the elastic Young's modulus of the material, β_F is the stabilisation penalty parameter which may vary from face to face [49] (set equal to 10 in this paper), p is the polynomial order of the element and h_F is the face length, and the positive outward facing normal matrix $[n]$ is as defined by Bird et al. [4] for SIPG. It should be noted that for CG only terms (P) and (Q) need to be evaluated.

Numerical integration is carried out using Gauss–Legendre quadrature, which requires n_{Gp} sampling Gauss points located throughout the element. For example Q is approximated using

$$(Q) \approx \left(\sum_{i=1}^{n_{Gp}} [B]^T[D][B]\det\Big([J]\Big)w_i\right)\{u\}, \tag{A.5}$$

where $[J]$ is the Jacobian mapping between the global and local coordinates and w_i is the weight associated with the Gauss point.

A.2. Internal force vector: elasto-plasticity

The element internal force vector is simply obtained by multiplying the element stiffness matrix by the displacement at the nodes. For CG finite element analysis the internal force of an element can be expressed as

$$\{f^e\} = \int_E [B]^T[D][B]\{u\}dx = \int_E [B]^T\{\sigma\}dx. \tag{A.6}$$

Note that in the case of elasto-plastic material deformation, the Cauchy stress is not linearly dependent on displacement.

For DG, the terms $R_{1\ldots4}$ and $S_{1\ldots4}$ must also be considered. The terms $R_{1\ldots4}$ can be evaluated in similar manner to (Q), taking R_1 as an example, it can be written that

$$\begin{aligned} \{f^e\}_{R_1} &= \frac{1}{2}\int_{\partial E\cap\mathcal{E}_h} [N^+]^T[n^+][D^+][B^+]\{u^+\}\,ds \\ &= \frac{1}{2}\int_{\partial E\cap\mathcal{E}_h} [N^+]^T\underbrace{[n^+]\{\sigma^+\}}_{\{t^+\}}\,ds, \end{aligned} \tag{A.7}$$

where $\{f^e\}_{R_1}$ is the R_1 component of the element internal force calculation and $\{\sigma^+\}$ and $\{t^+\}$ are the stress vector and traction vector corresponding to the element E^+, respectively.

For terms $S_{1\ldots4}$, the element internal force components are as stated in (A.4). This is because the penalty terms do not contain a non-linear component, as they simply penalise the displacement jump terms between adjacent elements. In this paper the non-linear equilibrium equation (3) is solved using a fully implicit Newton–Raphson algorithm. This requires linearisation of the equilibrium equation (3) in order to obtain the global stiffness matrix, $[K]$. The form of the global stiffness matrix for elasto-plasticity is the same as that given in (A.2) but the material stiffness matrix, $[D]$, is now non-linear in the nodal displacements. This is explained in more detail in Section 3.4.

References

[1] C. Moler, MATLAB Incorporates LAPACK, 2000, http://uk.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html?refresh=true. (Accessed September 2018).
[2] M. Dabrowski, M. Krotkiewski, D.W. Schmid, MILAMIN: MATLAB-based finite element method solver for large problem, Geochem. Geophys. Geosystems 9 (2008).
[3] T. Davis, Suite Sparse, 2013, http://faculty.cse.tamu.edu/davis/research.html. (Accessed September 2018).
[4] R. Bird, W. Coombs, S. Giani, Fast native-MATLAB stiffness assembly for SIPG linear elasticity, Comput. Math. Appl. 74 (2017) 3209–3230.
[5] M. Krotkiewski, M. Dabrowski, Parallel symmetric sparse matrix–vector product on scalar multi-core CPUs, Parallel Comput. 36 (2010) 181–198.
[6] T. Rahman, J. Valdman, Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements, Appl. Math. Comput. 219 (2013) 7151–7158, ESCO 2010 Conference in Pilsen, June 21–25, 2010.
[7] I. Anjam, J. Valdman, Fast MATLAB assembly of FEM matrices in 2D and 3D: Edge elements, Appl. Math. Comput. 267 (2015) 252–263, The Fourth European Seminar on Computing (ESCO 2014).
[8] E. Andreassen, A. Clausen, M. Schevenels, B.S. Lazarov, O. Sigmund, Efficient topology optimization in MATLAB using 88 lines of code, Struct. Multidiscip. Optim. 43 (2011) 1–16.
[9] F. Cuvelier, C. Japhet, G. Scarella, An efficient way to assemble finite element matrices in vector languages, BIT 56 (2015) 1–32, Cited By 2.
[10] M. Adamuszek, M. Dabrowski, D.W. Schmid, Folder: A numerical tool to simulate the development of structures in layered media, J. Struct. Geol. 84 (2016) 85–101.
[11] M. Ainsworth, R. Rankin, Constant free error bounds for nonuniform order discontinuous Galerkin finite-element approximation on locally refined meshes with hanging nodes, IMA J. Numer. Anal. 31 (2011) 254–280.
[12] W. Coombs, R. Crouch, C. Augarde, 70-line 3D finite deformation elasto-plastic finite element code, in: Proc. Numerical Methods in Geotechnical Engineering, NUMGE, Trondheim, Norway, 2010, pp. 151–156.
[13] R. von Mises, Mechanik der festen Körper im plastisch-deformablen Zustand, Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Math.-Physikalische Klass 1913 (1913) 582–592.
[14] M. Safaei, M.G. Lee, W.D. Waele, Evaluation of stress integration algorithms for elastic–plastic constitutive models based on associated and non-associated flow rules, Comput. Methods Appl. Mech. Engrg. 295 (2015) 414–445.
[15] R. Krieg, D. Krieg, Accuracies of numerical solution methods for the elastic-perfectly plastic model, J. Press. Vessel Technol. 99 (1977) 510–515.
[16] A. Anandarajah, Computational Methods in Elasticity and Plasticity: Solids and Porus Media, Springer, 2010.
[17] A.A. Ilyushin, Some problems in the theory of plastic deformation, Prikl. Mat. Mekh. (1943) 245–272.
[18] A. Mendelson, Plasticity: Theory and Application, The Macmillan Co., NY, 1968.
[19] G. Nayak, O. Zienkewicz, Elasto-plastic stress analysis. a generalization for various constitutive relations including strain softening, Internat. J. Numer. Methods Engrg. 5 (1972) 113–135.
[20] M. Vrh, M. Halilovič, B. Štok, Improved explicit integration in plasticity, Internat. J. Numer. Methods Engrg. 81 (2010) 910–938.
[21] M. Wilkins, Calculation of elastic–plastic flow, in: S. Fernback, M. Rotenberg (Eds.), in: Methods of Computational Physics, vol. 3, Academic Press, New York, 1964.
[22] M. Ortiz, E. Popov, Accuracy and stability of integration algorithms for elastoplastic constitutive relations, Internat. J. Numer. Methods Engrg. 21 (1985) 1561–1576.
[23] J.C. Simo, M. Ortiz, A unified approach to finite deformation elastoplastic analysis based on the use of hyperelastic constitutive equations, Comput. Methods Appl. Mech. Engrg. 49 (1985) 221–245.
[24] M. Kojić, The governing parameter method for implicit integration of viscoplastic constitutive relations for isotropic and orthotropic metals, Comput. Mech. 19 (1996) 49–57.
[25] M. Kojić, Stress integration procedures for inelastic material models within the finite element method, Appl. Mech. Rev. 55 (2002) 389–414.
[26] M. Kojić, K. Bathe, Inelastic Analysis of Solids and Structures, Springer Berlin Heidelberg New York, 2005.
[27] W.M. Coombs, R.S. Crouch, C.E. Augarde, Reuleaux plasticity: analytical backward Euler stress integration and consistent tangent, Comput. Methods Appl. Mech. Engrg. 199 (2010) 1733–1743.
[28] W.M. Coombs, R.S. Crouch, Non-associated Reuleaux plasticity: analytical stress integration and consistent tangent for finite deformation mechanics, Comput. Methods Appl. Mech. Engrg. 200 (2011) 1021–1037.
[29] R. Crouch, H. Askes, T. Li, Analytical CPP in energy-mapped stress space: application to a modified Drucker–Prager yield surface, Comput. Methods Appl. Mech. Engrg. 198 (2009) 853–859.
[30] W.M. Coombs, O.A. Petit, Y.G. Motlagh, NURBS plasticity: yield surface representation and implicit stress integration for isotropic inelasticity, Comput. Methods Appl. Mech. Engrg. 304 (2016) 342–358.
[31] W.M. Coombs, Y.G. Motlagh, NURBS plasticity: yield surface evolution and implicit stress integration for isotropic hardening, Comput. Methods Appl. Mech. Engrg. 324 (2017) 204–220.
[32] W.M. Coombs, Y.G. Motlagh, NURBS plasticity: non-associated plastic flow, Comput. Methods Appl. Mech. Engrg. 336 (2018) 419–443.
[33] M. Crisfield, Non-linear Finite Element Analysis of Solids and Structures, in: Essentials, vol. 1, John Wiley & Sons Ltd, 1991.
[34] J.C. Simo, T.J.R. Hughes, Computational Inelasticity, Springer, New York, 1998.
[35] Z. Wei, D. Perić, D.R.J. Owen, Consistent linearization for the exact stress update of Prandtl–Reuss non-hardening elastoplastic models, Internat. J. Numer. Methods Engrg. 39 (1996) 1219–1235.
[36] L. Szabó, A semi-analytical integration method for J2 flow theory of plasticity with linear isotropic hardening, Comput. Methods Appl. Mech. Engrg. 198 (2009) 2151–2166.
[37] A. Kossa, L. Szabó, Exact integration of the von Mises elastoplasticity model with combined linear isotropic-kinematic hardening, Int. J. Plast. 25 (2009) 1083–1106.
[38] A. Kossa, L. Szabó, Numerical implementation of a novel accurate stress integration scheme of the von Mises elastoplasticity model with combined linear hardening, Finite Elem. Anal. Des. 46 (2010) 391–400.
[39] B. Loret, J. Pré, Accurate numerical solutions for Drucker–Prager elastic plastic models, Comput. Methods Appl. Mech. Engrg. 54 (1986) 259–277.
[40] J. Nagtegaal, On the implementation of inelastic constitutive equations with special reference to large deformation problems, Comput. Methods Appl. Mech. Engrg. 33 (1982) 469–484.
[41] J.C. Simo, R.L. Taylor, Consistent tangent operators for rate-independent elastoplasticity, Comput. Methods Appl. Mech. Engrg. 48 (1985) 101–118.
[42] T. Duretz, A. Souche, R. d. Borst, L.L. Pourhiet, The benefits of using a consistent tangent operator for viscoelastoplastic computations in geodynamics, Geochem. Geophys. Geosystems 19 (2018) 4904–4924.
[43] J. Nagtegaal, D. Parks, J. Rice, On numerically accurate finite element solutions in the fully plastic range, Comput. Methods Appl. Mech. Engrg. 4 (1974) 153–177.
[44] R. Bird, W. Coombs, S. Giani, A quasi-static discontinuous Galerkin configurational force crack propagation method for brittle materials, Internat. J. Numer. Methods Engrg. 113 (2018) 1061–1080.

[45] R.E. Bird, W.M. Coombs, S. Giani, A posteriori discontinuous Galerkin error estimator for linear elasticity, Appl. Math. Comput. 344–345 (2019) 78–96.
[46] C. Dawson, S. Sun, M.F. Wheeler, Compatible algorithms for coupled flow and transport, Comput. Methods Appl. Mech. Engrg. 193 (2004) 2565–2580.
[47] R. Liu, M. Wheeler, C. Dawson, R. Dean, A fast convergent rate preserving discontinuous Galerkin framework for rate-independent plasticity problems, Comput. Methods Appl. Mech. Engrg. 199 (2010) 3213–3226.
[48] R. Liu, M.F. Wheeler, I. Yotov, On the spatial formulation of discontinuous Galerkin methods for finite elastoplasticity, Comput. Methods Appl. Mech. Engrg. 253 (2013) 219–236.
[49] P. Hansbo, M.G. Larson, Energy norm a posteriori error estimates for discontinuous Galerkin approximations of the linear elasticity problem, Comput. Methods Appl. Mech. Engrg. 200 (2011) 3026–3030.