Archives of Computational Methods in Engineering (2022) 29:1713–1726
https://doi.org/10.1007/s11831-021-09636-0

SURVEY ARTICLE

Julia Language in Computational Mechanics: A New Competitor

Lei Xiao[1] · Gang Mei[1] · Ning Xi[1] · Francesco Piccialli[2]

Received: 26 March 2021 / Accepted: 1 August 2021 / Published online: 13 August 2021

Abstract
Numerical methods are the most popular tools in computational mechanics and have been used to tackle various practical engineering problems. However, the most common programming languages used for implementing numerical methods do not effectively balance the demands of productivity and efficiency. To address the most computationally intensive areas of numerical computing with the increased abstraction and productivity provided by a high-level language, the Julia language was released by the Massachusetts Institute of Technology (MIT) in 2012. The Julia language is an open-source programming language that presents simple syntax and satisfactory performance; this is particularly useful for scientific computing. In this paper, we present a comprehensive survey on the use of the Julia language in computational mechanics. First, we introduce the existing numerical computing packages developed in the Julia language and their relevant applications. Second, we analyze the capabilities of the Julia language in the development of software packages for computational mechanics. Finally, we discuss the open issues regarding the Julia language and the challenges faced when using the Julia language in computational mechanics.

Abbreviations

BEM	Boundary element method
COO	COOrdinate
CPU	Central processing unit
CSC	Compressed sparse column
CSR	Compressed sparse row
DDA	Discontinuous deformation analysis
DEM	Discrete element method
DG-FEM	Discontinuous Galerkin finite element method
EFG	Element Free Galerkin
FLAC	Fast Lagrangian analysis of continua
FDM	Finite difference method
FEM	Finite element method
FVM	Finite volume method
GNN	Graph neural networks
GPU	Graphics processing unit
GUI	Graphical user interface
IDE	Integrated development environment
JIT	Just-in-time
LLVM	Low-level virtual machine
LOC	Lines-of-code
MKL	Math Kernel Library
MLPG	Meshfree local Petrov Galerkin
MPI	Massage passing interface
MPM	Material point method
PDE	Partial differential equation
PyPI	Python package index
RPIM	Radial point interpolation method
S-FEM	Smoothed finite element method
XFEM	eXtended finite element method

✉ Gang Mei
gang.mei@cugb.edu.cn

[1] School of Engineering and Technolgy, China University of Geosciences, Beijing, China

[2] Department of Mathematics and Applications "R. Caccioppoli", University of Naples Federico II, Naples, Italy

1 Introduction

With the development and improvement of traditional theories and basic equations, many common mechanical problems have been solved. However, for practical engineering, accurate analytical solutions are still intractable due to complex conditions and enormous computations. To address this situation, the field of computational mechanics, powered by traditional mechanics, mathematics, and computer science, has emerged. Fueled by the maturity of modern computers and related numerical methods, various computational mechanics methods have attained superiority in terms of obtaining numerical results that are close to experimental or monitoring results [1].

Typically, a complete computational mechanics procedure involves three components: a computational model that can reflect the essence of the given problem; a numerical algorithm considering applicability, accuracy, and efficiency; and an appropriate programming language for implementation. There is usually a best choice for the model and algorithm applied to a specific mechanical problem, while different programming languages lead to different implementations of the same computational methods.

Existing computational mechanical software packages are written primarily in C/C++, Fortran, MATLAB, or Python. Static languages such as C/C++ and Fortran run quickly; however, they are relatively difficult to use, and a long learning time is needed, which is not conducive to their use by nonprofessional developers. High-level dynamic languages, such as Python and MATLAB, have more advantages in terms of low learning costs, strong visualizations and interactions, which have led to their prevalent use in modern scientific computing. However, they are usually inefficient when handling computationally intensive problems [2, 3]. Moreover, as commercial software, MATLAB is often too expensive to purchase and is more likely to be used by large research institutions or enterprises. It is not realistic for individuals to spend large sums of money on programming languages. The merits and shortcomings of these languages are shown in Fig. 1.

As described above, there are several drawbacks presented by the commonly used programming languages for computational mechanics. None of these languages can balance productivity and performance. An alternative approach is to combine static and dynamic languages (i.e., by using fast languages to achieve the efficiency needed for the underlying code and productive languages for the remaining parts). However, unexpected errors during porting and high demands for developers can make this solution infeasible.

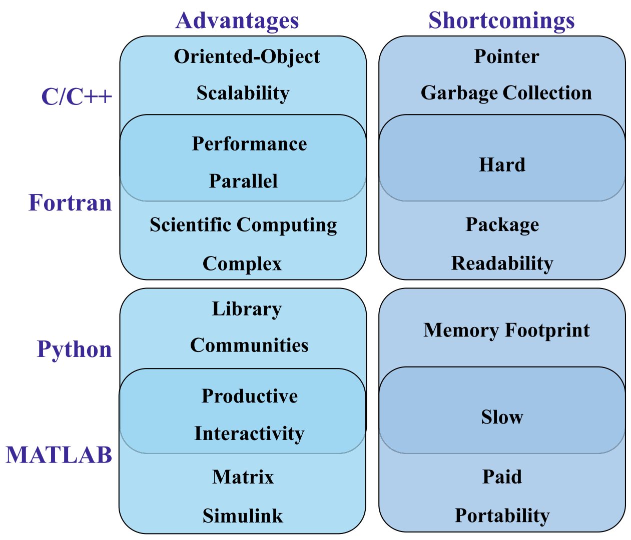

Fig. 1 Advantages and shortcomings of the common programming language in computational mechanics

To overcome this "two-language problem", the open-source Julia language was developed by the MIT. The original intention of the Julia language was to combine productivity and efficiency [3]. Julia integrates the advantages of other languages and focuses on scientific computing and data analysis. Julia is syntactically similar to dynamic languages, such as MATLAB and Python, which makes it relatively easy to learn. In terms of efficiency, Julia matches the performance of C/C++ and FORTRAN, and it is much faster than MATLAB and Python [4, 5], as shown in Fig. 2. Moreover, Julia can directly call C++ and Python with interfaces or specific packages. For high-performance computing, Julia provides parallel computing solutions, including distributed computing. In summary, Julia is a competitive programming language for scientific computing and has wide application prospects in computational mechanics.

In this paper, we present a systematic survey of the development of the Julia language in computational mechanics that focuses on the following issues:

1. Numerical computing packages developed in the Julia language,
2. The applications of some computational mechanics methods using Julia,
3. Some common problems in computational mechanics software development and feasible solutions in the Julia language, and
4. Open issues and future challenges regarding the implementation of Julia in computational mechanics.

The rest of the paper is organized as follows. A brief introduction to the Julia language is given in Sect. 2. Then, the implementations of various computational mechanics methods and relevant applications are investigated in Sect. 3. In Sect. 4, the advantages and capabilities of the Julia language with respect to the development of computational mechanics software packages are discussed. Open issues and future challenges involving the use of Julia in computational mechanics are outlined in Sect. 5. Finally, this survey is concluded.

2 Background: Brief Introduction to the Julia Language

Work on the Julia language was initiated in 2009, and after three years of development and optimization, it was officially released in 2012 [6]. The motivation for its development was the use of modern technologies to run dynamic languages efficiently. High-level dynamic scripting languages, such as

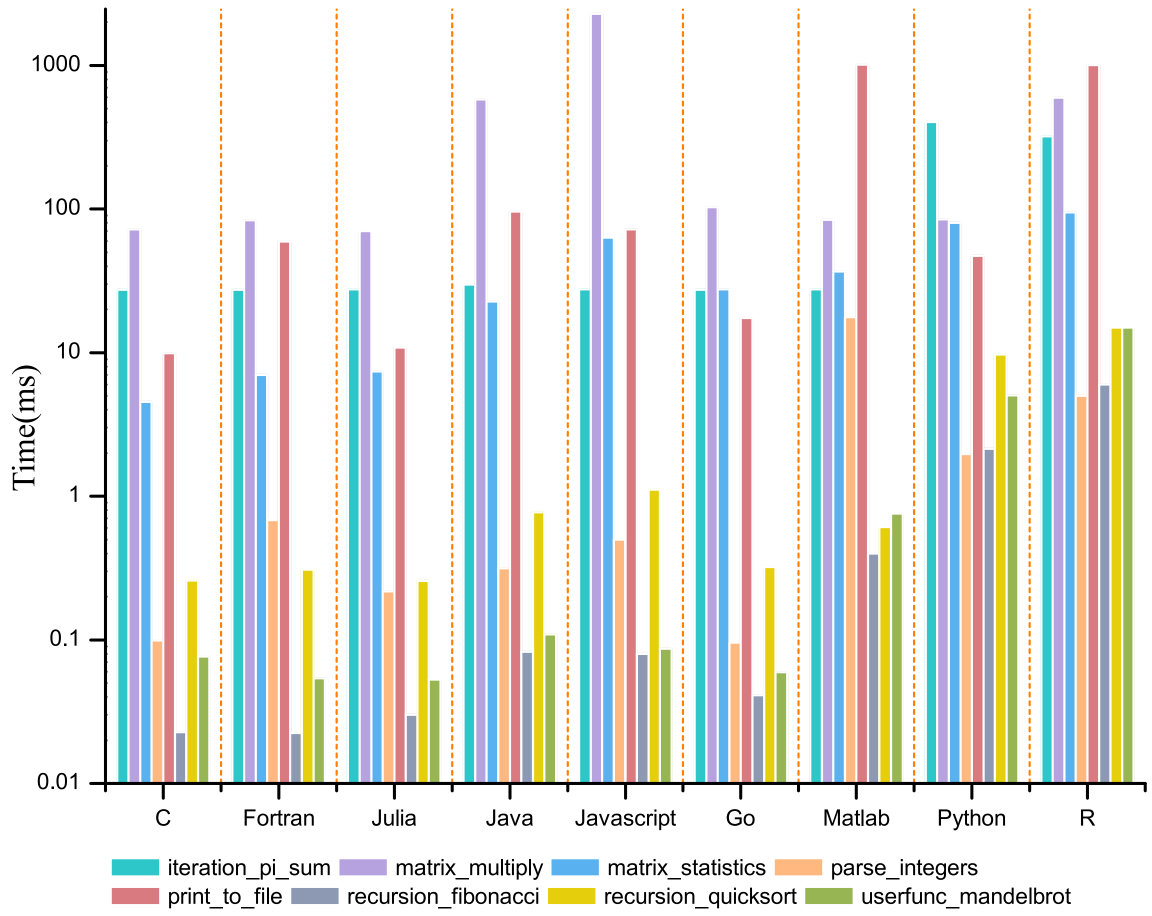

Fig. 2 Julia micro-benchmarks for performance test on a range of common code patterns, available from https://julialang.org/benchmarks/

Python and JavaScript, use interpreters, which take a high-level program and translate it into machine-readable code. The implementation of partial changes does not require the recompilation of all the code, although the code cannot run separately from the interpreter. The corresponding static languages need special compilers (such as gcc, g++, and gfortran) to compile the source code into machine code immediately, that is, to compile the code before executing it. This requirement indicates that the whole code must be recompiled when any changes are made. In contrast, in dynamic languages, the code can be run independently of the compiler. That is, dynamic languages are easy to use and more human-friendly, while static languages are more machine-friendly, which makes static languages the dominant choice for use in computationally intensive problems [7], as shown in Fig. 1.

Julia possesses the advantages of both types of languages. Its syntax is simple, easy, and expressive, similar to that of high-level languages. In addition, the Julia language also simplifies some tedious syntax. Using a flight simulation program, Sells [4] compared the amount of code used by Julia with that of other languages, and the results showed that the Lines-Of-Code (LOC) of C++, Java, and Python were 2.5, 2, and 1.5 times greater respectively, than that of Julia for the same model. Moreover, because of the Just-In-Time (JIT) compilation approach based on a Low-Level Virtual Machine (LLVM) [8], Julia is slightly slower than static languages but much faster than other dynamic languages [9]. The related benchmarks are shown in Fig. 2.

In addition, to solve intensive computing problems, parallel computing is essential in scientific computing. Julia provides efficient built-in features to help developers write high-performance parallel code. Currently, there are three Central Processing Unit (CPU)-based parallel strategies in Julia: asynchronous routines or coroutines, multithreading [10, 11], and distributed computing [12]. Developers can reasonably choose different solutions according to their own requirements. Through the use of specific packages, such as `GPUArrays.jl` and `CUDA.jl`, kernel functions and parallel arrays are conveniently supported in the Julia language for Graphics Processing Unit (GPU)-level parallel computing [13, 14].

After the stable version v1.0 was released at the end of 2018, Julia became a competitive programming language that is productive, efficient, easy-to-use, and open-source. Julia has broad application prospects in the field of high-performance scientific computing.

3 Julia Language in Computational Mechanics: Current Packages and Applications

Generally, numerical computing methods can be divided into two categories: (1) mesh-based methods and (2) meshfree/me-shless methods [15], as shown in Fig. 3. The former is represented by the Finite Element Method (FEM) and Finite Difference Method (FDM), which use meshes to discretize the study domain. Mesh-based methods usually obtain accurate descriptions of the desired physical and structural responses with high efficiency in continuity problems and small-scale deformation problems [16]. Meshfree methods use discrete field nodes instead of strictly-defined computational meshes to construct approximations to overcome the limitations of mesh-based methods in solving problems related to remeshing, mesh discontinuities, large deformations, and large displacements, such as the crack propagation and rock slope failure [17].

There are also other ways to classify the available approaches, such as the continuum-based methods and the corresponding discontinuum-based methods. However, this paper bases the discussion predominantly on the classification types discussed above. Because of the development of computer technology, these methods have been programmed into a variety of software packages for numerical computing. This section mainly introduces the computational mechanics packages developed in the Julia language.

3.1 Mesh-Based Methods

3.1.1 Finite Element Method

The FEM is a popular and widely developed method in computational mechanics. The FEM is used to construct approximations of Partial Differential Equations (PDEs) via discretization [18–20]. As one of the most successful numerical methods, the FEM has been used for many engineering problems [21–23].

The efficiency of Julia-based FEM software packages is satisfactory. For example, Ramabathiran [24] compared Julia's performance with that of competing languages using an FEM code for a simple 2D Poisson equation. The experimental results showed that the Julia code could match the performance of two mature open-source packages: `FEniCS` and `FreeFEM++` (these packages are mainly written in C++). Notably, this implementation was performed with an earlier version of the Julia language. Frontelius et al. [25] developed an open-source FEM package, `JuliaFEM`. `JuliaFEM` makes full use of the powerful Massage Passing Interface (MPI) programming in Julia to achieve efficient parallel computing. In addition, Rapo et al. [26] showed that the natural frequencies of `JuliaFEM` are not significantly different from those of commercial FEM software packages, which indicates that the performance of `JuliaFEM` is satisfactory. `JuliaFEM` supports ABAQUS-format inputs, showing good compatibility.

Furthermore, `JuAFEM.jl` is another FEM package based on Julia. It is simple and practical with fast updates and development. The developer of the popular PDE platform FEniCS [27, 28] also noticed the great potential of Julia and developed a corresponding wrapper in Julia

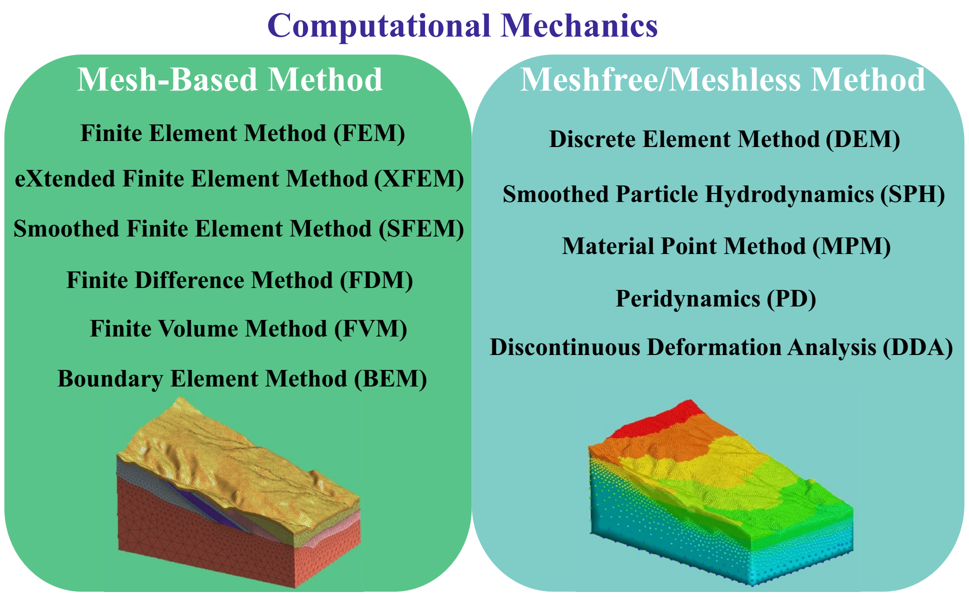
Computational Mechanics
Mesh-Based Method
Finite Element Method (FEM)
eXtended Finite Element Method (XFEM)
Smoothed Finite Element Method (SFEM)
Finite Difference Method (FDM)
Finite Volume Method (FVM)
Boundary Element Method (BEM)
Meshfree/Meshless Method
Discrete Element Method (DEM)
Smoothed Particle Hydrodynamics (SPH)
Material Point Method (MPM)
Peridynamics (PD)
Discontinuous Deformation Analysis (DDA)

Fig. 3 Common mesh-based methods and meshfree methods

called `FEniCS.jl`. `FEniCS.jl` provides open-source finite element development tools such as high-performance differential equation solvers for researchers. Moreover, there are also helpful packages, including `FEMQuad.jl`, `FEMSparse.jl`, and `FinEto-ols.jl`, for finite element programming. An auxiliary package for tensor computation was presented in [29]. `Tens-ors.jl` provides automatic differentiation and computation for both symmetric and nonsymmetric common tensors.

As an improvement of the FEM, the eXtended Finite Element Method (XFEM) reflects the discontinuities of elements by extending the shape function to enable solutions to be obtained for strongly discontinuous problems, such as crack expansions [30, 31]. The XFEM not only takes advantage of the FEM in handling various elastic-plastic problems but can also handle discontinuous problems that cause difficulties when the FEM is used. The XFEM has been widely used in the field of fracture mechanics, such as for crack propagation in metal or rock [32–34]. However, there is no current mature Julia package for the XFEM.

The Smoothed Finite Element Method (S-FEM) is another improvement of the FEM proposed by Liu [35]. The S-FEM draws on the strain smoothing techniques of meshfree methods. The compatible strain field is modified by different types of smoothing domains to improve upon the drawbacks of the traditional FEM [36–38]. Specifically, (1) S-FEM models are softer than FEM models in terms of enabling solutions to be obtained for nearly incompressible material problems; (2) the S-FEM considers the discontinuity of the strain field at the junctions of elements, enabling the attainment of more accurate solutions; and (3) the FEM demands a high-quality mesh. However, when large deformations or failures occur, elements are seriously distorted, resulting in a poor Jacobian matrix. There is no mapping process in the S-FEM; combined with the poor condition of the Jacobian matrix, the S-FEM therefore exhibits no difficulties when handling large deformations [39, 40]. The S-FEM has been widely used in limit analysis [41, 42] and problems with large deformations [43–45].

Quite recently, Huo et al. developed `juSFEM.jl`, a parallel S-FEM package for elastic problems based on Julia [11]. More specifically, compared with ABAQUS, the most commonly used commercial FEM software today, the serial `juSFEM` is 10–15% faster. The speedup of the parallel version is also satisfactory. The experimental results indicated the following: (1) `juSFEM` surpasses commercial FEM software in performance (due to the advantages of both the S-FEM and Julia language), and (2) `juSFEM` uses a multicore strategy for parallel computing, and the maximum speedup is 20.8 with a 24-core CPU. The core usage of multicore parallel computing is greater than 80%, which represents very high performance.

As the most popular computational mechanics method, the standard FEM is widely implemented in the Julia language; however, there are few packages for the XFEM and S-FEM. Moreover, most packages are more efficient than the commonly used FEM software.

3.1.2 Finite Difference Method

The FDM also incorporates the idea of discrete approximation, but it uses nodes instead of elements [46, 47]. The FDM is simple for solving PDEs with rapid convergence speed. In addition, the FDM has great flexibility in terms of handling complex initial or boundary conditions, which are quite common in real engineering situations. Therefore, FDM-based software is used extensively in different fields, such as the Fast Lagrangian Analysis of Continua (FLAC) in rock engineering [48].

Ranocha developed `SummationByPartsOpera-tors.jl` [49]. This package is used to obtain stable discretized PDEs for finite differences. `DiffEqOperators.jl` and the relative `DifferentialEquations.jl` are packages provided by SciML to solve differential equations using numerical methods [50]. The former constructs a finite difference operator and discretizes the given PDEs and boundary conditions; the latter is used to solve discrete ordinary differential equations. The package supports various forms of inputs and implements GPU-based parallel computing to ensure the efficiency of solving large-scale problems. Another package for solving PDEs using the FDM is called `Partial-Diffe-rential-Equations`.

The implementation of the FDM in the Julia language is mostly aimed at some processes or steps, and currently, there is no mature package for real analysis.

3.1.3 Finite Volume Method

Whereas the FDM is based on nodal relations for differential equations, the Finite Volume Method (FVM) balances the forces acting on control volumes by directly discretizing the integral forms of conservation laws. The FVM has good compatibility with irregular meshes and is suitable for complex practical engineering problems [51]. Moreover, compared with other computational mechanics methods based on differential equations, the integral-based FVM has a better conservation effect for achieving higher accuracy and has been applied to fluid mechanics [52, 53].

In contrast to C/C++, MATLAB, and Python, there are few implementations of the FVM in Julia. `FiniteVol-ume.jl` is a simple FVM package in the Julia language. It focuses on subsurface hydrology problems and provides an example of a box model. `VoronoiFVM.jl` is more mature, and it uses the Voronoi-based FVM to solve coupled nonlinear PDEs. This package has been very frequently updated in the past year. It is vital for a package to provide a large number of operational and computational demonstrations for beginners. Wang et al. [54] implemented the Discontinuous Galerkin Finite Element Method (DG-FEM), a method based on a combination of the FEM and FVM, to overcome the difficulties in obtaining stable schemes for high-order elements using the Julia language. Experimental results proved that the Julia implementation was more accurate and less efficient than the same method used in MATLAB.

3.1.4 Boundary Element Method

The Boundary Element Method (BEM) transforms a PDE boundary value problem into a corresponding boundary integral equation. The BEM reduces the dimensionality of the obtained solution and thus simplifies the mesh construction process. Compared with domain-based methods, the BEM discretizes only the boundary and not the interior of the physical domain. Most importantly, the BEM shows natural advantages in solving open boundary problems [55]. The BEM has been utilized in several fields, including acoustics [56, 57] and fracture mechanics [58].

Krcools developed `CompScienceMeshes.jl` [59]. The `CompScienceMeshes.jl` provides mesh construction and parameter queries for FEM/BEM calculations. Another mature package for the BEM is `BEAST.jl`. `BEAST.jl` includes common basis functions and assembly routines and supports the space-time Galerkin method for solving time-domain integral equations. A research team at the University of Brasília developed `BEM_base` using different Lagrangian elements. The repository is still in development and provides solutions for simple Helmholtz and Laplace equations. Gonzalez et al. [60] used the Julia-based BEM to compute acoustic scattering with swim bladder fish. In addition to these BEM packages, `JuliaBEM-old.jl` simply implements a direct BEM.

3.2 Meshfree Methods

3.2.1 Discrete Element Method

The Discrete Element Method (DEM) is a numerical method that considers the interactions between particles according to Newton's second law, and it has been one of the most popular meshfree methods [61]. Compared with the FEM, the DEM can handle the deformation and failure of granular objects and discontinuous materials without an element mesh. DEMs are usually divided into two categories: (1) the particle flow-based DEM and (2) the block DEM. The particle flow method is suitable for fluid flow problems and rock mass failure mechanisms [62, 63], while the block DEM has been used with some success in soil mechanics and rock mechanics [64, 65].

Currently, there is little research work on developing DEM software packages in the Julia language. Xiong et al. [66] proposed a new micromechanical model based on the kinematic hypothesis programmed it with Julia. Experimental results showed that the proposed model could save considerable computational resources compared with the DEM model. Compared with mesh-based algorithms such as the FEM, the DEM calculates a large number of particle velocities and displacements, which demands high computer memory and computational efficiency; thus, the DEM offers complex implementations, and the applicability for large-scale engineering is limited. Julia is easy to use, efficient, and powerful for parallel computing, which is an appropriate development environment for DEMs.

3.2.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) represent the state in a continuous fluid or solid according to interacting particles [67, 68]. To observe the mechanical behavior of a given system, governing equations are discretized in SPH such that particle motions can be calculated [69]. As a meshfree method, SPH reduces the requirements for particle arrangement and is easy to implement using numerical models. In addition, SPH is also a pure Lagrangian method, which is more suitable for discontinuous problems than the Euler-based meshfree method [70].

SPH demands the computation of the motions of many particles. Therefore, efficiency is an important issue regarding its implementation. Eschnett and Rjkat developed two packages called `SPH.jl` [71, 72]. `Juss` is an SPH framework written using Julia. `Juss` draws on `PySPH` [73], a popular and mature SPH framework in Python. The relative development of the approach for system analysis code coupling is still in progress.

3.2.3 Material Point Method

As another commonly used meshfree method, the Material Point Method (MPM) possesses the advantages of both the Lagrangian and Euler methods. The MPM uses Lagrangian material points for description, while nodal momentum equations are solved with an Eulerian background grid [74, 75]. This method alleviates the difficulty of the boundary conditions of SPH and avoids the mesh distortion induced by the FEM [76]. Similar to other meshfree methods, the MPM is widely applied to high-speed collision [77], material deformation [78], and large deformation problems in geotechnics [79].

`MPM-Julia` is an MPM package written in the Julia language by Sinaie [80]. To test the efficiency of the

Julia language, the same MPM code was implemented in MATLAB. The experimental results of three typical solid mechanics examples proved that the Julia code was more efficient than the corresponding MATLAB code in all cases, and the highest speedup achieved was 8.5. Notably, this library was written in Julia version v0.4.5, which is not a stable version. However, the author did not conduct maintenance. Additional modification is required for this library to be used in a new version of the language. The `MPM` package applies the MPM to fluid and hyperelastic materials. In contrast to `MPM-Julia`, this package was developed recently and is supported only by Julia version 1.5 or later.

3.2.4 Discontinuous Deformation Analysis

The Discontinuous Deformation Analysis (DDA) method was first proposed by Shi in 1988 [81, 82]. It combines the advantages of the FEM and DEM to avoid the decreased accuracy caused by mesh distortion and is successfully applied to large deformations and displacements of discontinuous rock-soil masses [83–85]. Rock masses are divided into different blocks according to their structural planes in DDA. In addition, the displacement problem is solved by the principle of minimum potential energy. DDA is more flexible and simpler to use for obtaining satisfactory results in rock analysis than the FEM and DEM, which makes DDA very suitable for related research.

However, to the best of the authors' knowledge, there is no current implementation of computational mechanics analysis using DDA in the Julia language. Further research work is expected.

3.2.5 Peridynamics

Peridynamics, which aims to address complicated discontinuous problems, was proposed by Silling [86]. Peridynamics is based on integral equations instead of the derivatives of displacement components; therefore, the governing equations are not affected by discontinuities [87, 88]. Peridynamics plays significant roles in discontinuity problems, including fatigue analysis and crack propagation problems [89–91].

Compared with other numerical methods, peridynamic methods are relatively new. Currently, the `peridynamics` developed by Annereinarz and `Peridynamics.jl` developed by Johntfoster are two of the few packages available.

3.2.6 Other Meshfree Methods

Other meshfree methods, such as the weak-form based meshfree methods including the Element Free Galerkin (EFG) method [92], the Meshfree Local Petrov Galerkin (MLPG) method [93, 94], and the Radial Point Interpolation Method (RPIM) [95, 96], solve PDEs by reducing the orders of derivatives; in contrast, the strong form methods apply an approximate strategy to partial differential operators. These meshfree methods are more flexible in complex model simulations than mesh-based methods since predefined meshes are discarded. In conclusion, weak-form-based meshfree methods are used predominantly in applications with large deformations, fracture mechanisms, and hydrodynamics [15, 17].

Currently, there is no mature Julia-based package for weak-form-based meshfree methods, and related works are mostly developed in C/C++ or MATLAB.

4 Julia Language in Computational Mechanics: Capabilities and Advantages in Software Development

Computational mechanics involves several aspects and faces many problems in terms of software development. Some common problems (shown in Fig. 4) are described in this section, which shows examples to introduce how Julia, as a language for scientific computing, has advantages when used for the development of computational mechanics-based software packages.

4.1 Solving Large Linear Systems

Solving systems of linear equations is critical in most numerical methods and is usually the most time-consuming sub-procedure [11, 97, 98]. Therefore, efficient solutions for large-scale linear equations are important. The Julia language provides a variety of ways to solve equations.

1. Julia integrates a series of simple and practical linear algebra operations in the `LinearAlgebra` module, including many matrix factorizations. With the appropriate factorizations, a linear solution can be calculated by using the "\" operation, which is fast and simple. More details can be found in the linear algebra section of the Julia language documentation [99].
2. When a certain computational efficiency requirement is imposed, Julia can call the `PARDISO` Library derived from the `Intel math kernel library` (MKL) [100] using the `Pardiso.jl` package [101]. Based on CPU parallelism, `PARDISO` provides users with a high-performance solver with low memory usage.
3. If the CPU-based parallel solver is still unsatisfactory, `CUDA.jl` of NVIDIA can be an optimal choice [13, 14]. The package contains the efficient linear equation solvers `cuSparse` and `cuSolver`, which solve large-scale equations using the power of high-performance GPU-based massive parallelism.

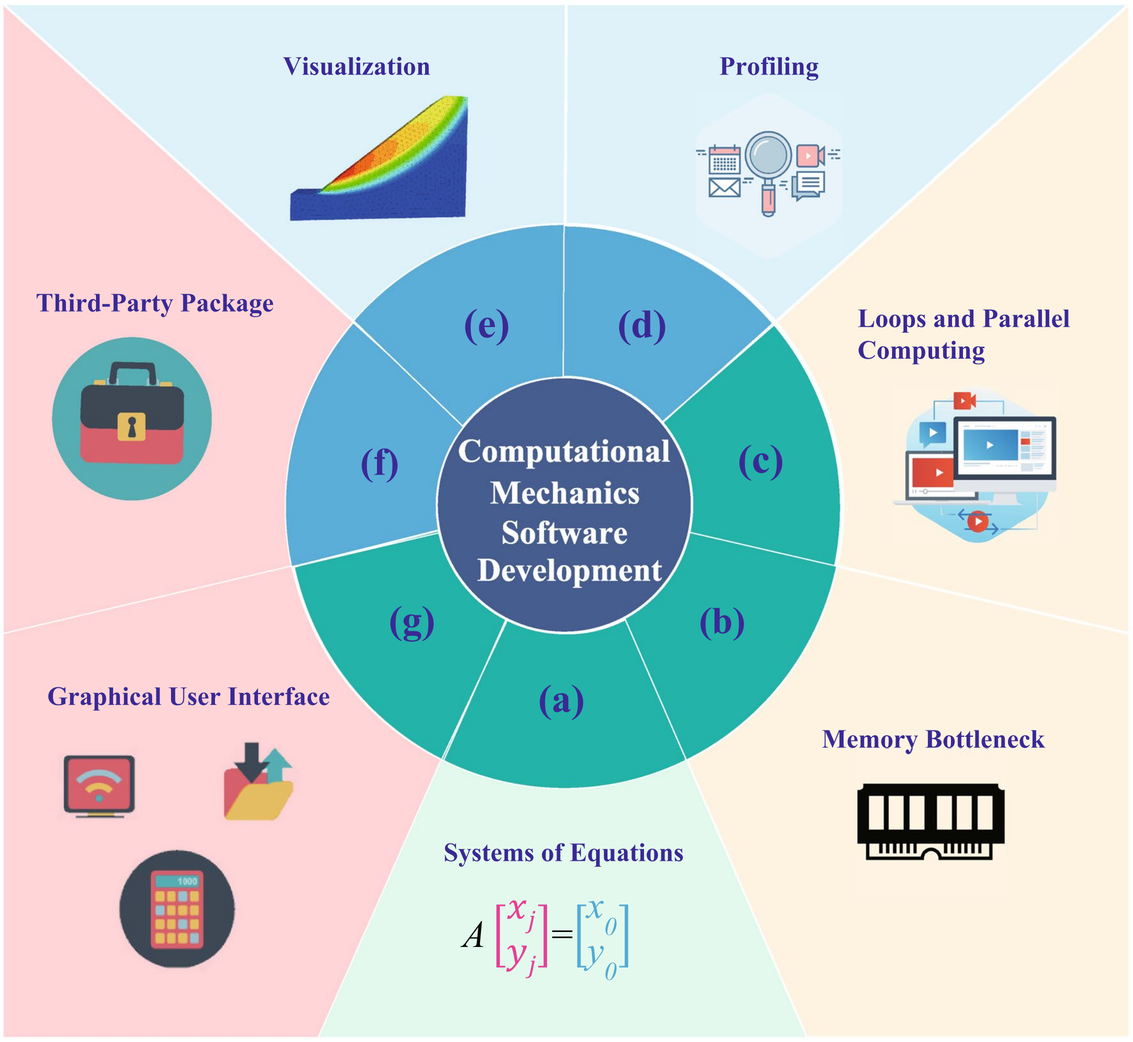
Visualization
Profiling
Third-Party Package
Loops and Parallel Computing
Computational Mechanics Software Development
(a)
(b)
(c)
(d)
(e)
(f)
(g)
Graphical User Interface
Memory Bottleneck
Systems of Equations
$A\begin{bmatrix}x_j\\y_j\end{bmatrix}=\begin{bmatrix}x_0\\y_0\end{bmatrix}$

Fig. 4 Common problems in computational mechanics software development

4. `DifferentialEquations.jl` [102] is suitable for general PDEs, and `DiffEqGPU.jl` provides a parallel solution for PDEs with a single GPU or multiple GPUs.

4.2 Addressing Memory Bottlenecks

Large-scale matrix operations lead to insufficient memory in computational mechanics programs [97, 103]. Thus, a sparse matrix is required to reduce the corresponding memory usage [15]. Julia, which is simple but powerful, is very similar to MATLAB in terms of matrix operations. Developers can choose the COOrdinate (COO), Compressed Sparse Column (CSC), and Compressed Sparse Row (CSR) formats to store large matrices. `MKLSparse.jl` [104] provides powerful and efficient sparse matrix operations. `cuSparse` [105] provides a GPU-based sparse matrix function. Matrices are stored in Julia as columns first, which is different from those in C/C++ and Python. Therefore, choosing the CSC format instead of the CSR format is more conducive to memory access and improves performance. In addition, Julia has garbage collectors similar to those in Python and Java that reduce the extra workload required by the developer.

4.3 Optimizing Loops

Loops are usually the most time-consuming procedures of computer program development due to the lack of type information [3, 106]. Julia incorporates a special loop optimization process using transparent data and performance models. Sinaie compared the performance of "for loops" and "vectorization" in large arrays and matrices [80]. The experimental results showed that the "j-i loops" are twice as fast as "vectorization", while the "i-j" form without aligned memory access reduced the performance, which was only 1/6 of that achieved with aligned memory access in the worst case. Therefore, the appropriate use of "for loops" instead of "vectorization" significantly improves efficiency. Parallel computing can also be used to optimize loops when there is no data dependency. `GPUifyLoops.jl` can be used to write loop code in heterogeneous computing to improve performance. Currently, it is integrated into `KernelAbstraction-s.jl` [107], a package for GPU kernel writing.

4.4 Use of Parallel Computing

Parallel computing is very important when developing computational mechanics packages [108]. There are powerful and mature parallel CPU capabilities in Julia that provide asynchronous routines or coroutines, multithreading, and distributed computing. Multithreading allows tasks to run simultaneously on a CPU using shared memory. Julia provides two different parallel primitives, the `@threads` and `@splash`, to implement static and dynamic scheduling mechanisms, which are similar to the "static" and "dynamic" mechanisms of C++ OpenMP, respectively. Because of its powerful compiler, Julia can yield a high speedup that is usually 50–80% as fast as the theoretical maximum. In parallel GPU programming, Julia supports mainstream GPU platforms, including NVIDIA, Intel, and AMD, as shown in Table 1. Among these platforms, NVIDIA's CUDA is the most mature, and developers generally choose it. `CUDA.jl` integrates a common toolkit for GPU programming according to different needs, such as `CuArray` for users with no experience and the CUDA kernel for special requirements.

4.5 Profiling

The Julia profile module applies the sampling and backtracking mechanisms to measure the relative running time of a particular line of code. The sampling mechanism enables the program to perform analysis with little performance loss. In addition, the memory allocation of each line of code can also be queried by the built-in `@time` and `@allocated` primitives to reduce unnecessary expenses. Through profiling, it is easier for developers to find the bottleneck of a program, which is very important in efficiency-driven computational mechanics.

4.6 Visualization

Visualizations, such as stress-strain cloud maps, are necessary for further analysis after preliminary computing is conducted using various computational mechanics methods [109]. Julia provides manifold packages for plotting, including `PyPlot`, `GR`, `PlotlyJS`, and others. `ParaView` is an open-source visualization software. Taking the commonly used `vtu` format as an example, `ParaView` requires only the displacement, stress, strain, and model mesh exported from the `VTK` package for visualization.

4.7 Package

Julia provides the `Pkg` module, a built-in package manager. As with Python `pips`, developers can enjoy complete installations, updates, and deletions by simply adding a package name. The package manager can prevent various errors in early configuration phase. Compared with other languages, Julia is still in the early stage of development. Thus, the quantity of available packages is insufficient. In addition, many high-quality, mature numerical computing libraries are written in static languages such as C and Fortran. The aforementioned facts result in a higher demand for powerful interfaces. Fortunately, Julia developers are well aware of this, and several interface packages have been developed to achieve a certain balance due to the lack of third-party libraries. Currently, Julia can directly invoke existing libraries from popular languages without many glue codes, which is very beneficial for software package development in the field of computational mechanics.

4.8 Graphical User Interface

Graphical User Interfaces (GUIs), which are directly related to user experience and determine popularization, have always been an important aspect of software development [110]. `QML.jl` and `Gtk.jl` are two main GUI packages in Julia. The former allows for calling Qt in Julia using an interface. Qt was developed in C++, which has good cross-platform support for users to design beautiful UIs, and it is one of the most popular GUI development frameworks. `Gtk.jl`, which has strong portability and rich content, provides an interface to the GTK library. GTK has a mature

Table 1 Different packages for GPU computing in Julia language

		Platform				
		NVIDIA	AMD	Intel	Other	
Integrated Package		CUDA.jl	AMDGPU.jl	oneAPI.jl	OpenCL.jl	ArrayFire.jl
Abstraction Level	Array	CuArray	ROCArrays.jl	oneArray	N/A	N/A
	Kernel	CUDA Kernel	AMDGPUnative.jl	oneapi	N/A	N/A

community that ensures certain updates and is also a commonly used GUI library.

5 Julia Language in Computational Mechanics: Open Issues and Challenges

In contrast to previous programming languages, Julia balances efficiency and simplicity. Increasingly many developers are choosing Julia, and more projects are based on Julia. Certainly, Julia is not stagnant: it has been updated every four months on average since the release of the stable version.

In addition, with continuous developments in the Julia language, there are still some open issues to be resolved. Moreover, the future development of computational mechanics also presents new challenges and opportunities for the Julia language. The aforementioned issues and challenges are discussed in the following sections.

5.1 Open Issues for the Julia Language

5.1.1 Immature Development of the Programming Environment

Although it has been nine years since Julia was officially released, this is a small developmental period for a programming language. In contrast, Python was first developed in 1989 and is currently in version 3.9.1; MATLAB was developed in 1984 and has maintained its biannual update frequency in recent years; Java was developed in 1995, and JDK16 was recently released; and C/C++ and Fortran were developed even earlier. Julia is in its infancy; its grammar is still being updated and optimized, and some old functions are removed or rendered invalid in a new version, which also causes difficulties when performing updates. In addition, Julia lacks a specialized integrated development environment (IDE) [111]. Third-party platform editors and IDEs are not appropriate choices for users wanting a convenient programming experience.

5.1.2 Lack of Third-Party Packages

The number of available packages largely reflects the popularity of a programming language. The higher the quality and the wider the scope of a language's libraries, the more a language can attract developers to use it. This is similar to the sustainable development of a programming language. As the most commonly used language (according to the PopularitY of Programming Languages [112]), Python has 279634 projects on the Python Package Index (PyPI) (as of December 30, 2020) and 3341337 related projects on GitHub, while Julia has only 15643 projects on GitHub. As shown by this huge gap, Julia developers still need time to improve its ecosystem. Due to the strong compatibility of Julia, packages such as `PyCall.jl` and `Cxx.jl` provide efficient and convenient interfaces with other languages. Notably, certain efficiency losses may be caused by interfacing. The Julia language still has a long way to go to catch up with other mature languages.

5.1.3 Requirements for an Intelligence Compiler

Julia is specifically developed for high-performance scientific computing. Compared with those of other languages, a Julia developer is more likely to be a scientific researcher than a professional computer software programmer [13]. For these researchers, it is not a reasonable choice to spend much time away from scientific research to learn how to write efficient code; thus, a smart and convenient compiler is required. In addition, high-performance compilers are also conducive to the real-time compilation and rewriting of productivity language code with high efficiency [113]. More intelligent and efficient compilers will attract more researchers and promote the development of the Julia ecosystem.

5.2 Challenges for the Julia Language

The Julia language provides concise syntax with expressive features, enabling the required codes to be much smaller than those written in C/C++ for the same implementation. In addition, some numerical computing packages in Julia are similar to commercial software in terms of efficiency. With increasingly many computational mechanics implementations emerging, Julia shows exciting potential in this field. However, challenges remain.

5.2.1 Challenges Regarding Imbalanced Implementation

Although the mesh-based method is mature, the requirement of high-quality meshes for complex models and its poor performance on large deformations and extremely discontinuous problems limit its applications in engineering. The meshfree method appears to overcome these drawbacks. Nevertheless, few meshfree packages use the Julia language, and the existing packages are mostly in the early stages of development. In general, the meshfree method requires large amounts of computations and a complicated program design [15, 114], and using Julia for meshfree software package development is a relative challenge for researchers.

5.2.2 Challenge Regarding Parallel Computing

Currently, computational models tend to be large and complicated, especially for real engineering, which presents a challenge to mechanics researchers. To reduce the required computation time, some trade-offs must be made with respect to the quality of the models and the accuracy of the results. Parallel computing can alleviate the aforementioned problems to a certain extent [108, 115]. Julia provides powerful and multifold parallel computing routines. However, only a few studies use these strategies.

5.2.3 Challenge Regarding Legacy Code

Due to the earlier development of computational mechanics, legacy code, which is usually too old to be maintained or ported, still widely exists. Therefore, it takes considerable effort for researchers to switch programming languages, which is the main obstacle for the development of the Julia language in computational mechanics. An ideal opportunity awaits in terms of the use of new technology, for which traditional language is rarely involved; thus, the impact of legacy code will be minimized, and the unique advantages of Julia can be maximized.

5.2.4 Challenge Regarding Deep Learning

In recent years, with the development of deep learning, especially the Graph Neural Networks (GNNs), some complex physical processes have been simulated [116, 117]. For example, Pestourie et al. [118] proposed an active learning algorithm for composite materials. Deep learning methods provide at least a two orders of magnitude speedup over the numerical PDE approach. Ye et al. [119] used a convolutional neural network for flow analysis, and their experiment showed that the proposed method could achieve results with similar precision to that of the results of computational fluid dynamics. More complex mechanical processes might be realized by deep learning in the future [120]. If the mechanical mechanism of a small model obtained by deep learning can be extended to the simulation of a larger model, even until practical engineering can be achieved, a more efficient and explanatory computational mechanics analysis process will be established. It will be important to be an early bird when developing deep learning-based computational mechanics.

6 Conclusion

In this paper, we have presented a comprehensive survey on the development of the Julia programming language in computational mechanics. There are multiple contributions made in this article. (1) Existing Julia packages, in which various popular computational mechanics methods have been implemented for related applications, have been investigated. (2) Feasible solutions to common problems that have been generated while developing computational mechanics programs in the Julia language have been presented. (3) Several open issues and potential opportunities for both the Julia language and its applications in computational mechanics have been discussed.

This survey shows that the Julia language is very competitive for use in scientific computing. The JIT compilation based on LLVM and the special type system allow Julia to handle computationally intensive problems similar to traditional static languages. Moreover, standard Julia packages can surpass some commercial software in terms of efficiency, which is an extraordinary advantage for computational mechanics programming. In addition, Julia is succinct and easy to use, similar to other modern high-level dynamic languages. The Julia language certainly balances the trade-off between productivity and efficiency. However, the applications of Julia to computational mechanics are insufficient in depth, and most existing works have implemented only a few algorithms. In addition, FEM packages constitute a large portion of these outputs, and there are few meshfree packages. Moreover, in parallel computing, the CPU-based Julia implementation is more dominant. The Julia language is in the early stage, and both the language itself and the community are developing continuously, while computational mechanics tends to be mature field with respect to theories and applications, even after decades. In this combination of a new language and a mature subject, frontier technologies such as deep learning-based simulation may be the most potent aspects.

In summary, for researchers who are new to computational mechanics or exploring new directions, the Julia language is highly recommended. Researchers who have engaged in related research for years will have difficulty rewriting or porting all the legacy code; therefore, they will not readily change their programming language. However, applying Julia for some simple implementations can give these researchers possible solutions to overcoming future obstacles.

Acknowledgements This research was jointly supported by the Natural Science Foundation of China (Grant No. 11602235), the Fundamental Research Funds for China Central Universities (2652018091), and Major Program of Science and Technology of Xinjiang Production and Construction Corps (2020AA002). The authors would like to thank the editor and the reviewers for their comments.

References

1. Liu GR (2016) An overview on meshfree methods: for computational solid mechanics. Int J Comput Methods 13(5):1630001
2. Lubin M, Dunning I (2015) Computing in operations research using Julia. INFORMS J Comput 27(2):238–248
3. Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: a fresh approach to numerical computing. SIAM Rev 59(1):65–98

4. Sells R (2020) Julia programming language benchmark using a flight simulation. In: 2020 IEEE aerospace conference, IEEE, pp 1–8
5. Dogaru I, Dogaru R (2015) Using python and Julia for efficient implementation of natural computing and complexity related algorithms, pp 599–604
6. Bezanson J, Karpinski S, Shah VB (2012) A fast dynamic language for technical computing, Julia
7. Perkel JM (2019) Julia: come for the syntax, stay for the speed. Nature 572(7767):141–142
8. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation, pp 75–86
9. Moura RAR, Schroeder MAO, Silva SJS, Nepomuceno EG, Vieira PHN, Lima ACS (2019) The usage of Julia programming in grounding grids simulations: an alternative to MATLAB and Python
10. Barros DA, Bentes C (2020) Analyzing the loop scheduling mechanisms on Julia multithreading, pp 257–264
11. Huo Z, Mei G, Xu N (2021) juSFEM: a Julia-based open-source package of parallel smoothed finite element method (S-FEM) for elastic problems. Comput Math Appl 81:459–477
12. Kratochvíl M, Hunewald O, Heirendt L, Verissimo V, Vondrášek J, Satagopam VP, Schneider R, Trefois C, Ollert M (2020) GigaSOM.jl: high-performance clustering and visualization of huge cytometry datasets. GigaScience 9(11):1–8
13. Besard T, Churavy V, Edelman A, Sutter BD (2019) Rapid software prototyping for heterogeneous and distributed platforms. Adv Eng Softw 132:29–46
14. Besard T, Foket C, De Sutter B (2019) Effective extensible programming: unleashing Julia on GPUs. IEEE Trans Parallel Distrib Syst 30(4):827–841
15. Xu N, Mei G, Qin J, Li Y, Xu L (2021) GeoMFree3D: a package of meshfree local radial point interpolation method (RPIM) for geomechanics. Comput Math Appl 81:113–132
16. Geuzaine C, Remacle J-F (2009) Gmsh: a 3-D finite element mesh generator with built-in pre- and post-processing facilities. Int J Numer Methods Eng 79(11):1309–1331
17. Chen J-S, Hillman M, Chi S-W (2017) Meshfree methods: progress made after 20 years. J Eng Mech 143(4):04017001
18. Munjiza A (2004) The combined finite-discrete element method, vol 12
19. Liu GR, Quek SS (2013) The finite element method: a practical course, 2nd edn
20. Hughes TMD, Thomasj R (2000) The finite element method: linear static and dynamic finite element analysis
21. Klaus-Jürgen B (2006) Finite element procedures. Klaus-Jurgen Bathe, Berlin
22. Belytschko T, Liu WK, Moran B, Elkhodary K (2000) Nonlinear finite elements for continua and structures. Wiley, New York
23. Li ZC, Cui XY, Cai Y (2018) Analysis of heat transfer problems using a novel low-order FEM based on gradient weighted operation. Int J Therm Sci 132:52–64
24. (2013) Ramabathiran AA (2013) Finite Element programming in Julia. https://www.codeproject.com/articles/579983/finite-element-programming-in-julia
25. Frondelius T, Aho J (2017) JuliaFEM-open source solver for both industrial and academia usage. Rakenteiden Mekaniikka 50:229
26. Rapo M, Aho J, Frondelius T (2017) Natural frequency calculations with JuliaFEM. Rakenteiden Mekaniikka 50:300
27. Alnæs M, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, Richardson C, Ring J, Rognes M, Wells G (2015) The FEniCS project version 1.5. 3
28. Anders L, Garth W, Kent-Andre M (2011) Automated solution of differential equations by the finite element method. FEniCS Book 84:04
29. Carlsson K, Ekre F (2019) Tensors.jl- tensor computations in Julia. J Open Res Softw 7(1)
30. Belytschko T, Black T (1999) Elastic crack growth in finite elements with minimal remeshing. Int J Numer Meth Eng 45(5):601–620
31. Nicolas M, Ted B (2002) Extended finite element method for cohesive crack growth. Comput Methods Appl Mech Eng 69:01
32. Obara Y, Nakamura K, Yoshioka S, Sainoki A, Kasai A (2020) Crack front geometry and stress intensity factor of semi-circular bend specimens with straight through and chevron notches. Rock Mech Rock Eng 53(2):723–738
33. Xiaoping Z, Junwei C (2019) Extended finite element simulation of step-path brittle failure in rock slopes with non-persistent en-echelon joints. Eng Geol 250:02
34. Qinglei Z, Zhanli L, Tao W, Yue G, Zhuo Z (2018) Fully coupled simulation of multiple hydraulic fractures to propagate simultaneously from a perforated horizontal wellbore. Comput Mech 61:02
35. Liu GR, Dai KY (2007) A smoothed finite element method for mechanics problems. Comput Mech 39:859–877
36. Liu GR (2019) The smoothed finite element method (S-FEM): a framework for the design of numerical models for desired solutions. Front Struct Civ Eng 13:01
37. Zeng W, Liu GR (2018) Smoothed finite element methods (S-FEM): an overview and recent developments. Arch Comput Methods Eng 25:397–435
38. Gang W, Xiang-Yang C, Hui F, Li GY (2015) A stable node-based smoothed finite element method for acoustic problems. Comput Methods Appl Mech Eng 297:09
39. Liu GR (2010) Element smoothed finite methods
40. Cui XY, Chang S (2015) Edge-based smoothed finite element method using two-step Taylor Galerkin algorithm for lagrangian dynamic problems. Int J Comput Methods 12:1550028
41. Thien V-M (2020) A stable node-based smoothed finite element method for stability analysis of two circular tunnels at different depths in cohesive-frictional soils. Comput Geotech 129:11
42. Nguyen-Xuan H, Rabczuk T (2015) Adaptive selective ES-FEM limit analysis of cracked plane-strain structures. Front Struct Civ Eng 9:478–490
43. Tian F, Tang X, Xu T, Li L (2020) An adaptive edge-based smoothed finite element method (ES-FEM) for phase-field modeling of fractures at large deformations. Comput Methods Appl Mech Eng 372
44. FanP HuangW, Zhang ZQ, Guo T, Ma YE (2020) Phase field simulation for fracture behavior of hyperelastic material at large deformation based on edge-based smoothed finite element method. Eng Fracture Mech 238:107233
45. Jiang C, Han GR, Xu L, Zhi-Qian Z, Wei Z (2015) A smoothed finite element method for analysis of anisotropic large deformation of passive rabbit ventricles in diastole. Int J Numer Methods Biomed Eng 31:n/a-:01
46. Mohammadnejad M, LiuH, Chan A, Dehkhoda S, Fukuda D (2018) An overview on advances in computational fracture mechanics of rock. Geosyst Eng 1–24
47. Elmo D (2006) Evaluation of a hybrid FEM/DEM approach for determination of rock mass strength using a combination of discontinuity mapping and fracture mechanics modelling, with particular emphasis on modelling of jointed pillars, vol 01, University of Exeter, UK
48. Hossein H (2001) Rock characterisation facility (rcf) shaft sinking-numerical computations using flac. Int J Rock Mech Min Sci 38:59–65
49. Ranocha P. SummationByPartsOperators.jl. https://github.com/ranocha/SummationByPartsOperators.jl

50. Rackauckas C, Nie Q (2017) Differential equations.jl: a performant and feature-rich ecosystem for solving differential equations in Julia. J Open Res Softw 5:05
51. Cardiff P, Demirdžić I (2021) Thirty years of the finite volume method for solid mechanics. Arch Comput Methods Eng
52. Hashemi-Tilehnoee M, Dogonchi AS, Seyyedi SM, Sharifpur M (2020) Magneto-fluid dynamic and second law analysis in a hot porous cavity filled by nanofluid and nano-encapsulated phase change material suspension with different layout of cooling channels. J Energy Storage 31:101720
53. Matsunaga T, Yuhashi N, Shibata K, Koshizuka S (2020) A wall boundary treatment using analytical volume integrations in a particle method. Int J Numer Meth Eng 121:05
54. Wang Y, Liu M, Li H, Liang S, Cao Q (2015) Implementation of DG-fem with dynamic Julia language for accurate EM simulation, pp 1850–1851
55. Liu Y, Mukherjee S, Nishimura N, Schanz M, Ye W, Sutradhar A, Pan E, Dumont N, Sáez A (2011) Recent advances and emerging applications of the boundary element method. Appl Mech Rev 64:1001
56. Citarella R, Federico L, Cicatiello A (2007) Modal acoustic transfer vector approach in a FEM-BEM vibro-acoustic analysis. Eng Anal Bound Elem 31:248–258
57. Li H, Zixiao M, Ke Y, Tian Y, Luo W (2019) A fast optimization algorithm of FEM/BEM simulation for periodic surface acoustic wave structures. Information 10:90
58. García-Sánchez F, Zhang C (2007) A comparative study of three BEM for transient dynamic crack analysis of 2-D anisotropic solids. Comput Mech 40:753–769
59. Krcools. CompScienceMeshes.jl. https://github.com/krcools/CompScienceMeshes.jl
60. Gonzalez J, Lavia E, Blanc S, Maas M, Madirolas A (2020) Boundary element method to analyze acoustic scattering from a coupled swimbladder-fish body configuration. J Sound Vib 486:115609
61. Weerasekara N, Powell M, Cleary P, Tavares L, Evertsson M, Morrison R, Quist J, Carvalho R (2013) The contribution of DEM to the science of comminution. Powder Technol 248:3–24
62. Zhao L, Zhang S, Huang D, Wang X, Zhang Y (2020) 3D shape quantification and random packing simulation of rock aggregates using photogrammetry-based reconstruction and discrete element method. Construct Build Mater 262:119986
63. Bahaaddini M, Sharrock G, Hebblewhite B (2013) Numerical investigation of the effect of joint geometrical parameters on the mechanical properties of a non-persistent jointed rock mass under uniaxial compression. Comput Geotech 49:206–225
64. Hanbin W, Bin Z, Gang M, Nengxiong X (2019) A statistics-based discrete element modeling method coupled with the strength reduction method for the stability analysis of jointed rock slopes. Eng Geol 264:08
65. Xifei D, Jianbo Z, Shougen C, Jian Z (2012) Some fundamental issues and verification of 3DEC in modeling wave propagation in jointed rock masses. Rock Mech Rock Eng 45:09
66. Xiong H, Yin ZY, Nicot F (2020) Programming a micro-mechanical model of granular materials in Julia. Adv Eng Softw 145:102816
67. Liu M, Liu GR (2010) Smoothed particle hydrodynamics (SPH): an overview and recent developments. Arch Comput Methods Eng 17:25–76
68. Monaghan J (1992) Smoothed particle hydrodynamics. Annu Rev Astron Astrophys 30:543–574
69. Angelos M, Nikolaos K, Emmanouil-Lazaros P (2019) Meshless methods for the simulation of machining and micro-machining: a review. Arch Comput Methods Eng 27:03
70. Xinyan P, Pengcheng Yu, Guangqi C, Mingyao X, Yingbin Z (2020) Development of a coupled DDA-SPH method and its application to dynamic simulation of landslides involving solid-fluid interaction. Rock Mech Rock Eng 53:01
71. eschnett. SPH.jl. https://github.com/eschnett/SPH.jl
72. archermarx. SPH.jl. https://github.com/rjkat/SPH.jl
73. Ramachandran P, Bhosale A, Puri K et al (2020) A python-based framework for smoothed particle hydrodynamics
74. Zhang X, Chen Z, Liu Y (2017) The material point method, pp 37–101
75. Rodríguez PJ, Josep C, Jonsén P (2018) Numerical methods for the modelling of chip formation. Arch Comput Methods Eng 27:12
76. Shyamini K, Kenichi S (2016) Implicit formulation of material point method for analysis of incompressible materials. Comput Methods Appl Mech Eng 313:10
77. Ravindra A, Xiaofei P, Huang Y, Xiong Z (2012) Application of material point methods for cutting process simulations. Comput Mater Sci 57:05
78. Duan Z, Tilak D (2017) Shock waves simulated using the dual domain material point method combined with molecular dynamics. J Comput Phys 334:01
79. Sołowski W, Sloan S (2015) Evaluation of material point method for use in geotechnics. Int J Numer Anal Methods Geomech 39:685–701
80. Sinaie S, Nguyen VP, Nguyen CT, Bordas S (2017) Programming the material point method in Julia. Adv Eng Softw 105:01
81. Shi GH (1992) Discontinuous deformation analysis: a new numerical model for the statics and dynamics of deformable block structures. Eng Comput 9:157–168
82. Shi GH, Goodman R (1985) Two dimensional discontinuous deformation analysis. Int J Numer Anal Methods Geomech 9:541–556
83. Tsesarsky M, Hatzor Y, Sitar N (2005) Dynamic displacement of a block on an inclined plane: analytical, experimental and DDA results. Rock Mech Rock Eng 38:153–167
84. Guangqi Chen L, Zhang ZY, Jian W (2013) Numerical simulation in rockfall analysis: a close comparison of 2-D and 3-D DDA. Rock Mech Rock Eng 46:05
85. David D, Nicholas S (2004) Time integration in discontinuous deformation analysis. J Eng Mech-Asce 130:03
86. Silling SA (2000) Reformulation of elasticity theory for discontinuities and long-range forces. J Mech Phys Solids 48:175–209
87. Yehui B, Xiang-Yang C, Li ZC (2017) A coupling approach of state-based peridynamics with node-based smoothed finite element method. Comput Methods Appl Mech Eng 331:12
88. Yehui B, She L, Xin H, Xiang-Yang C (2019) An implicit dual-based approach to couple peridynamics with classical continuum mechanics. Int J Numer Methods Eng 120:07
89. Zhou X, Shou Y (2016) Numerical simulation of failure of rock-like material subjected to compressive loads using improved peridynamic method. Int J Geomech 17:04016086
90. Zhou W, Dahsin L, Ning L (2017) Analyzing dynamic fracture process in fiber-reinforced composite materials with a peridynamic model. Eng Fract Mech 178:04
91. Hu YL, Erdogan M (2016) Peridynamics for fatigue life and residual strength prediction of composite laminates. Compos Struct 160:10
92. Belytschko T, Lu YY, Gu L (1994) Element-free Galerkin methods. Int J Numer Methods Eng 37:229–256
93. Atluri S, Zhu T (1998) A new meshless local Petrov–Galerkin (MLPG) approach. Comput Mech 22:117–127
94. Atluri S, Zhu T (2000) The meshless local Petrov-Galerkin (MLPG) approach for solving problems in elasto-statics. Comput Mech 25:169–179
95. Liu GR, Gu YT (2001) A local radial point interpolation method (LRPIM) for free vibration analyses of 2-D solids. J Sound Vib 246:29–46

96. Cui XY, Feng H, Li GY, Feng SZ (2015) A cell-based smoothed radial point interpolation method (CS-RPIM) for three-dimensional solids. Eng Anal Bound Elem 50:474–485
97. Harari I, Grosh K, Hughes TJR, Malhotra M, Pinsky PM, Stewart JR, Thompson LL (1996) Recent developments in finite element methods for structural acoustics. Arch Comput Methods Eng 3(2–3):131–309
98. Tchonkova M, Sture S (2001) Classical and recent formulations for linear elasticity. Arch Comput Methods Eng 8:41–74
99. Julia Documentation. https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/
100. Intel Math kernel library. https://software.intel.com/en-us/mkl
101. JuliaSparse. Pardiso.jl. https://github.com/JuliaSparse/Pardiso.jl
102. SciML. DifferentialEquations.jl. https://github.com/SciML/DifferentialEquations.jl
103. Serban G, Peter C, Hiroshi O (2013) GPU acceleration for FEM-based structural analysis. Arch Comput Methods Eng 20:05
104. JuliaSparse. MKLSparse.jl. https://github.com/JuliaSparse/MKLSparse.jl
105. JuliaAttic. CuSparse.jl. https://github.com/JuliaAttic/CUSPARSE.jl
106. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J (2015) Optimizing FPGA-based accelerator design for deep convolutional neural networks, pp 161–170
107. JuliaGPU. KernelAbstractions.jl. https://github.com/JuliaGPU/KernelAbstractions.jl
108. Cai Y, Cui XY, Li G, Liu W (2017) A parallel finite element procedure for contact-impact problems using edge-based smooth triangular element and GPU. Comput Phys Commun 225:12
109. Trescher D (2008) Development of an efficient 3-D CFD software to simulate and visualize the scavenging of a two-stroke engine. Arch Comput Methods Eng 15:03
110. Urick B, Sanders T, Hossain S, Zhang Y, Hughes T (2017) Review of patient-specific vascular modeling: template-based isogeometric framework and the case for CAD. Archiv Comput Methods Eng 26:1–24
111. Gao K, Mei G, Piccialli F, Cuomo S, Tu J, Huo Z (2020) Julia language in machine learning: algorithms, applications, and open issues. Comput Sci Rev 37:100254–100259
112. PYPL. PopularitY of programming language. https://pypl.github.io/PYPL.html
113. Tinnerholm J, Sjölund M, Pop A (2019) Towards introducing just-in-time compilation in a Modelica compiler, pp 11–19
114. Yao J (2020) SDOT: an explicit mesh-free detonation tracking package in 2D and 3D, vol 2272, p 070056
115. Ullah Z, Coombs W, Augarde C (2016) Parallel computations in nonlinear solid mechanics using adaptive finite element and meshless methods. Eng Comput 33:1161–1191
116. Sanchez-Gonzalez A, Godwin J, Pfaff T, Ying R, Leskovec J, Battaglia PW (2020) Learning to simulate complex physics with graph networks
117. Alexiadis A, Simmons M, Stamatopoulos K, Batchelor H, Moulitsas I (2020) The duality between particle methods and artificial neural networks. Sci Rep 10:10
118. Pestourie R, Mroueh Y, Nguyen TV, Das P, Johnson SG (2020) Active learning of deep surrogates for PDES: application to metasurface design. NPJ Comput Mater 6(1):1–7
119. Ye S, Zhang Z, Song X, Wang Y, Chen Y, Huang C (2020) A flow feature detection method for modeling pressure distribution around a cylinder in non-uniform flows by using a convolutional neural network. Sci Rep 10:03
120. Liu GR (2019) Two-way deepnets for real-time computations for both forward and inverse mechanics problems. Int J Comput Methods 16:05