Proceedings 10th NUMGE 2023
10th European Conference on Numerical Methods in Geotechnical Engineering
Zdravković L, Konte S, Taborda DMG, Tsiampousi A (eds)
ISSN
© Authors: All rights reserved, 2023
doi: 10numge-2023-Y-XXXX

An open-source Julia code for geotechnical MPM

N.D. Gavin, R.E. Bird, W.M. Coombs, C.E. Augarde
Department of Engineering, Durham University, Durham, UK

ABSTRACT: There is considerable interest in the Material Point Method (MPM) in the computational geotechnics community since it can model problems involving large deformations, e.g. landslides, collapses etc. without being too far from the standard finite element method, which can struggle with large deformation problems. The open-source code AMPLE developed at Durham University in recent years is a compact set of MATLAB functions that "address the severe learning curve for researchers wishing to understand, and start using, the MPM". It is well known that MATLAB can be very slow hence limiting its utility for major studies of large problems, so here we introduce an MPM code with the same aims as AMPLE but written in the relatively new language Julia, specifically for fast runtimes. We highlight areas where MATLAB code constructs are inefficient if just transferred to Julia and show that to unlock large speed gains with Julia, one needs to code in a different way and we demonstrate this on a geotechnical problem. While this paper is concerned with the MPM, the advice regarding coding using Julia is transferable to other computational geotechnics methods and tools.

Keywords: Material Point Method; Julia; AMPLE

1 INTRODUCTION

The Material Point Method (MPM), originally developed by Sulsky and co-workers in the 1990s (Sulsky et al., 1994), models a problem domain as a collection of *material points* at which all information pertaining to that location in the domain is stored. It is not however a meshless method, as sometimes supposed since the calculations to determine deformation response to a load event (in the case of quasi-statics for instance) are carried out on a background Finite Element (FE) mesh or grid. Information is mapped from the material points to the grid nodes, a standard FE solve carried out and the results mapped back to the material points. The advantage of this arrangement is that a new undistorted background mesh can be used for the subsequent load step/increment regardless of the magnitude of deformations of material points and therefore the key issue of mesh distortion met with in standard FE methods is totally avoided.

This key feature, and hence its utility for problems involving large deformation, has promoted considerable interest in the computational geotechnics community as evidenced by an increasing number of publications and two recent conferences (Fern et al., 2019). Recent examples of its use can be found for landslides (e.g. Xu et al., 2018; Conte et al., 2019), site investigations (e.g. Ceccato et al., 2016; Francesca et al., 2020) and offshore foundations (e.g. Brinkgreve et al. 2017, Galavi et al. 2019). Its close relation to standard FE methods means it is easy to transfer material models and other numerical methods to run in MPM from an existing FE code. Having said this, the MPM is not without its challenges, for instance much effort is being expended in the MPM research community trying to address solutions to problems such as poor system conditioning due to low numbers of material points in a background element (Coombs, 2022).

The open-source code AMPLE (Coombs & Augarde, 2020) is a MPM implementation for solid mechanics developed to specifically address the steep learning curve met by those wishing to experiment with the MPM. AMPLE is a MATLAB implementation of the MPM with relatively few options, but with the emphasis on code clarity and lack of ambiguity. While MATLAB is a good framework in which to develop code and prototype, it is not fast and actually using it to carry out large analyses (in terms of numbers of degrees of freedom) is difficult. In this paper we describe a new version of AMPLE written in the Julia language, which combines the clarity of the MATLAB implementation but with a much lower computational cost. We do not claim here to be the first developers of an MPM code in Julia, as there are other examples, e.g. Sinaie et al. (2017) covers similar ground and in greater detail, but for an explicit MPM code rather than implicit as in AMPLE and most geotechnical FEA. Other useful guidance on the use of Julia can be found in Xiong et al. (2020) and Xiao et al. (2022).

2 THE JULIA LANGUAGE

The Julia language was developed initially in 2009 to "address the needs of high-performance numerical and scientific computing." (Core Julia development team, 2023a). It has since gathered a large following and has a strong and supportive user community. It is also an exciting, flexible, and relatively new language. Julia has a modern, expressive syntax, automatic memory management, and built-in support for parallel computing. Julia also has a growing ecosystem of packages and libraries, making it well-suited for a wide range of applications.

2.1 Key differences

Since the syntax for MATLAB and Julia look similar (as both do to Python), it is easy to assume that behaviour expected from MATLAB will also happen for a similar-looking code fragment in Julia and that can lead to surprising annoyances in code development and debugging. Some of these differences are now described.

A key syntax difference that is immediately obvious if moving a code from MATLAB to Julia is the use of brackets. For example, for indexing of arrays, a MATLAB code fragment

```
bc(node*2-1,:)=[node*2-1 0]
```

would have the equivalent in Julia of

```
bc[node*2-1,:]=[node*2-1, 0]
```

which is almost the same apart from the use of square rather than curved brackets for the left hand side. Function calling also looks a little different: the MATLAB call to a function `form2D` with four arguments would look like this

```
[ep,cd] = form2D(nx,ny,lx,ly)
```

and in Julia like this

```
ep,cd= form2D(nx,ny,lx,ly).
```

Julia functions also by default return the last value calculated so it is important to override that default and ask explicitly for what you want returning, e.g. the fragment at the end of a function

```
else
  Svp =0
  dSvp=0
end
return Svp, dSvp
end
```

without the `return` statement will just return `dSvp.`

In the original AMPLE, all data for material points is held in a structure array (a “struct”) containing 20 fields. In Julia the equivalent is a *mutable struct*, the “mutable” indicating that field values can be changed during execution.

One pitfall to be aware of is how Julia handles the assignment of one variable to another. If variable `A` is an array and is assigned to variable `B`, MATLAB will create a new memory address for the new variable. However, Julia simply creates a “shallow copy” meaning that `B` becomes a reference to the memory address in which `A` is located. This means that errors can arise when working with both variables. For example, if `A` is an array of integers and we assign `A` to variable `B`, `B` will produce the same output as `A`,

```
A = [1,2,3,4]
B = A
>> B = [1,2,3,4]
```

Now, if an element of `B` is altered, the corresponding element in the variable `A` will also be altered

```
B[1] = 5
>> B = [5,2,3,4]
>> A = [5,2,3,4]
```

This is inevitably going to cause errors within the code if `A` and `B` are to be used in separate calculations. In order to overcome this, the `copy()` (Core Julia development team 2023f) function should be used, this will create a new memory address for `B` in which the contents of `A` will be stored when assigning to the new variable.

```
A = [1,2,3,4]
B = copy(A)
B[1] = 5
>> A = [1,2,3,4]
>> B = [5,2,3,4]
```

By copying the variable `A` into `B`, it is now possible to alter the elements of `A` (or `B`) without affecting the contents of `B` (or `A`).

3 JULIA NUANCES

Moving on from the syntactical differences, some of which have been covered above, it is useful to be aware that many optimisation practices that are used in more traditional languages have already been considered in the development of Julia and are simple to apply once one is aware. This includes; making use of contiguous memory, vectorisation, cache optimisation and memory allocation and reuse. Additionally, due to the large open-source community that uses Julia, and its inherent speed, it is simple to include a range of open source optimised numerical packages, and also produce your own, with relative ease. Some of the optimisation procedures inherent in Julia are now discussed.

3.1 *Predefining variable types*

Variables in Julia belong to “types”, and this can be exploited to make parametric and hierarchical code (Core Julia development team, 2023d), a particularly powerful tool for numerical modelling. By default, variable and function types are ambiguous making the code very flexible and powerful, however if the type is ambiguous then high performance compiled code is unlikely to exist, as additional decisions and operations will be performed at runtime, thus slowing the code. It is therefore routine to define types wherever possible. An example of a variable defined with a type is,

```
A:: Matrix{Float64},
```

where `A` is a matrix of 64-bit floating point (IEEE 754 standard) values. `Matrix` and `Vector` in Julia are based on the mutable data type `Array{T,N}` where `T` is the type (e,g, `Float64`) and `N` is the number of dimensions; `Matrix` has `N=2` and `Vector`, `N=1`. Different results are obtained for slight changes in assignments
e.g. `a=[1 2 3]` will give a 1×3 `Matrix`, while `a=[1,2,3]` or `a=[1;2;3]` will give a 3-element `Vector` (i.e. a 3×1 `Array`).

3.2 *Predefining variable memory*

Predefining variable memory is another way to optimise compiler performance and is achieved by allocating a size to the variable’s definition

```
A  = zeros(6,6)::Matrix{Float64}.
```

Predefining memory has two purposes. Firstly, if correct, it prevents reallocation of memory during a calculation since the variable’s size, and attributed memory, has already been defined. Secondly, it allows for memory reuse during repeated rewrites to a variable, therefore preventing unnecessary memory allocation which significantly slows code (Core Julia development team, 2023d). As an example, a function that does as much as it can to supply useful information for the compiler, that squares the `Float64` `a`, to produce the output `Float64` `b`, is

```
a = 2::Float64
b = 0::Float64
function sqr_flt!(a::Float64,b::Float64)
  b=a^2
end
```

The exclamation mark ! in the function definition prevents memory allocation when the function is called, it allows variables in the function to be edited directly and prevents new memory being allocated each time the function is called. This is particularly critical if the function is called multiple times.

3.3 *Macros*

Macros in Julia provide a mechanism to include generated code in the final body of a program; they change existing source code or generate entirely new code (Core Julia development team, 2023b). Julia optimisation packages have been written so they can be deployed with a macro, in most cases this means a significantly optimised version of the code can be achieved with an edit to a single line. One of the most useful packages is `LoopVectorization` (Elrod, 2023), which is used with macro `@turbo` and demonstrated below.

One of the most common calculations in implicit MPM and finite element codes is the element stiffness contribution at an integration point *i* (FEs) or a material point *i* (MPM).

$$k_i^e \approx B^\top DB \tag{1}$$

where k_i^e is the local stiffness matrix contribution from point i to the element e, B is the shape function derivative matrix and D is the material stiffness matrix. The approximation in Equation (1) represents the fact that there will be weighting of the matrix triple product, e.g. Gauss quadrature weights for FEs and volume/mass for the MPM. For 3D elasticity B is $9 \times 3n$, where n is the number of element nodes, and D is 9×9. Within a MPM or finite element code Equation (1) is a small matrix operation undertaken many times. In the MPM case, Equation (1) is calculated at each material point in every iteration of the non-linear solve.

A performance test of the calculation of Equation (1) is used here to demonstrate the `@turbo` macro. The timing of the code was performed with the benchmarking toolbox `BenchmarkTools` (Churavy, 2023) on a single core 2.10 GHz machine. Two different methods to multiply matrices are considered. First, to multiply the matrix `A`, of size $I \times K$, with matrix `B`, of size $K \times J$, to form `C`, the following code segment is used:

```
function my_mul!(C::Matrix{Float64},
A::Matrix{Float64},B::Matrix{Float64},
I::Int64,J::Int64,K::Int64)
    @turbo for j in 1:I
        for k in 1:J
            for i in 1:K
                C[i,j] += A[i,k]*B[k,j]
            end
        end
    end
end
```

With the `@turbo` macro initiated on the first for loop of `my_mul!` To calculate the two matrix multiplications in Equation (1) `my_mul!` is called twice.

Secondly, this is compared to the native implementation:

```
function k_mul!(
k::Matrix{Float64},
B::Matrix{Float64},
BT::Matrix{Float64},
D::Matrix{Float64})
  k = BT*D*B
end
```

where the multiplication operator * is called from the native `LinearAlgebra` package, which in turn calls LAPACK (Anderson et al., 1999). The speed of `k_mul!` is compared to `my_mul!` for the calculation of k_i^e when $n = 10$, representing a linear tetrahedral element. The run times of the two code segments are compared, and presented, in Table 1. The table clearly shows the improved speed from a bespoke user multiplication with the @turbo macro.

Table 1. Run times of native multiplication and `@turbo`.

Function used	**Time (ns) no @turbo**	**Time (ns) @turbo**
`k_mul!`	671.3	n/a
`my_mul!`	1334.9	158.0

This example also demonstrates where Julia performs well with a nest of loops, which may feel counter intuitive.

3.4 *The dot syntax*

Normally, vectorised code needs to be structured as such during writing, and one of the most useful features of Julia is the dot syntax (Core Julia development team, 2023a) which allows for the vectorisation of code without the overhead of writing code in a vectorised form and the subsequent lower readability.

However, the syntax is much more powerful than just as an improvement to readability (Core Julia development team, 2023a). The dot allows for vectorised operations to be recognised at the syntactic level, and hence loop vectorisation is a syntactic guarantee, not a compiled optimisation that might occur. Using the condensed example from Johnson (2017), operations for the vectorized code

```
f(X) = 2*X.^2
```

will be

```
tmp1 = X.^2
tmp2 = 2*tmp1
X = f(tmp2)
```

This both requires memory allocation for `tmp1` and `tmp2`, but also means that loops over the array `X` occur separately and sequentially over `X`. This in turn will cause repeated memory transfers to and from the RAM to CPU cache for values within `X` (assuming that `X` does not fit in the CPU cache). Rewriting `f` as

```
f(X) = 2*X^2
```

and calling it with `f.(X)`, fuses the loops that would exist for tmp1 and tmp2. This means that each value in the array is called into the cache once, all operations are performed and then it is stored back into the RAM; increasing code speed. The true power of the dot syntax, which is unique to Julia (Johnson, 2017), is that this can be applied to any function type, even those created by the user.

3.5 *Preallocating and reusing variables*

It can often be the case in loops that some variables only exist within the loop and are recalculated in every instance of the loop, this means that an allocation will occur each time, for example

```
C = zeros(Float64,3,3,100)
for i in 1:100
  A = rand(Float64,3,3)
  B = rand(Float64,3,3)
  C[:,:,i] = A * B
end
```

Here, variables A and B are allocated 100 times, however, if the sizes of A, B and C are known before entering the loop, these matrices can be pre-allocated and the contents of the matrices can be altered within each loop rather than reallocating the variables each time. This is done by using the broadcast operator `.=` (Core Julia development 2023g), reducing the number of allocations by rewriting the values within the 3x3 matrices stored at the pre-existing memory address. Using the broadcast operator along with the `my_mul!` function presented in section 3.3, an improved code becomes

```
A = zeros(Float64,3,3)
B = zeros(Float64,3,3)
C = zeros(Float64,3,3,100)
for i in 1:100
  A .= rand(Float64,3,3)
  B .= rand(Float64,3,3)
  my_mul!(C[:,:,i],A,B)
end
```

Knowing the sizes of the variables used throughout a code can be useful, as variables can be used throughout by storing variables in a "tuple" (Core Julia development team 2023e), which are immutable collections of variables. Once created, the contents of the tuple cannot be changed, meaning that variables cannot be added to the tuple or variables within the tuple cannot be removed or altered, but the contents of the variables can be changed. For example, a tuple is created with two variables, `X` (a 3x3 matrix) and `Y` (a 3x1 vector), the values of within `X` and `Y` can be changed using the broadcast operator, but the sizes and types of the two variables cannot change. The variables held within the tuple can be used in the same way as a struct in MATLAB.

```
tpl = (X = zeros(Float64,3,3),
Y = zeros(Float64,3))
>> tpl.X = [0 0 0; 0 0 0; 0 0 0]
>> tpl.Y = [0, 0, 0]

tpl.X .= Diagonal([1, 2, 3])
>> tpl.X = [1 0 0; 0 2 0; 0 0 3]

tpl.Y .= [4, 5, 6]
>> tpl.Y = [4, 5, 6]

Z = tpl.X * tpl.Y
>> Z = [4, 10, 18]
```

This tuple can be passed into every function of a code and the variables can be used as many times as required, reducing the total number of allocations and thus improving the performance. Tuples are also useful to hold key variables that are used throughout the code without having to pass them in and out of functions. In AMPLE for example, the number of material points is constant throughout an analysis (unlike the number of active nodes) and therefore one can exploit this feature to set up tuples for variables containing material point data, zeroing all of the contents of the pre-allocated variable at the start of a load step and altering its contents in each Newton-Raphson iteration rather than creating a new variable every load step.

4 AN EXAMPLE

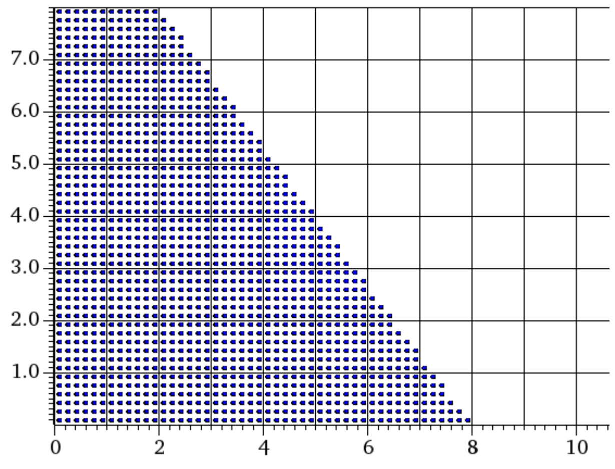
To demonstrate the Julia version of AMPLE a very simple geotechnical problem is modelled, and the computational cost in terms of runtime measured. The problem is somewhat artificial for simplicity, but includes material non-linearity and involves large deformations. An embankment of material 8 units high is modelled (using symmetry to reduce the problem domain modelled by half) where the base is assumed to be supported on a surface with zero friction. Two discretisations of bilinear quad elements are used: "small" where the element size is 1 unit and "large" where the element size is 0.5 units. The starting material point distribution is a 6×6 grid in each element and the total numbers of material points are 1440 and 5760 respectively.

Using compatible units, the material has a density of 1000 and is elastic-perfectly plastic with a von Mises failure criterion, with a deviatoric yield stress of $\rho_y = 2 \times 10^4$ where the yield surface is defined as

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

where $\rho = \sqrt{2J_2}$, $J_2 = \frac{1}{2}tr([s][s])$ &

$$[s] = [\tau] - \frac{1}{3}tr([\tau])$$



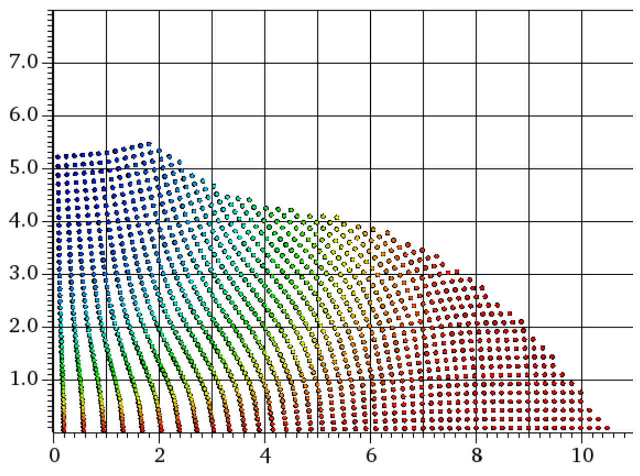


Figure 1. Example problem (small version) (a) original configuration; (b) final slumped shape: colours represent horizontal displacement.

in which $[\tau]$ is the Kirchhoff stress tensor. Elastic properties are Young's modulus, $E = 10^6$ and Poisson's ratio, $\nu = 0.3$. The embankment is loaded from increasing the gravitational acceleration over forty increments. As expected the embankment material slumps outwards as shown in Figure 1 and large deformations are evident.

Table 2 shows the runtimes (mean of five instances) for this problem for three versions of AMPLE: A. Original AMPLE in MATLAB, B. AMPLE in Julia where none of the nuances of Julia cited in Section 4 have been used and C. AMPLE in Julia where they have. The results show the huge advantages of the third implementation where large time savings (even for these relatively small problems) are achieved for intelligent Julia code over MATLAB, i.e. around 1.5-1.7 times faster. It also demonstrates that plain conversion from MATLAB to Julia is not a good idea.

Table 2. Runtimes (in seconds) for the example problems

Code	Small problem	Large problem
A	35.226	145.335
B	125.828	547.869
C	20.160	92.596

5 CONCLUSIONS

Julia is an ideal language in which developers of computational geotechnics technology can work being almost as clear as MATLAB in syntax (as compared to C++ for instance) but used correctly, can be much faster, as demonstrated here, via the use of a few simple-to-understand commands and structures.

Work is now underway in the group at Durham to expand the capabilities of Julia implementations to include multiphase materials, contact and friction and HPC. The Julia code used here will be made available via Github in the near future (https://wmcoombs.github.io/).

6 ACKNOWLEDGEMENTS

This work is supported by activity in the Computational Mechanics Research Node, Department of Engineering, Durham University, Author 1 is supported by a UK EPSRC studentship under grant EP/T518001/1 while author 2 is supported by UK EPSRC project grant EP/W000970/1.

7 REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Del, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D. 1999. *LAPACK Users' Guide*, SIAM, Philadelphia, PA.

Brinkgreve, R., Burg, M., Lim, L. J., & Andreykiv, A. 2017. On the practical use of the Material Point Method for offshore geotechnical applications. In *Proc.19th ICSMGE*, 2269-2272.

Ceccato, F., Beuth, L., Vermeer, P. A., Simonini, P., 2016. Two-phase material point method applied to the study of cone penetration. *Computers & Geotechnics*, 80, 440-452.

Churavy, V., 2023. *BenchmarkTools.jl.* GitHub repository, https://github.com/vchuravy

Conte, E., Pugliese, L., Troncone, A., 2019. Post-failure stage simulation of a landslide using the material point method. *Engineering Geology*, 253, 149-159.

Coombs, W.M. 2022. Ghost stabilisation of the Material Point Method for stable quasi-static and dynamic analysis of large deformation problems, https://arxiv.org/abs/2209.10955.

Coombs, W.M., Augarde, C.E. 2020. AMPLE: A Material Point Learning Environment, *Advances in engineering software*, 139:102748.

Core Julia development team 2023a. *Functions*. https://docs.julialang.org/en/v1/manual/functions/

Core Julia development team 2023b. *Julia Homepage*. https://julialang.org/

Core Julia development team 2023c. *Metaprogramming.* Julia 1.8 documentation. https://docs.julialang.org/en/v1/manual/metaprogramming/

Core Julia development team 2023d. *Julia Performance Tips. Julia 1.8 documentation*. https://docs.julialang.org/en/v1/manual/performance-tips/

Core Julia development team 2023e. *Types. Julia 1.8 documentation.* https://docs.julialang.org/en/v1/ manual/types/

Core Julia development team 2023f, *Essentials. Julia 1.8 documentation.* https://docs.julialang.org/en/v1/base/base/#Essentials

Core Julia development 2023g. *Broadcasting*. *Julia 1.8 documentation*. https://docs.julialang.org/en/v1/manual/arrays/#Broadcasting

Elrod, C., 2023. *LoopVectorization.jl.* GitHub repository, https://github.com/JuliaSIMD/LoopVectorization.jl

Fern, J., Rohe, A., Soga, K., Alonso, E., 2019. *The Material Point Method for Geotechnical Engineering*. CRC Press.

Francesca, C., Lars, B., Paolo, S., 2020. Analysis of Piezocone Penetration under Different Drainage Conditions with the Two-Phase Material Point Method. *Journal of Geotech & Geoenv Engineering*, 142, 04016066.

Galavi, V., Martinelli, M., Elkadi, A., Ghasemi, P., Thijssen, R. 2019. Numerical simulation of impact driven offshore monopiles using the material point method. *Proc XVII ECSMGE.*

Johnson, S.G. 2017. More Dots: Syntactic Fusion in Julia. Julia Blog. More Dots: Syntactic Loop Fusion in Julia (julialang.org)

Sinaie, S., Nguyen, V.P., Nguyen, C.T., Bordas, S. 2017.Programming the material point method in Julia. *Advances in Engineering Software*, 105, 17-29,

Sulsky, D., Chen, Z., Schreyer, H.L., 1994. A particle method for history-dependent materials. *Computer*

Methods in Applied Mechanics and Engineering, 118, 179-196.

Xiao, L., Mei, G., Xi, N., Piccialli, F. 2022. Julia Language in Computational Mechanics: A New Competitor. *Arch. Computational Methods in Engineering*, 29, 1713–1726.

Xiong, H., Yin, Z-Y, Nicot, F. 2020. Programming a micro-mechanical model of granular materials in Julia. *Advances in Engineering Software*, 145, 2020,102816.

Xu, X., Jin, F., Sun, Q., Soga, K., Zhou, G.G., 2018. Three-dimensional material point method modelling of runout behavior of the Hongshiyan landslide. *Canadian Geotechnical Journal*, 56(9): 1318-1337.