Computational Particle Mechanics (2021) 8:767–789
https://doi.org/10.1007/s40571-020-00369-8

Karamelo: an open source parallel C++ package for the material point method

Alban de Vaucorbeil[1] · **Vinh Phu Nguyen**[2] · **Chi Nguyen-Thanh**[3]

Received: 3 August 2020 / Revised: 17 September 2020 / Accepted: 28 September 2020 / Published online: 14 October 2020

Abstract
A simple and robust C++ code for the material point method (MPM) called `Karamelo` is presented here. It was designed to provide an open source, fast, light and easy-to-modify framework for both conducting research on the MPM and research using the MPM, instead of a finite element package. This paper presents the overall philosophy, the main design choices and some of the original algorithms implemented in `Karamelo`. Simulations of solids and fluids involving extreme deformation are provided to illustrate the capabilities of the code.

Keywords Material point method · MPM · Solids, fluids, ductile fracture · Damage

1 Introduction

The material point method is a meshfree method [7] which is one of the latest developments in particle-in-cell (PIC) methods. The first PIC technique was developed in the early 1950s by [19] and was used primarily in fluid mechanics. It suffered from excessive energy dissipation which was overcome later by [9] with the introduction of FLIP-the fluid implicit particle method. FLIP was later modified and tailored for applications in solid mechanics by Sulsky and co-workers [54,55] and has since been referred to as the material point method (MPM) [52].

The MPM is best suited for problems exhibiting very large deformation and contacts. This is due to the fact that, in the MPM, solids are discretized by a set of Lagrangian particles moving over a fixed Cartesian grid. Since 1994, many improved instances of the MPM have been developed [4,12,36,42,50,58] and the MPM has found applications in many fields ranging from geoengineering [15], mechanical engineering [25,46,47] to the movie industry [21]. For more details, we refer to the comprehensive review of the method in [13].

Computational scientists constantly face the trade-off between rapid prototyping and good software engineering. The former allows researchers to code quickly (so as to try different ideas and rapid publication of the work), but inevitably leads to issues such as unsatisfactory performance, poor portability and maintainability. The latter results in reusable codes for future projects, but slows down the research progress due to low-level engineering. Open source MPM codes belonging to the former category can be found for example in [45], who presented a Julia implementation, and at https://csmbrannon.net/2018/11/09/matlab-and-mms-source-files/ for a MATLAB code. There exist a few MPM implementations belonging to the second category. For example, [37] developed a parallel MPM code using message passing interface (MPI) called `Uintah` with excellent scalability of more than 1000 processors as demonstrated in [38] for simulations having about 16 million particles. [20] described a parallel MPM using OpenMP for shared memory machines and [28] presented a 3D parallel Fortran MPM code using MPI. [30] described an object-oriented C++ implementation of MPM. Most recently [14] presented a GPU implementation. Amongst all the open-source MPM codes, `Uintah` is probably the most efficient one to date. Unfortunately, since `Uintah` is not just an MPM code, but rather a large software suite for the simulation of both chemical and physical phenomena, it is too large of a code to understand and even more to modify.

✉ Alban de Vaucorbeil
alban.devaucorbeil@deakin.au.edu

1 Institute for Frontier Materials, Deakin University, Geelong, VIC 3216, Australia

2 Department of Civil Engineering, Monash University, Clayton 3800, VIC, Australia

3 Institute of Research and Development, Duy Tan University, Da Nang 550000, Vietnam

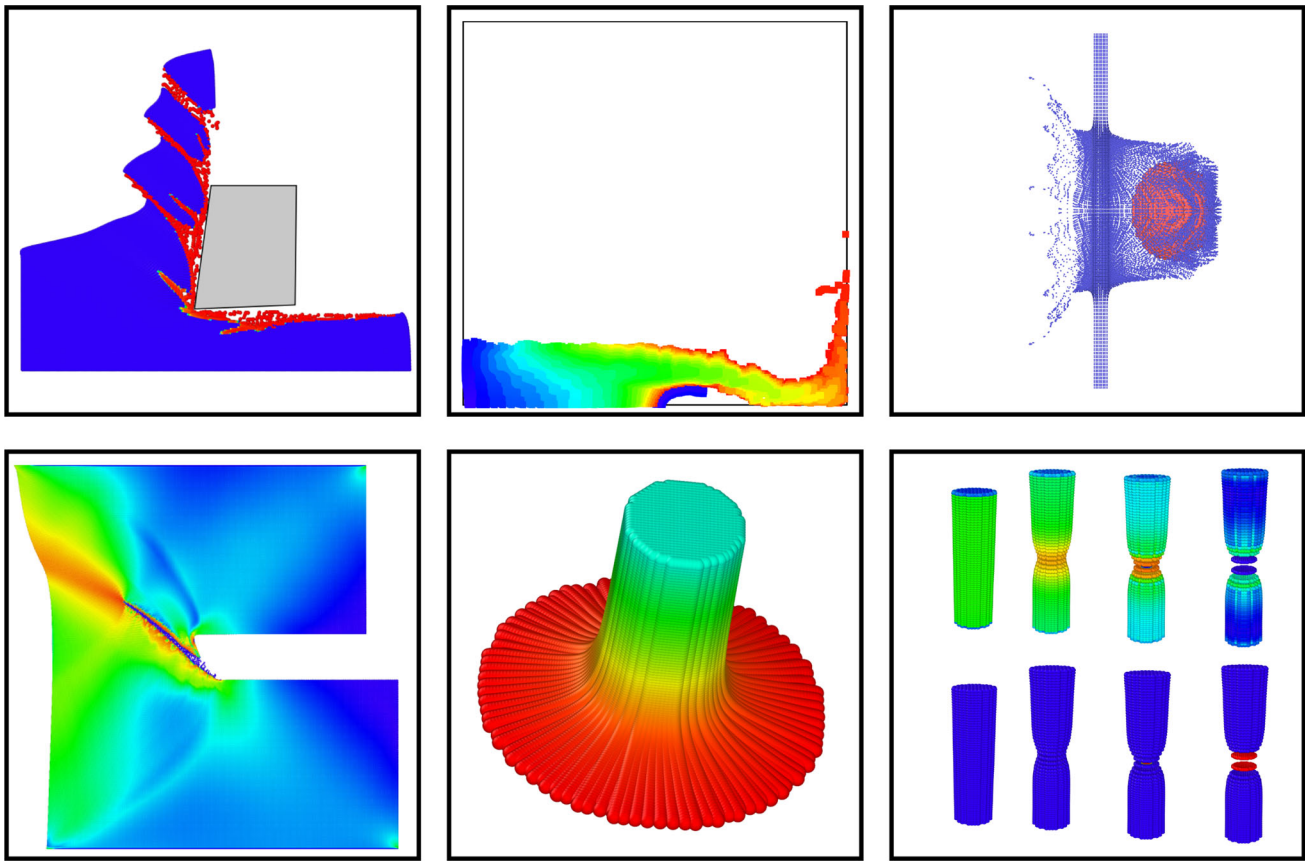


Fig. 1 Representative simulations that can be done using `Karamelo`. We refer to Sect. 4 for details

Therefore, there was a need to develop a portable, efficient, and easy to modify code that can be used in either 1D, 2D or 3D. To this end, a new in-house code called `Karamelo` was developed. The structure of this code is based on that of the popular molecular dynamics simulator `LAMMPS` [39]. `LAMMPS` is a highly parallel simulator that can be used on multi-CPU machines that support MPI (message passing interface), as well as on GPUs (graphic processing units). Moreover, through its ingenious hierarchical class system, it is easy to add new functionalities in `LAMMPS`. This is illustrated by the fact that although `LAMMPS` is a molecular dynamics code, modules for particle-based methods such as Smooth Particle Hydrodynamics (SPH) and peridynamics, see e.g., [44], have been added to it [16]. Because of the use of a background grid, it would be difficult to directly implement the MPM in `LAMMPS`, this is why it was decided to create a new code that uses `LAMMPS`'s hierarchical class system, but adapted to the requirements of the MPM. `Karamelo` was born.

The aim of this paper is to describe the implementation of `Karamelo`. Furthermore, new simulations, not reported in [13] where `Karamelo` was first introduced, are presented. The code has been used to successfully model challenging solid mechanics problems (e.g., high-velocity impacts, cold spray, material wearing), some free surface flow problems and fluid–structure interaction problems, see Fig. 1. Since `Karamelo` is an open-source code available at https://github.com/adevaucorbeil/karamelo/, we hope that the engineering community will make use of it, and even contribute to its development.

As of today, `Karamelo` only implements the explicit MPM algorithm which is briefly recalled in Sect. 2. Implementation details of `Karamelo` are described in Sect. 3, followed by representative simulations given in Sect. 4. Input files are presented in the appendices to explain the syntax.

The most commonly used symbols in this paper are listed in Table 1.

2 The material point method algorithm

This section briefly presents the explicit dynamics MPM formulation. For details, we refer to [54] or the recent review of [13].

In the MPM, the weak form of the momentum equation is solved. This formulation is identical to the weak form solved by the updated Lagrangian finite element method (FEM) [8].

Table 1 List of the most commonly used symbols in the paper

Symbol	Description
t	Time
Δt	time step
n_p	Number of material points (or particles)
$\mathbf{x}_p$	Position of particle p
$\mathbf{v}$	Velocity
m	Mass
V	Volume
ρ	Density
$\mathbf{F}$	Deformation gradient
$\mathbf{L}$	Gradient velocity tensor
$\boldsymbol{\sigma}$	Cauchy stress tensor
$\boldsymbol{\epsilon}$	Strain tensor
$\boldsymbol{\epsilon}'$	Deviatoric strain tensor
T	Temperature
ϕ_I	Weighting or shape function at node I
$\nabla\phi_I$	Derivative of the shape function at node I
$\mathbf{f}_I^{\text{int}}$	Internal force vector at node I
$\mathbf{f}_I^{\text{ext}}$	External force vector at node I
$\mathbf{f}_I$	Nodal force vector at node I
$\boldsymbol{\sigma}_f$	Flow stress
λ and μ	Lamé's constants
K and G	Bulk and shear moduli
D	Damage variable
$\hat{p}$	Hydrostatic pressure
α	Mixing ratio between PIC and FLIP

The MPM is built on the two main concepts already used in PIC that are the use of Lagrangian material points carrying physical information, and a background Eulerian grid used for the discretization of continuous fields (i.e., displacement field). These two concepts are treated in Sect. 2.1, followed by Sect. 2.2 which provides a complete algorithm for the MPM. This algorithm is applicable to any weighting functions (linear, B-splines, etc.). Section 2.3 presents the so-called hat functions (i.e. linear weighting functions), to complete the formulation. Other weighting functions such as B-splines are described in [13]. This MPM algorithm can be used to model a wide range of materials (metals, rubbers, concretes, ceramics, water, gases etc.) and a constitutive model is needed to differentiate them. Section 2.4 presents the few ones used here.

2.1 Lagrangian particles and Eulerian grid

In the MPM, a continuum body is discretized by a finite set of n_p Lagrangian material points (or particles) that are tracked throughout the deformation process. The terms *particle* and *material point* will be used interchangeably throughout this paper. In the original MPM, the subregions represented by the particles are not explicitly defined. Only their mass and volume are tracked. However, the shape of these subregions is tracked in advanced MPM formulations such as the generalized interpolation material point (GIMP) method [4] and the convected particle domain interpolation (CPDI), see [35,42,43]. Each material point has an associated position $\mathbf{x}_p^t$ $(p = 1, 2, \ldots, n_p)$, mass m_p, density ρ_p, velocity $\mathbf{v}_p$, deformation gradient $\mathbf{F}_p$, Cauchy stress tensor $\boldsymbol{\sigma}_p$, temperature T_p, and any other internal state variables necessary for the constitutive model. Collectively, these material points provide a Lagrangian description of the continuum body. Since each material point contains a fixed amount of mass at all time, mass conservation is automatically satisfied.

The original MPM developed by Sulsky is effectively an updated Lagrangian scheme, also called Updated Lagrangian MPM (ULMPM) thereafter. For this MPM, the space that the simulated body occupies and will occupy during deformation is discretized by a background grid (see Fig. 2) where the equation of balance of momentum is solved. On the other hand, in the Total Lagrangian MPM (TLMPM) presented in [12], the background grid covers only the space occupied by the body in its reference configuration as illustrated in Fig. 2. The use of a grid allows the method to be quite scalable by eliminating the need for directly computing particle–particle interactions. Indeed, the particles interact with other particles of the same body, with other solid bodies, or with fluids through a background Eulerian grid. Most often, for efficiency reasons, a fixed regular Cartesian grid is used throughout the simulation; and this fixed Cartesian background grid is the only option currently supported by `Karamelo`.

2.2 The basic explicit MPM algorithm

The MPM was originally developed to solve fast transient impact solid mechanics problems [54]. Therefore, the MPM has been developed using an explicit solver which is more efficient than an implicit one for such problems. From the updated Lagrangian MPM, the total Lagrangian MPM is obtained by making only slight modifications. Indeed, in this case, the weak form of the momentum equation to solve differs only by the fact that it is expressed in the reference coordinate system and the use of the first Piola-Kirchhoff *in lieu* of the Cauchy stress [8,12]. This weak form is similar to that used in the total Lagrangian FEM. Currently, `Karamelo` supports only an explicit solver and thus only this solver is presented here.

A typical explicit ULMPM computational cycle consists of four steps (see Fig. 3). The first step is to map the information (mass, momentum and internal and external forces) from the particles to the grid (P2G), since the grid is reset at every cycle. Next, the discrete equations of momentum are

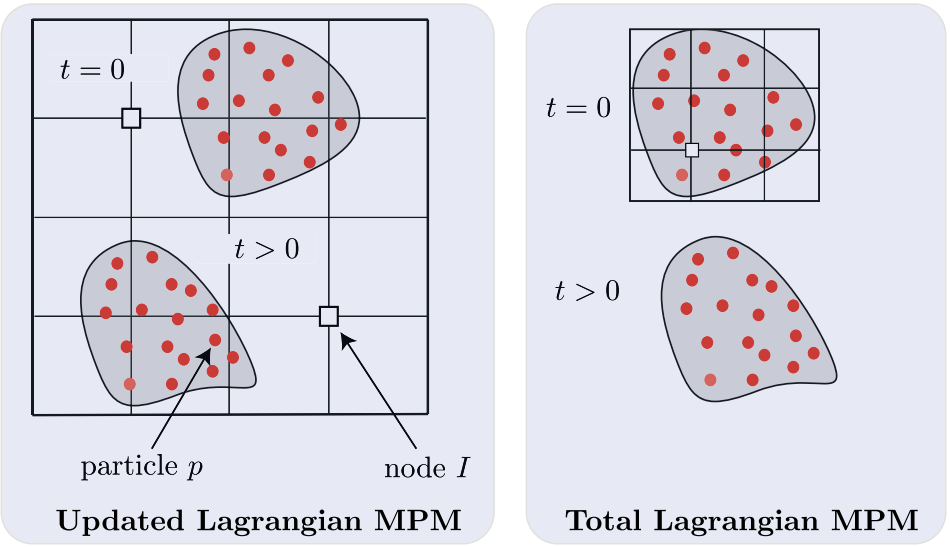


Fig. 2 The MPM discretization: the space is discretized by a background grid which can be either a Cartesian grid or an unstructured grid (not shown), while a solid is discretized using particles. The updated Lagrangian MPM grid covers the entire deformation space, whereas the total Lagrangian MPM only covers the initial configuration

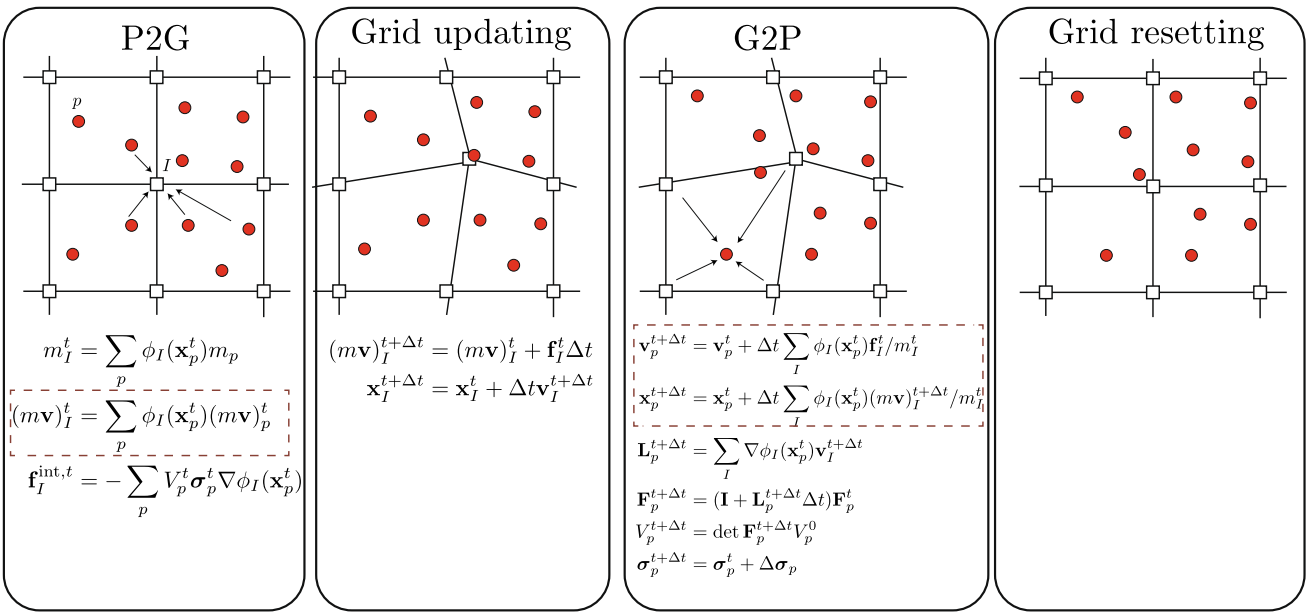


Fig. 3 Material point method: a computational step consists of four steps: (1) P2G (particle to grid) in which information is mapped from particles to nodes, (2) grid updating in which momentum equations are solved for the nodes, (3) G2P (Grid to particles) where the updated nodes are then mapped back to the particles to update their positions and velocities and (4) grid resetting where the grid is reset. The operations in dashed boxes are not present in the ULFEM

solved on the grid nodes (Grid updating). Then, the particles' position, velocity, volume, density, deformation gradient, stresses and all relevant internal variables are updated (G2P). These last two steps are equivalent to the updated Lagrangian FEM [8]. Finally, the grid is reset to its original state. Due to this grid resetting, mesh distortion never occurs, making the MPM a good method for large deformation problems. A complete flowchart of the explicit MPM using the so-called modified update stress last formulation of [55] is given in Algorithm 1. Note that this algorithm only allows no-slip, no-penetration contacts. Including frictional contacts requires a slight modification to this algorithm, see [13] for detail. In this algorithm, $0 \leq \alpha \leq 1$ was introduced to mix the good properties of PIC and FLIP following [26,48].

The flowchart of the TLMPM is quite similar to the one of the ULMPM except that the first Piola-Kirchhoff stress tensor is used in the internal force vector, and the spatial derivatives are performed with respect to the original configuration [12], not the current (deformed) one. Note that

contrary to the ULMPM, the TLMPM algorithm does not have contact capacities built-in.

Throughout this paper, subscripts $_p$ and $_I$ are used to refer to quantities at the particle or node positions, respectively. And upperscripts ^{t} and $^{t+\Delta t}$ refer to quantities at timesteps t and $t+\Delta t$, respectively.

Algorithm 1 Solution procedure of explicit ULMPM (MUSL).

Initialization
  Set up the Cartesian grid, set time $t = 0$
  Set up particle data: $\mathbf{x}_p^0, \mathbf{v}_p^0, \boldsymbol{\sigma}_p^0, \mathbf{F}_p^0, V_p^0, m_p, \rho_p^0$
end
while $t < t_f$ **do**
  Reset grid quantities: $m_I^t = 0, (m\mathbf{v})_I^t = \mathbf{0}, \mathbf{f}_I^{\text{ext},t} = \mathbf{0}, \mathbf{f}_I^{\text{int},t} = \mathbf{0}$
  Mapping from particles to nodes (P2G)
    Compute nodal mass $m_I^t = \sum_p \phi_I(\mathbf{x}_p^t) m_p$
    Compute nodal momentum $(m\mathbf{v})_I^t = \sum_p \phi_I(\mathbf{x}_p^t)(m\mathbf{v})_p^t$
    Compute external force $\mathbf{f}_I^{\text{ext},t} = \sum_p \phi_I(\mathbf{x}_p) m_p \mathbf{b}(\mathbf{x}_p)$
    Compute internal force $\mathbf{f}_I^{\text{int},t} = -\sum_p V_p^t \boldsymbol{\sigma}_p^t \nabla \phi_I(\mathbf{x}_p^t)$
    Compute nodal force $\mathbf{f}_I^t = \mathbf{f}_I^{\text{ext},t} + \mathbf{f}_I^{\text{int},t}$
  end
  Update the momenta $(m\tilde{\mathbf{v}})_I^{t+\Delta t} = (m\mathbf{v})_I^t + \mathbf{f}_I^t \Delta t$
  Fix Dirichlet nodes I e.g., $(m\mathbf{v})_I^t = \mathbf{0}$ and $(m\tilde{\mathbf{v}})_I^{t+\Delta t} = \mathbf{0}$
  Update particle velocities and grid velocities (double mapping)
    Get nodal velocities $\tilde{\mathbf{v}}_I^{t+\Delta t} = (m\tilde{\mathbf{v}})_I^{t+\Delta t} / m_I^t$
    Update particle positions $\mathbf{x}_p^{t+\Delta t} = \mathbf{x}_p^t + \Delta t \sum_I \phi_I(\mathbf{x}_p^t) \tilde{\mathbf{v}}_I^{t+\Delta t}$
    Update particle velocities $\mathbf{v}_p^{t+\Delta t} = \alpha\big(\mathbf{v}_p^t + \sum_I \phi_I(\mathbf{x}_p^t)\left[\tilde{\mathbf{v}}_I^{t+\Delta t} - \mathbf{v}_I^t\right]\big) + (1-\alpha)\sum_I \phi_I(\mathbf{x}_p^t)\tilde{\mathbf{v}}_I^{t+\Delta t}$
    Update grid momenta $(m\mathbf{v}_I)^{t+\Delta t} = \sum_p \phi_I(\mathbf{x}_p^t)(m\mathbf{v})_p^{t+\Delta t}$
    Fix Dirichlet nodes $(m\mathbf{v})_I^{t+\Delta t} = \mathbf{0}$
  end
  Update particles (G2P)
    Get nodal velocities $\mathbf{v}_I^{t+\Delta t} = (m\mathbf{v})_I^{t+\Delta t} / m_I^t$
    Compute gradient velocity $\mathbf{L}_p^{t+\Delta t} = \sum_I \nabla \phi_I(\mathbf{x}_p^t) \mathbf{v}_I^{t+\Delta t}$
    Updated gradient deformation tensor $\mathbf{F}_p^{t+\Delta t} = (\mathbf{I} + \mathbf{L}_p^{t+\Delta t} \Delta t)\mathbf{F}_p^t$
    Update volume $V_p^{t+\Delta t} = \det \mathbf{F}_p^{t+\Delta t} V_p^0$
    Update stresses $\boldsymbol{\sigma}_p^{t+\Delta t} = \boldsymbol{\sigma}_p^t + \Delta \boldsymbol{\sigma}_p$
  end
  Advance time $t = t + \Delta t$
end while

2.3 Weighting functions

Multiple weight functions are implemented in `Karamelo` such as linear hat functions, cubic B-splines, quadratic Bernstein, and CPDI. Here, we present only the simplest of them: the linear hat function.

Although any grid can be used in the MPM, a Cartesian grid is usually chosen for computational convenience reasons. The MPM weighting functions are conveniently (and efficiently) defined in the global coordinate system. In 1D, the linear weighting functions are defined as

$$\phi_I^x(x) = \begin{cases} 1 - |x - x_I|/h_x & \text{if } |x - x_I| \le h_x \\ 0 & \text{else} \end{cases} \tag{2.1}$$

where h_x denotes the nodal spacing or element size in the x direction. Its derivatives are given by

$$\phi_{I,x}^x(x) \equiv \frac{\mathrm{d}\phi_I^x(x)}{\mathrm{d}x} = \begin{cases} -\text{sign}(x - x_I)/h_x & \text{if } |x - x_I| \le h_x \\ 0 & \text{else} \end{cases} \tag{2.2}$$

where sign(x) is the signum function.

For 2D and 3D, the shape functions are simply the tensor-product of the shape functions along the x, y, and z directions

$$\phi_I(x, y) = \phi_I^x(x)\phi_I^y(y) \quad \text{(2D)} \tag{2.3}$$

$$\phi_I(x, y, z) = \phi_I^x(x)\phi_I^y(y)\phi_I^z(z) \quad \text{(3D)} \tag{2.4}$$

2.4 Constitutive models

Several constitutive models are available in `Karamelo` at the time this article was written: isotropic linear elastic material; Neo-Hookean hyperelastic material; small strain J2 elasto-plastic material; large strain hypoelastic plastic materials; and EOS for weakly compressible fluids and gases. This section briefly presents a few of them, those used in the simulations provided in Sect. 4. We refer to [13] for more details.

2.4.1 Linear elastic isotropic material

For isotropic linear elastic materials, the stress tensor is given by

$$\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}, \quad \boldsymbol{\sigma} = (\lambda \text{tr}\boldsymbol{\epsilon})\mathbf{I} + 2\mu\boldsymbol{\epsilon} \tag{2.5}$$

where λ and μ are the Lamé's constants. The stress tensor is also often written in terms of a hydrostatic and a deviatoric part

$$\boldsymbol{\sigma} = (K\,\text{tr}\boldsymbol{\epsilon})\mathbf{I} + 2G\boldsymbol{\epsilon}' \tag{2.6}$$

where $\boldsymbol{\epsilon}'$ denotes the deviatoric strain tensor, and the bulk modulus K and shear modulus G are given by

$$K := \frac{3\lambda + 2\mu}{3}, \quad G := \mu \tag{2.7}$$

2.4.2 Elasto-plastic materials

This section presents a temperature-dependent hypoelastic-damage-plastic material model. The model is applicable to large strain and large rotation problems and is suitable for problems when the elastic deformation is negligible compared with the plastic one.

In this model, the Cauchy stress tensor $\boldsymbol{\sigma}$ is expressed as the sum of its isotropic part, i.e., the hydrostatic pressure ($\hat{p}$), and the traceless symmetric deviatoric stress $\boldsymbol{\sigma}^d$. An equation of state (EOS) is used to determine the hydrostatic pressure. The deviatoric response is determined using a plastic flow rule in combination with a yield condition. Here, the plastic flow rule is given by the Johnson–Cook material model, while the yield condition is given by the von Mises condition. Moreover, when fracture is taken into account, it is modelled using the classic continuum damage mechanics approach. That is, the stress tensor scales linearly with a damage variable D [24] satisfying the condition $0 \leq D \leq 1$.

The hydrostatic pressure is determined using the Mie–Grüneisen EOS modified to account for damage [57]:

– If $\hat{p} \geq 0$:

$$\hat{p} = \frac{\rho_0(1-D)c_0^2(\eta-1)\left[\eta - \frac{\Gamma_0}{2}(\eta-1)\right]}{[\eta - S_\alpha(\eta-1)]^2} + \Gamma_0 e \tag{2.8}$$

$$\text{with } \eta = \frac{\rho(1-D)}{\rho_0},$$

– if $\hat{p} < 0$:

$$\hat{p} = \frac{\rho_0 c_0^2(\eta-1)[\eta - \frac{\Gamma_0}{2}(\eta-1)]}{[\eta - S_\alpha(\eta-1)]^2} + \Gamma_0 e \tag{2.9}$$

$$\text{with } \eta = \frac{\rho}{\rho_0},$$

where in both cases c_0 is the bulk speed of sound, Γ_0 the Grüneisen Gamma in the reference state. S_α is the linear Hugoniot slope coefficient and e is the internal energy. Note that positive pressure corresponds to compression.

The internal energy is written as

$$e = C_v\rho_0(T - T_r) \tag{2.10}$$

where C_v denotes the specific heat at constant volume and T_r denotes the reference temperature.

According to the Johnson–Cook's flow stress model scaled with damage [22], the equivalent von Mises flow stress is written as

$$\sigma_f(\varepsilon_p, \dot{\varepsilon}_p, T) = \left[A + B\left(\varepsilon_p\right)^n\right] \times \left[1 + C\ln\dot{\varepsilon}_p^*\right]\left[1 - (T^*)^m\right](1-D) \tag{2.11}$$

where ε_p is the equivalent plastic strain, ε_p^* is the normalized plastic strain rate, A the yield stress, B and n the strain hardening parameters, C the strain rate parameter, and m a temperature coefficient. This model has five experimentally determined parameters that describe quite well the response of a number of metals.

The normalized plastic strain rate and the homologous temperature T^* are given by:

$$\dot{\varepsilon}_p^* = \dot{\varepsilon}_p/\dot{\varepsilon}_0, \quad T^* = \frac{T - T_r}{T_m - T_r} \tag{2.12}$$

where $\dot{\varepsilon}_p$ and $\dot{\varepsilon}_0$ are the plastic strain rate, and the user-defined reference plastic strain rate, respectively; and T_m is

the reference melting temperature. Unless otherwise stated, $\dot{\varepsilon}_0 = 1.0\ \mathrm{s}^{-1}$.

Assuming that the adiabatic condition prevails, the temperature increase can be computed as follows

$$\Delta T = \frac{\chi}{\rho C_p}\sigma_f \Delta\varepsilon_p \tag{2.13}$$

where $0 < \chi \leq 1$ is the Taylor–Quinney coefficient that determines how much the plastic work is converted into heat. For metals, $\chi = 0.9$ is often used. The specific heat is denoted by C_p.

The amount of damage (D) in each particle is determined using the Johnson–Cook damage model, widely used for engineering applications [22]. It is a strain rate-dependent phenomenological model based on the local accumulation of plastic strain. According to this model, damage initiates when the accumulated equivalent plastic strain reaches the equivalent strain at failure ε_f:

$$D_{\text{init}} := \sum \frac{\Delta\varepsilon_p}{\varepsilon_f} = 1 \tag{2.14}$$

where $\Delta\varepsilon_p$ is the equivalent plastic strain increment. The equivalent strain at failure ε_f is given by Johnson–Cook's empirical equation:

$$\varepsilon_f = \left[D_1 + D_2 \exp(D_3\sigma^*)\right][1 + D_4 \ln(\dot{\varepsilon}_p^*)][1 + D_5 T^*] \tag{2.15}$$

and where D_1, …, D_5 are five material constants, $\sigma^* = -\hat{p}/\sigma_{\text{eq}}$ is the stress triaxiality with σ_{eq} is the equivalent von Mises stress.

As this model only describes damage initiation, in order to have a complete model of the fracture phenomenon, a damage evolution model is required. Here, it was assumed that the damage variable is given by:

$$D = \begin{cases} 0 & \text{when } 0 \leq D_{\text{init}} < 1 \\ 10\,(D_{\text{init}} - 1) & \text{when } D_{\text{init}} \geq 1 \end{cases} \tag{2.16}$$

Even if this assumption affects the details of the damage propagation, it does not change the fundamentals of the implementation. Other forms for the damage evolution can be used.

2.4.3 Fluids

For fluids, the stress field is defined as

$$\begin{aligned} \boldsymbol{\sigma}_f &= 2\mu_N\dot{\boldsymbol{\epsilon}} - \frac{2\mu_N}{3}\mathrm{tr}(\dot{\boldsymbol{\epsilon}})\mathbf{I} - \hat{p}\mathbf{I} \\ &= 2\mu_N\left[\dot{\boldsymbol{\epsilon}} - \frac{1}{3}\mathrm{tr}(\dot{\boldsymbol{\epsilon}})\mathbf{I}\right] - \hat{p}\mathbf{I} \end{aligned} \tag{2.17}$$

where the term in the bracket is the deviatoric part of the strain rate tensor defined as $\dot{\boldsymbol{\epsilon}} = 0.5(\mathbf{L} + \mathbf{L}^T)$ where $\mathbf{L}$ denotes the gradient velocity tensor; μ_N is the shear viscosity, A superscript f was used to label the fluid stress, as in an FSI problem, one has to deal with two stress fields—one for the solid and one for the fluid.

The pressure of the fluid particle is updated by an EOS. In the case of water and air, the EOS is given by [10,33]

$$\hat{p} = \kappa\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right] \tag{2.18}$$

where ρ_0 is the initial density and κ is the bulk modulus chosen such that the fluid is nearly incompressible and $\gamma = 7$ for water and $\gamma = 1.4$ for air. As this EOS provides a direct relationship between pressure and density, there is no need to solve any additional equation for the pressure, which is a big advantage.

The bulk modulus can be very high for a nearly incompressible fluid, such as water, resulting in a very small time step. To increase the time step, a reduced bulk modulus can be used as long as the change in density is less than 3%.

2.5 Adaptive time step

As explicit time integrations are only conditionally stable, explicit dynamics MPM must employ a time step smaller than a critical value so that errors will not be so amplified from time step to time step that the error will quickly swamp the solution. In typical explicit MPM simulations, an adaptive time step is employed i.e., the time step is adjusted according to the particle velocities instead of being fixed. One first computes the dilatational wave speed c_{dil}:

$$c_{\text{dil}} = \sqrt{\frac{\lambda + 2\mu}{\rho}} = \sqrt{\frac{K + \frac{4}{3}G}{\rho}} \tag{2.19}$$

Next, one computes the maximum wave speed using the following equation [1]

$$c = \left(\max_p(c_{\text{dil}} + |v_{xp}|), \max_p(c_{\text{dil}} + |v_{yp}|), \max_p(c_{\text{dil}} + |v_{zp}|)\right) \tag{2.20}$$

where v_{xp} is the x component of particle p's velocity. For hyper-velocity impact problems, the above equation, where the particle velocity is taken into account, is very much needed.

The time step Δt is then chosen as follows

$$\Delta t = \alpha\frac{h}{c} \tag{2.21}$$

where h is the cell spacings and α is a time step multiplier ranging from 0 to 1. This factor is needed as the stability analysis was done for linear problems.

Usually, α is taken as a constant. However, for problems involving plastic deformation under high shear stresses, this is not satisfactory. Indeed, when materials deform plastically, there is little to no volume change, which means that c_{dil} remains nearly constant. However, the density of particles along certain axes will change, causing the need for Δt to be adjusted to avoid instabilities. In `Karamelo`, this is done by scaling α with the lowest eigenvalue of the deformation matrix $F_{\lambda_{\min}}$:

$$\Delta t = \alpha \frac{h}{c} = \alpha' F_{\lambda_{\min}} \frac{h}{c} \tag{2.22}$$

where α' is a constant multiplier ranging from 0 to 1.

3 Implementation

This section presents the design and implementation of `Karamelo`. We start with Sect. 3.1 that outlines the code. The code's particular class system inherited from `LAMMPS` is described in Sect. 3.2. Next, pre-processing and post-processing is discussed in Sect. 3.3. Then, the user interface is introduced in Sect. 3.4 by detailing the input file's syntax. `Karamelo` is fully parallelized using MPI. The way this is done is explained in Sect. 3.5. Compiling an open-source project is always a daunting operation. In `Karamelo`, compilation is easy as you will later see in Sect. 3.6. `Karamelo` has been created to be easy to extend for rapid prototyping of new ideas while using an efficient core. How to extend it is finally presented in Sect. 3.7.

3.1 `Karamelo` in a nutshell

`Karamelo` is an explicit dynamics MPM code using a Cartesian background grid. It implements various weighting functions including hat functions, cubic B-splines, quadratic Bernstein, and CPDI. It can be used in an updated Lagrangian description or in a total Lagrangian description. One-, two- and three-dimensional simulations are supported, and for 2D problems, plane strain and axi-symmetric conditions are available.

A no-slip no penetration contact is inherent of the updated Lagrangian MPM algorithm. Frictional contacts for ULMPM have been developed, but are not yet implemented in `Karamelo` [5,6]. However, a slip contact model is available for the total Lagrangian model as described in [11].

The simulations' output is `LAMMPS` dump files and can therefore be visualized using `Ovito` [49]. At the time this article was written, the following material models are provided

- isotropic linear elastic material;
- Neo-Hookean hyperelastic material;
- small strain J2 elasto-plastic material;
- large strain hypoelastic plastic materials;
- EOS for weakly compressible fluids and gases.

This list is in constant evolution. Please check the code's official webpage for the updated list of supported materials: www.karamelo.org.

3.2 Hierarchical class system

`Karamelo` is a C++ code that uses a hierarchical class system directly inherited from `LAMMPS`. At its centre lies the Pointers class. All the main classes (to the exception of the classes MPM and Var) are inherited from it as shown in Fig. 4. This structure allows all these classes to access elements from all the other classes, while being independent.

When `Karamelo` is launched, it creates the class MPM which contains all the pointers to the other main classes which are (see Fig. 4):

- `Universe`: sets up partitions of processors so that multiple simulations can be run, each on a subset of the processors allocated for a run, e.g. by the mpirun command
- `Domain`: stores the simulation dimensions (i.e. 1D, 2D or 3D), the simulation box geometry, the list of the user defined geometric regions and solids, as well as the background grid if updated Lagrangian is used.
- `Grid`: stores all information related to the background grid(s): number of cells as well as all the nodes' properties such as position, velocity, mass, etc. It also updates the Grid updating step (Fig. 3).
- `Material`: stores all the user defined Equations of State, elasto-plastic, damage, and temperature laws as well as the different materials which are a combination of the formers.
- `Error`: prints all error and warning messages.
- `Input`: reads an input script, stores variables, and invokes stand-alone commands that are children classes of the other main classes.
- `Modify`: stores the list of compute and fixes classes, both of which are parent styles.
- `Update`: stores everything related to time steps as well as the scheme and method classes.
- `Method`: parent class of all the MPM methods supported: ULMPM, TLMPM, ULCPDI and TLCPDI, to date.

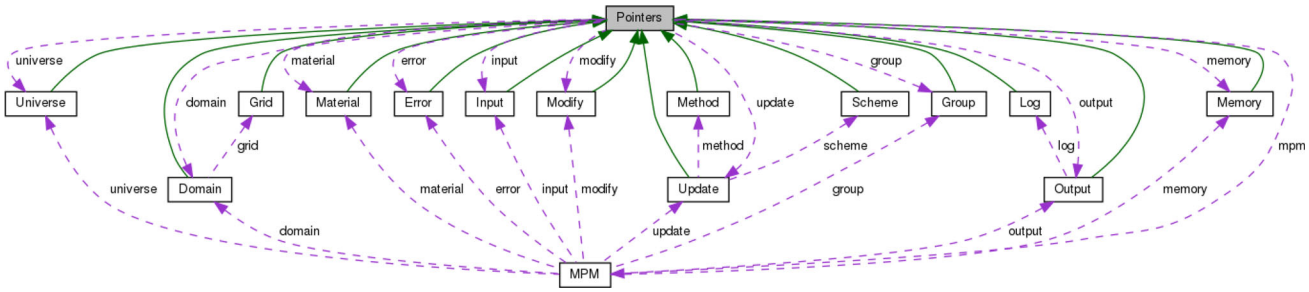


Fig. 4 Class hierarchy within Karamelo source code. All classes are derived from the class `Pointers` (parent class). This diagram was automatically generated by `Doxygen`. Green arrows show link between child classes and their parent class , while the dashed purple ones show pointers meaning that the class pointed to by the arrow is known from the class from where the arrow is emitted (for instance, the class `Update` knows the existence of `Method` and access and use its functions and variables—those declared as public)

- `Scheme`: parent class of all the computational cycle schemes supported: modified update stress last to date.
- `Group`: manipulates groups that particles or nodes are assigned to via the group command. It also computes various attributes of groups of particles or nodes.
- `Log`: it is used to generate the screen printed outputs and the log files.
- `Output`: it is used to generate 4 kinds of output from a simulation: information printed to the screen and log file, dump file snapshots, plots, and restart files.
- `Memory`: handles allocation of all large vectors and arrays.

In order to alter some property of the system during time-stepping "fixes" are used. They are the Karamelo mechanism (directly inherited from `LAMMPS`) for tailoring the operations of a time-step for a particular simulation. Essentially everything that happens during a simulation besides the regular MPM algorithm, and output, is a "fix". Example of operations are: setting boundary conditions or setting up a virtual indenter.

If a certain quantity needs to be output, "computes" need to be used. "Computes" are called to calculate the desired quantities only when a log output is requested, contrary to fixes. Each compute can return one or multiple scalar values. Some examples of computes are: calculating the total strain energy, or calculating the maximum plastic strain.

3.3 Pre and post-processing

`Karamelo` supports two types of particle generation. For simple geometries, it can directly generate the particles. The algorithm is simple: First, a Cartesian grid is built, and the user decides how many particles will populate each cell. Then, the code generates particles for all cells and discard those that fall outside the boundary of the geometry. Supported geometries include spheres, cylinders and blocks in 3D. The corresponding 2D geometries are disks and rectangles. For complex geometries, `Karamelo` can read a finite element mesh and generate particles as centroids of these elements. Currently it supports `Gmsh`—an open source unstructured mesh generator [17].

Similar to `LAMMPS`, the simulation snapshots are visualized using `Ovito` [49]. Furthermore, the `PyPlot` graphical package [23] is used to generate graphs featuring the evolution of user-defined quantities.

3.4 Input files

Interacting with `Karamelo` is performed through input files using an easy and flexible syntax. One can for example intuitively add new variables that can be constant:

```
E = 211
```

or depend on internal variables such as time—using the `time` variable—or the particle positions—using the `x`, `y` or `z` variables:

```
T      = 1
r      = sqrt(x*x+y*y)
g      = sin(PI*time/T)
```

Everything else is controlled through functions. For instance, the global dimensionality of the simulation, the domain's size, and the background grid cell size are set by the dimension(…) command:

```
dimension(2, xlo, xhi, ylo, yhi, cellsize)
```

A typical input file is given in Listing 1. Basically, it performs the following actions:

- define some constants (lines 4–8, 12 and 13, 20, 30 and 31);
- define the method (ULMPM or TLMPM or CPDI) together with the basis function (line 10);
- define the dimension together with the computational domain (line 14);

– define regions (or geometries) (lines 16 and 26). Simple geometries (blocks, cylinders, spheres) are supported;
– define materials (line 18)
– define solids using regions and materials (line 21);
– define particle and node groups (lines 23 and 27);
– define initial conditions and boundary conditions on the particle/node groups (lines 24 and 28);
– define outputs (line 32).
– define the CFL factor and the total simulation time (lines 34 and 35);

We refer to the appendices for complete input files of some simulations presented in Sect. 4.

Listing 1: A compact input file.

```
1  ######################################################
2  #                  UNITS: MPa, mm, s                 #
3  ######################################################
4  E    = 1e+3                    # Young's modulus
5  L    = 1                       # a dimension
6  hL   = 0.5*L
7
8  FLIP=1.0
9  #------------ SET METHOD ---------------------#
10 method(ulmpm, FLIP, linear, FLIP)
11 #------------ SET DIMENSION ------------------#
12 N         = 20      # 20 cells per direction
13 cellsize = L/N     # cell size
14 dimension(2,-hL, hL, -hL, hL, cellsize)  # 2D problem, which the computational domain is LxL
15 #------------ SET REGIONS--------------------#
16 region(rBall2, cylinder, hL-R, hL-R, R)
17 #------------ SET MATERIALS------------------#
18 material(mat1, linear, rho, E, nu)
19 #------------ SET SOLID ----------------------#
20 ppc1d = 2
21 solid(sBall1, region, rBall1, ppc1d, mat1, cellsize,0)
22 #------------ IMPOSE INITIAL CONDITIONS ------#
23 group(gBall1, particles, region, rBall1, solid, sBall1)  # define particle group1
24 fix(v0Ball1, initial_velocity_particles, gBall1,  0.1,  0.1, NULL)
25 # ------------------- IMPOSE BOUNDARY CONDITIONS ------------------------
26 region(region1, block, INF, INF, INF, cellsize/4, INF, INF)
27 group(groupn1, nodes, region, region1, solid, plate)
28 fix(BC_bot,  velocity_nodes, groupn1, NULL, NULL, NULL)
29 #------------OUTPUT-----------------------------------------#
30 N_log = 50
31 dumping_interval = N_log*1
32 dump(dump1, all, particle, dumping_interval, dump_p.*.LAMMPS, x, y, z)
33 #------------ RUN ----------------------------#
34 set_dt(0.001)   # constant time increments of 0.001
35 run_time(3.5)   # run for a period of 3.5 seconds
```

3.5 Parallelization using MPI

Similarly to `LAMMPS`, `Karamelo` supports multi-CPU computations through the use of MPI (Message Passing Interface). As of today, and contrary to `LAMMPS`, it does not support GPUs yet. In `Karamelo`, the total domain defined by the span of the background grid is equally split amongst the different CPU used (see Fig. 5). The connection between the different sub-domains is performed by the means of ghost nodes. The use of ghost nodes was preferred to that of ghost particles since [41] observed that the later is superior over the former for scaling to large-scale problems. Therefore, the nodes making of all the boundary mesh cells are shared amongst multiple CPUs (Fig. 5). The particles are stored in the processor in which their respective background cell is also stored. First, the interaction between these nodes and the local particles (i.e., the particles present in the domain allocated to the current CPU) are calculated. Then, the results are reduced over all the CPUs sharing the same nodes. In one time step, reduction is done once after calculating the nodes' mass, and every time their forces and velocities are computed. In the case of CPDI, however, as the domain of a given particle can be overlapping multiple CPUs, both ghost nodes and ghost particle's approach are used.

3.6 Compilation

Building an open-source project can be a daunting task for many users. However, with `Karamelo`, there is no hardship as building it has been made easy with the use of `CMake` (www.cmake.org). `CMake` automatically checks dependencies and generates the Makefile.

As `Karamelo` relies on two non-standard libraries that are `gzstream` (https://www.cs.unc.edu/Research/compgeom/gzstream/) and `matplotlib-cpp` (https://github.com/lava/matplotlib-cpp), the only user intervention needed to correctly build `Karamelo` is to make sure that these libraries' source code and object file are present and that

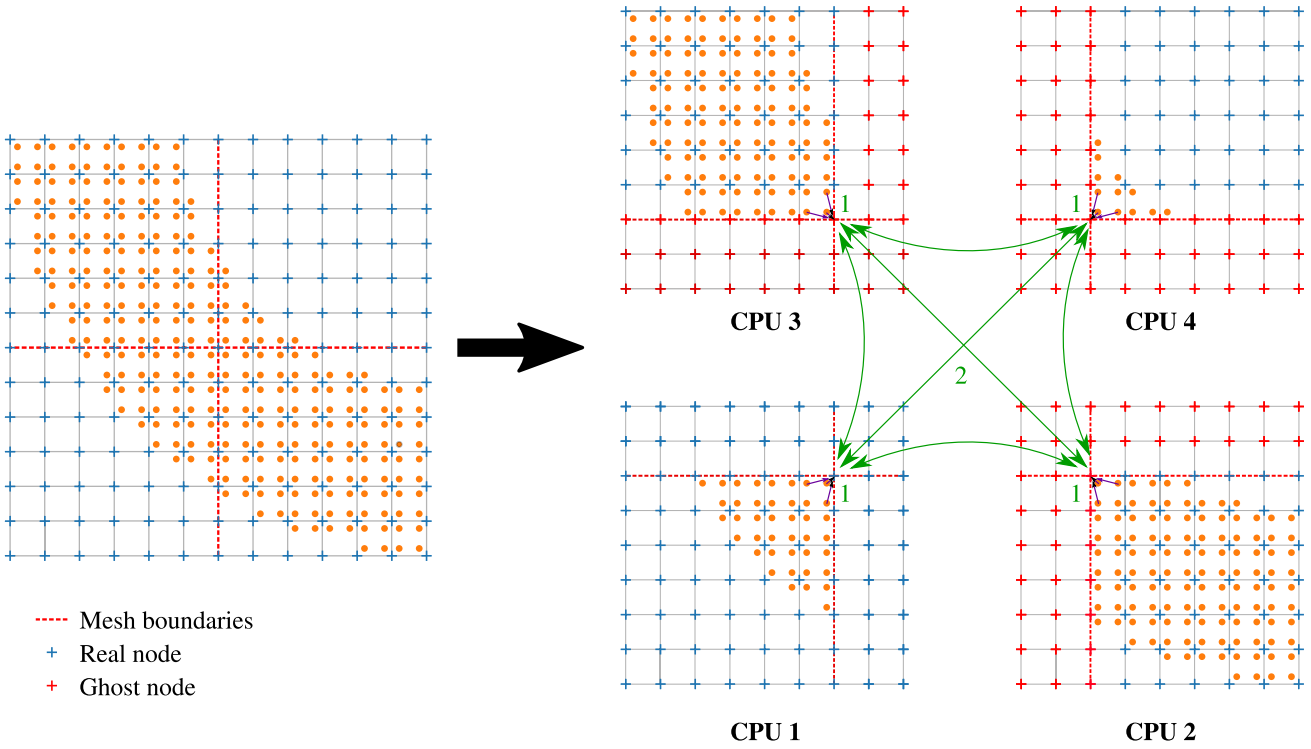


Fig. 5 Illustration of the mapping from particles to node order for nodes that are shared amongst different CPUs. 1: mapping is done locally, 2: the mass, or forces and velocities are reduced. The CPU that on which resides the real node is the one that manages the reduction

the link to it's directory is provided in both the `CPATH` and `LIBRARY_PATH`.

3.7 Extending Karamelo

`Karamelo` has been created to combine rapid prototyping of new ideas with the use of efficient software engineering. This has been achieved using the ingenious hierarchical class system inherited from `LAMMPS`. Extending `Karamelo` is therefore relatively easy as is explained in the following.

The easiest way to extend `Karamelo` is by the implementation of new fixes. Using fixes, users can implement many operations that can alter the system such as: changing particles or node attributes (position, velocity, forces, etc.), implement boundary conditions, reading/writing data, or even saving information about particles for future use (previous positions, for instance).

Adding a Fix All fixes are derived from class `Fix` and must have a constructor with the signature: `FixNew(class MPM *, vector<string>)`.

Every fix must be registered in `Karamelo` by writing the following lines of code in the header before include guards:

```
#ifdef FIX_CLASS
FixStyle(fix_name,FixNew)
#else
```

where "fix_name" is a name of the fix in the script and FixNew is the name of its class. This little piece of code allows `Karamelo` to find the new fix when it reads an input file. In addition, the fix header must be included in the file "style_fix.h". Once this is done and the project build again, the fix can be used using the following function:

```
fix(fix_ID, fix_name, arg_1, arg_2,  ..., arg_N)
```

Remark 1 The number of arguments the fix function takes is variable.

Adding a Compute All computes are derived from class Compute and must have a constructor with the signature: `ComputeNew(class MPM *, vector<string>)`.

Every compute must be registered in `Karamelo` by writing the following lines of code in the header before include guards:

```
#ifdef COMPUTE_CLASS
ComputeStyle(compute_name,ComputeNew)
#else
```

where "compute_name" is a name of the compute in the script and ComputeNew is the name of its class. This little piece

of code allows Karamelo to find the new compute when it reads an input file. In addition, the compute header must be included in the file "style_compute.h". Once this is done and the project build again, the compute can be used using the following function:

```
compute(compute_ID, compute_name, group_ID)
```

where group_ID is the name of the group over which the desired quantity will be calculated.

Adding a computational cycle schemes New computational cycle schemes can be added as easily as fixes. This could allow a given user to simply test a new scheme or add one that is not currently supported. Schemes are derived from the class Scheme. They must have constructor with the signature: SchemeNew(class MPM *, vector<string>) and feature their own definition of the setup() and run(Var condition) functions. The former is used to do any required setup at the beginning of the simulation, before time-stepping start. The later features the implementation of the time-stepping loop and takes a condition as only argument.

Similarly to fixes, every scheme must be registered by writing the following lines of code in the header before include guards:

```
#ifdef SCHEME_CLASS
SchemeStyle(scheme_name,SchemeNew)
#else
```

where "scheme_name" is a name of the new scheme in the script and SchemeNew is the name of its class. Also, the scheme header must be included in the file "style_scheme.h". The use of the new added scheme is done by invocating the following:

```
scheme(scheme_name)
```

Remark 2 The scheme selected by default is modified update stress last (MUSL).

Adding an MPM variant The same process used for adding fixes and schemes is used to add MPM variants. The class corresponding to a variant is derived from the class Method. It must have a signature of the type NewMPM(class MPM *, vector<string>) and feature the definition of a number of functions corresponding to the different steps of the algorithm. New variants must be registered as follows:

```
#ifdef METHOD_CLASS
MethodStyle(newmpm,NewMPM)
#else
```

where "newmpm" is a name of the added variant. Just like for fixes and schemes its header must be included in the file "style_method.h". In the input file, the function method defines the use of a given MPM variant. For example the function

```
method(newmpm, PIC, linear)
```

tells Karamelo to use the "newmpm" variant of MPM with the PIC integration scheme and linear shape functions.

4 Examples

This section presents simulations that involve large deformation, contacts and fracture which are signature application of MPMs (or any meshfree methods). Furthermore, some simulations are practical engineering problems of relevance. Concretely, the following problems are presented

- Collision of two elastic disks (Sect. 4.1);
- Upsetting of a cylindrical billet (Sect. 4.2);
- Cold spraying (Sect. 4.3);
- Machining of a ductile material (Sect. 4.4);
- Dam break problem (Sect. 4.5);
- Scalability test (Sect. 4.6).

The first example is a simple collision of two elastic disks. The second example is billet upsetting used in manufacturing as a means to pre-form workpieces prior to applying another operation such as rolling or extrusion. It demonstrate both 3D and 2D axi-symmetric formulations. The third example considers the simulation of cold spraying process. The fourth example is the machining of a ductile material and the prediction of chip formation. All the first four examples are applications of Karamelo in solid mechanics. The fifth example is an applications in fluid mechanics. Finally, the scalability of Karamelo is demonstrated for problems involving 74 thousands particles. The PIC-FLIP mixing parameter was chosen to be: $\alpha = 0.99$ for solid mechanics problems and $\alpha = 1.0$ for fluid mechanics problems. Unless stated otherwise, the ULMPM was used.

As the paper focuses on the code and not the physics, we do not present validation of our simulations (except the dam break example which is the first fluid mechanics simulation carried out using Karamelo). Note that the MPM has been validated against experiments, see for instance [25,46,47], and solid mechanics simulations carried out by Karamelo have been validated in our previous work [12].

4.1 Collision of two elastic disks

The problem of collision of two elastic bodies is perhaps the easiest 2D problem that can be used to check the imple-

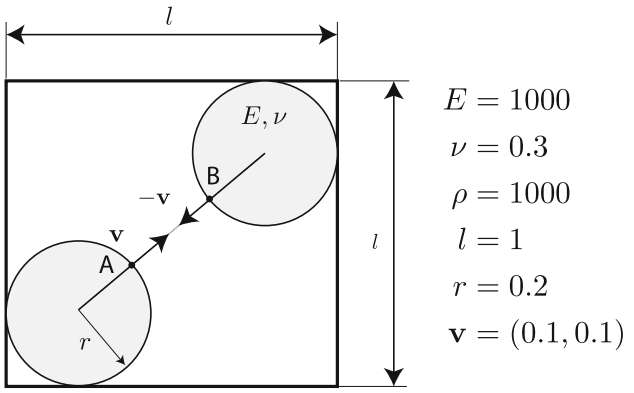


Fig. 6 Impact of two elastic bodies: problem statement. The computational domain is a unit square and the radius of the disks is 0.2 mm. Any set of consistent units is sufficient

mentation of a 2D MPM code [54]. As shown in Fig. 6, two identical elastic disks, moving in opposite directions towards each other, will collide with each other and rebound. The computational domain is a square, of which side is one millimetre. There is no boundary conditions in this problem since the simulation stops after impact and before the particles move out of the computational box. A plane strain condition is assumed.

The computational domain is discretized into 20×20 square elements. Four particles are used per elements resulting in a total of 416 particles. Simulations are performed in the absence of gravity. A constant time step $\Delta t = 0.001$ s was used. 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. A time step of 0.001 s was used. The input file for this problem is provided in Appendix A.

In order to check the energy conservation, the strain and kinetic energy are computed for each time step. They are defined as

$$E_s = \sum_{p=1}^{n_p} u_p V_p, \quad E_k = \frac{1}{2} \sum_{p=1}^{n_p} \mathbf{v}_p \cdot \mathbf{v}_p m_p \tag{4.1}$$

where u_p denotes the strain energy density of particle p, $u_p = 1/2 \sigma_{p,ij} \epsilon_{p,ij}$.

Figure 7 shows the movement of two disks. The collision occurs in a physically realistic fashion, although no contact law has been specified. Figure 8 plots the evolution of the kinetic, strain and total energy. The initial kinetic energy is $K = 2 \times (0.5 \times (v^2 + v^2) \times \rho \times \pi \times r^2) = 2.513$. The kinetic energy decreases during impact and is then mostly recovered after separation. The strain energy reaches its maximum value at the point of maximum deformation during impact and then decreases to a value associated with the free vibration of the disk. The result is identical to the ones reported in

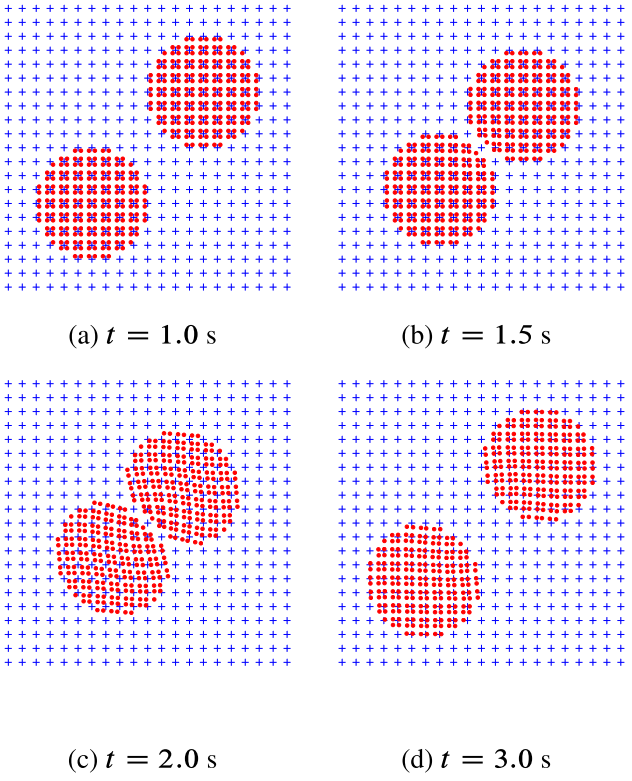


Fig. 7 Impact of two elastic bodies: simulation snapshots. These images were created using the `PyPlot` graphical package [23]

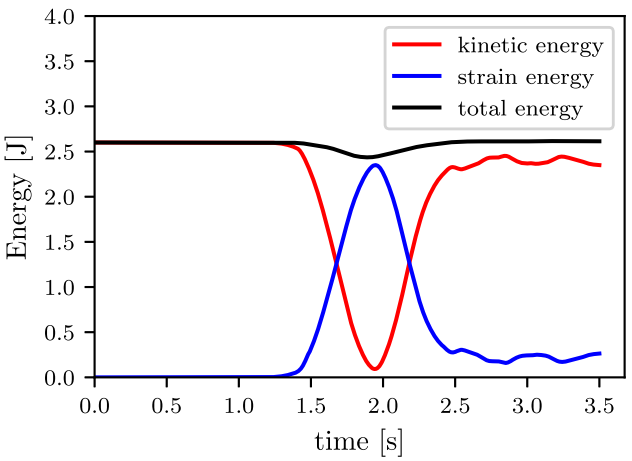
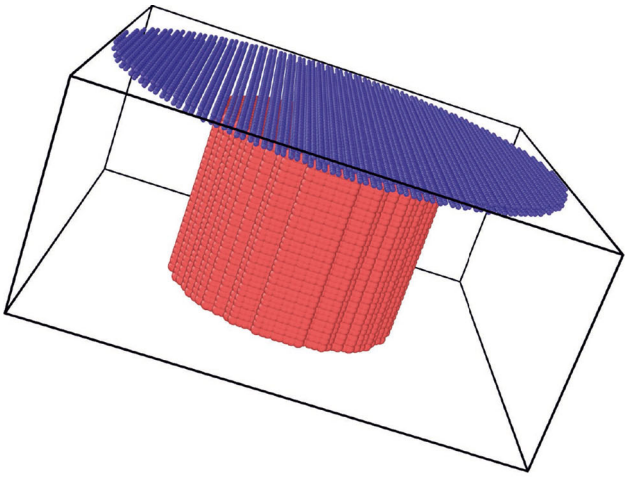
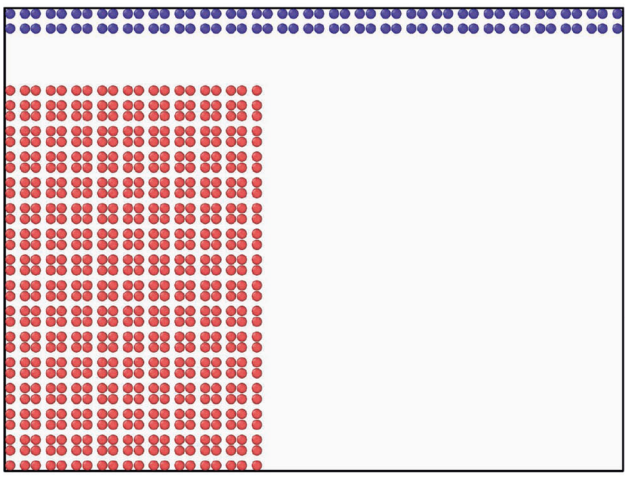


Fig. 8 Impact of two elastic bodies: evolution of kinetic, strain and total energies

[54] which confirms the correctness of the implementation. However the contact did occur earlier than it should have been. The correct contact time is $t = AB/(2\sqrt{2}v) = 1.5858$ s where $v = 0.1$. In the simulation contact happened at $t = 1.3$ s. This result is expected as the contact is resolved at the grid nodes and not at the particles i.e., contact is detected when the particles of the two bodies are one cell way from each other.



(a) Full 3D model



(b) 2D axi-symmetric model

Fig. 9 Upsetting of a cylindrical billet: initial particle distribution

4.2 Upsetting of a cylindrical billet

In upsetting, a cylindrical billet is compressed between two platens. This process is used in manufacturing as a means to pre-form workpieces prior to applying another operation such as rolling or extrusion. The benchmark problem consists of a steel cylinder 20 mm in diameter and 30 mm in height. Due to symmetry, only half of the cylinder is modelled. The initial distribution of the particles is shown in Fig. 9 for both 3D and axi-symmetric simulations.

The platen starts above the workpiece and moves down at a constant velocity of 1 m/s, compressing the billet. For simplicity, it is assumed that there is no slip between the platen and the cylindrical workpiece. The billet is made of steel with a Young's modulus of 200 GPa, Poisson's ratio of 0.3, density of 7800 kg/m^3, and an initial yield strength of 0.70 GPa with a strain hardening slope of 0.30 GPa. That is, in the Johnson–Cook model [22], $A = 0.70$ GPa, $B = 0.30$ GPa, $C = 0$ and $n = 1.0$.

Table 2 Material parameters for copper [2]

Material parameters	
Density	8960 kg/m^3
Young's modulus	124 GPa
Poisson's ratio	0.3
Johnson–Cook flow stress model parameters	
A	90 MPa
B	292 MPa
C	0.025
n	0.31
m	1.09
EOS parameters	
c_0	3940 m/s
S_α	1.489
Γ_0	2.02
Johnson–Cook damage parameters	
D_1	0.54
D_2	4.89
D_3	−3.03
D_4	0.014
D_5	1.12
Temperature model parameters	
T_r	298 K
T_m	1356 K

The 3D simulation consists of 47,744 particles (including the rigid particles modelling the platen), whereas the axi-symmetric one contains only 696 particles. And yet, their results, given in Fig. 10, are very similar to that of [51].

4.3 Cold spraying

Cold spraying (CS) is a coating deposition method in which solid powders (1–50 μm in diameter) travelling at velocities up to 1200 m/s impinge on a substrate. During this impact, particles undergo plastic deformation and adhere to the surface. Unlike thermal spraying techniques, e.g., plasma spraying, flame spraying, or high velocity oxygen fuel, the powders are not melted during the spraying process.

The coating is formed from many individual feed-stock impact events. Therefore, the understanding of a single feed-stock impact and its resulting morphology is vital to shed lights on the key parameters affecting bulk coating properties.

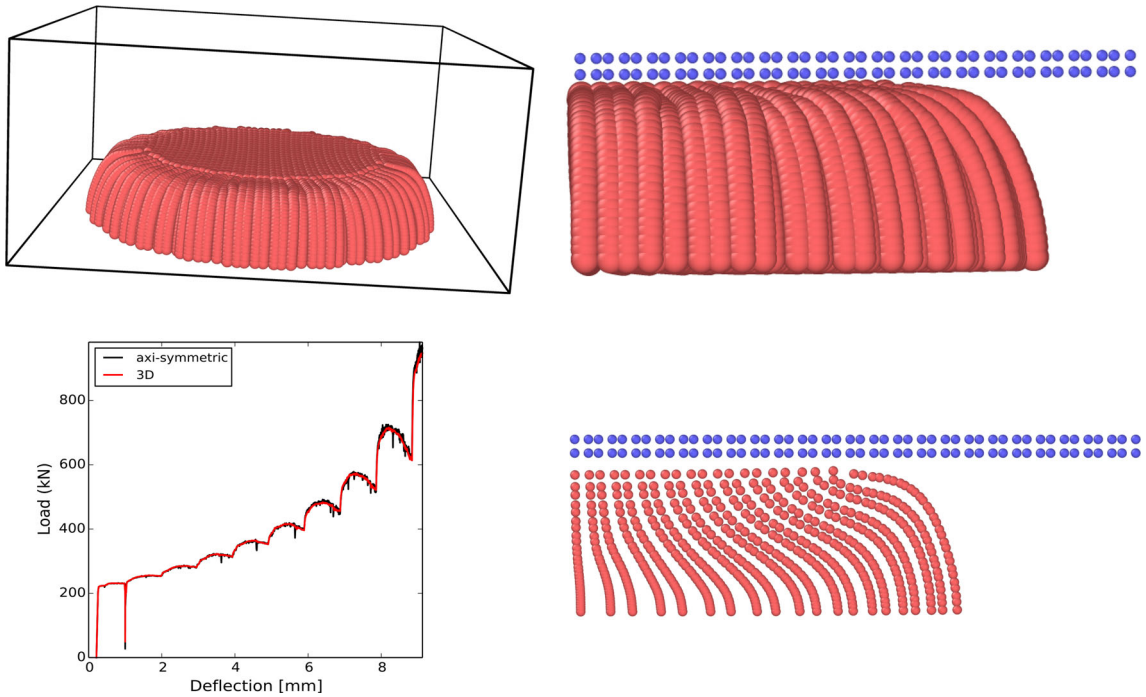
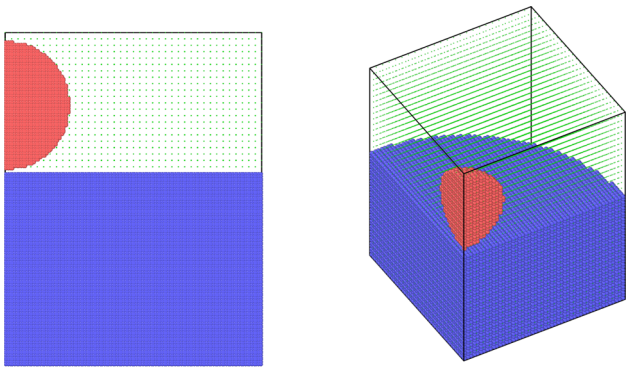


Fig. 10 Upsetting of a cylindrical billet: results in terms of deformed shape and load-deflection curves. The deflection is the averaged displacement of the particles on the top surface of the cylinder and the load is computed as a sum of all nodal internal force in the cylinder direction of all nodes locating on the bottom surface of the cylinder



(a) axi-symmetric model (b) 1/4 3D model

Fig. 11 Cold spraying with a single impact: 2D axi-symmetric model and 1/4 3D model. The bottom surface of the substrate is fixed, whereas in the symmetric planes, the nodes are fixed in the direction normal to these planes. The green dots denote the background grid nodes

It is very difficult to observe the whole deformation process via experiments because of the extremely short impact time scale. Numerical simulations thus play an important role in studying the spray powder deformation process. Common numerical methods include Lagrangian FEM, see e.g., [2], Eulerian methods [27] and Smoothed Particle Hydrodynamics (SPH), see e.g., [18,31]. From the review article of [59], Eulerian and SPH methods are more accurate than the FEM.

The most common scenario considered in the literature is a single impact: a spherical powder particle impacts a cylinder substrate. The spherical powder has a radius r of 10 μm and the cylindrical substrate is $8r$ in diameter and $3r$ in height. Both the powder and the substrate are made of copper, as it is the most widely used material in cold spray simulations. The material parameters used in this paper are given in Table 2 taken from [2].

Both 2D axi-symmetric model and 3D model are considered, see Fig. 11. For the 2D case, the domain is discretized by 40×52 cells with 9 particles per cell (12,218 particles). For the 3D case, it is discretized by $25 \times 25 \times 33$ cells with 8 particles per cell (74,919 particles).

As heat conduction is not yet coded in `Karamelo`, only adiabatic condition is considered and the temperature increase is due only to plastic deformation. Note that the contact between the powder and the substrate is frictionless as frictional contact is not available yet.

Some results are shown in Figs. 12 and 13. Figure 13 presents the evolution of the plastic strain and temperature in time. The axi-symmetric and 3D results are quite similar.

4.4 Machining simulations

Machining is a controlled material removal process in which a piece of material is cut into a desired final shape and size.

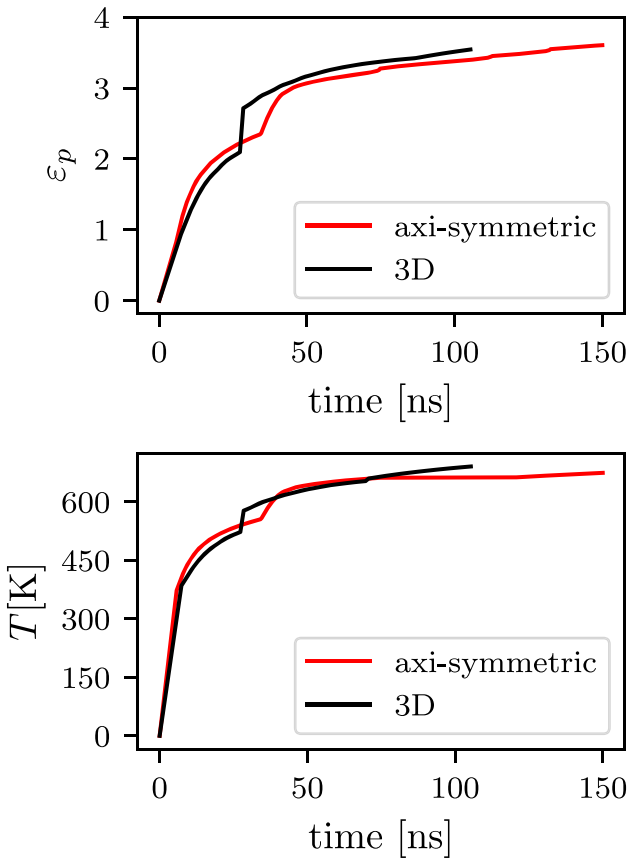


Fig. 12 Cold spraying with a single impact: evolution of plastic strain and temperature

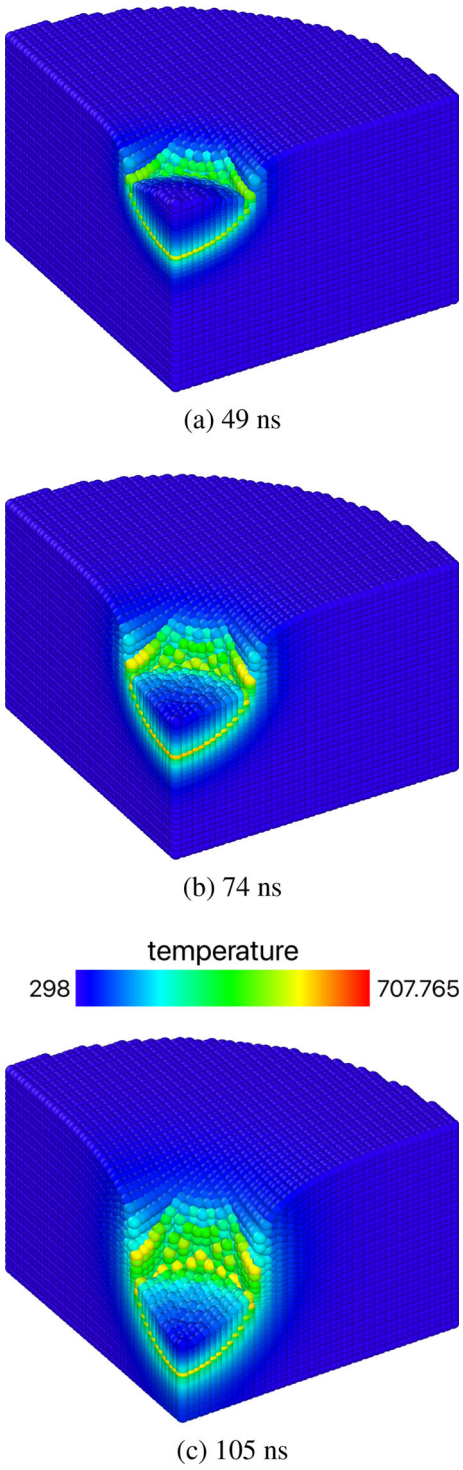


Fig. 13 Cold spraying with a single impact: deformation process (3D model)

Moreover, it is one of the most prominent industrial applications in the manufacturing field. Numerical simulations play an important role in improving this crucial process. During the material cutting process, severe deformations occur. Thus, accurate simulations of the process with mesh-based methods (i.e. finite element methods) are complicated owing to mesh distortion problems. Therefore, particle-based methods like the MPM seem more appropriate for this kind of simulations.

The total Lagrangian MPM is here used to simulate the cutting process of a workpiece which is a slab of 1 mm in length, and 0.3 mm in height (Fig. 14). The material of the workpiece is a high strength steel of which elasto-plastic and damage parameters, taken from [3], are given in Table 3. The cutting depth and speeds are 0.15 mm and 50 mm/ms, respectively.

To save computational time, 2D simulations are considered. The domain is discretized by 320 × 80 cells with one particle per cell (25,600 particles).

Here, only the workpiece is discretized. The cutting tool is not modelled by particles, but rather by as a domain impenetrable to particles. This domain is described by the area between the half-lines $[TA)$ and $[TB)$, where T is the cutting edge of the tool, and A and B are points lying on the flank and rack face, respectively (see Fig. 14). A given particle would have entered the cutting tool's domain when:

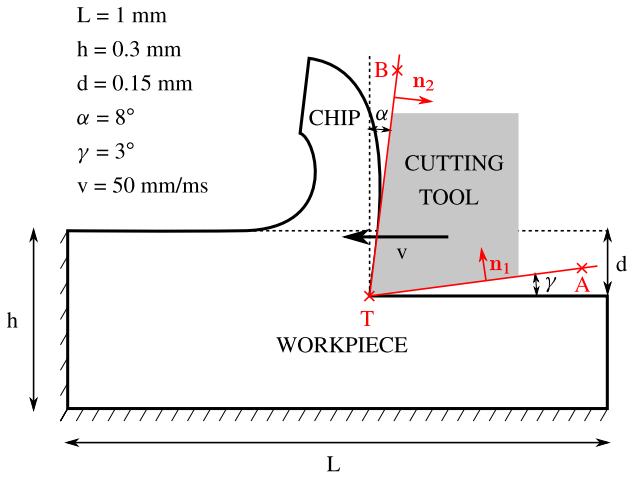


Fig. 14 Schematic of the 2D machining simulation setup

Table 3 Material parameters for the high strength steel used in the machining simulation

Material parameters	
Density	7750 kg/m^3
Young's modulus	211 GPa
Poisson's ratio	0.33
Johnson–Cook flow stress model parameters	
A	980 MPa
B	2000 MPa
C	0.0
n	0.83
EOS parameters	
c_0	5166 m/s
S_α	1.5
Γ_0	2.17
Johnson–Cook damage parameters	
D_1	0.05
D_2	0.8
D_3	−0.44
D_4	0
D_5	0

$$p_1 = \frac{(y_A - y_T)x_p + (x_T - x_A)y_p + y_T x_A - x_T y_A}{\sqrt{(y_A - y_T)^2 + (x_T - x_A)^2}} \geq 0 \tag{4.2}$$

and

$$p_2 = \frac{(y_B - y_T)x_p + (x_T - x_B)y_p + y_T x_B - x_T y_B}{\sqrt{(y_B - y_T)^2 + (x_T - x_B)^2}} \geq 0 \tag{4.3}$$

If a particle has entered the tool's domain (i.e. Eqs. (4.2) and (4.3) are satisfied), a repulsive force $\mathbf{F}_{\text{tool}}$ is applied at its centre. This force is derived from Hertz' contact theory as:

$$F_{\text{tool}} = \frac{\pi}{2} G \left(\frac{1+\nu}{1-\nu^2} \right) p^{3/2} \tag{4.4}$$

where G is the substrate's shear modulus, ν its Poisson's ration, and $p = \min(p_1, p_2)$ the penetration distance between a particle and the cutting tool. The contact between the tool and the workpiece is supposed to be frictionless, therefore the direction of $\mathbf{F}_{\text{tool}}$ is normal to the tool's cutting surface, i.e.:

$$\begin{cases} \mathbf{F}_{\text{tool}} = -F_{\text{tool}} \mathbf{n}_1, & \text{if } p_1 < p_2 \\ \mathbf{F}_{\text{tool}} = -F_{\text{tool}} \mathbf{n}_2, & \text{otherwise} \end{cases} \tag{4.5}$$

where $\mathbf{n}_1$ and $\mathbf{n}_2$ are the inside-facing normals to the flank and rack face, respectively (see Fig. 14).

The use of TLMPM is here motivated by the necessity to have material separation that is independent of the background grid size, i.e. described only by physic-based damage equations. This is because the ULMPM usually suffers from numerical fracture when particles are separated sufficiently far from each others. The results of these simulations showcases the abilities of the TLMPM implementation in `Karamelo` to simulate the formation and detachment of chips from the workpiece (see Fig. 15).

Remark 3 It should be noted that we did not introduce discrete cracks into the model. In other words, the entire deformation process of cutting was modelled using the continuum damage mechanics framework. We admit that using a local damage model could result in mesh-dependent result, but as modelling fracture is not the focus of the paper, we do not want to go deeply into the difficult problem of post-localization treatment. One way is to add discrete cracks as in the cracking particle method of [40], and discrete cracks can be introduced in the MPM as shown by [34].

4.5 Dam break problem

A simplified dam break simulation with a barrier is depicted in Fig. 16. A water column of initial height h_0 and length l_0 is initially constrained by a gate and rests on a smooth, flat surface. At an arbitrary starting time, say $t = 0$, the gate is removed and the water is allowed to flow freely under the

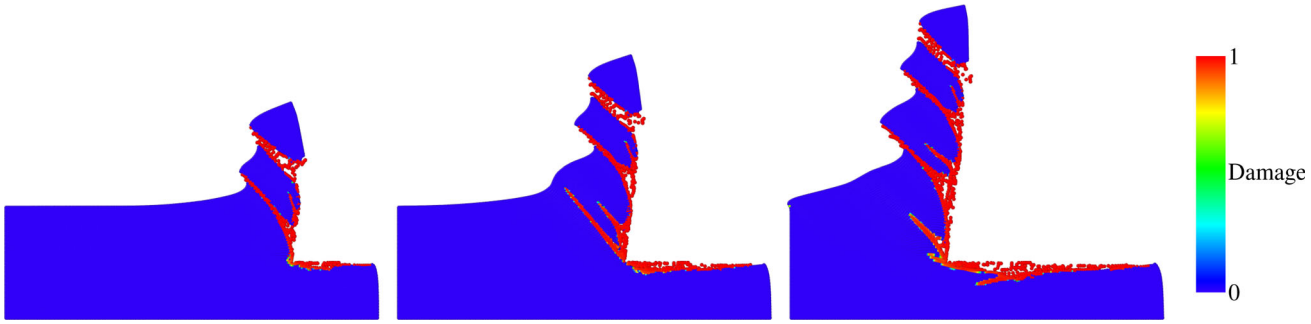


Fig. 15 Cutting process of a high strength steel (2D model)

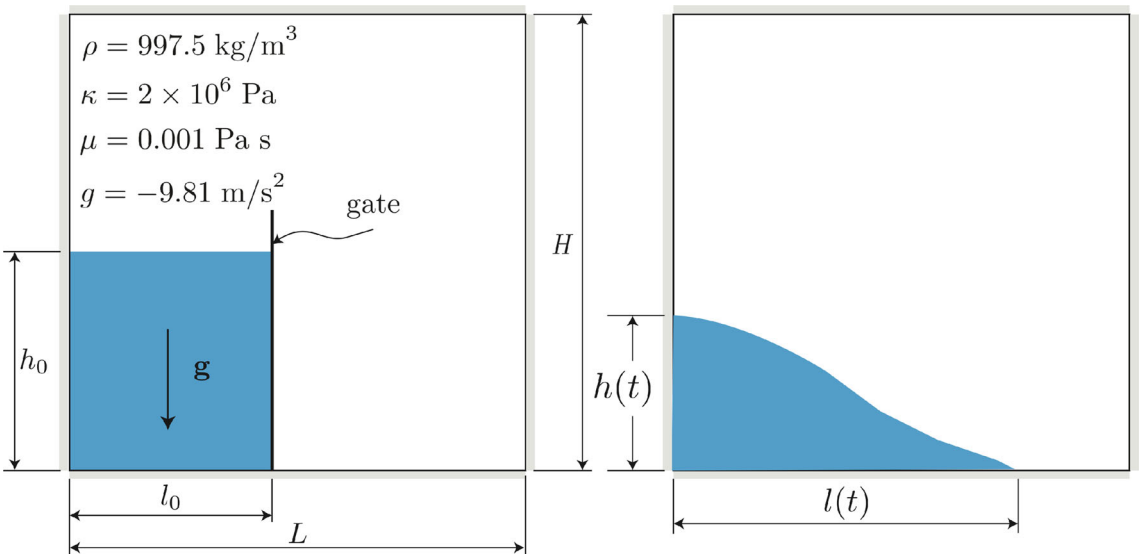


Fig. 16 Dam break problem: $L = 1.61$ m, $H = 0.6$ m, $l_0 = 0.6$ m, $h_0 = 0.6$ m according to [56]

force of gravity. This problem has been tackled extensively in the SPH literature and also studied using the MPM by e.g., [32,56].

The water pressure follows an artificial equation of state, see [13,56] for detail. The domain is discretized by 100×100 cells with 9 particles per cell (12,432 particles). A constant time step of $\Delta = 0.1 h_x/c$ with $c = \sqrt{\kappa/\rho}$ was used. The component normal to the boundaries of the grid velocities are set to zero. The aims of this example are twofold. First, it is demonstrated that `Karamelo` can do fluid simulations. Second, its solution is validated against the experiment carried out by [29].

For a quantitative assessment of the MPM, the following dimensionless quantities are calculated

$$L(T) := \frac{l(t)}{l_0}, \quad T := t\sqrt{\frac{h_0 g}{l_0^2}} \tag{4.6}$$

where $l(t)$ is defined as in Fig. 16.

The water profile at different time instants are given in Fig. 17 where it can be seen that the MPM solution is in qualitative good agreement with the experiment. Quantitatively, the simulation result and the experiment are compared in Fig. 18.

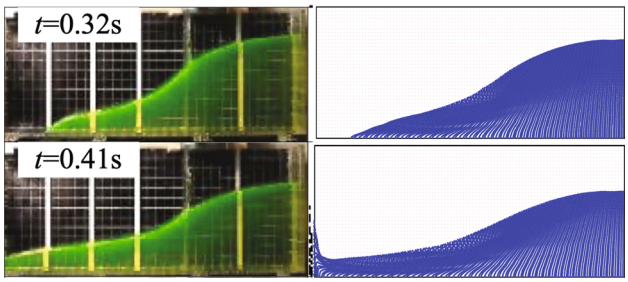


Fig. 17 Dam break problem: flow profiles of experiment results (left column) and MPM results (right column)

4.6 Scalability tests

As Moore's law is flattening (the power of CPUs do not double every two years anymore), chip manufacturers are relying on packing more computational cores onto a single

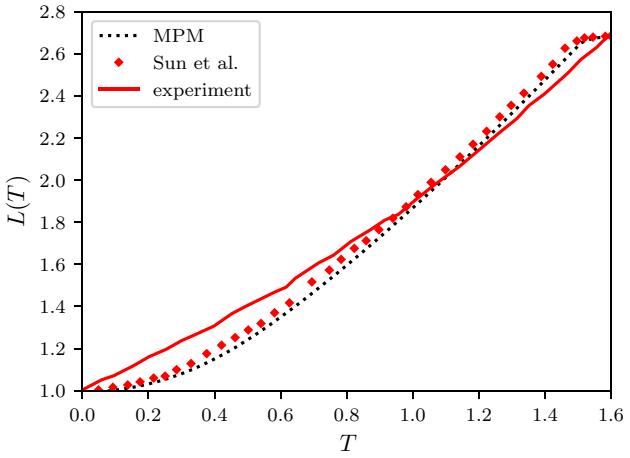


Fig. 18 Dam break problem: evolution of the MPM wave front in comparison with experiment result of [29]

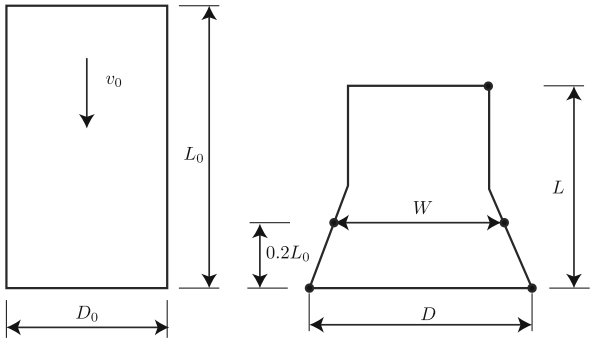


Fig. 19 Taylor bar impact for an elasto-plastic OFHC copper cylinder given an initial downward velocity of 190 m/s

chip to increase their computational power. Modern application therefore needs to be fully parallelized to make the best use of the technology available nowadays. In Sect. 3.5, the way `Karamelo` was parallelized has been presented. Here, the speedup gain obtained as the number of cores increase is presented, using the well known Taylor bar impact test. These tests are done using both TLMPM and ULMPM.

The Taylor bar impact test involves a cylinder hitting a fixed and rigid wall at a high velocity (Fig. 19). It was used by [53] to show the capabilities of ULMPM, and more recently by [12] to show those of TLMPM. The bar is a cylinder of original length $L_0 = 25.4$ mm, original diameter $D_0 = 7.6$ mm, made of OFHC Copper. This material is modelled using the Johnson–Cook elasto-plastic constitutive model, but without damage. We refer to [12] for the material parameters used in the simulations.

Similarly to what was done in the work by [12], the mesh cell size is $h = 0.25$ mm with 1 material point per element for a total of 74 052 points. Moreover, linear shape functions are used. The initial velocity is set at $v_0 = 190$ m/s. When using

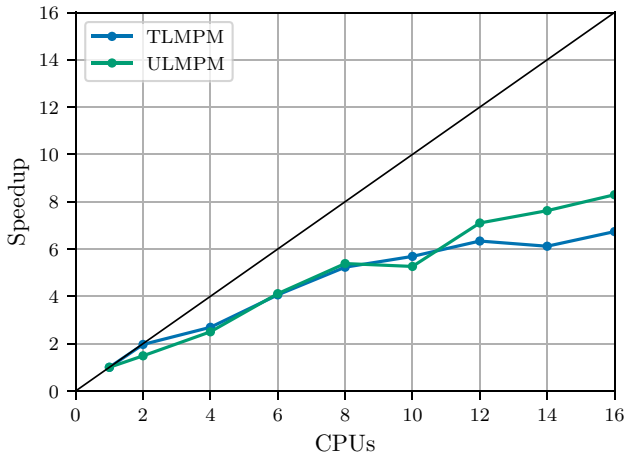


Fig. 20 Scalability of the TLMPM and the ULMPM compared using a Xeon-E5-2667-v3 type CPUs. The black line corresponds to the maximum possible speedup gain

ULMPM, the simulation cell is $25.4 \times 15.0 \times 15.0$ mm^3. All simulations are ran for a total of 20,000 steps.

These scalability tests have been performed using a Xeon-E5-2667-v3 chip made of 16 cores. The same simulations for TLMPM and ULMPM, respectively, were performed using a variable number of core, i.e. from 1 to 16. The simulation times were recorded and divided by those obtained when using a single core. As one can see in Fig. 20, the speedup gain for ULMPM and TLMPM is comparable. The increase of speed is substantial when the number of cores is lower than 10. When this number is higher, though the performances tend to plateau. This suggests that the computation time is dominated by the time passing messages, or waiting for messages.

The parallel performance of `Karamelo` is modest, but could definitely be improved. As the authors are not computer scientists, optimizing parallel execution is out of the scope of their research. However, any contribution on this aspect would be more than welcomed.

5 Conclusions

In this paper, we have presented the overall architecture of `Karamelo`—an open source parallel C++ package for the material point method—as well as some of its specific features. We hope that the engineering community will make use of the present code in ways that we cannot anticipate. The source code and examples are available at https://github.com/adevaucorbeil/karamelo/.

Although the examples presented here involved a number of particles lower than 100,000, `Karamelo` has been successfully used for simulations featuring up to 5 million particles. Moreover, the number of CPUs the problem to

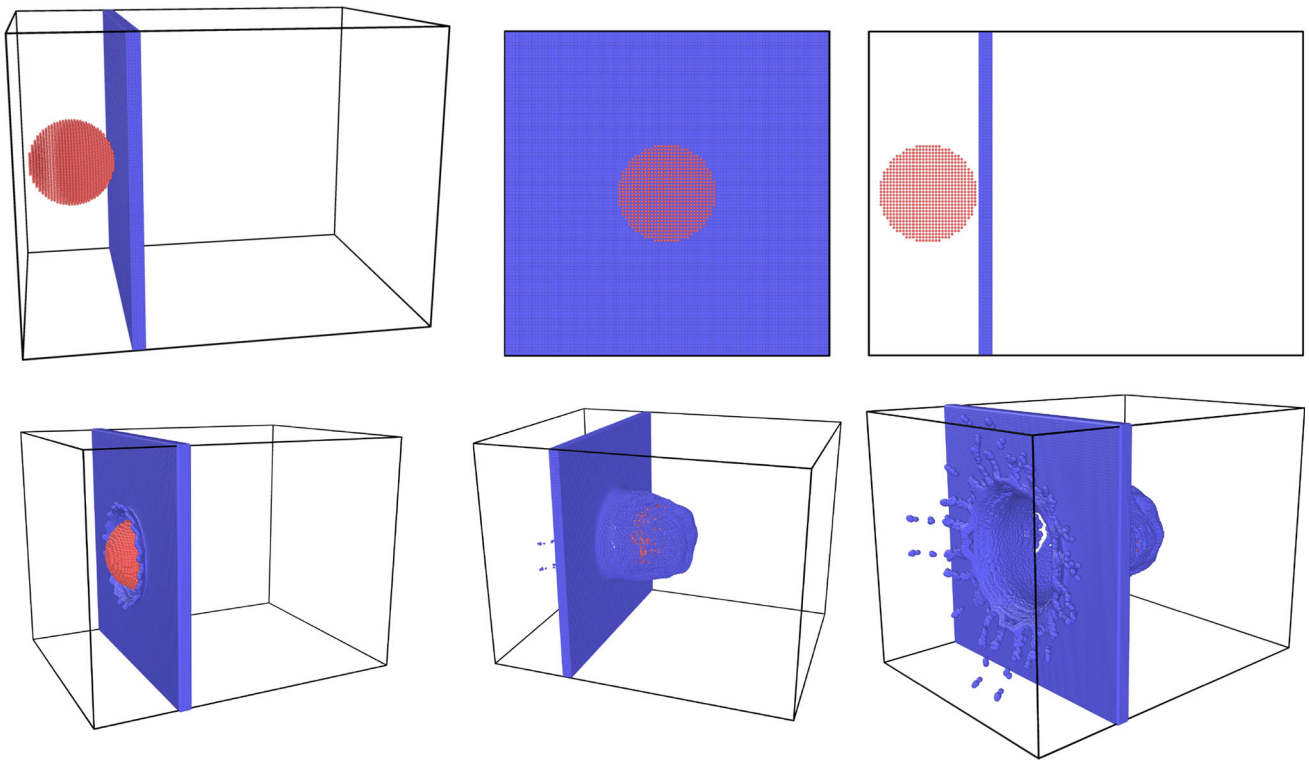


Fig. 21 Steel plate penetration problem: the plate is a block of $50 \times 50 \times 2$. The sphere radius is 7.5 mm

simulate can be distributed to is only limited by the hardware available. However, the current parallel performance of `Karamelo` is modest, but could definitely be improved. As the authors are not computer scientists, optimizing parallel execution is out of the scope of their research. However, any contribution on this aspect would be more than welcomed.

`Karamelo` is a new MPM code and is under constant development. For example, to illustrate the capability of `Karamelo` in the simulation of hyper-velocity impact problems, Fig. 21 shows the simulation of the impact of a spherical steel lead projectile with a velocity of 6.58 km/s on a thin steel plate.

Acknowledgements The first author gratefully acknowledges the financial support of the Australian Research Council (ARC) Training Centre in Alloy Innovation for Mining Efficiency (IC160100036). The second author (V.P. Nguyen) thanks the funding support from the Australian Research Council via DECRA Project DE160100577.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

A Input file for the two-disk collision problem

The input file used to simulate the two-disk collision problem is given in Listing 2.

Listing 2: Input file for the two disk collision problem.

```
#######################################################
#                 UNITS: MPa, mm, s                   #
#######################################################
E   = 1e+3                      # Young's modulus
nu  = 0.3                       # Poisson's ratio
rho = 1000                      # density

L   = 1                         # a dimension
hL  = 0.5*L

FLIP=1.0
#------------ SET METHOD ---------------------#
method(ulmpm, FLIP, linear, FLIP)
#------------ SET DIMENSION ------------------#
N         = 20      # 20 cells per direction
cellsize = L/N     # cell size
dimension(2,-hL, hL, -hL, hL, cellsize)  # 2D problem, which the computational domain is LxL
#------------ SET REGIONS--------------------#
R = 0.2
region(rBall1, cylinder, -hL+R, -hL+R, R)
region(rBall2, cylinder,  hL-R,  hL-R, R)
#------------ SET MATERIALS------------------#
material(mat1, linear, rho, E, nu)
#------------ SET SOLID ----------------------#
ppcld = 2
solid(sBall1, region, rBall1, ppcld, mat1, cellsize,0) # the last param sets the initial temperature
solid(sBall2, region, rBall2, ppcld, mat1, cellsize,0)
#------------ IMPOSE INITIAL CONDITIONS ------#
group(gBall1, particles, region, rBall1, solid, sBall1)  # define particle group1
group(gBall2, particles, region, rBall2, solid, sBall2)  # define particle group2
v = 0.1
fix(v0Ball1, initial_velocity_particles, gBall1,  v,  v, NULL)
fix(v0Ball2, initial_velocity_particles, gBall2, -v, -v, NULL)
#------------OUTPUT-------------------------------------#
N_log = 50
dumping_interval = N_log*1
dump(dump1, all, particle, dumping_interval, dump_p.*.LAMMPS, x, y, z)
dump(dump2, all, grid,     dumping_interval, dump_g.*.LAMMPS, x, y, z)
dump(dump3, all, pyplot, 1, dump.*.pdf, 500, 500)
#------------OUTPUTS------------------------#
fix(Ek, kinetic_energy, all)
fix(Es, strain_energy, all)
Etot = Ek_s + Es_s
#------------ RUN ----------------------------#
set_dt(0.001)   # constant time increments of 0.001

log_modify(custom, step, dt, time, Ek_s, Es_s)
plot(Ek, N_log, time, Ek_s)
plot(Es, N_log, time, Es_s)
plot(Etot, N_log, time, Etot)
save_plot(plot.pdf)
log(N_log)
run_time(3.5)  # run for a period of 3.5 seconds
```

References

1. Anderson CE Jr (1987) An overview of the theory of hydrocodes. Int J Impact Eng 5(1–4):33–59
2. Assadi H, Gärtner F, Stoltenhoff T, Kreye H (2003) Bonding mechanism in cold gas spraying. Acta Mater 51(15):4379–4394
3. Banerjee A, Dhar S, Acharyya S, Datta D, Nayak N (2015) Determination of Johnson Cook material and failure model constants and numerical modelling of Charpy impact test of armour steel. Mater Sci Eng A 640:200–209. https://doi.org/10.1016/j.msea.2015.05.073
4. Bardenhagen S, Kober E (2004) The generalized interpolation material point method. Comput Model Eng Sci 5(6):477–495
5. Bardenhagen S, Brackbill J, Sulsky D (2000) The material-point method for granular materials. Comput Methods Appl Mech Eng 187(3–4):529–541
6. Bardenhagen S, Guilkey J, Roessig K, Brackbill J, Witzel W, Foster J (2001) Improved contact algorithm for the material point method and application to stress propagation in granular material. Comput Model Eng Sci 2(4):509–522
7. Belytschko T, Krongauz Y, Organ D, Fleming M, Krysl P (1996) Meshless methods: an overview and recent developments. Comput Methods Appl Mech Eng 139:3–47
8. Belytschko T, Liu WK, Moran B (2000) Nonlinear finite elements for continua and structures. Wiley, Chichester
9. Brackbill J, Ruppel H (1986) FLIP: a method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. J Comput Phys 65(2):314–343
10. Cueto-Felgueroso L, Colominas I, Mosqueira G, Navarrina F, Casteleiro M (2004) On the Galerkin formulation of the smoothed particle hydrodynamics method. Int J Numer Methods Eng 60(9):1475–1512

11. de Vaucorbeil A, Nguyen VP (2020) A modelling contacts with a total Lagrangian material point method. Comput Methods Appl Mech Eng (under review)
12. de Vaucorbeil A, Nguyen VP, Hutchinson CR (2020) A total-Lagrangian material point method for solid mechanics problems involving large deformations. Comput Methods Appl Mech Eng 360:112783
13. de Vaucorbeil A, Nguyen VP, Sinaie S, Wu JY (2020) Material point method after 25 years: theory, implementation and applications. In: Balint D, Bordas S (eds) Advances in applied mechanics, vol 53. Elsevier, Amsterdam
14. Dong Y, Grabe J (2018) Large scale parallelisation of the material point method with multiple GPUs. Comput Geotech 101:149–158
15. Fern J, Rohe A, Soga K, Alonso E (2019) The material point method for geotechnical engineering: a practical guide. CRC Press, Boca Raton
16. Ganzenmüller GC (2014) Smooth-Mach-dynamics package for LAMMPS. Fraunhofer Ernst-Mach Institute for High-Speed Dynamics, Freiburg im Breisgau
17. Geuzaine C, Remacle JF (2009) GMSH: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. Int J Numer Methods Eng 79(11):1309–1331
18. Gnanasekaran B, Liu G-R, Fu Y, Wang G, Niu W, Lin T (2019) A smoothed particle hydrodynamics (SPH) procedure for simulating cold spray process—a study using particles. Surf Coat Technol 377:124812
19. Harlow F (1964) The particle-in-cell computing method for fluid dynamics. Methods Comput Phys 3:319–343
20. Huang P, Zhang X, Ma S, Wang H (2008) Shared memory OpenMP parallelization of explicit MPM and its application to hypervelocity impact. Comput Model Eng Sci 38(2):119–147
21. Jiang C, Schroeder C, Teran J, Stomakhin A, Selle A (2016) The material point method for simulating continuum materials. In ACM SIGGRAPH 2016 courses. ACM, p 24
22. Johnson GR, Cook WH (1985) Fracture characteristics of three metals subjected to various strains, strain rates, temperatures and pressures. Eng Fract Mech 21(1):31–48
23. Johnso SG (2012) PyPlot module for Julia. https://github.com/stevengj/PyPlot.jl
24. Lemaitre J (1985) A continuous damage mechanics model for ductile fracture. J Eng Mater Technol 107(1):83–89. https://doi.org/10.1115/1.3225775
25. Lemiale V, Nairn J, Hurmane A (2010) Material point method simulation of equal channel angular pressing involving large plastic strain and contact through sharp corners. Comput Model Eng Sci 70(1):41–66
26. Leroch S, Eder SJ, Ganzenmüller G, Murillo L, Ripoll MR (2018) Development and validation of a meshless 3D material point method for simulating the micro-milling process. J Mater Process Technol 262:449–458
27. Li W, Yang K, Yin S, Guo X (2016) Numerical analysis of cold spray particles impacting behavior by the Eulerian method: a review. J Therm Spray Technol 25(8):1441–1460
28. Li X, Sulsky D (2000) A parallel material-point method with application to solid mechanics. In: Ingber CBM, Power H (eds) Cmputational science-ICCS 2002, volume 2331 of applications of high-performance computing in engineering VI. WIT Press, Southampton
29. Lobovskỳ L, Botia-Vera E, Castellana F, Mas-Soler J, Souto-Iglesias A (2014) Experimental investigation of dynamic pressure loads during dam break. J Fluids Struct 48:407–434
30. Ma ZT, Zhang X, Huang P (2010) An object-oriented MPM framework for simulation of large deformation and contact of numerous grains. Comput Model Eng Sci 55(1):61–87
31. Mason LS (2015) Modelling cold spray splat morphologies using smoothed particle hydrodynamics. PhD thesis, Heriot-Watt University
32. Mast C, Mackenzie-Helnwein P, Arduino P, Miller G, Shin W (2012) Mitigating kinematic locking in the material point method. J Comput Phys 231(16):5351–5373
33. Monaghan JJ (1994) Simulating free surface flows with SPH. J Comput Phys 110(2):399–406
34. Nairn JA (2003) Material point method calculations with explicit cracks. Comput Model Eng Sci 4(6):649–663
35. Nguyen VP, Nguyen CT, Rabczuk T, Natarajan S (2017) On a family of convected particle domain interpolations in the material point method. Finite Elem Anal Des 126:50–64
36. Nguyen VP, de Vaucorbeil A, Nguyen-Thanh C, Mandal TK (2020) A generalized particle in cell method for explicit solid dynamics. Comput Methods Appl Mech Eng 371:113308
37. Parker S (2002) A component-based architecture for parallel multi-physics PDE simulation. In: Sloot P, Hoekstra A, Tan C, Dongarra J (eds) Computational science—ICCS 2002, volume 2331 of lecture notes in computer science. Springer, Berlin, pp 719–734
38. Parker S, Guilkey J, Harman T (2006) A component-based parallel infrastructure for the simulation of fluid–structure interaction. Eng Comput 22(3–4):277–292
39. Plimpton S (1995) Fast parallel algorithms for short-range molecular dynamics. J Comput Phys 117(1):1–19
40. Rabczuk T, Belytschko T (2004) Cracking particles: a simplified meshfree method for arbitrary evolving cracks. Int J Numer Methods Eng 61(13):2316–2343
41. Ruggirello KP, Schumacher SC (2014) A comparison of parallelization strategies for the material point method. In: 11th world congress on computational mechanics, pp 20–25
42. Sadeghirad A, Brannon RM, Burghardt J (2011) A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations. Int J Numer Methods Eng 86(12):1435–1456
43. Sadeghirad A, Brannon R, Guilkey J (2013) Second-order convected particle domain interpolation (CPDI2) with enrichment for weak discontinuities at material interfaces. Int J Numer Methods Eng 95(11):928–952
44. Silling SA (2000) Reformulation of elasticity theory for discontinuities and long-range forces. J Mech Phys Solids 48(1):175–209
45. Sinaie S, Nguyen VP, Nguyen CT, Bordas S (2017) Programming the material point method in Julia. Adv Eng Softw 105:17–29
46. Sinaie S, Ngo TD, Nguyen VP, Rabczuk T (2018) Validation of the material point method for the simulation of thin-walled tubes under lateral compression. Thin-Walled Struct 130:32–46
47. Sinaie S, Ngo TD, Kashani A, Whittaker AS (2019) Simulation of cellular structures under large deformations using the material point method. Int J Impact Eng 134:103385
48. Stomakhin A, Schroeder C, Chai L, Teran J, Selle A (2013) A material point method for snow simulation. ACM Trans Graph 32(4):1
49. Stukowski A (2009) Visualization and analysis of atomistic simulation data with ovito-the open visualization tool. Model Simul Mater Sci Eng 18(1):015012
50. Sulsky D, Gong M (2016) Improving the material-point method. In: Pandolfi A, Weinberg K (eds) Innovative numerical approaches for multi-field and multi-scale problems. Springer, Berlin, pp 217–240
51. Sulsky D, Kaul A (2004) Implicit dynamics in the material-point method. Comput Methods Appl Mech Eng 193(12–14):1137–1170
52. Sulsky D, Schreyer H (1996) Axisymmetric form of the material point method with applications to upsetting and Taylor impact problems. Comput Methods Appl Mech Eng 139:409–429
53. Sulsky D, Schreyer HL (1996) Axisymmetric form of the material point method with applications to upsetting and Taylor impact

problems. Comput Methods Appl Mech Eng 139(1–4):409–429. https://doi.org/10.1016/s0045-7825(96)01091-2
54. Sulsky D, Chen Z, Schreyer H (1994) A particle method for history-dependent materials. Comput Methods Appl Mech Eng 5:179–196
55. Sulsky D, Zhou S, Schreyer HL (1995) Application of a particle-in-cell method to solid mechanics. Comput Phys Commun 87(1–2):236–252
56. Sun Z, Li H, Gan Y, Liu H, Huang Z, He L (2018) Material point method and smoothed particle hydrodynamics simulations of fluid flow problems: a comparative study. Progr Comput Fluid Dyn Int J (PCFD) 18(1):1–18
57. Wilkins ML (1999) Computer simulation of dynamic phenomena. Springer, Berlin
58. Wobbes E, Möller M, Galavi V, Vuik C (2019) Conservative Taylor least squares reconstruction with application to material point methods. Int J Numer Methods Eng 117(3):271–290
59. Yin S, Wang X-F, Xu B-P, Li W-Y (2010) Examination on the calculation method for modeling the multi-particle impact process in cold spraying. J Therm Spray Technol 19(5):1032–1041