`mpm_3d` Documentation

Version 3.0

Aaron S. Baumgarten

November 2018

Contents

Chapter 1

Overview

In its default configuration, `mpm_3d` is a program which solves time-dependent mechanics problems using the material point method. Additionally, `mpm_3d` is built on a highly customizeable code framework and allows users to drastically change the default behavior of the program through user-defined classes.

This document will cover the basics of installing the program and running simulations as well as introduce developers to the underlying code framework, including how to customize it.

For questions or comments, please email asbaum@mit.edu.

Chapter 2

Installation

2.1 Downloading the Code

mpm_3d is maintained in a git repository here: http://jabroni.mit.edu/gitlab/asbaumgarten/mpm_3d.git. After setting up a git account at http://jabroni.mit.edu and following the instructions to add your SSH key (http://jabroni.mit.edu/gitlab/help/gitlab-basics/README.md), you can clone the git repo on the command line:

```
$ git clone git@jabroni.mit.edu:asbaumgarten/mpm_3d.git
```

2.2 Code Dependencies

mpm_3d is built with CMake 3.2.2 using gcc 5.2.1, though earlier versions may be supported. mpm_3d also requires the Eigen linear algebra library for C++. CMake and Eigen can be installed on Linux using the following commands:

```
$ sudo apt-get install cmake
$ sude apt-get install libeigen3-dev
```

If Eigen is installed somewhere other than /usr/include/eigen3, you will need to edit the CMakeLists.txt file in the main project folder. In particular, you will need to add:

```
include_directories(PATH-TO-EIGEN3)
```

2.3 Building the Code

`mpm_3d` can be built from the command line using CMake. After cloning the git repository, navigate to the main project directory and make a build directory:

```
$ cd mpm_v3
$ mkdir build
```

To build the code, simply call `cmake` from the build directory, then `make`:

```
$ cd build
$ cmake ..
$ make
```

If all goes according to plan, `mpm_3d` should now be installed!

Chapter 3

Running Simulations

3.1 Problem Definition

In mechanics, we are often interested in the time-dependent behavior of systems that obey the following PDEs:

$$\begin{aligned}
\dot{\boldsymbol{x}} &= \boldsymbol{v} \\
\dot{\boldsymbol{\varepsilon}} &= \tfrac{1}{2}(\operatorname{grad}\boldsymbol{v} + \operatorname{grad}\boldsymbol{v}^\top) \\
\boldsymbol{w} &= \tfrac{1}{2}(\operatorname{grad}\boldsymbol{v} - \operatorname{grad}\boldsymbol{v}^\top) \\
\dot{\boldsymbol{\sigma}} &= \hat{\dot{\boldsymbol{\sigma}}}(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}, \boldsymbol{w}, \boldsymbol{\sigma}, \xi) \\
\dot{\xi} &= \hat{\dot{\xi}}(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}, \boldsymbol{w}, \boldsymbol{\sigma}, \xi) \\
\rho\dot{\boldsymbol{v}} &= \operatorname{div}(\boldsymbol{\sigma}) + \boldsymbol{b} \\
\dot{\rho} &= \rho\operatorname{div}(\boldsymbol{v})
\end{aligned} \tag{3.1}$$

where $\boldsymbol{x}$ represents the material motion function $\boldsymbol{\chi}(\boldsymbol{X}, t)$ for an initial material point $\boldsymbol{X} = \boldsymbol{\chi}(\boldsymbol{X}, 0)$, $\boldsymbol{v}$ represents the velocity field $\boldsymbol{v}(\boldsymbol{x}, t)$, $\boldsymbol{\varepsilon}$ represents the strain-rate field $\boldsymbol{\varepsilon}(\boldsymbol{x}, t)$, $\boldsymbol{w}$ represents the spin field $\boldsymbol{w}(\boldsymbol{x}, t)$, ρ represents the density field $\rho(\boldsymbol{x}, t)$, $\boldsymbol{\sigma}$ represents the Cauchy stress field $\boldsymbol{\sigma}(\boldsymbol{x}, t)$, $\boldsymbol{b}$ represents the body force field $\boldsymbol{b}(\boldsymbol{x}, t)$, and ξ represents the internal state field $\xi(\boldsymbol{x}, t)$. In (3.1), the derivative operator is taken to be the material derivative:

$$\dot{\psi} = \frac{\partial\psi}{\partial t} + \boldsymbol{v} \cdot \operatorname{grad}\psi$$

for some scalar ψ.

In the material point method (see [3]) these fields are discretized using two function spaces, a point space, $\{U_p(\boldsymbol{x}, t)\ \forall p \in [1, N]\}$, and a node space, $\{\phi_i(\boldsymbol{x}, t)\ \forall i \in$

$[1, n]\}$, where N is the number of material points and n is the number of nodes. The motion field is defined implicitly by the material points as follows,

$$\boldsymbol{x_p} = \boldsymbol{\chi}(\boldsymbol{X_p}, t)$$

where $\boldsymbol{X_p}$ is the initial material point position. The kinematic fields $\boldsymbol{v}(\boldsymbol{x}, t)$, $\dot{\boldsymbol{v}}(\boldsymbol{x}, t)$, $\dot{\boldsymbol{\varepsilon}}(\boldsymbol{x}, t)$, and $\boldsymbol{w}(\boldsymbol{x}, t)$ are defined on the node function space as follows,

$$\boldsymbol{v}(\boldsymbol{x}, t) = \sum_{i=1}^{n} \boldsymbol{v_i} \phi_i(\boldsymbol{x}, t)$$

$$\dot{\boldsymbol{v}}(\boldsymbol{x}, t) = \sum_{i=1}^{n} \dot{\boldsymbol{v}}_{\boldsymbol{i}} \phi_i(\boldsymbol{x}, t)$$

$$\dot{\boldsymbol{\varepsilon}}(\boldsymbol{x}, t) = \sum_{i=1}^{n} \mathrm{Sym}(\boldsymbol{v_i} \otimes \mathrm{grad}\, \phi_i(\boldsymbol{x}, t))$$

$$\boldsymbol{w}(\boldsymbol{x}, t) = \sum_{i=1}^{n} \mathrm{Skw}(\boldsymbol{v_i} \otimes \mathrm{grad}\, \phi_i(\boldsymbol{x}, t))$$

and the remaining fields are defined on the point function space as follows,

$$\boldsymbol{\sigma}(\boldsymbol{x}, t) = \sum_{p=1}^{N} \boldsymbol{\sigma_p} U_p(\boldsymbol{x}, t)$$

$$\boldsymbol{b}(\boldsymbol{x}, t) = \sum_{p=1}^{N} \boldsymbol{b_p} U_p(\boldsymbol{x}, t)$$

$$\rho(\boldsymbol{x}, t) = \sum_{p=1}^{N} \rho_p U_p(\boldsymbol{x}, t)$$

$$\xi(\boldsymbol{x}, t) = \sum_{p=1}^{N} \xi_p U_p(\boldsymbol{x}, t)$$

For more details about the default behavior of `mpm_3d`, see [1].

3.2 Points File

To integrate the PDEs in (3.1) through time, `mpm_3d` needs to define a set of initial conditions. In its default configuration, `mpm_3d` will do this by reading a space-delimited points file with the following form,

N
$m_1 \;\; v_1 \;\; \boldsymbol{X_1} \;\; \boldsymbol{v_1} \;\; \texttt{1}$
$m_2 \;\; v_2 \;\; \boldsymbol{X_2} \;\; \boldsymbol{v_2} \;\; \texttt{1}$
$\dots$
$m_N \;\; v_N \;\; \boldsymbol{X_N} \;\; \boldsymbol{v_N} \;\; \texttt{1}$

where the sets of coefficients $\{m_p\}$, $\{v_p\}$, $\{\boldsymbol{X_p}\}$, and $\{\boldsymbol{v_p}\}$ are defined as follows,

$$\begin{aligned}
m_p &= \int_\Omega \rho_p U_p(\boldsymbol{x}, 0) dv \\
v_p &= \int_\Omega U_p(\boldsymbol{x}, 0) dv \\
\boldsymbol{X_p} &= \frac{1}{m_p} \int_\Omega \rho_p \boldsymbol{x} U_p(\boldsymbol{x}, 0) dv \\
\boldsymbol{v_p} &= \frac{1}{m_p} \int_\Omega \rho_p \boldsymbol{v}(\boldsymbol{x}, 0) U_p(\boldsymbol{x}, 0) dv
\end{aligned}$$

`mpm_3d` can be run in 1D (uniaxial-strain), 2D (plane-strain), or 3D. Defining the dimension of a simulation will be covered in more detail in section 3.4; however, the number of coefficients that define the set of vectors $\{\boldsymbol{X_p}\}$ or $\{\boldsymbol{v_p}\}$ in the points file will change according to the choice of simulation dimension.

To generate a points file, there are a number of poorly commented python scripts in the `mpm_pygen` folder. In particular, any python script ending in '_v2' will provide a good starting point for generating your own files.

3.3 Output Directory

While the `mpm_3d` is running, it will output simulation snapshots. By default, these simulation snapshots are saved as individual '`.vtk`' files. It is often useful to direct these files to an new output directory.

```
$ mkdir output
```

3.4 Configuration File

The last step in setting up a simulation for `mpm_3d` is the creation of a configuration file. The configuration file contains information about the various objects and parameters that `mpm_3d` will need to create to run the simulation. A comprehensive list of currently implemented objects/classes can be found in section 5.

Every configuration file begins with the definition of the '`job`' object as follows,

```
job
{
    t = 0
    dt = 1e-3
    TYPE = 2
}
```

The '`t`' parameter denotes the start time for the simulation and the '`dt`' parameter denotes the length of the discrete time integration steps. (Note that all parameters are assumed to be defined in terms of kilograms, meters, and seconds.)

The '`TYPE`' parameter is an integer between 1 and 5 that defines the type of simulation. `1` is used for 1D (uniaxial-strain) simulations, `2` is used for 2D (plane-strain) simulations, `3` is used for 3D simulations, `4` is used for axisymmetric simulations, and `5` is used for 2D simulations with out-of-plane motion.

This is followed by the definition of a '`serializer`' object,

```
serializer
{
    class = "DefaultVTK"
    #frames-per-second
    properties = {60}
    int-properties = {}
    #{output-dir, save-dir, sim-name}
    str-properties = {"output", "save", "example"}
}
```

The '`serializer`' is the first object that can be user-defined and governs the input/output behavior of `mpm_3d`. Here the '`DefaultVTK`' serializer class is implemented. A complete list of serializer options and definitions can be found in section 5.1. The '`properties`' keyword takes in a comma-separated list of floating-point numbers, which for '`DefaultVTK`' is a single value for the rate at which simulation snapshots are saved. The '`int-properties`' keyword takes in a comma-separated list of integers (which is left empty for '`DefaultVTK`'), and the '`str-properties`' keyword takes in a comma-separated list of strings, which for '`DefaultVTK`' is the output directory for the simulation snapshots, a non-implemented save directory, and a simulation name. (Note that comments can be written in the configuration file using '`#`' symbol.)

The '`driver`' object is defined with a similar structure to the '`serializer`' object as follows,

```
driver
{
    class = "DefaultDriver"
    properties = {1}
    int-properties = {}
    str-properties = {}
}
```

The '`driver`' object acts like a while-loop for time-integration; it will call the '`solver`' to integrate the system of equations in (3.1) during each time-step until the simulation has completed. Here the '`DefaultDriver`' driver class is implemented. A complete list of driver options can be found in section 5.2. '`DefaultDriver`' requires only one '`properties`' item, the stop time for the simulation.

Now the '`solver`' object is defined,

```
solver
{
    class = "ExplicitUSL"
    properties = {}
    int-properties = {1, 1}
    str-properties = {}
}
```

As stated above, the '`solver`' object integrates the system of equations in (3.1) through one time-step. The class '`ExplicitUSL`' implements a forward euler integration scheme. A complete list of solver options can be found in section 5.3.

The next item in the configuration file is the '`grid`' object,

```
grid
{
    class = "CartesianLinear"
    #Lx, Ly
    properties = {1.0, 1.0}
    #Nx, Ny
```

```
        int-properties = {20, 20}
        str-properties = {}
    }
```

The '`grid`' object defines the simulation domain and nodal function space used to discretize the kinematic fields in the simulation. The '`CartesianLinear`' grid class defines linear (for `TYPE = 1`), bi-linear (for `TYPE = 2`, `4`, or `5`), or tri-linear (for `TYPE = 3`) functions on a cartesian grid. The side lengths of the grid are given by the '`properties`' keyword and the number of linear elements on each edge are given by the '`int-properties`' keyword. A complete list of grid options can be found in section 5.4.

The next objects to be defined in the configuration file are the '`body`' objects,

```
    body
    {
        class = "DefaultBody"
        name = "block_0"
        properties = {}
        int-properties = {}
        str-properties = {}
        point-file = "block_0.points"

        point-class = "DefaultPoints"
        point-props = {}
        point-int-props = {}
        point-str-props = {}

        node-class = "DefaultNodes"
        node-props = {}
        node-int-props = {}
        node-str-props = {}

        material-class = "IsotropicLinearElasticity"
        #{E, ν}
        material-props = {1e5,0.3}
        material-int-props = {}
        material-str-props = {}

        boundary-class = "CartesianBox"
        boundary-props = {}
        boundary-int-props = {}
        boundary-str-props = {}
    }

    body
    {
        class = "DefaultBody"
        name = "block_1"
        point-file = "block_1.points"

        point-class = "DefaultPoints"
        node-class = "DefaultNodes"

        material-class = "IsotropicLinearElasticity"
        material-props = {1e8,0.4}

        boundary-class = "CartesianBox"
    }
```

The '`body`' objects represent continuum bodies within the simulated domain. (Note that multiple bodies can be simulated at the same time). Here the '`DefaultBody`' class is implemented. Each body is assigned a *unique* '`name`' and points file (using the '`point-file`' keyword). As stated above, the points file contains the initial conditions for the continuum body. In addition, each '`body`' definition contains a '`node-class`', '`point-class`', '`material-class`', and '`boundary-class`'. The '`point-class`' (here set to '`DefaultPoints`') and '`node-class`' (here set to '`DefaultNodes`') are containers for the coefficients defining the point and node fields respectively. The '`material-class`', here set to '`IsotropicLinearElasticity`', defines the update rule for the stress in the body as given by the expression,

$$\dot{\boldsymbol{\sigma}} = \hat{\dot{\boldsymbol{\sigma}}}(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}, \boldsymbol{w}, \boldsymbol{\sigma}, \xi) \tag{3.2}$$

The '`boundary-class`', here set to '`CartesianBox`', defines the boundary conditions for the body.

The final objects defined in the configuration file are the '`contact`' objects,

```
    contact
    {
        name = "collision"
        class = "ContactHuang"
        properties = {0.4}
        int-properties = {}
        str-properties = {"block_0","block_1"}
    }
```

The '`contact`' object, here set to '`ContactHuang`' defines the kinematic constraints on bodies due to their interactions with other bodies in the simulation. The '`ContactHuang`' contact class implements the constraints given in [2] and takes in a single floating point property and two string properties (the coefficient of friction between the two bodies and the '`name`' of the bodies). If more than one contact rule is given (required if there are more than two bodies in a simulation), then the constraints will be applied in the order they appear in the configuration file. If everything has been written correctly (including the points files), then the simulation is ready to run!

3.5 Calling Program

3.6 Visualizing Results

Chapter 4

Code Framework and Customization

4.1 General Problem Definition

4.2 `main` Program

4.3 `Job` Class

4.4 `Registry` and `Configurator` Classes

4.5 `MPMObjects` Class

4.6 `Serializer` Class

4.7 `Driver` Class

a

4.8 `Solver` Class

4.9 `Grid` Class

4.10 `Contact` Class

4.11 `Body` Class

4.12 `Points` Class

4.13 `Nodes` Class

4.14 `Material` Class

4.15 `Boundary` Class

Chapter 5

User-Defined Classes

5.1 Serializers

5.2 Drivers

5.3 Solvers

5.4 Grids

5.5 Contacts

5.6 Bodies

5.7 Points

5.8 Nodes

5.9 Materials

5.10 Boundaries

Bibliography

[1] Sachith Dunatunga and Ken Kamrin. Continuum modeling of projectile impact and penetration in dry granular media. *Journal of the Mechanics and Physics of Solids*, 100:45–60, 2017.

[2] Peng Huang, X Zhang, S Ma, and X Huang. Contact algorithms for the material point method in impact and penetration simulation. *International journal for numerical methods in engineering*, 85(4):498–517, 2011.

[3] Deborah Sulsky, Zhen Chen, and Howard L Schreyer. A particle method for history-dependent materials. *Computer methods in applied mechanics and engineering*, 118(1-2):179–196, 1994.