Programming the material point method in Julia

Article *in* Advances in Engineering Software · January 2017

DOI: 10.1016/j.advengsoft.2017.01.008

CITATIONS
23

READS
7,931

4 authors:

Sina Sinaie
University of Melbourne
21 PUBLICATIONS **810** CITATIONS
SEE PROFILE

Vinh Phu Nguyen
Monash University (Australia)
128 PUBLICATIONS **5,814** CITATIONS
SEE PROFILE

Chi Thanh Nguyen
Delft University of Technology
18 PUBLICATIONS **694** CITATIONS
SEE PROFILE

Stéphane Pierre Alain Bordas
University of Luxembourg
434 PUBLICATIONS **17,244** CITATIONS
SEE PROFILE

Programming the material point method in Julia

Sina Sinaie[a,1], Vinh Phu Nguyen [a,2,*], Chi Thanh Nguyen[a,3], Stephane Bordas[b,4]

[a] *Department of Civil Engineering, Monash University, Clayton, Victoria 3800, Australia*
[b] *Faculté des Sciences, de la Technologie et de la Communication, University of Luxembourg, 6, rue Richard Coudenhove-Kalergi, 1359 , Luxembourg City, Luxembourg*

Abstract

This article presents the implementation of the material point method (MPM) using Julia. Julia is an open source, multi-platform, high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to Matlab and Python programmers. MPM is a hybrid particle-grid approach that combines the advantages of Eulerian and Lagrangian methods and is suitable for complex solid mechanics problems involving contact, impact and large deformations. We will show that a Julia based MPM code, which is short, compact and readable and uses only Julia built in features, performs much better (with speed up of up to 8) than a similar Matlab based MPM code for large strain solid mechanics simulations. We share our experiences of implementing MPM in Julia and demonstrate that Julis is a very interesting platform for rapid development in the field of scientific computing.

Keywords: Julia, material point method (MPM), high-performance dynamic programming language, technical computing

1. Introduction

Many researchers, including us, today do their day-to-day work in dynamic languages such as Matlab [1], Python [2], Mathmematica [3]. The reasons are several: (i) these languages are easy to use, (ii) they provide a friendly user interface that integrates computing and graphics into one single platform, (iii) they are ideal for rapid prototyping and (iv) they can be used perfectly for educational purposes. However they are usually slow and not suitable for computationally intensive problems as static languages such as Fortran and C/C++. The 'ideal' programming language, from the point of view of a researcher, is the one that is as easy to use as Matlab/Python and as fast as Fortran/C++ so that time would be spent on testing new scientific ideas rather than on studying difficult programming topics and code optimization techniques. In the search for such a language, Julia was created in 2012 [4, 5]. Julia is designed to be easy and fast thanks to the LLVM-based just-in-time (JIT) compilation [6]. In other words, with Julia, one can have machine performance without sacrificing human convenience [4]. The aim of this article is, within the context of computational solid mechanics, to test the high performance capacity of Julia as reported in [7] by comparing it with Matlab–a very popular dynamic language for scientic computing. To this end, we implement the Material Point Method (MPM) for large deformation solid mechanics in Julia and Matlab. Different aspects of using Julia for implementing MPM such as vectorized vs de-vectorized codes, efficient use of composite types and the choice of concrete types over abstract types etc. are discussed. Many interesting features of Julia including multiple dispatch, generic programming, easy use of existing Fortran/C routines are, however, not used in our code as we aimed for a simple, easy to understand Julia code that is relatively

*Corresponding author
[1] sina.sinaie@monash.edu
[2] phu.nguyen@monash.edu
[3] chi.t.nguyen@monash.edu
[4] stephane.bordas@alum.northwestern.edu

similar to our Matlab code for a fair comparison. There are a few attempts in comparing the performance of Julia finite element codes with C++ codes. For example, in [8], a simple finite element code in Julia for solving the two dimensional Poisson equation over a unit square was presened with performance as good as the C++ FEM code FEniCS [9] for N–the number of elements– ranging from 10^2 to 10^6.

MPM is a hybrid grid/particle method in which a continuum body is discretized by a finite set of Lagrangian material points (or particles) in the original configuration that is overlaid over a background grid [10, 11, 12]. The terms particle and material point will be used interchangeably throughout this manuscript. Since each material point contains a fixed amount of mass during the course of the simulation, mass conservation is automatically satisfied. In MPM, the space where the simulated bodies might occupy is discretized by a finite element grid where the equation of balance of momentum is solved. This is in contrast to other meshfree/particle methods where the momentum equations are solved directly on the particles. In other words, particles do not interact with each other directly, but rather the particle information is accumulated onto the grid, where the equations of motion are integrated over time. This avoids the time-consuming neighbor search in the evaluation of meshfree approximants/interpolants using Eulerian kernels [13, 14, 15, 16, 17, 18, 19]. The particles interact with other particles in the same body, with other solid bodies, or with fluids through a background grid. Most often, a fixed regular Cartesian grid is used throughout the simulation. This is the key difference between MPM and the updated Lagrange FEM and makes the method distortion-free. MPM has found applications in large strain problems including landslide [20], silo discharging [21, 22, 23], anchor pull-out [24], large deformations of saturated porous media [25, 26, 27, 28]. In the context of contact mechanics, [29, 30] used their contact model [31, 32] to model the compression of foam micro-structures to full densification. Applications of the MPM are emerging in sea ice models for climate simulation [33] as well as more traditional explosive-related simulations e.g., explosive welding [34], cutting processes [35], high melting explosive with cavities [36], blast and fragmentation [37, 38, 39], high strain rate penetration problems [40] and biomechanics [41].

The results reported in this article demonstrate that a Julia based MPM code, which is short, compact, readable and uses only Julia standard library, performs much better (with speed up of up to 8) than a similar Matlab based MPM code. For a high velocity impact simulation that involves 15000 particles, the Julia code took about half an hour whereas the Matlab code completed in 5 hours. This plus the fact that Julia is open source with a growing user/developer community[5] makes Julia a very promising language for scientific computing, particularly for rapid prototyping of complex scientific algorithms. As MPM is similar to explicit updated Lagrangian finite elements [42], our finding on the superior performance of Julia codes carries over to explicit dynamics simulations using finite elements.

The remainder of the paper is organized as follows. MPM is briefly presented in Section 2 followed by Section 3 which is devoted to the Julia implementation of MPM. Numerical examples are presented in Section 4, followed by concluding remarks in the last section.

2. A brief introduction to the material point method

This section gives a brief introduction to MPM for solid mechanics. We refer to [12] for a comprehensive treatment of MPM and to the note [43] for beginners where implementation details with Matlab codes are presented. The discussion is confined to explicit time integration as it is the most widely used time integrator in MPM. The space possibly occupied by the solid under consideration is discretized by a Cartesian grid and the solid is represented by a set of material points as illustrated in Fig. 1 for two dimensional problems. Subscript p refers to 'particle' whereas subscript I to the 'grid nodes'.

2.1. General procedure

An MPM computational cycle consists of three phases as illustrated in Fig. 2, (i) initialization phase where particle data are mapped (or projected) to the grid as the grid does not carry permanent data, (ii) Lagrangian phase in which the momentum equation is solved and (iii) convective phase where the grid is

[5] As the time of this writing there is 1064 registered packages at http://pkg.julialang.org.

reset. Algorithmically, an MPM cycle from step t to step $t+\Delta t$ proceeds as follows (steps a – c) according to the *update stress last* scheme [12] and note that with a slight change, the central difference time integration scheme can be included [44, 45].

(a) Particle to node mapping

$$\begin{aligned}
m_I^t &= \sum_p \phi_I(\mathbf{x}_p^t) M_p && \text{mass} \\
(m\mathbf{v})_I^t &= \sum_p \phi_I(\mathbf{x}_p^t)(M\mathbf{v})_p^t && \text{momentum} \\
\mathbf{f}_I^{\text{ext},t} &= \sum_p \phi_I(\mathbf{x}_p^t) M_p \mathbf{b} && \text{external force} \\
\mathbf{f}_I^{\text{int},t} &= \sum_p -V_p^t \boldsymbol{\sigma}_p^t \nabla\phi_I(\mathbf{x}_p^t) && \text{internal force}
\end{aligned} \tag{1}$$

(b) Update nodal momenta and fix Dirichlet nodes

$$\begin{aligned}
&(m\mathbf{v})_I^{t+\Delta t} = (m\mathbf{v})_I^t + \mathbf{f}_I^t \Delta t && \text{momentum} \\
&\begin{cases} (m\mathbf{v})_I^{t+\Delta t} = \mathbf{0} \\ \mathbf{f}_I^t = \mathbf{0} \end{cases} && \text{for fixed nodes } I
\end{aligned} \tag{2}$$

where $\mathbf{f}_I^t = \mathbf{f}_I^{\text{ext},t} + \mathbf{f}_I^{\text{int},t}$.

(c) Update particle positions, velocities and stresses

$$\begin{aligned}
\mathbf{v}_p^{t+\Delta t} &= \mathbf{v}_p^t + \Delta t \sum_I \frac{\phi_I(\mathbf{x}_p^t)\mathbf{f}_I^t}{m_I^t} && \text{velocity} \\
\mathbf{x}_p^{t+\Delta t} &= \mathbf{x}_p^t + \Delta t \sum_I \frac{\phi_I(\mathbf{x}_p^t)(m\mathbf{v})_I^{t+\Delta t}}{m_I^t} && \text{position} \\
\mathbf{v}_I^{t+\Delta t} &= \frac{(m\mathbf{v})_I^{t+\Delta t}}{m_I^t} && \text{nodal velocity} \\
\mathbf{L}_p^{t+\Delta t} &= \sum_I \nabla\phi_I(\mathbf{x}_p^t)\mathbf{v}_I^{t+\Delta t} && \text{gradient velocity} \\
\mathbf{F}_p^{t+\Delta t} &= (\mathbf{I} + \mathbf{L}_p^{t+\Delta t}\Delta t)\mathbf{F}_p^t && \text{gradient deformation} \\
V_p^{t+\Delta t} &= \det \mathbf{F}_p^{t+\Delta t} V_p^0 && \text{volume} \\
\boldsymbol{\sigma}_p^{t+\Delta t} &= \boldsymbol{\sigma}_p^t + \Delta\boldsymbol{\sigma}_p && \text{stress}
\end{aligned} \tag{3}$$

where $M_p, \mathbf{x}_p, \mathbf{v}_p, \boldsymbol{\sigma}_p, V_p, \mathbf{L}_p, \mathbf{F}_p$ are the particle mass, positions, velocities, Cauchy stress[6], volume, gradient velocity tensor and deformation gradient, respectively. Body forces are denoted by $\mathbf{b}$ and the time step is denoted by Δt. The weighting and gradient weighting functions are designated by ϕ_I and $\nabla\phi_I$, respectively. These functions are kept abstract for the moment as their forms depend on which MPM formulation is being used. Note that the grid node positions were not updated.

[6] Or an objective stress rate such as Jaumann or Truesdell rate can be used here.

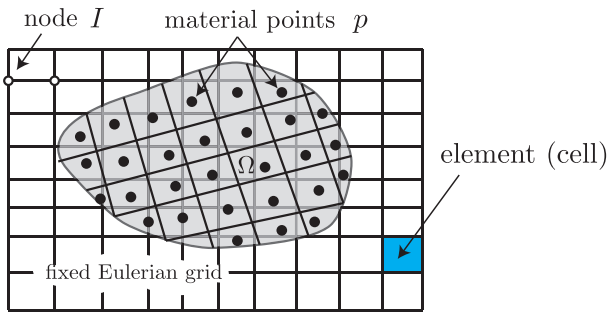

Figure 1: Material point method: Lagrangian material points overlaid on a Eulerian grid.

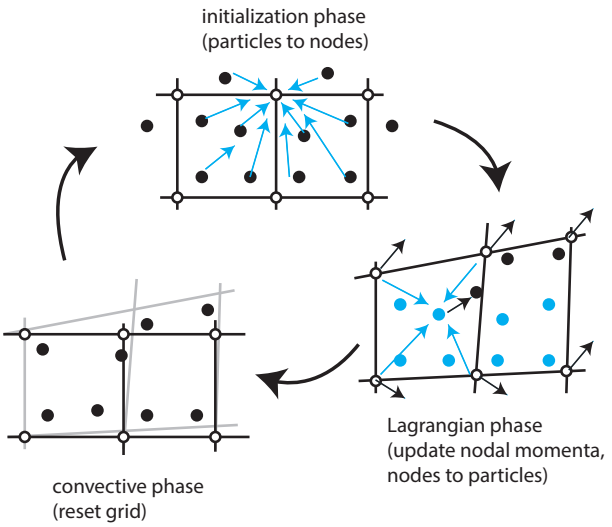

Figure 2: Material point method: a computational step consists of three phases: (1) initialization phase where information is mapped from particles to nodes, (2) momentum equations are solved for the nodes, the updated nodes are then mapped back to the particles to update their positions and velocities and (3) convective phase where the grid is reset.

2.2. Generalized interpolation material point method

In GIMP[7] the weighting and gradient weighting functions $\phi_I(\mathbf{x}_p)$ and $\nabla\phi_I(\mathbf{x}_p)$ are given by [46].

$$\begin{aligned}\phi_{Ip} \equiv \phi_I(\mathbf{x}_p) &= \frac{1}{V_p}\int_{\Omega_p} \chi(\mathbf{x}-\mathbf{x}_p) N_I(\mathbf{x})\mathrm{d}\Omega \\ \nabla\phi_{Ip} \equiv \nabla\phi_I(\mathbf{x}_p) &= \frac{1}{V_p}\int_{\Omega_p} \chi(\mathbf{x}-\mathbf{x}_p) \nabla N_I(\mathbf{x})\mathrm{d}\Omega\end{aligned} \tag{4}$$

where N_I represents the standard grid functions, which form a Partition of Unity (PoU), and $\chi(\mathbf{x})$ is the *particle characteristic function*. Note that short notation ϕ_{Ip} is used to represent $\phi_I(\mathbf{x}_p)$.

Typically, piece-wise constant particle characteristic functions, also known as top-hat functions, are used

$$\chi_p(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \Omega_p \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

which result in the following GIMP weighting functions (simplification of Equation (4))

$$\begin{aligned}\phi_{Ip} &= \frac{1}{V_p}\int_{\Omega_p} N_I(\mathbf{x})\mathrm{d}\Omega \\ \nabla\phi_{Ip} &= \frac{1}{V_p}\int_{\Omega_p} \nabla N_I(\mathbf{x})\mathrm{d}\Omega\end{aligned} \tag{6}$$

How the integrals in Equation (6) are evaluated and how the particle domains Ω_p are defined/updated result in various GIMP formulations. Fig. 3 illustrates existing GIMP methods to be discussed in what follows.

In the approach referred to as uGIMP (unchanged GIMP) Ω_p is a rectangle in 2D and is kept fixed. The integrals in Equation (6) can thus be exactly integrated resulting in analytical expressions for ϕ_{Ip} and $\nabla\phi_{Ip}$ [46]. A more complicated approach, known as cpGIMP (contiguous particle GIMP), updates the particle domain using the deformation gradient $\mathbf{F}$ without taking shear deformation into account. Even though cell-crossing noise is mitigated, as the material deforms, the particle domains cannot fill the material space under general loading conditions. Convected Particle Domain Interpolation (CPDI) [47, 48, 49] is the next logical development of GIMP where the integrals of Equation (6) are also analytically evaluated thanks to the use of alternative basis functions and the space is tiled without gaps.

2.3. Convected particle domain interpolations

The GIMP weighting function in Equation (6) now becomes

$$\begin{aligned}\phi_{Ip} &= \frac{1}{V_p}\int_{\Omega_p} N_I^{\mathrm{alt}}(\mathbf{x})\mathrm{d}\Omega \\ &= \frac{1}{V_p}\sum_{c=1}^{4}\left[\int_{\Omega_p} M_c(\mathbf{x})\mathrm{d}\Omega\right] N_I(\mathbf{x}_c) = \sum_{c=1}^{4} w_c^f N_I(\mathbf{x}_c)\end{aligned} \tag{7}$$

and similarly the gradient $\nabla\phi_{Ip}$ is written as

$$\begin{aligned}\nabla\phi_{Ip} &= \frac{1}{V_p}\int_{\Omega_p} \nabla N_I^{\mathrm{alt}}(\mathbf{x})\mathrm{d}\Omega \\ &= \frac{1}{V_p}\sum_{c=1}^{4}\left[\int_{\Omega_p} \nabla M_c(\mathbf{x})\mathrm{d}\Omega\right] N_I(\mathbf{x}_c) = \sum_{c=1}^{4} \mathbf{w}_c^g N_I(\mathbf{x}_c)\end{aligned} \tag{8}$$

What makes CPDI an efficient GIMP is the fact that the integrals in the square brackets of Equations (7) and (8) can be exactly computed in terms of the geometry of the particle domains.

[7] GIMP is short for generalized interpolation material point.

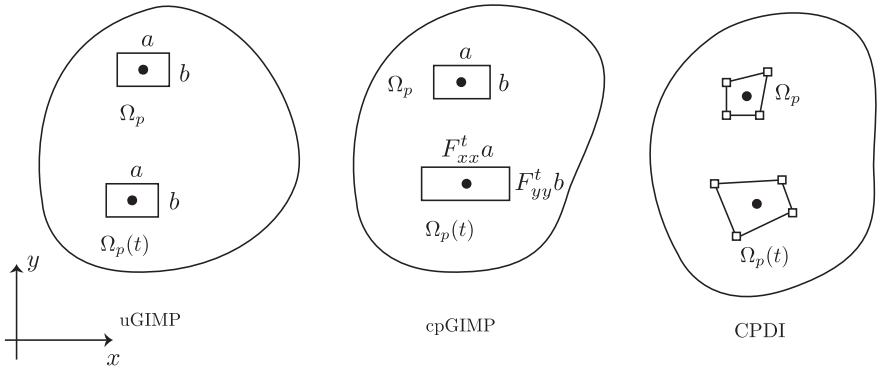

Figure 3: Tracking particle domains in GIPM: space cannot be tiled in a general multi-dimension domain using rectilinear Ω_p in uGIMP and cpGIMP. By consider each particle as a quadrilateral, in CPDI, space is filled without gaps.

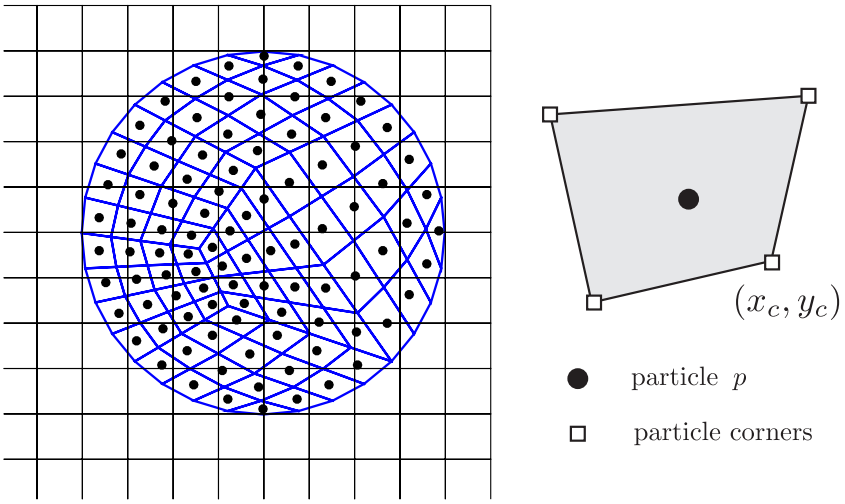

Figure 4: Two dimensional CPDI: background grid and particle domains. Each particle domain is a four-node quadrilateral. Note that particle corners do not carry any information. They are present to precisely track the particle geometries.

2.3.1. Two dimensional bilinear quadrilateral CPDI

If the particles are represented by four-node quadrilateral elements, the corresponding CPDI-Q4 weighting function and first derivatives are given by

$$\begin{aligned}\phi_{Ip} = \frac{1}{36V_p}\Big[&(4c_1+2c_2+2c_3+c_4)N_I(\mathbf{x}_1)\\ &+(2c_1+4c_2+c_3+2c_4)N_I(\mathbf{x}_2)\\ &+(c_1+2c_2+2c_3+4c_4)N_I(\mathbf{x}_3)\\ &+(2c_1+c_2+4c_3+2c_4)N_I(\mathbf{x}_4)\Big]\end{aligned} \tag{9}$$

$$\begin{aligned}\nabla\phi_{Ip} = \frac{1}{2V_p}\Big\{&N_I(\mathbf{x}_1)\begin{bmatrix}y_{24}\\x_{42}\end{bmatrix} + N_I(\mathbf{x}_2)\begin{bmatrix}y_{31}\\x_{13}\end{bmatrix}\\ &+N_I(\mathbf{x}_3)\begin{bmatrix}y_{42}\\x_{24}\end{bmatrix} + N_I(\mathbf{x}_4)\begin{bmatrix}y_{13}\\x_{31}\end{bmatrix}\Big\}\end{aligned} \tag{10}$$

where $c_1 = (x_{21}y_{41} - y_{21}x_{41})$, $c_2 = (x_{21}y_{32} - y_{21}x_{32})$, $c_3 = (x_{34}y_{41} - y_{34}x_{41})$, $c_4 = (x_{34}y_{32} - y_{34}x_{32})$ and $x_{ij} = x_i - x_j$ and $y_{ij} = y_i - y_j$ are the corner coordinate differences. The area of the particle domain is given by $V_p = 0.5[(x_1y_2 - x_2y_1) + (x_2y_3 - x_3y_2) + (x_3y_4 - x_4y_3) + (x_4y_1 - x_1y_4)]$.

2.3.2. Two dimensional constant triangular CPDI

If the particles are represented by three-node triangular elements, the corresponding CPDI-T3 weighting and gradient weighting functions are given by [50]

$$\begin{aligned}\phi_{Ip} &= \frac{1}{V_p}\left[\frac{V_p}{3}N_I(\mathbf{x}_1) + \frac{V_p}{3}N_I(\mathbf{x}_2) + \frac{V_p}{3}N_I(\mathbf{x}_3)\right]\\ \nabla\phi_{Ip} &= \frac{1}{V_p}\Big\{N_I(\mathbf{x}_1)\frac{1}{2}\begin{bmatrix}y_{23}\\x_{32}\end{bmatrix} + N_I(\mathbf{x}_2)\frac{1}{2}\begin{bmatrix}y_{31}\\x_{13}\end{bmatrix}\\ &\quad + N_I(\mathbf{x}_3)\frac{1}{2}\begin{bmatrix}y_{12}\\x_{21}\end{bmatrix}\Big\}\end{aligned} \tag{11}$$

where V_p denote the area of the particle triangle.

2.3.3. Three dimensional linear tetraheron CPDI

If the particles are represented by linear tetrahedron elements, the corresponding CPDI-Tet4 weighting and gradient weighting functions are given by [51]

$$\begin{aligned}\phi_{Ip} &= \frac{1}{4}N_I(\mathbf{x}_1) + \frac{1}{4}N_I(\mathbf{x}_2) + \frac{1}{4}N_I(\mathbf{x}_3) + \frac{1}{4}N_I(\mathbf{x}_4)\\ \nabla\phi_{Ip} &= \frac{1}{6V_p}\Big\{N_I(\mathbf{x}_1)\begin{bmatrix}a_1\\b_1\\c_1\end{bmatrix} + N_I(\mathbf{x}_2)\begin{bmatrix}a_2\\b_2\\c_2\end{bmatrix}\\ &\quad + N_I(\mathbf{x}_3)\begin{bmatrix}a_3\\b_3\\c_3\end{bmatrix} + N_I(\mathbf{x}_4)\begin{bmatrix}a_4\\b_4\\c_4\end{bmatrix}\Big\}\end{aligned} \tag{12}$$

where $6V_p = x_{21}(y_{23}z_{34} - y_{34}z_{23}) + x_{32}(y_{34}z_{12} - y_{12}z_{34}) + x_{43}(y_{12}z_{23} - y_{23}z_{12})$; $a_1 = y_{42}z_{32} - y_{32}z_{42}$, $a_2 = y_{31}z_{43} - y_{34}z_{13}$, $a_3 = y_{24}z_{14} - y_{14}z_{24}$, $a_4 = y_{13}z_{21} - y_{12}z_{31}$, $b_1 = x_{32}z_{42} - x_{42}z_{32}$, $b_2 = x_{43}z_{31} - x_{13}z_{34}$,

$b_3 = x_{14}z_{24} - x_{24}z_{14}$, $b_4 = x_{21}z_{13} - x_{31}z_{12}$, $c_1 = x_{42}y_{32} - x_{32}y_{42}$, $c_2 = x_{31}y_{43} - x_{34}y_{13}$, $c_3 = x_{24}y_{14} - x_{14}y_{24}$ and $c_4 = x_{13}y_{21} - x_{12}y_{31}$. Note that V_p is a signed quantity and a proper node numbering was used to have a positive value.

3. Julia-based implementation of MPM

This section investigates some implementation aspects of the Julia programming language. With the intention of producing codes that are optimized for speed, some processes common to an MPM implementation are addressed. In order to help the transition from Matlab to Julia, we provide a list of some syntax differences between Matlab and Julia in Table A.1. This section will present some code snippets written in Julia where **boldface** font is used for the Julia built-in features. Julia can be downloaded for free, under the MIT licence, from the web page `http://julialang.org/downloads`. There are a couple of ways to interact with Julia. First, Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the julia executable. Alternatively, Julia can be launched from a web browser using this link `https://juliabox.org`. If an IDE (integrated development environment) is preferred there is Juno at `http://junolab.org`. In some code snippets one will see *variable*::*type_name* which is to declare a variable of a certain type such as x::Int64 to indicate that x is an 64 bit integer.

3.1. Using 'for' loops versus vectorization

Performing a fixed set of operations on large arrays of data (consisting of nodal data and material point data) encompasses a significant part of an MPM implementation. In Julia, similar to Matlab, such operations can be carried out using either *for-loops* or *vectorized operators*. While vectorized operators produce condensed code, *for-loops* are considered to be more intuitive. From this point of view, the choice is subjective. On the other hand, from a performance point of view, *for-loops* tend to be faster than vectorized operators in Julia. However, this depends on how the *for-loop* is implemented.

The performance of *for-loops* and a vectorized operator is examined using the code given in Listing 1. Herein, performance is evaluated in terms of 'Iterations/run-time second' i.e., the more iterations per second the more efficient the code. For this purpose, three arrays are created and assigned with initial values, where after a summation operation is performed on the first two arrays and assigned to the third array. A simple summation has been selected, so that the run time is more representative of the iterations, rather than the calculations. Each approach is iterated 1000 times, whereby the time duration is monitored using the *tic()* and *toc()* functions. The initial values assigned to the arrays are arbitrary.

Table 1 shows the results of comparing vectorized and *for-loop* operations. The first observation made by these results is the very poor performance of abstract data types. Although Julia allows the use of abstract data types such as 'Real', which has attractive design implication towards polymorphism, it is always best to declare variables using concrete types when performance is an issue. This is readily seen by comparing the first two rows of Table 1. As a result, the 'Float64' concrete data type is used in this study.

It is observed from Table 1 that, in Julia, *for-loops* can perform better than vectorized operators. For one-dimensional arrays the speed at which for loops are carried out is more than twice the speed of vectorized operators. However, for two-dimensional arrays, in order to have noticeable advantage over vectorized operators, nested *for-loops* have to be consistent with the 'column-major' order of which data is stored in memory. In other words, the inner loop should iterate over the first index of the two-dimensional array. Such order is termed as 'j-i loop' in Listing 1 and Table 1. Albeit irrelevant here, Julia is very well suited for non-vectorized problems such as those arising from stochastic processes modeled by Monte Carlo methods.

3.2. Composite types versus arrays

Given the common use of composite types (similar to *struct* in C and can be thought of as roughly equivalent to a class without behavior in object-oriented languages such as C++), the performance of an array of such types are compared to the performance of multi-dimensional arrays. The code used to benchmark the performance of two-dimensional arrays is similar to the one presented in Listing 1 (j-i for loop) and is not repeated here. Listing 2 shows the code use to compare different methods of access to

Table 1: Performance comparison of *for-loops* against vectorized operators on large data arrays in Julia.

		Iterations/sec.		
Array size	Type	i-j loop	j-i loop	vectorized
10^6	Real	2.70	-	2.52
10^6	Float64	668.90	-	293.00
$10^6 \times 2$	Float64	55.07	296.00	158.70
$10^6 \times 10$	Float64	33.39	60.90	32.05
10×10^6	Float64	9.17	35.57	32.21

Listing 1: Julia code listing for vectorized operations versus for loops. Lines with # at the beginning are comments. Julia discourages the used of semicolons to end statements.

```
# create and assign arrays
# values of m and n defined in Table 1
A01 = Array{Float64}(m,n) # m by n 2D array of Float64 numbers
A02 = Array{Float64}(m,n)
A03 = Array{Float64}(m,n)

for i = 1:1:m
  for j = 1:1:n
   A01[i,j] =  1.0*i
   A02[i,j] = -1.0*i
   A03[i,j] = 0
   end
end

Iterations = 1000

# using for i-j loop
tic()
for count in 1:1:Iterations
 for i = 1:1:m
   for j = 1:1:n
      A03[i,j] = A01[i,j] + A02[i,j]
   end
 end
end
toc()

# using for j-i loop, should be used
tic()
for count in 1:1:Iterations
 for j = 1:1:n
  for i = 1:1:m
     A03[i,j] = A01[i,j] + A02[i,j]
  end
 end
end
toc()

# using vectorization
tic()
for count in 1:1:Iterations
 A03 = A01 + A02
end
toc()
```

the members of a composite types, the results of which are given in Table 2. These results indicate a consistent performance advantage of multi-dimensional arrays over composite types. However, Table 2 also demonstrates that removing redundant access calls to the members of a composite type improves the performance of operations on composite types (compare column 5 with columns 3 and 4).

While the performance advantage of multi-dimensional arrays as opposed to composite types is significant enough to be taken advantage of, one also has to consider the benefits derived from composite types in terms of code organization and readability. Taking this into account, composite types are used for the purpose of this paper and for the numerical examples considered in the next section. Note that our aim is not to write a highly optimized MPM code in Julia but to have a fast MPM platform for future fundamental development of the method.

Table 2: Performance comparison of accessing elements of an array as opposed to members of a composite type. Methods 1–3 refer to different methods of access specified in Listing 2.

	Iterations/sec.			
Array size	array	method 1	method 2	method 3
$10^6 \times 10$	58.51	17.85	22.15	25.36
$10^6 \times 100$	5.77	1.88	2.50	3.01

3.3. A simple MPM code in Julia

Having discussed some implementation choices for common operations in MPM, we now proceed to a simple implementation of MPM in Julia for solid mechanics. Composite types are used in this study to store the attributes of grid points and material points. The declaration of these types are presented in Listings 3 and 4. Note that whether the vectors are initialized to be 2D or 3D depends on the problem that is being solved. Next, one defines an array of grid points and array(s) of particles. We refer to Listing 5 for an example of generating an array of particles. Listing 6 presents the implementation of a 2D MPM for elastic bodies. Extension to 3D and inelastic materials is straightforward. Note that for simplicity only constant time steps are coded but consideration of adaptive time steps should not pose any difficulties. The implementation of the CPDI-Tet4 for a Neo-Hookean hyperelastic material is given in Listing 7. Note that the implementation of CPDI-Q4 and CPDI-T3 is similar and they are programmed in our code even though not used in the reported examples. The code organization is discussed in Appendix B. We would like to emphasize that this is just our experimental MPM code in Julia.

4. Numerical examples

This section presents some numerical examples demonstrating the performance advantage of Julia as a development tool for MPM simulations. All simulations are carried out using in-house MPM codes written in Matlab and Julia. The Matlab codes are described in [43]. In Julia, just like in Python, accessing external packages is very simple. Most of the images presented in the following text, are created in Julia using the PyPlot graphical package [52].

There are three primary variants affecting the accuracy of MPM solutions: the time-step size, the grid density and the particle density. Decreasing the time-step size will increase the number of time-integration steps needed to complete the analysis, and hence, will increase run-time. Given the trivial relation between time-step size and run-time, i.e. a linear scaling between the two, there is no need to examine it here. However, what is worth examining is the scaling of run-time with grid and particle density. In order to eliminate the effect of time-step size in this process, performance is evaluated in terms of 'Iterations/run-time seconds'. Note that each iteration constitutes a single loop of the analysis as described in Section 2, and run-time refers to the actual time it takes to complete the analysis using a single thread on an Intel Core i7-6700 CPU with a RAM of 16 GB. We also provide the total runtime of all the conducted simulations. Julia commands used to measure the time of the following simulations are given in Appendix B.

Listing 2: Julia code listing for performance testing of operations on composite types.

```
# declaration of a user defined type, named 'DataStructure'
# with one member named M which is 1-dimensional array
# in what follows, values of n and m varied following Table 2
type DataStructure
   M :: Array{Float64} # the symbol :: used to define a TYPE
   function DataStructure()
      new(zeros(n))
   end
end

# create and assign type members
S01 = Array{DataStructure}(m)
S02 = Array{DataStructure}(m)
S03 = Array{DataStructure}(m)
# the following can be simplied: for i=1:m
for i = 1:1:m
   S01[i] = DataStructure()
   S02[i] = DataStructure()
   S03[i] = DataStructure()
   for j = 1:1:n
      S01[i].M[j] =  1.0*i
      S02[i].M[j] = -1.0*i
      S03[i].M[j] = 0
   end
end

# method 1, using composite type object directly
tic()
for count in 1:1:iIterations
   for i = 1:1:m
      for j = 1:1:n
         S03[i].M[j] = S01[i].M[j] + S02[i].M[j]
      end
   end
end
toc()

# method 2, using reference to composite type objects
tic()
for count in 1:1:iIterations
   for i = 1:1:m
     this01 = S01[i]
     this02 = S02[i]
     this03 = S03[i]
     for j = 1:1:n
      this03.M[j] = this01.M[j] + this02.M[j]
     end
   end
end
toc()

# method 3, using reference to the member of composite type objects
# method of choice
tic()
for count in 1:1:iIterations
   for i = 1:1:m
      this01 = S01[i].M
      this02 = S02[i].M
      this03 = S03[i].M
      for j = 1:1:n
         this03[j] = this01[j] + this02[j]
      end
   end
end
toc()
```

Listing 3: Julia code listing for the declaration of 'Grid Point' composite type.

```
# composite type declaration for Grid Point
type GridPoint
    Fixed     :: Array{Bool}     # fixation in different directions
    Mass      :: Float64
    Position  :: Array{Float64}
    Momentum  :: Array{Float64}
    Force     :: Array{Float64}  # total force=internal force+external force

    function GridPoint()
       # constructor, set all values to zero, the following is for 2D problems
         new (
              [false;false],
              0.0,
              zeros(2),
              zeros(2),
              zeros(2)
             );
    end
 end
```

Listing 4: Julia code listing for the declaration of 'Material Point' composite type.

```
# composite type declaration for Material Point
type MaterialPoint
   Mass            :: Float64
   VolumeInitial   :: Float64
   Volume          :: Float64
   Centroid        :: Array{Float64}
   Velocity        :: Array{Float64}
   Momentum        :: Array{Float64}
   ExternalForce   :: Array{Float64}
   DeformationGradient::Array{Float64}
  # strain/stress as vectors (Voigt notation)
   Stress          :: Array{Float64}
   Strain          :: Array{Float64}
   PlasticStrain   :: Array{Float64}
   Alpha           :: Float64 # equivalent internal plastic parameter

   ElasticModulus:: Float64
   PoissonRatio   :: Float64
   YieldStress    :: Float64
    # the following  for CPDIs
   Corner          :: Array{Float64}

   function MaterialPoint()
      # constructor, set all above variables to zero
   end
end
```

Listing 5: Julia code listing for generation of material points.

```
thisMaterialDomain          = Array{MaterialPoint}(0)
# repeat the following until the desired number of particles is obtained
thisMaterialPoint           = MaterialPoint()
thisMaterialPoint.Centroid = [1.0; 1.0]
# appending to the end of arrays, similar to push_back in C++ STL
push!(thisMaterialDomain, thisMaterialPoint)
```

Listing 6: Time-integration loop for standard MPM implementation. All grid points and material points are stored into arrays denoted by allGP and allMP, respectively. GP: Grid Point, MP: Material Point, AGP: Adjacent Grid Point. The prefix 'this' indicates the reference of the object currently being processed.

```
for Time = 0:dT:TimeEnd
# reset grid ------------------------------------------------------------------
for indexGP = 1:size(allGP)
  allGP[indexGP].Mass = 0.0
  allGP[indexGP].Momentum = [0.0; 0.0]
  allGP[indexGP].Velocity = [0.0; 0.0]
end
# material point to grid ------------------------------------------------------
for indexMP = 1:size(allMP)
  thisMP = allMP[indexMP]
  allAGP = getAGP(thisMP, allGP)
  for indexAGP = 1:size(allAGP)
     thisAGP  = allAGP[indexAGP]
     # shape function value and gradient
     BasisValue         = getBasisValue(thisMP, thisAGP)
     BasisGradient      = getBasisGradient(thisMP, thisAGP)
     thisAGP.Mass      += BasisValue*thisMP.Mass
     thisAGP.Momentum  += BasisValue*thisMP.Mass*thisMP.Velocity
     thisAGP.Force     += BasisValue*thisMP.Force-BasisGradient*thisMP.Stress*thisMP.Volume
  end
# update grid -----------------------------------------------------------------
for indexGP = 1:size(allGP)
  thisGP = allGP[indexGP]
  thisGP.Momentum += thisGP.Force * dT
  # apply boundary conditions with momentum = 0 and force = 0
end
# grid to material point ------------------------------------------------------
for indexMP = 1:size(allMP)
   thisMP = allMP[indexMP]
   allAGP = getAGP(thisMP, allGP)
   DeformationGradientIncrement = [1 0; 0 1]
   for indexAGP = 1:size(allAGP)
      thisAGP  = allAGP[indexAGP]
      # shape function value and gradient
      BasisValue    = getBasisValue(thisMP, thisAGP)
      BasisGradient = getBasisGradient(thisMP, thisAGP)
      if(thisAGP.mass > 1.0e-8)
         velocity           = thisAGP.Momentum / thisAGP.Mass
         thisMP.Velocity   += BasisValue * thisAGP.Force/thisAGP.Mass * dT
         PositionIncrement += BasisValue * thisAGP.Momentum/thisAGP.Mass * dT
      end
      DeformationGradientIncrement += velocity * BasisGradient' * dT
   end
   thisMP.Position  += PositionIncrement
   thisMP.Strain[1] += DeformationGradientIncrement[1,1] - 1.0
   thisMP.Strain[2] += DeformationGradientIncrement[1,1] - 1.0
   thisMP.Strain[3] += DeformationGradientIncrement[1,2] + DeformationGradient[2,1]
   thisMP.Stress = thisMP.StiffnessMatrix * thisMP.Strain
   thisMP.DeformationGradient = DeformationGradientIncrement * thisMP.DeformationGradient
   thisMP.Volume = det(thisMP.DeformationGradient) * thisMP.InitialVolume
 end
end
```

Listing 7: Time-integration loop for CPDI implementation. All grid points and material points are stored into arrays denoted by allGP and allMP, respectively. GP: Grid Point, MP: Material Point, AGP: Adjacent Grid Point. The prefix 'this' indicates the reference of the object currently being processed.

```
for Time = 0.0:dT:TimeEnd
  # reset grid ------------------------------------------------------------
  for indexGP in 1:1:thisGrid.iNodes
      allGP[indexGP].Mass = 0.0
      allGP[indexGP].Momentum = [0.0; 0.0; 0.0]
      allGP[indexGP].Force = [0.0; 0.0; 0.0]
  end
  # material to grid ------------------------------------------------------
  for indexMP = 1:1:length(allMP)
      thisMP = allMP[indexMP]
      allGP  = getAGP(thisMP,allGP)
      for indexAGP = 1:1:size(allAGP)
          thisAGP = allGP[indexAGP]
          BasisValue         = getBasisValue(thisMP,thisAGP)
          BasisGradient      = getBasisGradient(thisMP,thisAGP)
          thisAGP.Mass      += BasisValue*thisMP.Mass
          thisAGP.Momentum  += BasisValue*thisMP.Momentum
          thisAGP.Force     += BasisValue*thisMP.Force-BasisGradient*thisMP.Stress*thisMP.Volume
      end
  end
  # update grid momentum and apply boundary conditions, similar to standard MPM

  # grid to material points -----------------------------------------------
  for indexMP = 1:1:size(allMP)
      thisMP = allMP[indexMP]
      allAGP = getAGP(thisMP, allGP)
      DeformationGradientIncrement = [0.0; 0.0; 0.0]
      for indexAGP = 1:1:size(allAGP)
          thisAGP        = allGP[indexAGP]
          BasisValue     = getBasisValue(thisMP, thisAGP)
          BasisGradient = getBasisGradient(thisMP, thisAGP)
          if(thisAGP.Mass > 1.0e-16)
              velocity  = thisAGP.Momentum / thisAGP.Mass
              thisMP.Velocity += (BasisValue * thisGP.Force / thisGP.Mass) * dT
          end
          DeformationGradientIncrement += velocity * BasisGradient' * dT;
      end
      thisMP.DeformationGradient = DeformationGradientIncrement * thisMP.DeformationGradient
      # for a Neo_Hookean model
      E      = thisMP.ElasticModulus
      Nu     = thisMP.PoissonRatio
      Lame1 = 0.5*E / (1.0 + Nu)
      Lame2 = Nu*E / ((1+Nu)*(1.0-2.0*Nu))
      F      = thisMP.DeformationGradient
      J      = det(F)
      thisMP.Stress    = Lame2*log(J)/J * eye(3) + Lame1/J*(F*F'-eye(3))
      thisMP.Momentum = thisMP.Velocity * thisMP.Mass
  end
  # update corner positions
  for indexMP = 1:1:size(allMP)
      thisMP = allMP[indexMP]
      for indexCorner = 1:1:4
          thisCorner       = thisMP.Corner[:, indexCorner]
          CornerIncrement = [0.0; 0.0; 0.0]
          thisAGP          = getAGP(thisCorner, allGP)
          for indexAGP = 1:1:length(thisAGP)
              thisAGP     = allGP[indexAGP]
              BasisValue = getBasisValue_Classic(thisMP, thisAGP)
              if(thisGridPoint.fMass > 1.0e-16)
                  velocity = thisAGP.Momentum / thisAGP.Mass
              end
              CornerIncrement += BasisValue * velocity * dT
          end
          thisMP.Corner[:, indexCorner] += CornerIncrement
      end
      thisMP.Volume = det(thisMP.DeformationGradient) * thisMP.VolumeInitial
  end
end
```

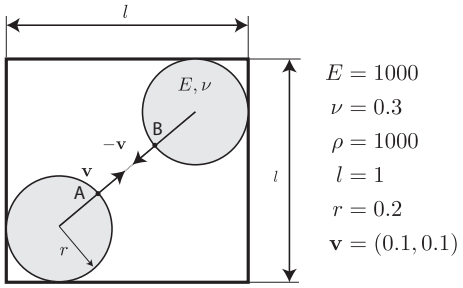

Figure 5: Geometry and initial conditions for the impact of two elastic bodies (units in N and mm).

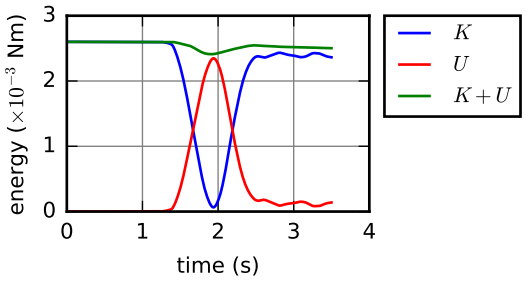

Figure 6: Variation of kinetic energy (K) and strain energy (U) over the time for the impact of two elastic bodies.

4.1. Impact of two elastic bodies

Fig. 5 shows the problem of impact between two identical elastic disks moving towards each other [10]. The computational domain is a square, of which side is 1 mm and is discretized into a mesh of 20×20 elements. The radius of the disks is 0.2 mm. A plane strain condition is assumed and simulations are performed in the absence of gravity. This problem serves to verify the MPM codes developed for this study. It is also used to evaluate the performance of Matlab and Julia codes and how they scale as the number of particles and grid points increases.

The 20×20 mesh and the particle distribution are shown in Fig. 7. In this case, each disk consists of 208 particles placed at regular intervals to create the circular domains. The initial condition for this problem is the initial velocities of the particles, $\mathbf{v}_p = \mathbf{v}$ for lower-left particles and $\mathbf{v}_p = -\mathbf{v}$ for upper-right particles. No boundary conditions are required here since the simulation is set to stop before the particles move out of the computational box after impact. A constant time step of $\Delta t = 0.001$ s is used over the entire simulation. In order to avoid numerical problems caused by division by very small numbers, a simple cutoff algorithm is used to detect small nodal masses [12]. A cutoff value of 10^{-8} is used in this case.

In order to check for energy conservation in this problem, the values of strain energy and kinetic energy are computed at each time step. These are defined as

$$U = \sum_{p=1}^{n_p} u_p V_p, \quad K = \frac{1}{2} \sum_{p=1}^{n_p} \mathbf{v}_p \cdot \mathbf{v}_p M_p \tag{13}$$

where $u_p = 1/2 \sigma_{p,ij} \epsilon_{p,ij}$ denotes the strain energy density of particle p, and is explicitly calculated by

$$u_p = \frac{1}{4\mu} \left[\frac{\kappa + 1}{4} (\sigma_{p,xx}^2 + \sigma_{p,yy}^2) - 2(\sigma_{p,xx}\sigma_{p,yy} - \sigma_{p,xy}^2) \right] \tag{14}$$

with μ and κ representing the shear modulus and the Kolosov constant, respectively.

The movement of the two disks and their impact are illustrated in Fig. 5. The collision occurs in a physically realistic fashion, although no contact law has been specified. Fig. 6 plots the evolution of the

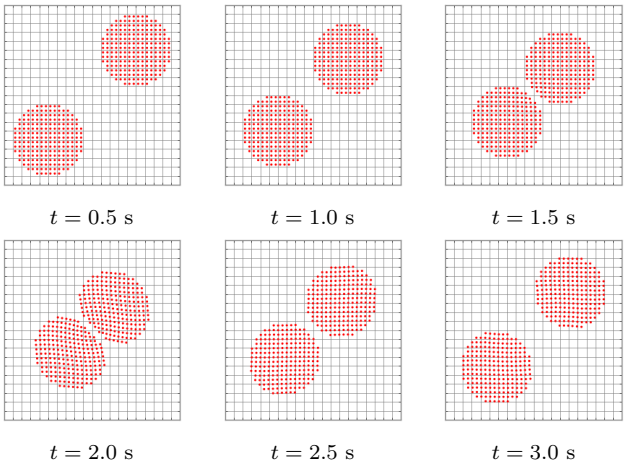

Figure 7: Snapshots for the impact of two elastic bodies before, during and after impact. These images were created using the PyPlot graphical package [52].

kinetic (K), strain (U) and total energy ($K + U$). All of the initial energy is kinetic energy and equal to $K = 2 \times (0.5 \times (v^2 + v^2) \times \rho \times \pi \times r^2) = 2.513$. The kinetic energy decreases during impact, but is mostly recovered upon separation. The strain energy reaches its maximum value at the point of maximum deformation during impact and then decreases to a value associated with free vibration of the disk. The result is consistent to the ones reported in [10, 53, 54] which confirms the implementation. However the contact does occur earlier than it should. The correct contact time is $t = AB/(2\sqrt{2}v) = 1.59$ s where $v = 0.1$. In the simulation, contact happened at $t = 1.24$ s. This result is expected as the contact is resolved at the grid nodes not the particles. This means that contact is detected even when the particles of the two bodies are one cell apart. Consequently, increasing the resolution of the grid will give a more accurate estimation for the time of impact.

Table 3 shows the performance results for different cases of number of particles and number of grid points, whereby it is shown that Julia code is consistently faster than Matlab. Note that increasing the number of particles or making the grid more dense, causes the performance ratio to vary. However, for all cases, Julia code is observed to perform more than 5 times faster than Matlab. In fact, the performance ratio is rather significant, especially when it comes to simulations that can potentially take hours or days to complete. For example, the simulation given in the last row of Table 3 took 2.6 hours with the Matlab code and only 0.46 hours with the Julia code. One might argue that MEX functions[8] can be used to enhance the computational efficiency of Matlab codes. However, this technique is not considered here even though it was coded in our Matlab code as we want to employ only dynamic languages for rapid development. If needed, existing quality C/Fortran functions can be called directly by Julia, with no glue code, or boilerplate code.

Table 3: Performance comparison of Julia and Matlab code for the two-disk impact problem.

		Iterations/sec.		Total time [s]		
Particles	Grid	Matlab	Julia	Matlab	Julia	Ratio
416	20x20	20.16	132.80	148.81	22.59	6.59
1624	20x20	5.30	33.37	566.36	89.90	6.30
1624	40x40	5.07	26.45	591.25	113.42	5.21
25784	80x80	0.32	1.82	9375.00	1651.07	5.68

[8] A MEX file is an interface between MATLAB and functions written in C or Fortran. It stands for "MATLAB executable".

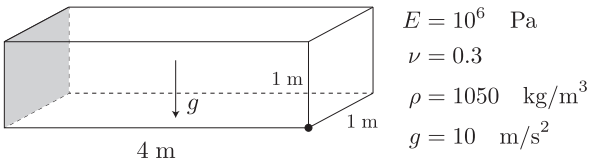

Figure 8: Compliant cantilever beam example. The left face (shaded) is fixed. The dot denotes the point of which vertical displacement is tracked in time.

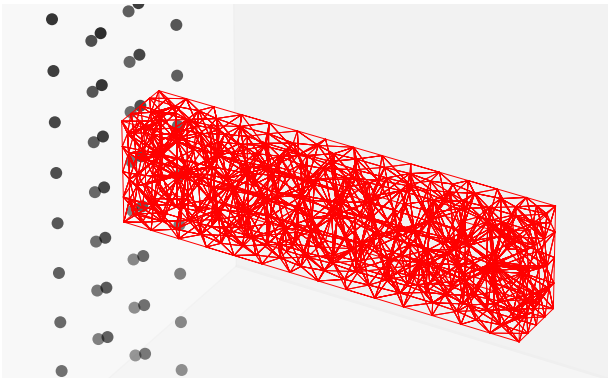

Figure 9: Cantilever beam comprising of 1041 tetrahedral elements and the grid points representing a fixed constraint (dark circles).

4.2. Cantilever beam

As another example, we consider the cantilever beam shown in Fig. 8. A Neo-Hookean constitutive model is used with $E = 1.0 \times 10^6$ Pa and $\mu = 0.3$. Large deformation vibration of this cantilever beam is induced by its own weight with the sudden activation of gravity ($g = 10$ m/s^2) at $t = 0$ s. This example is analyzed using constant time steps of $\Delta t = 0.001$ s up to $t = 3.0$ s. The problem makes use of 1041 material points and implements the CPDI-Tet4 formulation described in Section 2 as standard MPM and cpGIMP would result in numerical fractures [47, 48, 49]. Created using Gmsh [55], the material points consist of four-node tetrahedral elements, whereby the entire body is illustrated in Fig. 9. Note that neighbouring particles share nodes along the common faces in our implementation and thus particle separation (or fracture) cannot be captured. However, failure is not occurring in this example. If fracture is to be modelled with CPDI each particle must have their own nodes/corners and the free code described in [56] can be used for this purpose just as in cohesive crack modelling with interface elements, e.g., [57].

The codes used for this problem are verified against the results reported by [47], whereby the response is confirmed through Fig. 11. This figure shows the vertical displacement of the corner point at the free end of the beam. In order to show the performance advantages of Julia, three background grids of $16 \times 16 \times 2$, $32 \times 32 \times 4$ and $64 \times 64 \times 8$ cells are considered. The number of elements or particles is not changed for these analyses.

The runtime duration of Julia and Matlab codes are presented in Table 4. By comparison, the performance advantage of Julia is clearly evident. However, another important observation made from the results is the increasing performance advantage of Julia over Matlab, as the computations become more intense with the increase in the number of grid points–the last simulation took 3 hours with the Matlab code whereas the Julia code completed in just 0.6 hours.

4.3. High-velocity impact

The third and final example showcased in this study is a 2D simulation involving the impact of an elastic steel disk with a plastic aluminum target. The geometry and initial conditions are given in Fig. 12.

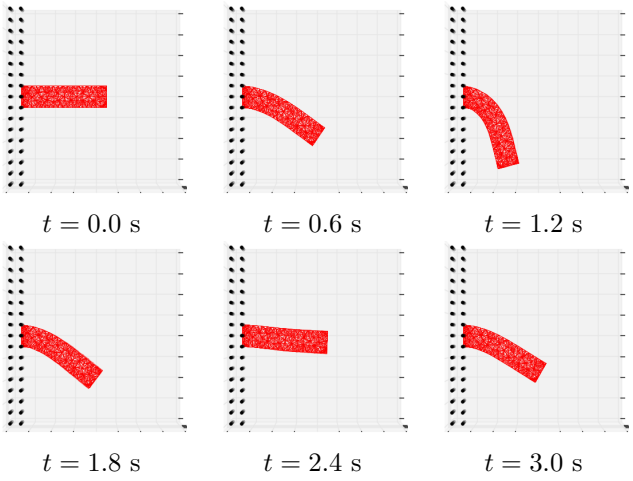

Figure 10: Snapshots of the dynamic response of the cantilever beam problem.

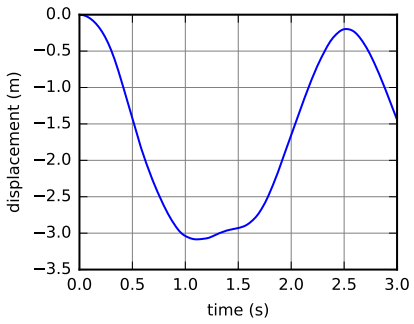

Figure 11: Time-deflection response for the cantilever beam problem. Deflection values correspond to the bottom corner of the free end (Fig. 8).

Table 4: Performance comparison of Julia and Matlab code for the cantilever beam problem.

		Iterations/sec.		Total time [s]		
Particles	Grid	Matlab	Julia	Matlab	Julia	Ratio
1041	16x16x2	0.63	2.51	4783.16	1195.22	4.00
1041	32x32x4	0.46	1.97	6574.62	1521.22	4.32
1041	64x64x8	0.29	1.36	10384.22	2204.26	4.71

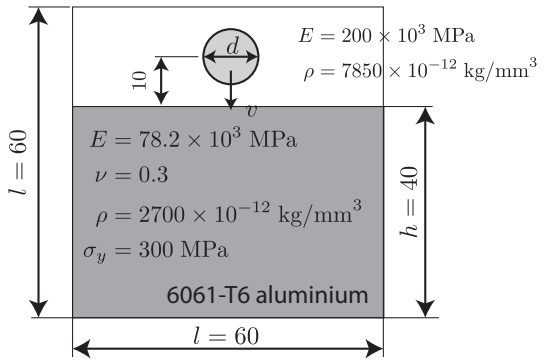

Figure 12: Setup and initial conditions for the high-velocity impact problem.

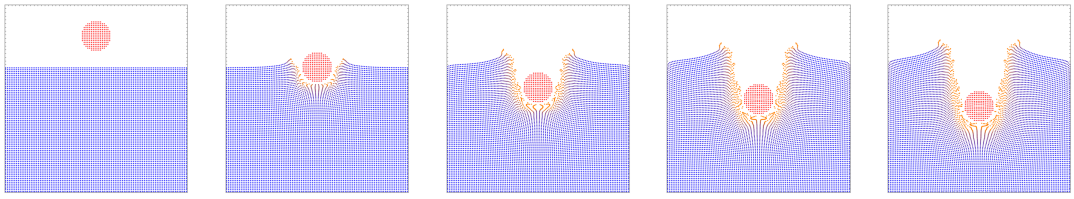

Figure 13: Snapshots for the high-velocity impact problem as the steel disk penetrated into the aluminum target. The color represents the equivalent plastic strain. These images were created using the PyPlot graphical package [52].

The background grid has a 50×50 layout, however, is not shown in the figures to avoid clutter. The simulation consists of a total of 6908 particles, 208 of which constitute the steel disk and 6700 particles for the aluminium target. The simulation was performed over constant time increments of $\Delta t = 10^{-8}$ s up to $t = 40.0 \times 10^{-6}$ s.

The code used to carry out this simulation is similar to the one used for the first example (impact of two disks), except for the stress update algorithm which now involves a perfectly plastic von-Mises model. Fig. 13 shows the penetration of the disk into the target at four different time states. The results are very similar to the ones reported by [54], which serves as verification for the codes implemented in this study.

Table 5 compares the performance of the Julia implementation to the one in Matlab. The results indicate a significant speed advantage of more than 7 times for Julia. Given that this problem is representative of actual engineering problems that can be solved using MPM, such an advantage is of significant interest where the run-time of simulations is in the order of hours or even days.

Table 5: Performance comparison of the Julia and Matlab code for the high-velocity impact problem.

		Iterations/sec.		Total time [s]		
Particles	Grid	Matlab	Julia	Matlab	Julia	Ratio
6908	50x50	0.54	4.27	7380.07	936.33	7.88
15448	50x50	0.23	1.92	17746.23	2088.77	8.50

5. Conclusions

This paper investigated the performance advantages of the Julia programming language as opposed to Matlab in the context of computational solid mechanics. Using similar implementations of the Material

Point Method (MPM) for codes written in Julia and Matlab, run time of the two platforms were compared. The following conclusions can be drawn from the reported results:

1. Both programming languages allow rapid development of models. This is enhanced by the fact that both languages provide graphical tools to easily visualize the results.
2. Codes written in Julia are generally faster than those written in Matlab. The understanding of how objects are accessed and passed through functions in Julia, allows some implementation techniques to be used to further enhance the performance of the program.
3. In contrast to Matlab where vectorized operations are more favorable against for-loops [58, 59], Julia code runs faster when iterative operations are not vectorized [5]. This is important when it comes to MPM implementations since such loops are extensively used to iterate over nodal grid points and material points.
4. Using composite types (synonymous to 'struct' in C) is commonly used in software design given its contribution to the organization and readability of the code. However, utilizing composite types instead of multi-dimensional arrays is shown to decrease the performance of the code. Through benchmark testing, this paper demonstrates that removing redundant calls to the members of composite types and replacing them with a single reference access can narrow the performance gap between composite types and multi-dimensional arrays.

In conclusion, Julia enables writing efficient codes without losing productivity and we no longer need to know two programming languages as people have been using Fortran/C in their Matlab/Python codes within the paradigm of mixed language programming. However, our debugging experiences with Julia is negative as Julia lacks a matured interactive debugger that Matlab provides. This is not suprising as Julia is a relatively young language of about 4 year old and the current version is only 0.4.5, and work has been initiated on a debugger for Julia (`https://github.com/toivoh/Debug.jl`).

Acknowledgements

Funding support from the Australian Research Council via project DE160100577 (Vinh Phu Nguyen) is gratefully acknowledged. The first author also acknowledges the financial support from the Civil Engineering Department at Monash University.

Appendix A. Some syntax differences between Matlab and Julia

Several typical syntax differences between Matlab and Julia are given in Table A.1. A comprehensive list can be found at `http://hyperpolyglot.org/numerical-analysis` even with a script to convert Matlab codes to Julia. Another source for this is `http://docs.julialang.org/en/release-0.4/manual/noteworthy-differences/`.

Appendix B. Code organization

Our complete MPM code is organized into four Julia files as shown in Table B.1. Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the julia executable. In order to run the simulations one launches the Julia REPL using the julia command, and change to the directory where the source code is found[9] and type *include("Main.jl")*. We used *@time include("Main.jl")* to measure the runtime of the examples presented in this article.

[9] If the code is in the folder /Users/vingu/my-codes/JuliaCodes/, then the command would be *cd("/Users/vingu/my-codes/JuliaCodes/")*.

Table A.1: Some syntax differences between Matlab and Julia

	Matlab	Julia
File extension	.m	.jl
Comments	% comment	# comment
Quotes	'Quote'	'Quote' "Quote"
In-line functions	f = @(x,y) x+y	f(x,y) = x+y

Functions	function [a,b] = f(x,y)
	a = x + y;
	b = x * y;
	end (optional)
		function f(x,y)
		a = x + y
		b = x * y;
		return [a b]
		end

Array index	K(i,j)	K[i,j]
Conditional expression	none	x >0 ? x : x

Table B.1: Organization of the presented MPM code.

Main.jl	main script	379 lines (including plotting)
Basis.jl	shape functions	53 lines
Grid.jl	grid and grid points	104 lines
MaterialPoint.jl	material points	139 lines

References

[1] MathWorks. http://www.mathworks.com.

[2] Python Software Foundation. https://www.python.org/.

[3] Mathematica. http://www.mathematica.com.

[4] J. Bezanson, S. K., V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[5] J. Bezanson, A. Edelman, S. K., and V. B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[6] C. Lattner and Vikram A. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–. IEEE Computer Society, 2004.

[7] J. B., S. K., V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[8] A. A. Ramabathiran. Finite element programming in Julia. http://www.codeproject.com/Articles/579983/Finite-Element-programming-in-Julia.

[9] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[10] D. Sulsky, Z. Chen, and H.L. Schreyer. A particle method for history-dependent materials. *Computer Methods in Applied Mechanics and Engineering*, 5:179–196, 1994.

[11] D. Sulsky, S.J. Zhou, and H. L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87(1-2):236–252, 1995.

[12] D. Sulsky and H.L. Schreyer. Axisymmetric form of the material point method with applications to upsetting and Taylor impact problems. *Computer Methods in Applied Mechanics and Engineering*, 139:409–429, 1996.

[13] T. Rabczuk, T. Belytschko, and S.P. Xiao. Stable particle methods based on Lagrangian kernels. *Computer Methods in Applied Mechanics and Engineering*, 193(12-14):1035 – 1063, 2004.

[14] T. Rabczuk and T. Belytschko. Cracking particles: a simplified meshfree method for arbitrary evolving cracks. *International Journal for Numerical Methods in Engineering*, 61(13):2316–2343, 2004.

[15] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139:3–47, 1996.

[16] T. Rabczuk, S. Bordas, and G. Zi. A three-dimensional meshfree method for continuous multiple-crack initiation, propagation and junction in statics and dynamics. *Computational Mechanics*, 40(3):473–495, 2007.

[17] T. Rabczuk and T. Belytschko. A three-dimensional large deformation meshfree method for arbitrary evolving cracks. *Computer Methods in Applied Mechanics and Engineering*, 196(29):2777–2799, 2007.

[18] V.P. Nguyen, T. Rabczuk, S. Bordas, and M. Duflot. Meshless methods: A review and computer implementation aspects. *Mathematics and Computers in Simulation*, 79(3):763–813, 2008.
[19] T. Rabczuk, G. Zi, S. Bordas, and H. Nguyen-Xuan. A simple and robust three-dimensional cracking-particle method without enrichment. *Computer Methods in Applied Mechanics and Engineering*, 199(37):2437–2455, 2010.
[20] S. Andersen and L. Andersen. Modelling of landslides with the material-point method. *Computational Geosciences*, 14(1):137–147, 2010.
[21] Z. Wieçkowski, Sung-kie Y., and Jeoung-heum Y. A Particle-in-cell solution to the silo discharging problem. *International Journal For Numerical Methods in Engineering*, 45:1203–1225, 1999.
[22] Z. Wieçkowski. The material point method in large strain engineering problems. *Computer Methods in Applied Mechanics and Engineering*, 193(39-41):4417–4438, 2004.
[23] H.-B. Mühlhaus, H. Sakaguchi, L. Moresi, and M. Fahey. Discrete and continuum modelling of granular materials. In P.A. Vermeer, H.J. Herrmann, S. Luding, W. Ehlers, S. Diebels, and E. Ramm, editors, *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*, volume 568 of *Lecture Notes in Physics*, pages 185–204. Springer Berlin Heidelberg, 2001.
[24] C. J. Coetzee, P. A. Vermeer, and A. H. Basson. The modelling of anchors using the material point method. *International Journal for Numerical and Analytical Methods in Geomechanics*, 29(9):879–895, 2005.
[25] H.W. Zhang, K.P. Wang, and Z. Chen. Material point method for dynamic analysis of saturated porous media under external contact/impact of solid bodies. *Computer Methods in Applied Mechanics and Engineering*, 198(17-20):1456–1472, 2009.
[26] L. Beuth, Z. Wieckowski, and P. A. Vermeer. Solution of quasi-static large-strain problems by the material point method. *International Journal for Numerical and Analytical Methods in Geomechanics*, 35(13):1451–1465, 2011.
[27] J. Ma, D. Wang, and M.F. Randolph. A new contact algorithm in the material point method for geotechnical simulations. *International Journal for Numerical and Analytical Methods in Geomechanics*, 38(11):1197–1210, 2014.
[28] A. Yerro, E.E. Alonso, and N.M. Pinyol. The material point method for unsaturated soils. *Géotechnique*, 65:201–217(16), 2015.
[29] S.G. Bardenhagen, A.D. Brydon, and J.E. Guilkey. Insight into the physics of foam densification via numerical simulation. *Journal of the Mechanics and Physics of Solids*, 53(3):597 – 617, 2005.
[30] A.D. Brydon, S.G. Bardenhagen, E.A. Miller, and G.T. Seidler. Simulation of the densification of real open-celled foam microstructures. *Journal of the Mechanics and Physics of Solids*, 53(12):2638 – 2660, 2005.
[31] S.G. Bardenhagen, J.U. Brackbill, and D. Sulsky. The material-point method for granular materials. *Computer Methods in Applied Mechanics and Engineering*, 187(3-4):529–541, 2000.
[32] S.G. Bardenhagen, J.E. Guilkey, K.M. Roessig, J.U. Brackbill, W.M. Witzel, and J.C. Foster. Improved contact algorithm for the material point method and application to stress propagation in granular material. *Computer Modeling in Engineering and Sciences*, 2(4):509–522, 2001.
[33] D. Sulsky, H.L. Schreyer, K. Peterson, R. Kwok, and M. Coon. Using the material-point method to model sea ice dynamics. *Journal of Geophysical Research*, 112(C2):C02S90, 2007.
[34] Y. Wang, H.G. Beom, M. Sun, and S. Lin. Numerical simulation of explosive welding using the material point method. *International Journal of Impact Engineering*, 38(1):51–60, 2011.
[35] R. Ambati, X. Pan, H. Yuan, and X. Zhang. Application of material point methods for cutting process simulations. *Computational Materials Science*, 57:102–110, 2012.
[36] X. F. Pan, A. Xu, G. Zhang, and J. Zhu. Generalized interpolation material point approach to high melting explosive with cavities under shock. *Journal of Physics D: Applied Physics*, 41(1):015401, 2008.
[37] W. Hu and Z. Chen. Model-based simulation of the synergistic effects of blast and fragmentation on a concrete wall using the MPM. *International Journal of Impact Engineering*, 32(12):2066 – 2096, 2006.
[38] B. Banerjee. Material point method simulations of fragmenting cylinders. In *17th ASCE Engineering Mechanics Conference*, University of Delaware, Newark, DE, June 2004.
[39] J.E. Guilkey, T.B. Harman, and B. Banerjee. An Eulerian-Lagrangian approach for simulating explosions of energetic devices. *Computers & Structures*, 85(11-14):660–674, 2007.
[40] J. Burghardt, B. Leavy, J. Guilkey, Z. Xue, and R. Brannon. Application of Uintah-MPM to shaped charge jet penetration of aluminum. *IOP Conference Series: Materials Science and Engineering*, 10(1):012223, 2010.
[41] James E. Guilkey, James B. Hoying, and Jeffrey A. Weiss. Computational modeling of multicellular constructs with the material point method. *Journal of Biomechanics*, 39(11):2074 – 2086, 2006.
[42] T. Belytschko, W. K. Liu, and B. Moran. *Nonlinear Finite Elements for Continua and Structures*. John Wiley & Sons, Chichester, England, 2000.
[43] V. P. Nguyen. Material point method: basics and applications. https://www.researchgate.net/publication/262415477_Material_point_method_basics_and_applications_Contents, 2014. Online.
[44] P.C. Wallstedt and J.E. Guilkey. An evaluation of explicit time integration schemes for use with the generalized interpolation material point method. *Journal of Computational Physics*, 227(22):9628 – 9642, 2008.
[45] M. Steffen, R. M. Kirby, and M. Berzins. Decoupling and balancing of space and time errors in the material point method (MPM). *International Journal for Numerical Methods in Engineering*, 82(10):1207–1243, 2010.
[46] S.G. Bardenhagen and E.M. Kober. The generalized interpolation material point method. *Computer Modeling in Engineering and Sciences*, 5(6):477–495, 2004.
[47] A. Sadeghirad, R. M. Brannon, and J. Burghardt. A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations. *International Journal for Numerical Methods in Engineering*, 86(12):1435–1456, 2011.

[48] A. Sadeghirad, R.M. Brannon, and J.E. Guilkey. Second-order convected particle domain interpolation (CPDI2) with enrichment for weak discontinuities at material interfaces. *International Journal for Numerical Methods in Engineering*, 95(11):928–952, 2013.
[49] M. A. Homel, R. M. Brannon, and J. Guilkey. Controlling the onset of numerical fracture in parallelized implementations of the material point method (MPM) with convective particle domain interpolation (CPDI) domain scaling. *International Journal for Numerical Methods in Engineering*, 107(1):31–48, 2017.
[50] V.P. Nguyen and G.D. Nguyen. A Voronoi cell material point method for large deformation solid mechanics problems. In *Advances of Computational Mechanics in Australia*, volume 846 of *Applied Mechanics and Materials*, pages 108–113, 2016.
[51] V.P. Nguyen, C.T. Nguyen, S. Najatarajan, and T. Rabczuk. On a family of convected particle domain interpolations in the material point method. *Finite Elements in Analysis and Design*, 126:50–64, 2017.
[52] S. G. Johnson. PyPlot module for Julia. https://github.com/stevengj/PyPlot.jl, 2012.
[53] O Buzzi, DM Pedroso, and A Giacomini. Caveats on the implementation of the generalized material point method. *Computer Modeling in Engineering and Sciences*, 1(1):1–21, 2008.
[54] C. J. Coetzee. *The Modelling of Granular Flow Using the Particle-in-Cell Method*. PhD thesis, University of Stellenbosch, 2003.
[55] C. Geuzaine and J. F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.
[56] V.P. Nguyen. An open source program to generate zero-thickness cohesive interface elements. *Advances in Engineering Software*, 74:27–39, 2014.
[57] V.P. Nguyen. Discontinuous Galerkin/Extrinsic cohesive zone modeling: implementation caveats and applications in computational fracture mechanics. *Engineering Fracture Mechanics*, 128:37–68, 2014.
[58] MathWorks. Using vectorization in Matlab. https://au.mathworks.com/help/matlab/matlab_prog/vectorization.html, 2016.
[59] MathWorks. Techniques to improve performance. https://au.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html, 2016.