



A fast and efficient MATLAB-based MPM solver: fMPMM-solver v1.1

Emmanuel Wyser¹, Yury Alkhimenkov^{1,2,3}, Michel Jaboyedoff^{1,2}, and Yury Y. Podladchikov^{1,2,3}

¹Institute of Earth Sciences, University of Lausanne, 1015 Lausanne, Switzerland

²Swiss Geocomputing Centre, University of Lausanne, 1015 Lausanne, Switzerland

³Faculty of Mechanics and Mathematics, Lomonosov Moscow State University, Moscow, 119899, Russia

Correspondence: Emmanuel Wyser (manuwyser@gmail.com)

Received: 3 June 2020 – Discussion started: 20 July 2020

Revised: 12 October 2020 – Accepted: 27 October 2020 – Published: 10 December 2020

Abstract. We present an efficient MATLAB-based implementation of the material point method (MPM) and its most recent variants. MPM has gained popularity over the last decade, especially for problems in solid mechanics in which large deformations are involved, such as cantilever beam problems, granular collapses and even large-scale snow avalanches. Although its numerical accuracy is lower than that of the widely accepted finite element method (FEM), MPM has proven useful for overcoming some of the limitations of FEM, such as excessive mesh distortions. We demonstrate that MATLAB is an efficient high-level language for MPM implementations that solve elasto-dynamic and elasto-plastic problems. We accelerate the MATLAB-based implementation of the MPM method by using the numerical techniques recently developed for FEM optimization in MATLAB. These techniques include vectorization, the use of native MATLAB functions and the maintenance of optimal RAM-to-cache communication, among others. We validate our in-house code with classical MPM benchmarks including (i) the elastic collapse of a column under its own weight; (ii) the elastic cantilever beam problem; and (iii) existing experimental and numerical results, i.e. granular collapses and slumping mechanics respectively. We report an improvement in performance by a factor of 28 for a vectorized code compared with a classical iterative version. The computational performance of the solver is at least 2.8 times greater than those of previously reported MPM implementations in Julia under a similar computational architecture.

1 Introduction

The material point method (MPM), developed in the 1990s (Sulsky et al., 1994), is an extension of a particle-in-cell (PIC) method to solve solid mechanics problems involving massive deformations. It is an alternative to Lagrangian approaches (updated Lagrangian finite element method) that is well suited to problems with large deformations involved in geomechanics, granular mechanics or even snow avalanche mechanics. Vardon et al. (2017) and Wang et al. (2016c) investigated elasto-plastic problems of the strain localization of slumping processes relying on an explicit or implicit MPM formulation. Similarly, Bandara et al. (2016), Bandara and Soga (2015), and Abe et al. (2014) proposed a poro-elasto-plastic MPM formulation to study levee failures induced by pore pressure increases. Additionally, Baumgarten and Kamrin (2019), Dunatunga and Kamrin (2017), Dunatunga and Kamrin (2015), and Więckowski (2004) proposed a general numerical framework of granular mechanics, i.e. silo discharge or granular collapses. More recently, Gaume et al. (2019, 2018) proposed a unified numerical model in the finite deformation framework to study the whole process (i.e. from failure to propagation) of slab avalanche releases.

The core idea of MPM is to discretize a continuum with material points carrying state variables (e.g. mass, stress and velocity). The latter are mapped (accumulated) to the nodes of a regular or irregular background finite element (FE) mesh, on which an Eulerian solution to the momentum balance equation is explicitly advanced forward in time. Nodal solutions are then mapped back to the material points, and the mesh can be discarded. The mapping from material points to nodes is ensured using the standard FE hat

function that spans over an entire element (Bardenhagen and Kober, 2004). This avoids a common flaw of the finite element method (FEM), which is an excessive mesh distortion. We will refer to this first variant as the standard material point method (sMPM).

MATLAB® allows a rapid code prototyping, although it is at the expense of significantly lower computational performance than compiled language. An efficient MATLAB implementation of FEM called MILAMIN (Million a Minute) was proposed by Dabrowski et al. (2008) that was capable of solving two-dimensional linear problems with 1 million unknowns in 1 min on a modern computer with a reasonable architecture. The efficiency of the algorithm lies in the combined use of vectorized calculations with a technique called blocking. MATLAB uses the Linear Algebra PACKage (LAPACK), written in Fortran, to perform mathematical operations by calling basic linear algebra subprograms (BLAS, Moler, 2000). The latter results in an overhead each time a BLAS call is made. Hence, mathematical operations over a large number of small matrices should be avoided, and operations on fewer and larger matrices should be preferred. This is a typical bottleneck in FEM when local stiffness matrices are assembled during the integration point loop within the global stiffness matrix. Dabrowski et al. (2008) proposed an algorithm in which a loop reordering is combined with operations on blocks of elements to address this bottleneck. However, data required for a calculation within a block should entirely reside in the CPU's cache; otherwise, additional time is spent on the RAM-to-cache communication, and the performance decreases. Therefore, an optimal block size exists, and it is solely defined by the CPU architecture. This technique of vectorization combined with blocking significantly increases the performance.

More recently, Bird et al. (2017) extended the vectorized and blocked algorithm presented by Dabrowski et al. (2008) to the calculation of the global stiffness matrix for discontinuous Galerkin FEM considering linear elastic problems using only native MATLAB functions. Indeed, the optimization strategy chosen by Dabrowski et al. (2008) also relied on non-native MATLAB functions, e.g. `sparse2` of the SuiteSparse package (Davis, 2013). In particular, Bird et al. (2017) showed the importance of storing vectors in a column-major form during calculation. Mathematical operations are performed in MATLAB by calling LAPACK, written in Fortran, in which arrays are stored in column-major order form. Hence, element-wise multiplication of arrays in column-major form is significantly faster; thus, vectors in column-major form are recommended whenever possible. Bird et al. (2017) concluded that vectorization alone results in a performance increase of between 13.7 and 23 times, whereas blocking only improved vectorization by an additional 1.8 times. O'Sullivan et al. (2019) recently extended the works of Bird et al. (2017) and Dabrowski et al. (2008) to optimized elasto-plastic codes for continuous Galerkin (CG) or discontinuous Galerkin (DG) methods. In particular, they proposed

an efficient native MATLAB function, `accumarray()`, to efficiently assemble the internal force vector. This kind of function constructs an array by accumulation. More generally, O'Sullivan et al. (2019) reported a performance gain of 25.7 times when using an optimized CG code instead of an equivalent non-optimized code.

As MPM and FEM are similar in their structure, we aim to improve the performance of MATLAB up to the level reported by Sinaie et al. (2017) using the Julia language environment. In principal, Julia is significantly faster than MATLAB for an MPM implementation. We combine the most recent and accurate versions of MPM: the explicit generalized interpolation material point method (GIMPM, Bardenhagen and Kober, 2004) and the explicit convected particle domain interpolation with second-order quadrilateral domains (CPDI2q and CPDI; Sadeghirad et al., 2013, 2011) variants with some of the numerical techniques developed during the last decade of FEM optimization in MATLAB. These techniques include the use of `accumarray()`, optimal RAM-to-cache communication, minimum BLAS calls and the use of native MATLAB functions. We did not consider the blocking technique initially proposed by Dabrowski et al. (2008), as an explicit formulation in MPM excludes the global stiffness matrix assembly procedure. The performance gain mainly comes from the vectorization of the algorithm, whereas blocking has a less significant impact over the performance gain, as stated by Bird et al. (2017). The vectorization of MATLAB functions is also crucial for a straight transpose of the solver to a more efficient language, such as the C-CUDA language, which allows the parallel execution of computational kernels of graphics processing units (GPUs).

In this contribution, we present an implementation of an efficiently vectorized explicit MPM solver, `fmpmm-solver` (v1.1 is available for download from Bitbucket at <https://bitbucket.org/ewyser/fmpmm-solver/src/master/>, last access: 6 October 2020), taking advantage of the vectorization capabilities of MATLAB®. We extensively use native functions of MATLAB®, such as `repmat()`, `reshape()`, `sum()` or `accumarray()`. We validate our in-house code with classical MPM benchmarks including (i) the elastic collapse of a column under its own weight; (ii) the elastic cantilever beam problem; and (iii) existing experimental and numerical results, i.e. granular collapses and slumping mechanics respectively. We demonstrate the computational efficiency of a vectorized implementation over an iterative one for an elasto-plastic collapse of a column. We compare the performance of the Julia and MATLAB language environments for the collision of two elastic discs problem.

2 Overview of the material point method (MPM)

2.1 A material point method implementation

The material point method (MPM), originally proposed by Sulsky et al. (1995, 1994) in an explicit formulation, is an extension of the particle-in-cell (PIC) method. The key idea is to solve the weak form of the momentum balance equation on an FE mesh while state variables (e.g. stress, velocity or mass) are stored at Lagrangian points discretizing the continuum, i.e. the material points, which can move according to the deformation of the grid (Dunatunga and Kamrin, 2017). MPM could be regarded as a finite element solver in which integration points (material points) are allowed to move (Guilkey and Weiss, 2003) and are, thus, not always located at the Gauss–Legendre location within an element, resulting in higher quadrature errors and poorer integration estimates, especially when using low-order basis functions (Steffen et al., 2008a, b).

A typical calculation cycle (see Fig. 1) consists of the three following steps (Wang et al., 2016a):

1. a mapping phase, during which properties of the material point (mass, momentum or stress) are mapped to the nodes;
2. an updated Lagrangian FEM (UL-FEM) phase, during which the momentum equations are solved on the nodes of the background mesh, and the solution is explicitly advanced forward in time;
3. a convection phase, during which (i) the nodal solutions are interpolated back to the material points, and (ii) the properties of the material point are updated.

Since the 1990s, several variants have been introduced to resolve a number of numerical issues. The generalized interpolation material point method (GIMPM) was first presented by Bardenhagen and Kober (2004). They proposed a generalization of the basis and gradient functions that were convoluted with a characteristic domain function of the material point. A major flaw in sMPM is the lack of continuity of the gradient basis function, resulting in spurious oscillations of internal forces as soon as a material point crosses an element boundary while entering into its neighbour. This is referred to as cell-crossing instabilities due to the C_0 continuity of the gradient basis functions used in sMPM. This issue is minimized by the GIMPM variant (Acosta et al., 2020).

GIMPM is categorized as a domain-based material point method, unlike the later development of the B-spline material point method (BSMPM, e.g. de Koster et al., 2020; Gan et al., 2018; Gaume et al., 2018; Stomakhin et al., 2013) which cures cell-crossing instabilities using B-spline functions as basis functions. Whereas only nodes belonging to an element contribute to a given material point in sMPM, GIMPM requires an extended nodal connectivity, i.e. the nodes of the

element enclosing the material point and the nodes belonging to the adjacent elements (see Fig. 2). More recently, the convected particle domain interpolation (CPDI and its most recent development CPDI2q) has been proposed by Sadeghirad et al. (2013, 2011).

We choose the explicit GIMPM variant with the modified update stress last scheme (MUSL; see Nairn, 2003, and Bardenhagen et al., 2000, for a detailed discussion), i.e. the stress of the material point is updated after the nodal solutions are obtained. The updated momentum of the material point is then mapped back to the nodes a second time in order to obtain an updated nodal velocity, which is further used to calculate derivative terms such as strains or the deformation gradient of the material point. The explicit formulation also implies the well-known restriction on the time step, which is limited by the Courant–Friedrichs–Lewy (CFL) condition to ensure numerical stability.

Additionally, we implemented a CPDI/CPDI2q version (in an explicit and quasi-static implicit formulation) of the solver. However, in this paper, we do not present the theoretical background of the CPDI variant nor the implicit implementation of an MPM-based solver. Therefore, interested readers are referred to the original contributions of Sadeghirad et al. (2013, 2011) for the background of the CPDI variant and to the contributions of Acosta et al. (2020), Charlton et al. (2017), Iaconeta et al. (2017), Beuth et al. (2008), and Guilkey and Weiss (2003) for an implicit implementation of an MPM-based solver. Regarding the quasi-static implicit implementation, we strongly adapted our vectorization strategy to some aspects of the numerical implementation proposed by Coombs and Augarde (2020) in the MATLAB code AMPLE v1.0. However, we did not consider blocking, because our main concern regarding performance is on the explicit implementation.

2.2 Domain-based material point method variants

Domain-based material point method variants could be treated as two distinct groups:

1. the material point's domain is a square for which the deformation is always aligned with the mesh axis, i.e. a non-deforming domain uGIMPM (Bardenhagen and Kober, 2004), or it is a deforming domain cpGIMPM, (Wallstedt and Guilkey, 2008), where the latter is usually related to a measure of the deformation, e.g. the determinant of the deformation gradient;
2. the material point's domain is either a deforming parallelogram that has its dimensions specified by two vectors, i.e. CPDI (Sadeghirad et al., 2011), or it is a deforming quadrilateral solely defined by its corners, i.e. CPDI2q (Sadeghirad et al., 2013). However, the deformation is not necessarily aligned with the mesh anymore.

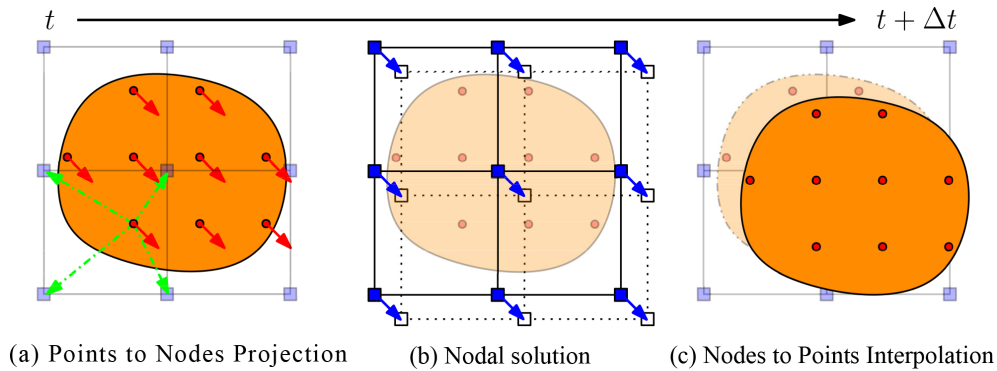


Figure 1. Typical calculation cycle of an MPM solver for a homogeneous velocity field, inspired by Dunatunga and Kamrin (2017). (a) The continuum (orange) is discretized into a set of Lagrangian material points (red dots), at which state variables or properties (e.g. mass, stress and velocity) are defined. The latter are mapped to an Eulerian finite element mesh made of nodes (blue square). (b) Momentum equations are solved at the nodes, and the solution is explicitly advanced forward in time. (c) The nodal solutions are interpolated back to the material points, and their properties are updated.

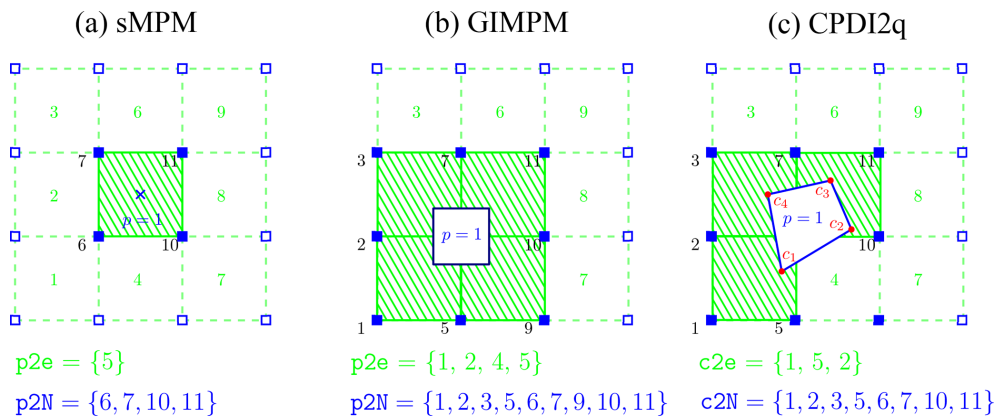


Figure 2. Nodal connectivities of the (a) standard MPM, (b) GIMPM and (c) CPDI2q variants. The material point’s location is marked by the blue cross. Note that the particle domain does not exist for sMPM (or BSMPPM), unlike GIMPM or CPDI2q (the blue square enclosing the material point). Nodes associated with the material point are denoted by filled blue squares, and the element number appears in green in the centre of the element. For sMPM and GIMPM, the connectivity array between the material point and the element is $p2e$, and the array between the material point and its associated nodes is $p2N$. For CPDI2q, the connectivity array between the corners (filled red circles) of the quadrilateral domain of the material point and the element is $c2e$, and the array between the corners and their associated nodes is $c2N$.

We first focus on the different domain-updating methods for GIMPM. Four domain-updating methods exists: (i) the domain is not updated, (ii) the deformation of the domain is proportional to the determinant of the deformation gradient $\det(F_{ij})$ (Bardenhagen and Kober, 2004), (iii) the domain lengths l_p are updated according to the principal component of the deformation gradient F_{ii} (Sadeghirad et al., 2011) or (iv) the domain lengths l_p are updated with the principal component of the stretching part of the deformation gradient U_{ii} (Charlton et al., 2017). Coombs et al. (2020) highlighted the suitability of generalized interpolation domain-updating methods according to distinct deformation modes. Four different deformation modes were considered by Coombs et al. (2020): simple stretching, hydrostatic compression or exten-

sion, simple shear and pure rotation. Coombs et al. (2020) concluded the following:

- Not updating the domain is not suitable for simple stretching and hydrostatic compression or extension.
- A domain update based on $\det(F_{ij})$ will results in an artificial contraction or expansion of the domain for simple stretching.
- The domain will vanish with increasing rotation when using F_{ii} .
- The domain volume will change under isochoric deformation when using U_{ii} .

Consequently, Coombs et al. (2020) proposed a hybrid domain update inspired by CPDI2q approaches: the corners of

the material point domain are updated according to the nodal deformation, but the midpoints of the domain limits are used to update the domain lengths l_p to maintain a rectangular domain. Even though Coombs et al. (2020) reported an excellent numerical stability, the drawback is to compute specific basis functions between nodes and material point's corners, which has an additional computational cost. Hence, we did not selected this approach in this contribution.

Regarding the recent CPDI/CPDI2q version, Wang et al. (2019) investigated the numerical stability under stretching, shear and torsional deformation modes. CPDI2q was found to be erroneous in some case, especially when the torsion mode was involved, due to distortion of the domain. In contrast, CPDI and even sMPM showed better performance with respect to modelling torsional deformations. Although CPDI2q can exactly represent the deformed domain (Sadeghirad et al., 2013), care must be taken when dealing with very large distortion, especially when the material has yielded, which is common in geotechnical engineering (Wang et al., 2019).

Consequently, the domain-based method and the domain-updating method should be carefully chosen according to the deformation mode expected for a given case. The domain-updating method will be clearly mentioned for each case throughout the paper.

3 MATLAB-based MPM implementation

3.1 Rate formulation and elasto-plasticity

The large deformation framework in a linear elastic continuum requires an appropriate stress–strain formulation. One approach is based on the finite deformation framework, which relies on a linear relationship between elastic logarithmic strains and Kirchhoff stresses (Coombs et al., 2020; Gaume et al., 2018; Charlton et al., 2017). In this study, we adopt another approach, namely a rate-dependent formulation using the Jaumann stress rate (e.g. Huang et al., 2015; Bandara et al., 2016; Wang et al., 2016c, b). This formulation provides an objective (invariant by rotation or frame-indifferent) stress rate measure (de Souza Neto et al., 2011) and is simple to implement. The Jaumann rate of the Cauchy stress is defined as

$$\frac{D\sigma_{ij}}{Dt} = \frac{1}{2} C_{ijkl} \left(\frac{\partial v_l}{\partial x_k} + \frac{\partial v_k}{\partial x_l} \right), \tag{1}$$

where C_{ijkl} is the fourth rank tangent stiffness tensor, and v_k is the velocity. Thus, the Jaumann stress derivative can be written as

$$\frac{D\sigma_{ij}}{Dt} = \frac{D\sigma_{ij}}{Dt} - \sigma_{ik}\omega_{jk} - \sigma_{jk}\omega_{ik}, \tag{2}$$

where $\omega_{ij} = (\partial_i v_j - \partial_j v_i)/2$ is the vorticity tensor, and $D\sigma_{ij}/Dt$ denotes the material derivative

$$\frac{D\sigma_{ij}}{Dt} = \frac{\partial \sigma_{ij}}{\partial t} + v_k \frac{\partial \sigma_{ij}}{\partial x_k}. \tag{3}$$

Plastic deformation is modelled with a pressure-dependent Mohr–Coulomb law with non-associated plastic flow, i.e. both the dilatancy angle ψ and the volumetric plastic strain ϵ_v^p are null (Vermeer and De Borst, 1984). We have adopted the approach of Simpson (2017) for a two-dimensional linear elastic, perfectly plastic (elasto-plasticity) continuum because of its simplicity and its ease of implementation. The yield function is defined as

$$f = \tau + \sigma \sin \phi - c \cos \phi, \tag{4}$$

where c is the cohesion, ϕ the angle of internal friction,

$$\sigma = (\sigma_{xx} + \sigma_{yy})/2 \tag{5}$$

and

$$\tau = \sqrt{(\sigma_{xx} - \sigma_{yy})^2/4 + \sigma_{xy}^2}. \tag{6}$$

The elastic state is defined when $f < 0$; however, when $f > 0$, plastic state is declared and stresses must be corrected (or scaled) to satisfy the condition $f = 0$, as $f > 0$ is an inadmissible state. Simpson (2017) proposed the following simple algorithm to return stresses to the yield surface:

$$\sigma_{xx}^* = \sigma + (\sigma_{xx} - \sigma_{yy})\beta/2, \tag{7}$$

$$\sigma_{yy}^* = \sigma - (\sigma_{xx} - \sigma_{yy})\beta/2, \tag{8}$$

$$\sigma_{xy}^* = \sigma_{xy}\beta, \tag{9}$$

where $\beta = (|c \cos \phi - \sigma \sin \phi|)/\tau$, and σ_{xx}^* , σ_{yy}^* and σ_{xy}^* are the corrected stresses, i.e. $f = 0$.

A similar approach is used to return stresses when considering a non-associated Drucker–Prager plasticity (see Huang et al., 2015, for a detailed description of the procedure). In addition, their approach also allows one to model associated plastic flows, i.e. $\psi > 0$ and $\epsilon_v^p \neq 0$.

3.2 Structure of the MPM solver

The solver procedure is shown in Fig. 3. In the `main.m` script, the respective `meSetup.m` and `mpSetup.m` functions define the geometry and related quantities such as the nodal connectivity (or element topology) array, e.g. the `e2N` array. The latter stores the nodes associated with a given element. As such, a material point p located in an element e can immediately identify which nodes n it is associated with.

After initialization, a while loop solves the elasto-dynamic (or elasto-plastic) problem until a time criterion T is reached. This time criterion could be restricted to the time needed for the system to reach an equilibrium or if the global kinetic energy of the system has reached a threshold.

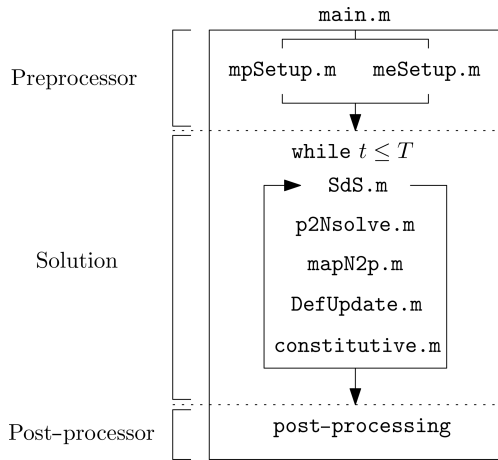


Figure 3. Workflow of the explicit GIMPM solver and the calls to functions within a calculation cycle. The role of each function is described in the text.

At the beginning of each cycle, a connectivity array $p2e$ between the material points and their respective element (a material point can only reside in a single element) is constructed. As (i) the nodes associated with the elements and (ii) the elements enclosing the material points are known, it is possible to obtain the connectivity array $p2N$ between the material points and their associated nodes, e.g. $p2N=e2N(p2e, :)$ in a MATLAB syntax (see Fig. 2 for an example of these connectivity arrays). This array is of dimension (n_p, n_n) . Here, n_p is the total number of material points; n_n is the total number of nodes associated with an element (16 in two-dimensional problems); and $n_{i,j}$ is the node number, where i corresponds to the material point and j corresponds to its j -th associated nodes, which results in the following:

$$p2N = \begin{pmatrix} n_{1,1} & \cdots & n_{1,n_n} \\ \vdots & \ddots & \vdots \\ n_{n_p,1} & \cdots & n_{n_p,n_n} \end{pmatrix}. \quad (10)$$

The following five functions are called successively during one calculation cycle:

1. `SdS.m` calculates the basis functions, the derivatives and assembles the strain-displacement matrix for each material point.
2. `p2Nsolve.m` projects the quantities of the material point (e.g. mass and momentum) to the associated nodes, solves the equations of motion and sets boundary conditions.
3. `mapN2p.m` interpolates nodal solutions (acceleration and velocity) to the material points with a double-mapping procedure (see Zhang et al., 2016, or Nairn, 2003, for a clear discussion of update stress first, update stress last and MUSL algorithms).

4. `DefUpdate.m` updates incremental strains and deformation-related quantities (e.g. the volume of the material point or the domain half-length) at the level of the material point based on the remapping of the updated material point momentum.

5. `constitutive.m` calls two functions to solve for the constitutive elasto-plastic relation, namely

- (a) `elastic.m`, which predicts an incremental objective stress assuming a purely elastic step, further corrected by
- (b) `plastic.m`, which corrects the trial stress by a plastic correction if the material has yielded.

When a time criterion is met, the calculation cycle stops and further post-processing tasks (visualization, data exportation) can be performed.

The numerical simulations are conducted using MATLAB® R2018a within a Windows 7 64-bit environment on an Intel Core i7-4790 (fourth-generation CPU with four physical cores of base frequency at 3.60 GHz up to a maximum turbo frequency of 4.00 GHz) with 4×256 kB L2 cache and 16 GB DDR3 RAM (clock speed 800 MHz).

3.3 Vectorization

3.3.1 Basis functions and derivatives

The GIMPM basis function (Coombs et al., 2018; Steffen et al., 2008a; Bardenhagen and Kober, 2004) results from the convolution of a characteristic particle function χ_p (i.e. the material point spatial extent or domain) with the standard basis function $N_n(x)$ of the mesh, which results in

$$S_n(x_p) = \begin{cases} 1 - (4x^2 + l_p^2)/(4hl_p) & \text{if } |x| < l_p/2 \\ 1 - |x|/h & \text{if } l_p/2 \leq |x| < h - l_p/2 \\ (h + l_p/2 - |x|)^2/(2hl_p) & \text{if } h - l_p/2 \leq |x| < h + l_p/2 \\ 0 & \text{otherwise} \end{cases}. \quad (11)$$

Here, l_p is the length of the material point domain; h is the mesh spacing; and $x = x_p - x_n$, where x_p is the coordinate of a material point and x_n the coordinate of its associated node n . The basis function of a node n with its material point p is constructed for a two-dimensional model, as follows:

$$S_n(\mathbf{x}_p) = S_n(x_p)S_n(y_p). \quad (12)$$

Here, the derivative is defined as

$$\nabla S_n(\mathbf{x}_p) = (\partial_x S_n(x_p)S_n(y_p), S_n(x_p)\partial_y S_n(y_p)). \quad (13)$$

Similar to the FEM, the strain-displacement matrix \mathbf{B} consists of the derivatives of the basis function and is assigned to each material point, which results in the following:

$$\mathbf{B}(\mathbf{x}_p) = \begin{pmatrix} \partial_x S_1 & 0 & \dots & \partial_x S_{n_n} & 0 \\ 0 & \partial_y S_1 & \dots & 0 & \partial_y S_{n_n} \\ \partial_y S_1 & \partial_x S_1 & \dots & \partial_y S_{n_n} & \partial_x S_{n_n} \end{pmatrix}, \quad (14)$$

where n_n is the total number of associated nodes to an element e , in which a material point p resides.

The algorithm outlined in Fig. 4 (the function `[mpD] = SdS(meD, mpD, p2N)` called at the beginning of each cycle; see Fig. 4) represents the vectorized solution of the computation of basis functions and their derivatives.

Coordinates of the material points `mpD.x(:, 1:2)` are first replicated and then subtracted by their associated nodes' coordinates, e.g. `meD.x(p2N)` and `meD.y(p2N)` respectively (lines 3 or 5 in Fig. 4). This yields the array `D` with the same dimension of `p2N`. This array of distance between the points and their associated nodes is sent as an input to the nested function `[N, dN] = NdN(D, h, lp)`, which computes the one-dimensional basis function and its derivative through matrix element-wise operations (operator `.*`) (either in line 4 for x coordinates or line 6 for y coordinates in Fig. 4).

Given the piece-wise Eq. (11), three logical arrays (`c1`, `c2` and `c3`) are defined (lines 21–24 in Fig. 4), whose elements are either one (the condition is true) or zero (the condition is false). Three arrays of basis functions are calculated (`N1`, `N2` and `N3`, lines 26–28) according to Eq. (12). The array of basis functions `N` is obtained through a summation of the element-wise multiplications of these temporary arrays with their corresponding logical arrays (line 29 in Fig. 4). The same holds true for the calculation of the gradient basis function (lines 31–34 in Fig. 4). It is faster to use logical arrays as multipliers of precomputed basis function arrays rather than using these in a conditional indexing statement, e.g. `N(c2==1) = 1-abs(dX(c2==1))./h`. The performance gain is significant between the two approaches, i.e. an intrinsic 30% gain over the wall-clock time of the basis functions and derivatives calculation. We observe an invariance of such gain with respect to the initial number of material points per element or to the mesh resolution.

3.3.2 Integration of internal forces

Another computationally expensive operation for MATLAB® is the mapping (or accumulation) of the material point contributions to their associated nodes. It is performed by the function `p2Nsolve.m` in the workflow of the solver.

The standard calculations for the material point contributions to the lumped mass m_n , the momentum \mathbf{p}_n , the external

```

1 function [mpD] = SdS(meD, mpD, p2N)
2 %% COMPUTE (X,Y)-BASIS FUNCTION
3 D = ( repmat(mpD.x(:,1),1,meD.nNe) - meD.x(p2N) );%
4 [Sx,dSx] = NdN(D,meD.h(1), repmat(mpD.1(:,1),1,meD.nNe));%
5 D = ( repmat(mpD.x(:,2),1,meD.nNe) - meD.y(p2N) );%
6 [Sy,dSy] = NdN(D,meD.h(2), repmat(mpD.1(:,2),1,meD.nNe));%
7 %% CONVOLUTION OF BASIS FUNCTIONS
8 mpD.S = Sx.* Sy ;%
9 mpD.dSx = dSx.* Sy ;%
10 mpD.dSy = Sx.* dSy ;%
11 %% B MATRIX ASSEMBLY
12 iDx = 1:meD.DoF:meD.nDoF(1)-1 ;%
13 iDy = iDx+1 ;%
14 mpD.B(1,iDx,:) = mpD.dSx' ;%
15 mpD.B(2,iDy,:) = mpD.dSy' ;%
16 mpD.B(3,iDx,:) = mpD.dSy' ;%
17 mpD.B(3,iDy,:) = mpD.dSx' ;%
18 end
19 function [N,dN]=NdN(dX,h,lp)
20 %% COMPUTE BASIS FUNCTIONS
21 lp = 2*lp ;%
22 c1 = ( abs(dX)<( 0.5*lp) ) ;%
23 c2 = ((abs(dX)>=( 0.5*lp)) & (abs(dX)<(h-0.5*lp))) ;%
24 c3 = ((abs(dX)>=(h-0.5*lp)) & (abs(dX)<(h+0.5*lp))) ;%
25 % BASIS FUNCTION
26 N1 = 1-((4*dX.^2+lp.^2)./(4*h.*lp)) ;%
27 N2 = 1-(abs(dX)./h) ;%
28 N3 = ((h+0.5*lp-abs(dX)).^2)./(2*h.*lp) ;%
29 N = c1.*N1+c2.*N2+c3.*N3 ;%
30 % BASIS FUNCTION GRADIENT
31 dN1 = -((8*dX)./(4*h.*lp)) ;%
32 dN2 = sign(dX).*(-1/h) ;%
33 dN3 = -sign(dX).*(h+0.5*lp-abs(dX))./(h*lp) ;%
34 dN = c1.*dN1+c2.*dN2+c3.*dN3 ;%
35 end
    
```

Figure 4. Code fragment 1 shows the vectorized solution to the calculation of the basis functions and their derivatives within `SdS.m`. Table B1 lists the variables used.

f_n^e and internal f_n^i forces are given by

$$\mathbf{m}_n = \sum_{p \in n} S_n(\mathbf{x}_p) m_p, \quad (15)$$

$$\mathbf{p}_n = \sum_{p \in n} S_n(\mathbf{x}_p) m_p \mathbf{v}_p, \quad (16)$$

$$\mathbf{f}_n^e = \sum_{p \in n} S_n(\mathbf{x}_p) m_p \mathbf{b}_p, \quad (17)$$

$$\mathbf{f}_n^i = \sum_{p \in n} v_p \mathbf{B}^T(\mathbf{x}_p) \boldsymbol{\sigma}_p, \quad (18)$$

where m_p is the material point mass, \mathbf{v}_p is the material point velocity, \mathbf{b}_p is the body force applied to the material point and $\boldsymbol{\sigma}_p$ is the material point Cauchy stress tensor in the Voigt notation.

Once the mapping phase is achieved, the equations of motion are explicitly solved forward in time on the mesh. Nodal accelerations \mathbf{a}_n and velocities \mathbf{v}_n are given by

$$\mathbf{a}_n^{t+\Delta t} = m_n^{-1} (\mathbf{f}_n^e - \mathbf{f}_n^i), \quad (19)$$

$$\mathbf{v}_n^{t+\Delta t} = m_n^{-1} \mathbf{p}_n + \Delta t \mathbf{a}_n^{t+\Delta t}. \quad (20)$$

Finally, boundary conditions are applied to the nodes that belong to the boundaries.

The vectorized solution comes from the use of the built-in function `accumarray()` of MATLAB® combined with `reshape()` and `repmat()`. The core of the vectorization is to use `p2N` as a vector (i.e. flattening the array `p2N(:)` results in a row vector) of subscripts with

```

1 function [meD] = p2Nsolve(meD,mpD,g,dt,l2g,p2N,bc)
2 %% INITIALIZATION
3 %% NODAL VECTOR INITIALIZATION
4 meD.m(:) = 0.0 ; meD.mr(:) = 0.0 ; meD.f(:) = 0.0 ; meD.d(:) = 0.0 ;%
5 meD.a(:) = 0.0 ; meD.p(:) = 0.0 ; meD.v(:) = 0.0 ; meD.u(:) = 0.0 ;%
6 %% CONTRIBUTION TO NODES
7 %% PREPROCESSING
8 m = reshape( mpD.S.*repmat( mpD.m,1,meD.nNe) ,mpD.n*meD.nNe ,1) ;%
9 p = reshape( (mpD.S.*repmat( mpD.p(:,1),1,meD.nNe) ;...
10 mpD.S.*repmat( mpD.p(:,2),1,meD.nNe) ],mpD.n*meD.nDoF(1),1) ;%
11 f = reshape( (mpD.S.*0.0 ;...
12 mpD.S.*repmat( mpD.m,1,meD.nNe) .*g ],mpD.n*meD.nDoF(1),1) ;%
13 fi = squeeze( sum( mpD.B.*repmat( reshape( mpD.s, size( mpD.s,1),1,mpD.n) ...
14 ,1,meD.nDoF(1) ),1) .*repmat( mpD.V',meD.nDoF(1),1)
15 %% CONTRIBUTION FROM p TO N
16 meD.m = accumarray( p2N(:,),m,[meD.nN 1]) ;%
17 meD.p = accumarray( l2g(:,),p,[meD.nDoF(2) 1]) ;%
18 meD.f = accumarray( l2g(:,),f,[meD.nDoF(2) 1]) ;%
19 for n = 1:meD.nNe
20 l = [(meD.DoF*p2N(:,n)-1);(meD.DoF*p2N(:,n))] ;%
21 meD.f = meD.f - accumarray( l,[fi(n*meD.DoF-1,:)] ;...
22 fi(n*meD.DoF ,:)] ,[meD.nDoF(2) 1]) ;%
23 end
24 %% SOLVE EXPLICIT MOMENTUM BALANCE EQUATION
25 %% UPDATE GLOBAL NODAL INFORMATIONS
26 iDx = 1:meD.DoF:meD.nDoF(2)-1 ;%
27 iDy = iDx+1 ;%
28 %% COMPUTE GLOBAL NODAL FORCE
29 meD.d(iDx) = sqrt( meD.f(iDx).^2+meD.f(iDy).^2) ;%
30 meD.d(iDy) = meD.d(iDx) ;%
31 meD.f = meD.f - meD.vd.*meD.d.*sign( meD.p) ;%
32 %% UPDATE GLOBAL NODAL MOMENTUM
33 meD.p = meD.p + dt.*meD.f ;%
34 %% COMPUTE GLOBAL NODAL ACCELERATION AND VELOCITY
35 meD.mr = reshape( repmat( meD.m',meD.DoF,1) ,meD.nDoF(2),1) ;%
36 iD = meD.mr==0 ;%
37 meD.a = meD.f./meD.mr ;%
38 meD.v = meD.p./meD.mr ;%
39 meD.a(iD) = 0.0 ;%
40 meD.v(iD) = 0.0 ;%
41 %% BOUNDARY CONDITIONS: FIX DIRICHLET BOUNDARY CONDITIONS
42 meD.a( bc.x(:,1) )=bc.x(:,2) ;%
43 meD.a( bc.y(:,1) )=bc.y(:,2) ;%
44 meD.v( bc.x(:,1) )=bc.x(:,2) ;%
45 meD.v( bc.y(:,1) )=bc.y(:,2) ;%
46 end

```

Figure 5. Code fragment 2 shows the vectorized solution to the nodal projection of material point quantities (e.g. mass and momentum) within the local function `p2Nsolve.m`. The core of the vectorization process is the extensive use of the built-in function of MATLAB© `accumarray()`, for which we detail the main features in the text. Table B1 lists the variables used.

`accumarray`, which accumulates material point contributions (e.g. mass or momentum) that share the same node.

In the function `p2Nsolve` (code fragment 2, shown in Fig. 5), the first step is to initialize nodal vectors (e.g. mass, momentum and forces) to zero (lines 4–5 in Fig. 5). Then, temporary vectors (`m`, `p`, `f` and `fi`) of material point contributions (namely, mass, momentum, and external and internal forces) are generated (lines 10–17 in Fig. 5). The accumulation (nodal summation) is performed (lines 19–26 in Fig. 5) using either the flattened `p2n(:)` or `l2g(:)` (e.g. the global indices of nodes) as the vector of subscripts. Note that for the accumulation of material point contributions of internal forces, a short for-loop iterates over the associated node (e.g. from 1 to `meD.nNe`) of every material point to accumulate their respective contributions.

To calculate the temporary vector of internal forces (`fi` at lines 15–17 in Fig. 5), the first step consists of the matrix multiplication of the strain-displacement matrix `mpD.B` with the material point stress vector `mpD.s`. The vectorized solution is given by (i) element-wise multiplications of `mpD.B` with a replication of the transposed stress vector `repmat(reshape(mpD.s, size(mpD.s,1) ,1, mpD.n) ,1,meD.nDoF(1))`, whose result is then (ii) summed by means of the built-in function `sum()` along the columns and, finally, multiplied by a repli-

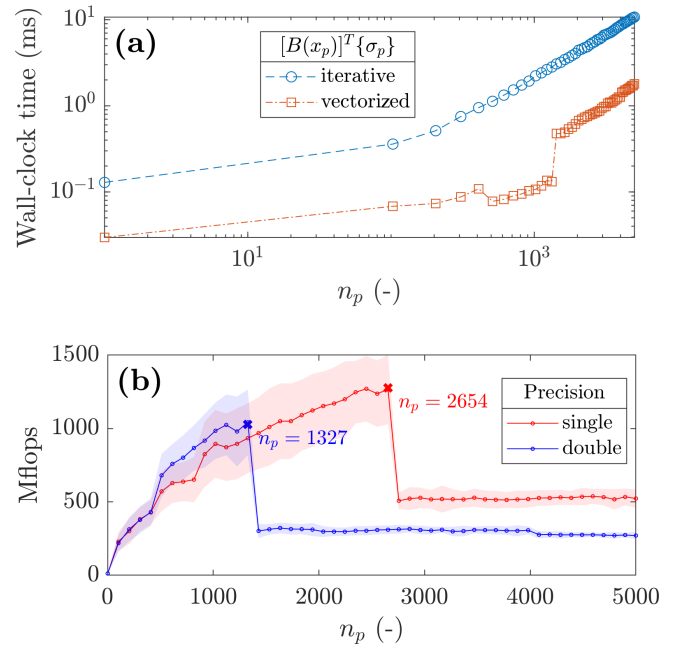


Figure 6. (a) The wall-clock time to solve for a matrix multiplication between a multidimensional array and a vector with an increasing number of the third dimension with a double arithmetic precision, and (b) the number of floating point operations per second (flops) for single and double arithmetic precisions. The continuous line represents the average values, and the shaded area denotes the standard deviation.

cated transpose of the material point volume vector, e.g. `repmat(mpD.V', meD.nDoF(1) ,1)` .

To illustrate the numerical efficiency of the vectorized multiplication between a matrix and a vector, we have developed an iterative and vectorized solution of $\mathbf{B}(\mathbf{x}_p)^T \boldsymbol{\sigma}_p$ with an increasing n_p and considering single (4 bytes) and double (8 bytes) arithmetic precision. The wall-clock time increases with n_p with a sharp transition for the vectorized solution around $n_p \approx 1000$, as shown in Fig. 6a. The mathematical operation requires more memory than available in the L2 cache (1024 kB under the CPU architecture used), which inhibits cache reuse (Dabrowski et al., 2008). A peak performance of at least 1000 Mflops (million floating point operations per second), shown in Fig. 6b, is achieved when $n_p = 1327$ or $n_p = 2654$ for simple or double arithmetic precision respectively, i.e. it corresponds exactly to 1024 kB for both precisions. Beyond this value, the performance drops dramatically to approximately half of the peak value. This drop is even more severe for a double arithmetic precision.

3.3.3 Update of material point properties

Finally, we propose a vectorization of the function `mapN2p.m` that (i) interpolates updated nodal solutions to the material points (velocities and coordinates) and (ii) the double-mapping (DM or MUSL) procedure (see Fern et al.,

2019). The material point velocity \mathbf{v}_p is defined as an interpolation of the solution of the updated nodal accelerations, which is given by

$$\mathbf{v}_p^{t+\Delta t} = \mathbf{v}_p^t + \Delta t \sum_{n=1}^{n_n} S_n(\mathbf{x}_p) \mathbf{a}_n^{t+\Delta t}. \quad (21)$$

The material point updated momentum is found by $\mathbf{p}_p^{t+\Delta t} = m_p \mathbf{v}_p^{t+\Delta t}$. The double-mapping procedure of the nodal velocity \mathbf{v}_n consists of the remapping of the updated material point momentum on the mesh, divided by the nodal mass, given as

$$\mathbf{v}_n^{t+\Delta t} = m_n^{-1} \sum_{p \in n} S_n(\mathbf{x}_p) \mathbf{p}_p^{t+\Delta t}, \quad (22)$$

and for which boundary conditions are enforced. Finally, the material point coordinates are updated based on the following:

$$\mathbf{x}_p^{t+\Delta t} = \mathbf{x}_p^t + \Delta t \sum_{n=1}^{n_n} S_n(\mathbf{x}_p) \mathbf{v}_n^{t+\Delta t}. \quad (23)$$

To solve for the interpolation of updated nodal solutions to the material points, we rely on a combination of element-wise matrix multiplication between the array of basis functions $\text{mpD} \cdot S$ with the global vectors through a transform of the $p2N$ array, i.e. $iDx = \text{meD} \cdot \text{DoF} * p2N - 1$ and $iDy = iDx + 1$ (lines 3–4 in code fragment 3 in Fig. 7), which are used to access the x and y components of global vectors.

When accessing global nodal vectors by means of iDx and iDy , the resulting arrays are naturally of the same size as $p2N$ and are, therefore, dimension-compatible with $\text{mpD} \cdot S$. For instance, a summation along the columns (e.g. the associated nodes of material points) of an element-wise multiplication of $\text{mpD} \cdot S$ with $\text{meD} \cdot a(iDx)$ results in an interpolation of the x component of the global acceleration vector to the material points.

This procedure is used for the velocity update (line 6 in Fig. 7) and for the material point coordinate update (line 22 in Fig. 7). A remapping of the nodal momentum is carried out (lines 11 to 14 in Fig. 7), which allows for the calculation of the updated nodal incremental displacements (line 15 in Fig. 7). Finally, boundary conditions of nodal incremental displacements are enforced (lines 19–20 in Fig. 7).

3.4 Initial settings and adaptive time step

Regarding the initial setting of the background mesh of the demonstration cases presented in the following, we select a uniform mesh and a regular distribution of material points within the initially populated elements of the mesh. Each element is evenly filled with four material points, e.g. $n_{pe} = 2^2$, unless otherwise stated.

In this contribution, Dirichlet boundary conditions are resolved directly on the background mesh, as in the standard

```

1 function [meD, mpD] = mapN2p(meD, mpD, dt, l2g, p2N, bc)
2 %% INTERPOLATE SOLUTIONS N to p
3 iDx = meD.DoF*p2N-1 ;%
4 iDy = iDx+1 ;%
5 % VELOCITY UPDATE
6 mpD.v = mpD.v+dt*[sum(mpD.S.*meD.a(iDx),2) sum(mpD.S.*meD.a(iDy),2)] ;%
7 % MOMENTUM UPDATE
8 mpD.p = mpD.v.*repmat(mpD.m,1,meD.DoF) ;%
9 % UPDATE NODAL MOMENTUM WITH UPDATED MP MOMENTUM (MUSL OR DOUBLE MAPPING) ;%
10 meD.p(:) = 0 ;%
11 p = reshape([mpD.S.*repmat(mpD.p(:,1),1,meD.nNe);...
12             mpD.S.*repmat(mpD.p(:,2),1,meD.nNe)];...
13             mpD.n*meD.nDoF(1),1) ;%
14 meD.p = accumarray(l2g(:,1),p,[meD.nDoF(2) 1]) ;%
15 meD.u = dt*(meD.p./meD.mr) ;%
16 iD = meD.mr==0 ;%
17 meD.u(iD)= 0 ;%
18 %% BOUNDARY CONDITIONS: FIX DIRICHLET BOUNDARY CONDITIONS
19 meD.u(bc.x(:,1))=bc.x(:,2) ;%
20 meD.u(bc.y(:,1))=bc.y(:,2) ;%
21 % UPDATE COORDINATE AND DISPLACEMENT
22 mpD.x = mpD.x+[sum(mpD.S.*meD.u(iDx),2) sum(mpD.S.*meD.u(iDy),2)] ;%
23 mpD.u = mpD.u+[sum(mpD.S.*meD.u(iDx),2) sum(mpD.S.*meD.u(iDy),2)] ;%
24 end
    
```

Figure 7. Code fragment 3 shows the vectorized solution for the interpolation of nodal solutions to material points with a double-mapping procedure (or MUSL) within the function `mapN2p.m`.

finite element method. This implies that boundary conditions are resolved only in contiguous regions between the mesh and the material points. Deviating from this contiguity or having the mesh not aligned with the coordinate system requires specific treatments for boundary conditions (Cortis et al., 2018). Furthermore, we ignore the external tractions as their implementation is complex.

As explicit time integration is only conditionally stable, any explicit formulation requires a small time step Δt to ensure numerical stability (Ni and Zhang, 2020), e.g. smaller than a critical value defined by the Courant–Friedrichs–Lewy (CFL) condition. Hence, we employ an adaptive time step (de Vaucorbeil et al., 2020), which considers the velocity of the material points. The first step is to compute the maximum wave speed of the material using (Zhang et al., 2016; Anderson Jr., 1987)

$$(c_x, c_y) = \left(\max_p (V + |(v_x)_p|), \max_p (V + |(v_y)_p|) \right), \quad (24)$$

where the wave speed is $V = ((K + 4G/3)/\rho)^{\frac{1}{2}}$, K and G are the bulk and shear moduli respectively, ρ is the material density, and $(v_x)_p$ and $(v_y)_p$ are the material point velocity components. Δt is then restricted by the CFL condition as follows:

$$\Delta t = \alpha \min \left(\frac{h_x}{c_x}, \frac{h_y}{c_y} \right), \quad (25)$$

where $\alpha \in [0; 1]$ is the time step multiplier, and h_x and h_y are the mesh spacings.

4 Results

In this section, we first demonstrate our MATLAB-based MPM solver to be efficient at reproducing results from other studies, i.e. the compaction of an elastic column (Coombs et al., 2020; e.g. quasi-static analysis), the cantilever beam

problem (Sadeghirad et al., 2011; e.g. large elastic deformation) and an application to landslide dynamics Huang et al., 2015; e.g. elasto-plastic behaviour). Then, we present both the efficiency and the numerical performance for a selected case, e.g. the elasto-plastic collapse. We offer conclusions on and compare the performance of the solver with respect to the specific case of an impact of two elastic discs previously implemented in a Julia language environment by Sinaie et al. (2017).

Regarding the performance analysis, we investigate the performance gain of the vectorized solver considering a double arithmetic precision with respect to the total number of material points for the following reasons: (i) the mesh resolution, i.e. the total number of elements n_{el} , influences the wall-clock time of the solver by reducing the time step due to the CFL condition, thereby increasing the total number of iterations. In addition, (ii) the total number of material points n_p increases the number of operations per cycle due to an increase in the size of matrices, i.e. the size of the strain-displacement matrix depends on n_p and not on n_{el} . Hence, n_p consistently influences the performance of the solver, whereas n_{el} determines the wall-clock time of the solver. The performance of the solver is addressed through both the number of floating point operations per second (flops) and via the average number of iteration per second (iterations s^{-1}). The number of floating point operations per second was manually estimated for each function of the solver.

4.1 Validation of the solver and numerical efficiency

4.1.1 Convergence: the elastic compaction of a column under its own weight

Following the convergence analysis proposed by Coombs and Augarde (2020), Wang et al. (2019) and Charlton et al. (2017), we analyse an elastic column of an initial height $l_0 = 10\text{ m}$ subjected to an external load (e.g. the gravity). We selected the cpGIMPM variant with a domain update based on the diagonal components of the deformation gradient. Coombs et al. (2020) showed that such a domain update is well suited for hydrostatic compression problems. We also selected the CPDI2q variant as a reference, because of its superior convergence accuracy for such problems compared with GIMPM (Coombs et al., 2020).

The initial geometry is shown in Fig. 8. The background mesh is made of bi-linear four-noded quadrilaterals, and roller boundary conditions are applied on the base and the sides of the column, initially populated by four material points per element. The column is 1 element wide and n elements tall, and the number of elements in the vertical direction is increased from 1 to a maximum of 1280 elements. The time step is adaptive, and we selected a time step multiplier of $\alpha = 0.5$, e.g. minimal and maximal time step values of $\Delta t_{\min} = 3.1 \times 10^{-4}\text{ s}$ and $\Delta t_{\max} = 3.8 \times 10^{-4}\text{ s}$ respectively for the finest mesh of 1280 elements.

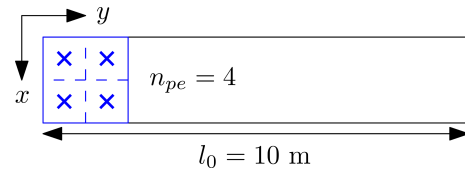


Figure 8. Initial geometry of the column.

To consistently apply the external load for the explicit solver, we follow the recommendation of Bardenhagen and Kober (2004), i.e. a quasi-static solution (given that an explicit integration scheme is chosen) is obtained if the total simulation time is equal to 40 elastic wave transit times. The material has a Young's modulus $E = 1 \times 10^4\text{ Pa}$ and a Poisson's ratio $\nu = 0$ with a density $\rho = 80\text{ kg m}^{-3}$. The gravity g is increased from zero to its final value, i.e. $g = 9.81\text{ m s}^{-2}$. We performed additional implicit quasi-static simulations (named iCPDI2q) in order to consistently discuss the results with respect to what was reported in Coombs and Augarde (2020). The external force is consistently applied over 50 equal load steps. The vertical normal stress is given by the analytical solution (Coombs and Augarde, 2020) $\sigma_{yy}(y_0) = \rho g(l_0 - y_0)$, where l_0 is the initial height of the column, and y_0 is the initial position of a point within the column.

The error between the analytical and numerical solutions is as follows:

$$\text{error} = \sum_{p=1}^{n_p} \frac{\|(\sigma_{yy})_p - \sigma_{yy}(y_p)\|(V_0)_p}{(\rho g l_0) V_0}, \quad (26)$$

where $(\sigma_{yy})_p$ is the stress along the y axis of a material point p (Fig. 8) of an initial volume $(V_0)_p$, and V_0 is the initial volume of the column, i.e. $V_0 = \sum_{p=1}^{n_p} (V_0)_p$.

The convergence toward a quasi-static solution is shown in Fig. 9a. It is quadratic for both cpGIMPM and CPDI2q; however, contrary to Coombs et al. (2020); Coombs and Augarde (2020), who reported a full convergence, it stops at error $\approx 2 \times 10^{-6}$ for the explicit implementation. This has already been outlined by Bardenhagen and Kober (2004) as a saturation of the error caused by resolving the dynamic stress wave propagation, which is inherent to any explicit scheme. Hence, a static solution could never be achieved, because unlike quasi-static implicit methods, the elastic waves propagate indefinitely and the static equilibrium is never resolved. This is consistent when compared to the iCPDI2q solution we implemented, whose behaviour is still converging below the limit error $\approx 2 \times 10^{-6}$ reached by the explicit solver. However, the convergence rate of the implicit algorithm decreases as the mesh resolution increases. We did not investigate this as our focus is on the explicit implementation. The vertical stresses of material points are in good agreement with the analytical solution (see Fig. 9b). Some oscillations are observed for a coarse mesh resolution, but these rapidly decrease as the mesh resolution increases.

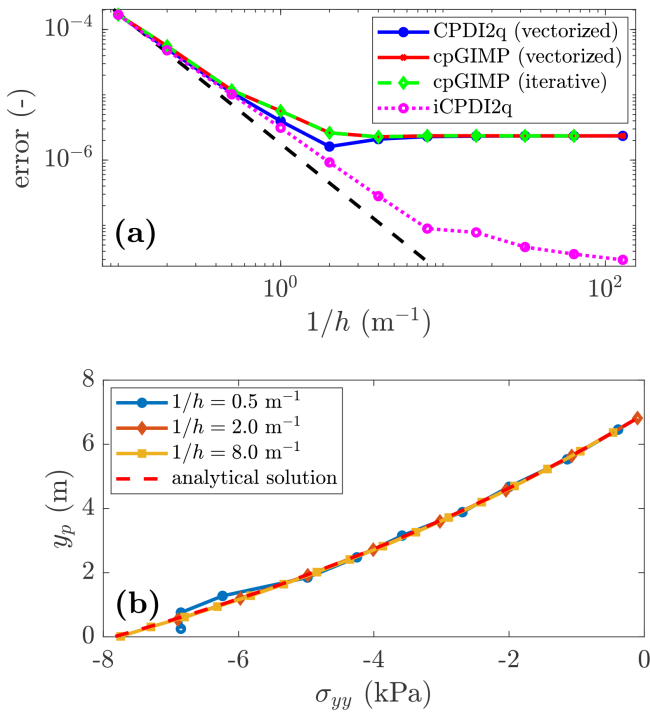


Figure 9. (a) Convergence of the error: a limit is reached at error $\approx 2 \times 10^{-6}$ for the explicit solver, whereas the quasi-static solution still converges. This was demonstrated in Bardenhagen and Kober (2004) as an error saturation due to the explicit scheme, i.e. the equilibrium is never resolved. (b) The stress σ_{yy} along the y axis predicted at the deformed position y_p by the CPDI2q variant is in good agreement with the analytical solution for a refined mesh.

We finally report the wall-clock time for the cpGIMPM (iterative), cpGIMPM (vectorized) and the CPDI2q (vectorized) variants. As claimed by Sadeghirad et al. (2013, 2011), the CPDI2q variant induces no significant computational cost compared to the cpGIMPM variant. However, the absolute value between vectorized and iterative implementations is significant. For $n_p = 2560$, the vectorized solution completed in 1161 s, whereas the iterative solution completed in 52 856 s. The vectorized implementation is roughly 50 times faster than the iterative implementation.

4.1.2 Large deformation: the elastic cantilever beam problem

The cantilever beam problem (Sinaie et al., 2017; Sadeghirad et al., 2011) is the second benchmark that demonstrates the robustness of the MPM solver. Two MPM variants are implemented, namely (i) the contiguous GIMPM (cpGIMPM), which relies on the stretching part of the deformation gradient (see Charlton et al., 2017) to update the particle domain as large rotations are expected during the deformation of the beam, and (ii) the convected particle domain interpolation (CPDI, Leavy et al., 2019; Sadeghirad et al., 2011). We selected the CPDI variant as it is more suitable to large

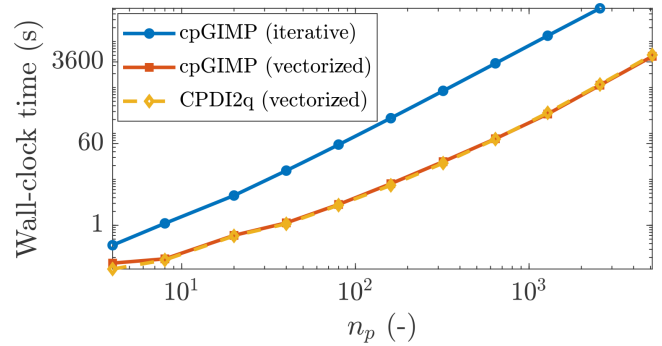


Figure 10. The wall-clock time for cpGIMPM (vectorized and iterative solutions) and the CPDI2q solution with respect to the total number of material points n_p . There is no significant differences between the CPDI2q and cpGIMPM variants regarding the wall-clock time. The iterative implementation is also much slower than the vectorized implementation.

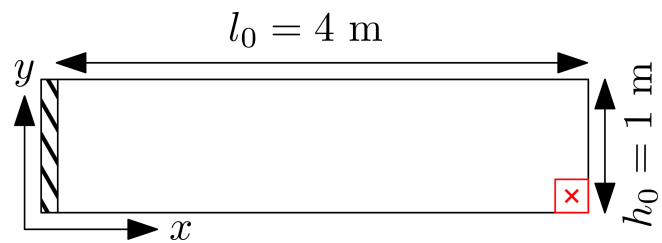


Figure 11. Initial geometry for the cantilever beam problem; the free end material point appears in red, and a red cross marks its centre.

torsional deformation modes (Coombs et al., 2020) than the CPDI2q variant. Two constitutive elastic models are selected, i.e. neo-Hookean (Guilkey and Weiss, 2003) or linear elastic (York et al., 1999) solids. For consistency, we use the same physical quantities as in Sadeghirad et al. (2011), i.e. an elastic modulus $E = 10^6 \text{ Pa}$, a Poisson's ratio $\nu = 0.3$, a density $\rho = 1050 \text{ kg/m}^3$, the gravity $g = 10.0 \text{ m/s}$ and a real-time simulation $t = 3 \text{ s}$ with no damping forces introduced.

The beam geometry is depicted in Fig. 11 and is discretized by 64 four-noded quadrilaterals, each of them initially populated by nine material points (e.g. $n_p = 576$) with a adaptive time step determined by the CFL condition, i.e. the time step multiplier is $\alpha = 0.1$, which yields minimal and maximal time step values of $\Delta t_{\min} = 5.7 \times 10^{-4} \text{ s}$ and $\Delta t_{\max} = 6.9 \times 10^{-4} \text{ s}$ respectively. The large deformation is initiated by suddenly applying the gravity at the beginning of the simulation, i.e. $t = 0 \text{ s}$.

As indicated in Sadeghirad et al. (2011), the cpGIMPM simulation failed when using the diagonal components of the deformation gradient to update the material point domain, i.e. the domain vanishes under large rotations as stated in Coombs et al. (2020). However, as expected, the cpGIMPM simulation succeeded when using the diagonal terms of the stretching part of the deformation gradient, as proposed by

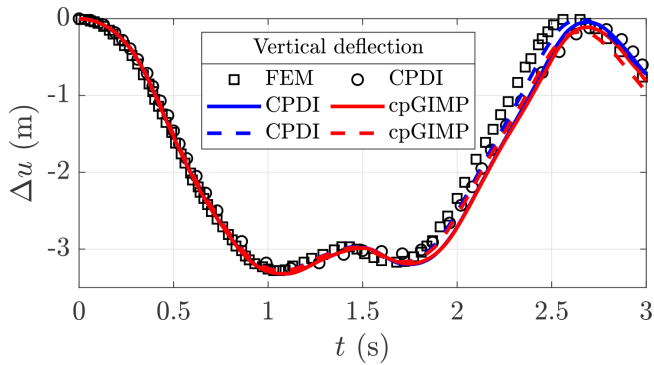


Figure 12. Vertical deflection Δu for the cantilever beam problem. The black markers denote the solutions of Sadeghirad et al. (2011) (circles for CPDI and squares for FEM). The line colour indicates the MPM variant (blue for CPDI and red for cpGIMP), solid lines refer to a linear elastic solid and dashed lines refer to a neo-Hookean solid. Δu corresponds to the vertical displacement of the bottom material point at the free end of the beam (the red cross in Fig. 11).

Coombs et al. (2020) and Charlton et al. (2017). The numerical solutions, obtained by the latter cpGIMPM and CPDI, to the vertical deflection Δu of the material point at the bottom free end of the beam (e.g. the red cross in Fig. 11) are shown in Fig. 12. Some comparative results reported by Sadeghirad et al. (2011) are depicted by black markers (squares for the FEM solution and circles for the CPDI solution), whereas the results of the solver are depicted by lines.

The local minimal and the minimal and maximal values (in timing and magnitude) are in agreement with the FEM solution of Sadeghirad et al. (2011). The elastic response is in agreement with the CPDI results reported by Sadeghirad et al. (2011), but it differs in timing with respect to the FEM solution. This confirms our numerical implementation of CPDI when compared to the one proposed by Sadeghirad et al. (2011). In addition, the elastic response does not substantially differ from a linear elastic solid to a neo-Hookean one. It demonstrates the incremental implementation of the MPM solver to be relevant in capturing large elastic deformations for the cantilever beam problem.

Figure 13 shows the finite deformation of the material point domain (panel a or c) and the vertical Cauchy stress field (panel b or d) for CPDI and cpGIMPM. The stress oscillations due to the cell-crossing error are partially cured when using a domain-based variant compared with the standard MPM. However, spurious vertical stresses are more developed in Fig. 13d than in Fig. 13b, where the vertical stress field appears even smoother. Both CPDI and cpGIMPM give a decent representation of the actual geometry of the deformed beam.

We also report quite a significant difference in execution time between the CPDI variant compared with the CPDI2q and cpGIMPM variants: CPDI executes in an average of 280.54 iteration per second, whereas both CPDI2q

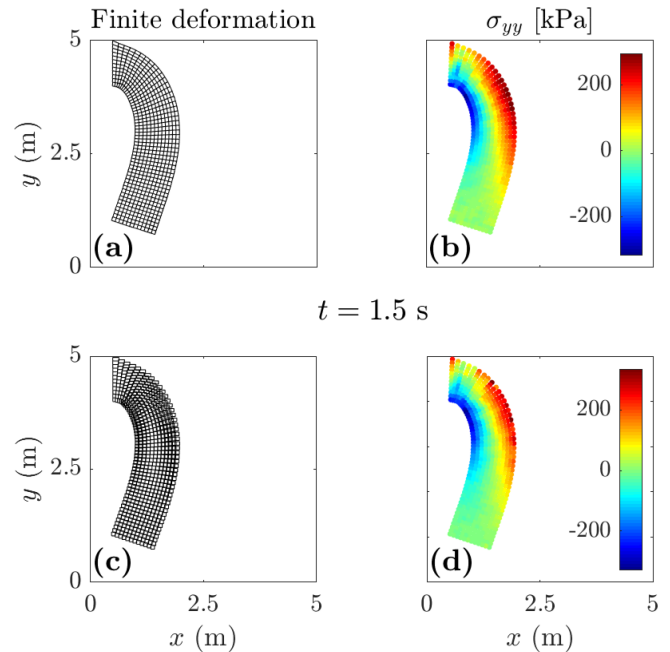


Figure 13. Finite deformation of the material point domain and vertical Cauchy stress σ_{yy} for CPDI (panels a and b respectively) and cpGIMPM (panels c and d respectively). The CPDI variant gives a better and more contiguous description of the material point's domain and a slightly smoother stress field compared with the cpGIMPM variant, which is based on the stretching part of the deformation gradient.

and cpGIMPM execute in an average of 301.42 iterations s^{-1} and an average of 299.33 iterations s^{-1} respectively.

4.1.3 Application: the elasto-plastic slumping dynamics

We present an application of the MPM solver (vectorized and iterative version) to the case of landslide mechanics. We selected the domain-based CPDI variant as it performs better than the CPDI2q variant in modelling torsional and stretching deformation modes (Wang et al., 2019) coupled to an elasto-plastic constitutive model based on a non-associated Mohr–Coulomb (M-C) plasticity (Simpson, 2017). We (i) analyse the geometrical features of the slump and (ii) compare the results (the geometry and the failure surface) to the numerical simulation of Huang et al. (2015), which is based on a Drucker–Prager model with tension cut-off (D-P).

The geometry of the problem is shown in Fig. 14; the soil material is discretized by 110×35 elements with $n_{pe} = 9$, resulting in $n_p = 21\,840$ material points. A uniform mesh spacing $h_{x,y} = 1$ m is used, and rollers are imposed at the left and right domain limits, while a no-slip condition is enforced at the base of the material. We closely follow the numerical procedure proposed in Huang et al. (2015), i.e. no local damping is introduced in the equation of motion and

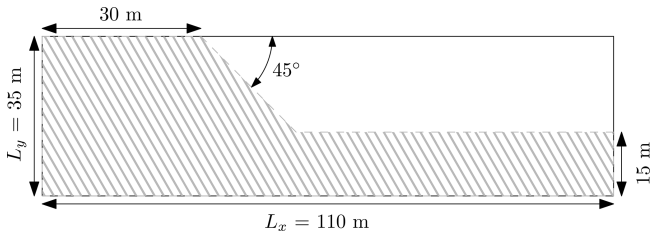


Figure 14. Initial geometry for the slump problem from Huang et al. (2015). Roller boundary conditions are imposed on the left and right of the domain, while a no-slip condition is enforced at the base of the material.

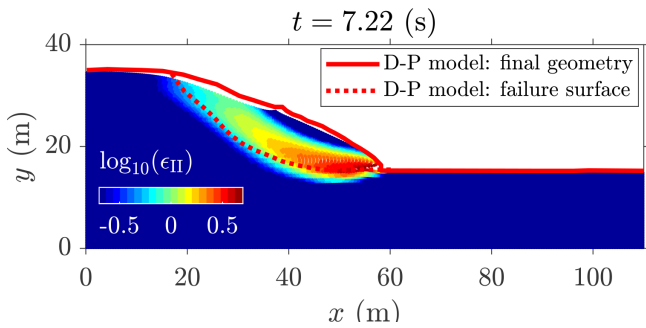


Figure 15. MPM solution to the elasto-plastic slump. The red lines indicate the numerical solution of Huang et al. (2015), and the coloured points indicate the second invariant of the accumulated plastic strain ϵ_{II} obtained by the CPDI solver. An intense shear zone progressively develops backwards from the toe of the slope, resulting in a circular failure mode.

the gravity is suddenly applied at the beginning of the simulation. As in Huang et al. (2015), we consider an elasto-plastic cohesive material of density $\rho = 2100 \text{ kg m}^{-3}$, with an elastic modulus $E = 70 \text{ MPa}$ and a Poisson’s ratio $\nu = 0.3$. The cohesion is $c = 10 \text{ Pa}$, and the internal friction angle is $\phi = 20^\circ$ with no dilatancy, i.e. the dilatancy angle is $\psi = 0$. The total simulation time is 7.22 s, and we select a time step multiplier $\alpha = 0.5$. The adaptive time steps (considering the elastic properties and the mesh spacings $h_{x,y} = 1 \text{ m}$) yield minimal and maximal values of $\Delta t_{\min} = 2.3 \times 10^{-3} \text{ s}$ and $\Delta t_{\max} = 2.4 \times 10^{-3} \text{ s}$ respectively.

The numerical solution to the elasto-plastic problem is shown in Fig. 15. An intense shear zone, highlighted by the second invariant of the accumulated plastic strain ϵ_{II} , develops at the toe of the slope as soon as the material yields and propagates backwards to the top of the material. It results in a rotational slump. The failure surface is in good agreement with the solution reported by Huang et al. (2015) (continuous and discontinuous red lines in Fig. 15), but we also observe differences, i.e. the crest of the slope is lower compared with the original work of Huang et al. (2015). This may be explained by the problem of spurious material separation when using sMPM or GIMP (Sadeghirad et al., 2011), with the latter being overcome with the CPDI variant, i.e. the crest

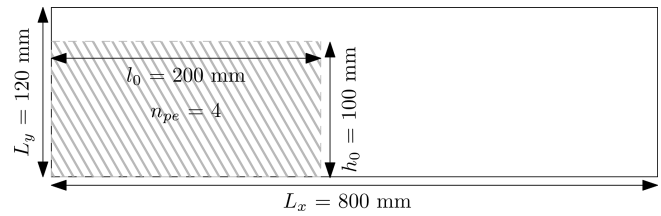


Figure 16. Initial geometry for the elasto-plastic collapse (Huang et al., 2015). Roller boundaries are imposed on the left and right boundaries of the computational domain, while a no-slip condition is enforced at the bottom of the domain. The aluminium-bar assemblage has dimensions of $l_0 \times h_0$ and is discretized by $n_{pe} = 4$ material points per initially populated element.

of the slope experiences considerable stretching deformation modes. Despite some differences, our numerical results appear coherent with those reported by Huang et al. (2015).

The vectorized and iterative solutions are resolved within approximately 630 s (a wall-clock time of $\approx 10 \text{ min}$ and an average of $4.20 \text{ iterations s}^{-1}$) and 14 868 s (a wall-clock time of $\approx 1 \text{ h}$ and an average of $0.21 \text{ iterations s}^{-1}$) respectively. This corresponds to a performance gain of 23.6. The performance gain is significant between an iterative and a vectorized solver for this problem.

4.2 Computational performance

4.2.1 Iterative and vectorized elasto-plastic collapses

We evaluate the computational performance of the solver, using the MATLAB version R2018a on an Intel Core i7-4790, with a benchmark based on the elasto-plastic collapse of the aluminium-bar assemblage, for which numerical and experimental results were initially reported by Bui et al. (2008) and Huang et al. (2015) respectively.

We vary the number of elements of the background mesh, which results in a variety of different regular mesh spacings $h_{x,y}$. The number of elements along the x and y directions are $n_{el,x} = [10, 20, 40, 80, 160, 320, 640]$ and $n_{el,y} = [1, 2, 5, 11, 23, 47, 95]$ respectively. The number of material points per element is kept constant, i.e. $n_{pe} = 4$, and this yields a total number of material points $n_p = [10, 50, 200, 800, 3200, 12800, 51200]$. The initial geometry and boundary conditions used for this problem are depicted in Fig. 16. The total simulation time is 1.0 s, and the time step multiplier is $\alpha = 0.5$. According to Huang et al. (2015), the gravity $g = 9.81 \text{ m s}^{-2}$ is applied to the assemblage, and no damping is introduced. We consider a non-cohesive granular material (Huang et al., 2015) of density $\rho = 2650 \text{ kg m}^{-3}$, with a bulk modulus $K = 0.7 \text{ MPa}$ and a Poisson’s ratio $\nu = 0.3$. The cohesion is $c = 0 \text{ Pa}$, the internal friction angle is $\phi = 19.8^\circ$ and there is no dilatancy, i.e. $\psi = 0$.

We conducted preliminary investigations using either uGIMP or cpGIMP variants – the latter with a domain

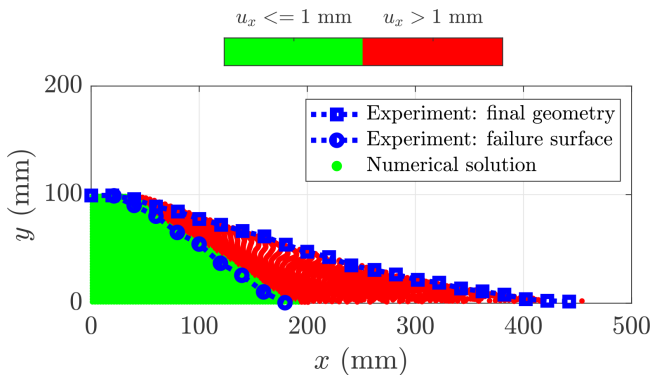


Figure 17. Final geometry of the collapse: in the intact region (horizontal displacement $u_x < 1$ mm), the material points are coloured in green; in the deformed region (horizontal displacement $u_x > 1$ mm), they are coloured in red and indicate plastic deformations of the initial mass. The transition between the deformed and undeformed regions marks the failure surface of the material. The experimental results of Bui et al. (2008) are depicted by the blue dotted lines. The computational domain is discretized by a background mesh made of 320×48 quadrilateral elements with $n_p = 4$ per initially populated element, i.e. a total $n_p = 12\,800$ material points discretize the aluminium assemblage.

update based either on the determinant of the deformation gradient or on the diagonal components of the stretching part of the deformation gradient. We concluded that the uGIMPM was the most reliable, even though its suitability is restricted to both simple shear and pure rotation deformation modes (Coombs et al., 2020).

We observe a good agreement between the numerical simulation and the experiments (see Fig. 17), considering either the final surface (blue square dotted line) or the failure surface (blue circle dotted line). The repose angle in the numerical simulation is approximately 13° , which is in agreement with the experimental data reported by Bui et al. (2008), who reported a final angle of 14° .

The vectorized and iterative solutions (for a total of $n_p = 12\,800$ material points) are resolved within approximately 1595 s (a wall-clock time of ≈ 0.5 h and an average of 10.98 iterations s^{-1}) and 43 861 s (a wall-clock time of ≈ 12 h and an average of 0.38 iterations s^{-1}) respectively. This corresponds to a performance gain of 28.24 for a vectorized code over an iterative code to solve this elasto-plastic problem.

The performance of the solver is demonstrated in Fig. 18. A peak performance of ≈ 900 Mflops is reached as soon as n_p exceeds 1000 material points, and a residual performance of ≈ 600 Mflops is further resolved (for $n_p \approx 50\,000$ material points). Every function provides an even and fair contribution on the overall performance, except the function `constitutive.m` for which the performance appears delayed or shifted. First of all, this function treats the elasto-plastic constitutive relation, in which the dimen-

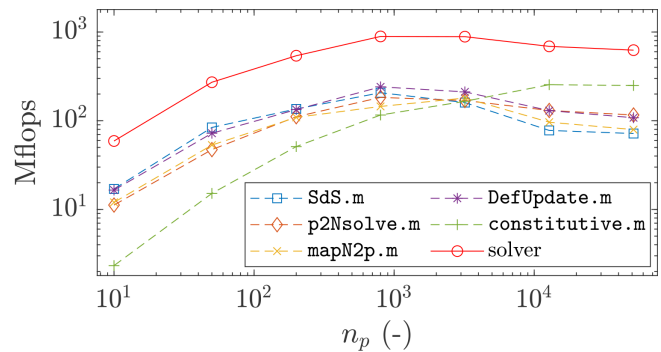


Figure 18. Number of floating point operations per second (flops) with respect to the total number of material points n_p for the vectorized implementation. The discontinuous lines refer to the functions of the solver, and the continuous line refer to the solver. A peak performance of 900 Mflops is reached by the solver for $n_p > 1000$, and a residual performance of 600 Mflops is further resolved for an increasing n_p .

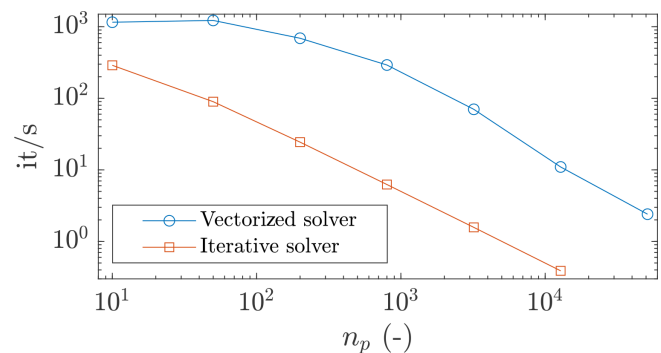


Figure 19. Number of iterations per second with respect to the total number of material points n_p . The greatest performance gain is reached around $n_p = 1000$, which is related to the peak performance of the solver (see Fig. 18). The gains corresponding to the peak performance and residual performance are 46 and 28 respectively.

sions of the matrices are smaller when compared with the other functions. Hence, the number of floating point operations per second is lower compared with other functions, e.g. `p2Nsolve.m`. This results in lower performance for an equivalent number of material points. It also requires a greater number of material points to increase the dimensions of the matrices in order to exceed the L2 cache maximum capacity.

These considerations provide a better understanding of the performance gain of the vectorized solver shown in Fig. 19: the gain increases, reaches a plateau and then ultimately decreases to a residual gain. This is directly related to the peak and the residual performance of the solver showed in Fig. 18.

Table 1. Efficiency comparison of the Julia implementation of Sinaie et al. (2017) and the MATLAB-based implementation for the two elastic disc impact problems.

Mesh	n_{pe}	n_p	Iterations s^{-1}		
			Julia	MATLAB	Gain
20×20	2^2	416	132.80	450.27	3.40
20×20	4^2	1624	33.37	118.45	3.54
40×40	2^2	1624	26.45	115.59	4.37
80×80	4^2	25 784	1.82	5.21	2.86

4.2.2 Comparison between Julia and MATLAB

We compare the computational efficiency of the vectorized CPDI2q MATLAB implementation and the computational efficiency reported by Sinaie et al. (2017) of a Julia-based implementation of the collision of two elastic discs problem. However, we note a difference between the actual implementation and the one used by Sinaie et al. (2017); the latter is based on a USL variant with a cut-off algorithm, whereas the present implementation relies on the MUSL (or double-mapping) procedure, which necessitates a double-mapping procedure. The initial geometry and parameters are the same as those used in Sinaie et al. (2017). However, the time step is adaptive, and we select a time step multiplier $\alpha = 0.5$. Given the variety of mesh resolution, we do not present minimal and maximal time step values.

Our CPDI2q implementation, in MATLAB R2018a, is at least 2.8 times faster than the Julia implementation proposed by Sinaie et al. (2017) for similar hardware (see Table 1). Sinaie et al. (2017) completed the analysis with an Intel Core i7-6700 (four cores with a base frequency of 3.40 GHz up to a turbo frequency of 4.00 GHz) with 16 GB RAM, whereas we used an Intel Core i7-4790 with similar specifications (see Sect. 2). However, the performance ratio between MATLAB and Julia seems to decrease as the mesh resolution increases.

5 Discussion

In this contribution, a fast and efficient explicit MPM solver is proposed that considers two variants (e.g. the uGIMPM/cpGIMPM and the CPDI/CPDI2q variants).

Regarding the compression of the elastic column, we report good agreement of the numerical solver with previous explicit MPM implementations, such as Bardenhagen and Kober (2004). The same flaw of an explicit scheme is also experienced by the solver, i.e. a saturation of the error due to the specific usage of an explicit scheme that resolves the wave propagation, thereby preventing any static equilibrium from being reached. This confirms that our implementation is consistent with previous MPM implementations. However, the implicit implementation suffers from a decrease in the

convergence rate for a fine mesh resolution. Further work would be needed to investigate this decrease in the convergence rate. This case also demonstrated that cpGIMPM and CPDI variants have a similar computational cost, and this confirms the suitability of cpGIMPM with respect to CPDI, as previously mentioned by Coombs et al. (2020) and Charlton et al. (2017).

For the cantilever beam, we report good agreement of the solver with the results of Sadeghirad et al. (2011), i.e. we report the vertical deflection of the beam to be very close in both magnitude and timing (for the CPDI variant) to the FEM solution. However, we also report a slower execution time for the CPDI variant when compared with both the cpGIMPM and CPDI2q variants.

The elasto-plastic slump also demonstrates the solver to be efficient at capturing complex dynamics in the field of geomechanics. The CPDI solution showed that the algorithm proposed by Simpson (2017) to return stresses when the material yields is well suited to the slumping dynamics. However, as mentioned by Simpson (2017), such return mapping is only valid under the assumption of a non-associated plasticity with no volumetric plastic strain. This particular case of isochoric plastic deformations raises the issue of volumetric locking. In the actual implementation, no regularization techniques are considered. As a result, the pressure field experiences severe locking for isochoric plastic deformations. One way to overcome locking phenomena would be to implement the regularization technique initially proposed by Coombs et al. (2018) for quasi-static sMPM and GIMPM implementations.

Regarding the elasto-plastic collapse, the numerical results demonstrate the solver to be in agreement with both previous experimental and numerical results (Huang et al., 2015; Bui et al., 2008). This confirms the ability of the solver to address elasto-plastic problems. However, the choice of whether to update the material point domain or not remains critical. This question remains open and would require a more thorough investigation of the suitability of each of these domain-updating variants. Nevertheless, the uGIMPM variant is a good candidate as (i) it is able to reproduce the experimental results of Bui et al. (2008), and (ii) it ensures numerical stability. However, one must consider its limited range of suitability regarding the deformation modes involved. If a cpGIMPM is selected, the splitting algorithm proposed in Gracia et al. (2019) and Homel et al. (2016) could be implemented to mitigate the amount of distortion experienced by the material point domains during deformation. We did not select the domain-updating method based on the corners of the domain as suggested in Coombs et al. (2020). This is because a domain-updating method necessitates the calculation of additional shape functions between the corners of the domain of the material point with their associated nodes. This results in an additional computational cost. Nevertheless, such a variant is of interest and should also be addressed when computational performance is not the main concern.

The computational performance comes from the combined use of the connectivity array `p2N` with the built-in function `accumarray()` to (i) accumulate material point contributions to their associated nodes or (ii) to interpolate the updated nodal solutions to the associated material points. When a residual performance is resolved, an overall performance gain (e.g. the number of iterations per second) of 28 is reported. As an example, the functions `p2nsolve.m` and `mapN2p.m` are 24 and 22 times faster than an iterative algorithm when the residual performance is achieved. The overall performance gain is in agreement with other vectorized FEM codes, i.e. O’Sullivan et al. (2019) reported an overall gain of 25.7 for a optimized continuous Galerkin finite element code.

An iterative implementation would require multiple nested for-loops and a larger number of operations on smaller matrices, which increase the number of BLAS calls, thereby inducing significant BLAS overheads and decreasing the overall performance of the solver. This is limited by a vectorized code structure. However, as shown by the matrix multiplication problem, the L2 cache reuse is the limiting factor, and it ultimately affects the peak performance of the solver due to these numerous RAM-to-cache communications for larger matrices. This problem is serious, and its influence is demonstrated by the delayed response in terms of performance for the function `constitutive.m`. However, we also have to mention that the overall residual performance was resolved only for a limited total number of material points. The performance drop of the function `constitutive.m` has never been achieved. Consequently, we suspect an additional decrease in the overall performance of the solver for larger problems.

The overall performance achieved by the solver is higher than expected, and it is even higher than what was reported by Sinaie et al. (2017). We demonstrate that MATLAB is even more efficient than Julia, i.e. a minimum 2.86 performance gain achieved compared with a similar Julia CPDI2q implementation. This confirms the efficiency of MATLAB for solid mechanics problems, provided that a reasonable amount of time is spent on the vectorization of the algorithm.

6 Conclusions

We have demonstrated the capability of MATLAB as an efficient language with regard to a material point method (MPM) implementation in an explicit formulation when bottleneck operations (e.g. calculations of the shape function or material point contributions) are properly vectorized. The computational performance of MATLAB is even higher than the performance previously reported for a similar CPDI2q implementation in Julia, provided that built-in functions such as `accumarray()` are used. However, the numerical efficiency naturally decreases with the level of complexity of the chosen MPM variant (sMPM, GIMPM or CPDI/CPDI2q).

The vectorization activities that we performed provide a fast and efficient MATLAB-based MPM solver. Such vectorized code could be transposed to a more efficient language, such as the C-CUDA language, which is known to efficiently take advantage of vectorized operations.

As a final word, a future implementation of a poro-elasto-plastic mechanical solver could be applied to complex geomechanical problems such as landslide dynamics while benefiting from a faster numerical implementation in C-CUDA, thereby resolving high three-dimensional resolutions in a decent and affordable amount a time.

Appendix A: Acronyms used throughout the paper

PIC	Particle in cell
FLIP	Fluid implicit particle
FEM	Finite element method
sMPM	Standard material point method
GIMPM	Generalized material point method
uGIMPM	Undeformed generalized material point method
cpGIMPM	Contiguous particle generalized material point method
CPDI	Convected particle domain interpolation
CPDI2q	Convected particle domain interpolation second-order quadrilateral

Appendix B: fMPMM-solver variables

Table B1. Variables of the structure arrays for the mesh meD and the material point mpD used in code fragments 1 and 2 shown in Figs. 4 and 5. nDF stores the local and global number of degrees of freedom, i.e. $nDF = [nNe, nN * DoF]$. The constant $nstr$ is the number of stress components, according to the standard definition of the Cauchy stress tensor using the Voigt notation, e.g. $\sigma_p = (\sigma_{xx}, \sigma_{yy}, \sigma_{xy})$.

	Variable	Description	Dimension
meD.	nNe	Nodes per element	(1)
	nN	Number of nodes	(1)
	DoF	Degree of freedom	(1)
	nDF	Number of DoF	(1, 2)
	h	Mesh spacing	(1, DoF)
	x	Node coordinates	(nN, 1)
	y	Node coordinates	(nN, 1)
	m	Nodal mass	(nN, 1)
	p	Nodal momentum	(nDF(2), 1)
	f	Nodal force	(nDF(2), 1)
mpD.	n	Number of points	(1)
	l	Domain half-length	(np, DoF)
	V	Volume	(np, 1)
	m	Mass	(np, 1)
	x	Point coordinates	(np, DoF)
	p	Momentum	(np, DoF)
	s	Stress	(np, nstr)
	S	Basis function	(np, nNe)
	dSx	Derivative in x	(np, nNe)
	dSy	Derivative in y	(np, nNe)
B	B matrix	(nstr, nDF(1), np)	

Code availability. The fMPMM-solver developed in this study is licensed under the GPLv3 free software licence. The latest version of the code is available for download from Bitbucket at <https://bitbucket.org/ewyser/fmpmm-solver/src/master/> (last access: 6 October 2020; Wyser et al., 2020b). The fMPMM-solver archive (v1.0 and v1.1) is available from a permanent DOI repository (Zenodo) at <https://doi.org/10.5281/zenodo.4068585> (Wyser et al., 2020a). The fMPMM-solver software includes the reproducible codes used for this study.

Author contributions. EW wrote the original draft of the paper. EW and YP developed the first version the solver (fMPMM-solver, v1.0). YA provided technical support, assisted EW with the revision of the latest version of the solver (v1.1) and corrected specific parts of the solver. EW and YA wrote and revised the final version of the paper. MJ and YP supervised the early stages of the study and provided guidance. All authors reviewed and approved the final version of the paper.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. Yury Alkhimenkov gratefully acknowledges support from the Swiss National Science Foundation (grant no. 172691). Yury Alkhimenkov and Yury Y. Podladchikov gratefully acknowledge support from the Russian Ministry of Science and Higher Education (project no. 075-15-2019-1890). The authors gratefully thank Johan Gaume for his comments that contributed to improving the overall quality of the article.

Financial support. This research has been supported by the Swiss National Science Foundation (grant no. 172691). This research has been supported by the Russian Ministry of Science and Higher Education (project no. 075-15-2019-1890).

Review statement. This paper was edited by Alexander Robel and reviewed by two anonymous referees.

References

Abe, K., Soga, K., and Bandara, S.: Material point method for coupled hydromechanical problems, *J. Geotechn. Geoenviron. Eng.*, 140, 04013033, [https://doi.org/10.1061/\(ASCE\)GT.1943-5606.0001011](https://doi.org/10.1061/(ASCE)GT.1943-5606.0001011), 2014.

Acosta, J. L. G., Vardon, P. J., Remmerswaal, G., and Hicks, M. A.: An investigation of stress inaccuracies and proposed solution in the material point method, *Comput. Mechan.*, 65, 555–581, 2020.

Anderson Jr., C. E.: An overview of the theory of hydrocodes, *Int. J. Impact Eng.*, 5, 33–59, 1987.

Bandara, S. and Soga, K.: Coupling of soil deformation and pore fluid flow using material point method, *Comput. Geotech.*, 63, 199–214, 2015.

Bandara, S., Ferrari, A., and Laloui, L.: Modelling landslides in unsaturated slopes subjected to rainfall infiltration using material point method, *Int. J. Num. Anal. Method. Geomechan.*, 40, 1358–1380, 2016.

Bardenhagen, S., Brackbill, J., and Sulsky, D.: The material-point method for granular materials, *Comput. Method. Appl. M.*, 187, 529–541, 2000.

Bardenhagen, S. G. and Kober, E. M.: The generalized interpolation material point method, *Comp. Model. Eng.*, 5, 477–496, 2004.

Baumgarten, A. S. and Kamrin, K.: A general fluid–sediment mixture model and constitutive theory validated in many flow regimes, *J. Fluid Mechan.*, 861, 721–764, 2019.

Beuth, L., Benz, T., Vermeer, P. A., and Więckowski, Z.: Large deformation analysis using a quasi-static material point method, *J. Theor. Appl. Mechan.*, 38, 45–60, 2008.

Bird, R. E., Coombs, W. M., and Giani, S.: Fast native-MATLAB stiffness assembly for SIPG linear elasticity, *Comput. Mathe. Appl.*, 74, 3209–3230, 2017.

Bui, H. H., Fukagawa, R., Sako, K., and Ohno, S.: Lagrangian meshfree particles method (SPH) for large deformation and failure flows of geomaterial using elastic–plastic soil constitutive model, *Int. J. Num. Anal. Method. Geomechan.*, 32, 1537–1570, 2008.

Charlton, T., Coombs, W., and Augarde, C.: iGIMP: An implicit generalised interpolation material point method for large deformations, *Comput. Struct.*, 190, 108–125, 2017.

Coombs, W. M. and Augarde, C. E.: AMPLE: A Material Point Learning Environment, *Adv. Eng. Softw.*, 139, 102748, <https://doi.org/10.1016/j.advengsoft.2019.102748>, 2020.

Coombs, W. M., Charlton, T. J., Cortis, M., and Augarde, C. E.: Overcoming volumetric locking in material point methods, *Comput. Method. Appl. Mechan.*, 333, 1–21, 2018.

Coombs, W. M., Augarde, C. E., Brennan, A. J., Brown, M. J., Charlton, T. J., Knappett, J. A., Motlagh, Y. G., and Wang, L.: On Lagrangian mechanics and the implicit material point method for large deformation elastoplasticity, *Comput. Method. Appl. Mechan.*, 358, 112622, <https://doi.org/10.1016/j.cma.2019.112622>, 2020.

Cortis, M., Coombs, W., Augarde, C., Brown, M., Brennan, A., and Robinson, S.: Imposition of essential boundary conditions in the material point method, *Int. J. Num. Method.*, 113, 130–152, 2018.

Dabrowski, M., Krotkiewski, M., and Schmid, D.: MIL-AMIN: MATLAB-based finite element method solver for large problems, *Geochem. Geophys. Geosyst.*, 9, 4, <https://doi.org/10.1029/2007GC001719>, 2008.

Davis, T. A.: Suite Sparse, available at: <https://people.engr.tamu.edu/davis/research.html> (last access: 6 October 2020), 2013.

de Koster, P., Tielen, R., Wobbes, E., and Möller, M.: Extension of B-spline Material Point Method for unstructured triangular grids using Powell–Sabin splines, *Comput. Part. Mechan.*, 1–16, <https://doi.org/10.1007/s40571-020-00328-3>, 2020.

de Souza Neto, E. A., Peric, D., and Owen, D. R.: Computational methods for plasticity: theory and applications, John Wiley & Sons, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom, 2011.

- de Vaucorbeil, A., Nguyen, V., and Hutchinson, C.: A Total-Lagrangian Material Point Method for solid mechanics problems involving large deformations, *Computer Methods in Applied Mechanics and Engineering*, 360, <https://doi.org/10.1016/j.cma.2019.112783>, 2020.
- Dunatunga, S. and Kamrin, K.: Continuum modelling and simulation of granular flows through their many phases, *J. Fluid Mechan.*, 779, 483–513, 2015.
- Dunatunga, S. and Kamrin, K.: Continuum modeling of projectile impact and penetration in dry granular media, *J. Mechan. Phys. Solids*, 100, 45–60, 2017.
- Fern, J., Rohe, A., Soga, K., and Alonso, E.: *The Material Point Method for Geotechnical Engineering*. Boca Raton: CRC Press, <https://doi.org/10.1201/9780429028090>, 2019.
- Gan, Y., Sun, Z., Chen, Z., Zhang, X., and Liu, Y.: Enhancement of the material point method using B-spline basis functions, *Int. J. Num. Method.*, 113, 411–431, 2018.
- Gaume, J., Gast, T., Teran, J., van Herwijnen, A., and Jiang, C.: Dynamic anticrack propagation in snow, *Nat. Commun.*, 9, 1–10, 2018.
- Gaume, J., van Herwijnen, A., Gast, T., Teran, J., and Jiang, C.: Investigating the release and flow of snow avalanches at the slope-scale using a unified model based on the material point method, *Cold Reg. Sci. Technol.*, 168, 102847, <https://doi.org/10.1016/j.coldregions.2019.102847>, 2019.
- Gracia, F., Villard, P., and Richefeu, V.: Comparison of two numerical approaches (DEM and MPM) applied to unsteady flow, *Comput. Part. Mechan.*, 6, 591–609, 2019.
- Guilkey, J. E. and Weiss, J. A.: Implicit time integration for the material point method: Quantitative and algorithmic comparisons with the finite element method, *Int. J. Num. Method.*, 57, 1323–1338, 2003.
- Homel, M. A., Brannon, R. M., and Guilkey, J.: Controlling the onset of numerical fracture in parallelized implementations of the material point method (MPM) with convective particle domain interpolation (CPDI) domain scaling, *Int. J. Num. Method.*, 107, 31–48, 2016.
- Huang, P., Li, S.-l., Guo, H., and Hao, Z.-m.: Large deformation failure analysis of the soil slope based on the material point method, *Comput. Geosci.*, 19, 951–963, 2015.
- Iaconeta, I., Larese, A., Rossi, R., and Guo, Z.: Comparison of a material point method and a galerkin meshfree method for the simulation of cohesive-frictional materials, *Materials*, 10, 1150, 2017.
- Leavy, R., Guilkey, J., Phung, B., Spear, A., and Brannon, R.: A convected-particle tetrahedron interpolation technique in the material-point method for the mesoscale modeling of ceramics, *Comput. Mechan.*, 64, 563–583, 2019.
- Moler, C.: MATLAB Incorporates LAPACK, available at: <https://ch.mathworks.com/de/company/newsletters/articles/matlab-incorporates-lapack.html?refresh=true> (last access: 6 October 2020), 2000.
- Nairn, J. A.: Material point method calculations with explicit cracks, *Comput. Model. Eng. Sci.*, 4, 649–664, 2003.
- Ni, R. and Zhang, X.: A precise critical time step formula for the explicit material point method, *Int. J. Num. Method.*, 121, 4989–5016, 2020.
- O’Sullivan, S., Bird, R. E., Coombs, W. M., and Giani, S.: Rapid non-linear finite element analysis of continuous and discontinuous galerkin methods in matlab, *Comput. Mathe. Appl.*, 78, 3007–3026, 2019.
- Sadeghirad, A., Brannon, R. M., and Burghardt, J.: A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations, *Int. J. Num. Method.*, 86, 1435–1456, 2011.
- Sadeghirad, A., Brannon, R., and Guilkey, J.: Second-order convected particle domain interpolation (CPDI2) with enrichment for weak discontinuities at material interfaces, *Int. J. Num. Method.*, 95, 928–952, 2013.
- Simpson, G.: *Practical finite element modeling in earth science using matlab*, Wiley Online Library, 2017.
- Sinaie, S., Nguyen, V. P., Nguyen, C. T., and Bordas, S.: Programming the material point method in Julia, *Adv. Eng. Softw.*, 105, 17–29, 2017.
- Steffen, M., Kirby, R. M., and Berzins, M.: Analysis and reduction of quadrature errors in the material point method (MPM), *Int. J. Num. Method.*, 76, 922–948, 2008a.
- Steffen, M., Wallstedt, P., Guilkey, J., Kirby, R., and Berzins, M.: Examination and analysis of implementation choices within the material point method (MPM), *Comput. Model. Eng. Sci.*, 31, 107–127, 2008b.
- Stomakhin, A., Schroeder, C., Chai, L., Teran, J., and Selle, A.: A material point method for snow simulation, *ACM Transactions on Graphics (TOG)*, 32, 1–10, 2013.
- Sulsky, D., Chen, Z., and Schreyer, H. L.: A particle method for history-dependent materials, *Comput. Method. Appl. Mechan. Eng.*, 118, 179–196, 1994.
- Sulsky, D., Zhou, S.-J., and Schreyer, H. L.: Application of a particle-in-cell method to solid mechanics, *Comput. Phys. Commun.*, 87, 236–252, 1995.
- Vardon, P. J., Wang, B., and Hicks, M. A.: Slope failure simulations with MPM, *J. Hydrodynam.*, 29, 445–451, 2017.
- Vermeer, P. A. and De Borst, R.: Non-associated plasticity for soils, concrete and rock, *HERON*, 29, 163–196, 1984.
- Wallstedt, P. C. and Guilkey, J.: An evaluation of explicit time integration schemes for use with the generalized interpolation material point method, *J. Computat. Phys.*, 227, 9628–9642, 2008.
- Wang, B., Hicks, M., and Vardon, P.: Slope failure analysis using the random material point method, *Géotech. Lett.* 6, 113–118, 2016a.
- Wang, B., Vardon, P., and Hicks, M.: Investigation of retrogressive and progressive slope failure mechanisms using the material point method, *Comput. Geotech.*, 78, 88–98, 2016b.
- Wang, B., Vardon, P. J., Hicks, M. A., and Chen, Z.: Development of an implicit material point method for geotechnical applications, *Comput. Geotech.*, 71, 159–167, 2016c.
- Wang, L., Coombs, W. M., Augarde, C., Cortis, E. M., Charlton, T. J., Brown, M. J., Knappett, J., Brennan, A., Davidson, C., Richards, and Blake, D. A.: On the use of domain-based material point methods for problems involving large distortion, *Comput. Method. Appl. Mechan. Eng.*, 355, 1003–1025, 2019.
- Więckowski, Z.: The material point method in large strain engineering problems, *Comput. Method. Appl. Mechan. Eng.*, 193, 4417–4438, 2004.
- Wyser, E., Alkhimenkov, Y., Jayboyedoff, M., and Podladchikov, Y.: fMPMM-solver, Zenodo, <https://doi.org/10.5281/zenodo.4068585>, 2020a.

Wyser, E., Alkhimenkov, Y., Jayboyedoff, M., and Podladchikov, Y.: fMPMM, available at: <https://bitbucket.org/ewyser/fmpmm-solver/src/master/>, last access: 6 October 2020.

York, A. R., Sulsky, D., and Schreyer, H. L.: The material point method for simulation of thin membranes, *Int. J. Num. Method.*, 44, 1429–1456, 1999.

Zhang, X., Chen, Z., and Liu, Y.: The material point method: a continuum-based particle method for extreme loading cases, Academic Press, ©2017 Tsinghua University Press Limited, Elsevier Inc., 2016.