Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2024-0027

Vol. **36**, No. 1, pp. 274-318
July 2024

COMPUTATIONAL SOFTWARE

AFEPack: A General-Purpose C++ Library for Numerical Solutions of Partial Differential Equations

Zhenning Cai[1], Yun Chen[2], Yana Di[3,4], Guanghui Hu[5,*], Ruo Li[6,7,*], Wenbin Liu[8], Heyu Wang[9], Fanyi Yang[10], Chengbao Yao[11] and Hongfei Zhan[12]

[1] *Department of Mathematics, National University of Singapore, Singapore.*
[2] *Shenyang National Laboratory for Material Science, Institute of Metal Research, Chinese Academy of Sciences, Shenyang, Liaoning Province, China.*
[3] *Research Center for Mathematics, Beijing Normal University, Zhuhai, China.*
[4] *Guangdong Key Laboratory of IRADS, BNU-HKBU United International College, Zhuhai, China.*
[5] *State Key Laboratory of Internet of Things for Smart City and Department of Mathematics, University of Macau, Macao SAR, China.*
[6] *CAPT, LMAM and School of Mathematical Sciences, Peking University, Beijing, China.*
[7] *Chongqing Research Institute of Big Data, Peking University, Chongqing, China.*
[8] *Division of Business and Management, BNU-HKBU United International College, Zhuhai, China.*
[9] *School of Mathematical Sciences, Zhejiang University, Hangzhou, Zhejiang Province, China.*
[10] *School of Mathematics, Sichuan University, Chengdu, China.*
[11] *Northwest Institute of Nuclear Technology, Xian, Shaanxi Province, China.*
[12] *School of Mathematical Sciences, Peking University, Beijing, China.*

Received 3 February 2024; Accepted (in revised version) 6 May 2024

Abstract. AFEPack is a general-purpose C++ library for numerical solutions of partial differential equations. With over two decades development, AFEPack has been successfully applied for scientific and engineering computational problems in a variety of areas such as computational fluid dynamics, electronic structure calculations, computational micromagnetics. In this paper, design philosophy of the library, algorithms and data structures used in the discretization of governing equations, numerical linear algebra for the discretized system, as well as the pre-processing and post-processing of the simulations, will be described systematically for the AFEPack. The realization of two main features of the library, i.e., adaptive mesh methods and parallel computing,

*Corresponding author. *Email addresses:* garyhu@um.edu.mo (G. Hu), rli@math.pku.edu.cn (R. Li)

will be introduced in detail. The potential of the library for large scale scientific/engineering problems would be demonstrated by several examples. The future works on developing the library will also be discussed.

AMS subject classifications: 65M08, 65M60, 65N08, 65N30, 65-04

Key words: AFEPack, C++ library, partial differential equations, scientific/engineering computing, software.

Program summary

Program title: AFEPack
Software licence: GPL 2.0
CiCP scientific software URL:
Developer's repository link: `http://dsec.pku.edu.cn/~rli/software.php`
Programming language(s): C, C++
Nature of problem: Numerical methods for partial differential equations have been playing a more and more important role in both scientific exploration and engineering applications, and the development of related numerical software is urged to catch up with the rapid development of hardware, as well as numerical algorithms.
Solution method: A C++ library entitled *AFEPack* is developed for the purpose. The package was designed originally based on finite element methods, and has been extended to finite volume methods, discontinuous Galerkin methods, spectral elements, etc. Features of the package include adaptive mesh techniques, and parallel computing. Based on AFEPack, several specific-purpose packages have been developed in computational fluid dynamics, electronic structure calculations, etc.

1 Introduction

Besides the theory and experiments, computational science has been becoming an indispensable methodology for science exploration. As an essential component of computational science, numerical solutions of partial differential equations (PDEs) have been playing an increasingly important role, not only in science exploration, but also in engineering applications, entertainments, etc.

Towards the scientific and engineering numerical simulations, there have been many mature software, covering a variety of research and application areas. For example, in computational fluid dynamics, Ansys Fluent [48], COMSOL [50], Autodesk CFD [49], etc., are popular commercial software, while OpenFOAM [35], SU2 [10], GeoClaw [4], etc., are popular open-source ones. In electronic structure calculations, there are commercial software such as Gaussian [51], Q-Chem [52], and Jaguar [53], and open-source software such as VASP [11], Quantum ESPRESSO [8], and ABINIT [1].

Different from aforementioned "purpose oriented" (specific-purpose) software, there have been a class of "method oriented" (general-purpose) ones in the market, in which the software is designed based on a class of specific methods for the education and academic research. For example, deal.II [3] and PHG [7] for finite element methods, MOOD [26] for finite volume methods, parDG [33] for discontinuous Galerkin methods, and DMSUITE [72] in MATLAB for spectral methods. These software bring scientists quality platforms for systematically studying the numerical methods, and for their research. Beyond that, these general-purpose software potentially serve as a starting point for the development of specific-purpose software. For example, DFT-FE [28] is a massively parallel adaptive finite element library in electronic structure calculations, which is built on top of the general-purpose finite element software deal.II [3]. GeoClaw [4] is a software for geophysical flow simulations, which is developed based on Clawpack [2] - a collection of finite volume methods for linear and nonlinear hyperbolic systems of conservation laws.

In this paper, we will introduce a competitive open-source general-purpose C++ library AFEPack (**A**daptive **F**inite **E**lement **Pack**age). The package was initially developed by Li and Liu for the study of moving mesh methods in solving PDEs, under finite element framework [57, 60, 61]. With the development over two decades, the functionality of the package has been greatly expanded, e.g., other grid based methods such as finite volume methods and discontinuous Galerkin methods are realized; solvers for both two dimensional and three dimensional PDEs can be coded in a unified framework with the help of templatization feature in C++ programming language; modules for adaptive mesh methods, including *r*-adaptivity and *h*-adaptivity, are available in the package; towards large scale simulations, both shared-memory and distributed-memory parallelism are supported; rich support for both the pre-possessing and the post-processing in solving PDEs is available from plenty of third-party software such as Gmsh [37] and EasyMesh [68] for mesh generation, and OpenDX [12] and ParaView [6] for the visualization of the numerical results; etc. All above features make AFEPack an ideal platform for studying numerical methods for PDEs, in which almost all aspects in numerical simulations, from mesh generation and discretization to solving linear system and results visualization, can be experienced.

Due to its general-purpose design and availability of the source code, AFEPack has attracted many scientists and engineers for the application of the software in their research and applications. For instances, in [31], Di et al. extended the moving mesh method to problems defined on a sphere through a perturbed harmonic mapping technique; in [30], a general moving mesh framework in 3D was established, and its application in simulating the mixture of multiphase flows was explored; in [70], the mesh sensitivity for numerical solutions of phase-field equations using *r*-adaptive finite element methods was investigated;in [29, 38, 39, 41, 43, 67], Hu et al. developed a finite volume framework for studying both steady and unsteady solutions of Euler equations; in [13–17], Bao et al. designed a finite element framework for the density functional theory, based on which both the ground state and the dynamics of a given electronic structure can be

numerically investigated; in [20, 46], by using the adaptive mesh module in AFEPack, Hu et al. numerically studied the porous medium flow problems; in [32, 47, 71], numerical simulations of the dendritic growth were studied based on *r*- and *h*-adaptive mesh methods, more results can be seen in [22] and [24]; in [63], an approximate solver for the hydro-elastoplastic solid material was proposed, in which a natural transformation between the fluid and solid for the phase transitions was designed; in [59, 64], a patch reconstruction technique has been proposed in the framework of discontinuous Galerkin methods, based on which a least square method was designed for a variety of problems; in [74,75], a finite element framework for computational micromagnetics was established and developed; in [19], an adaptive finite element DtN method has been developed for the three-dimensional acoustic scattering problem; in [22], the numerical methods developed based on AFEPack was applied in the research of drug delivery problem, etc. Furthermore, based on AFEPack, there have also been many examples towards developing "purpose oriented" software, such as AFEABIC [14, 16, 36, 76] in the electronic structure calculations, AFVM4CFD [38, 39, 41] in computational fluid dynamics, AFEMAG [74, 75] in computational micromagnetics.

In the rest of this paper, the design philosophy, the structure of the code, feature modules such as mesh adaptivity, as well as the application of AFEPack in a variety of areas, will be introduced and demonstrated by order.

2 A fundamental introduction to AFEPack based on solving a Poisson equation

It is well known that, with a given model PDE such as a Poisson equation, a standard procedure of numerically solving this PDE mainly consists of the generation of finite dimensional space for the approximate solution, the formation and solving of the system of linear equations, as well as the post-processing of numerical solutions.

More specifically, suppose that we have the following Poisson equation with a homogeneous Dirichlet boundary condition

$$\begin{cases} -\nabla^2 u(\mathbf{x}) = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ u(\mathbf{x}) = 0, & \mathbf{x} \in \partial\Omega, \end{cases} \tag{2.1}$$

where $\Omega \subset \mathbb{R}^d$ (d = 2 or 3), $f \in L_2(\Omega)$, and $u(\mathbf{x})$ is the unknown solution of the equation. In a finite element framework, the first step is to derive the variational form of the above equation, which is given by: To find $u \in V := H_0^1(\Omega)$, such that

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v d\mathbf{x} = \int_\Omega f v d\mathbf{x} =: (f,v), \quad \forall v \in V, \tag{2.2}$$

where $H_0^1(\Omega) := \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$ with $H^1(\Omega)$ a Hilbert space.

To solve the above equation (2.2), a finite dimensional trial space $V_h \subset V$ needs to be built, based on which a discrete variational form can be given by: To find $u_h \in V_h$, such that

$$a(u_h,v) = (f,v), \quad \forall v \in V_h, \tag{2.3}$$

where the subscript h stands for the characteristic of V_h, and u_h is the approximation of u. It is noted that (2.3) is equivalent to a system of linear equations

$$Lu_h = f, \tag{2.4}$$

from which the approximate solution u_h can be obtained for the post-processing. It is noted that the same notations u_h and f are used here for the convenience.

To code the above algorithm, many modules need to be handled well, such as the generation and management of mesh grids, the design of the template elements for the trail space, the design of the linear algebra module for efficiently storing and solving the system of linear equations, etc. A flowchart of the algorithm for solving a partial differential equation mentioned above is given in Fig. 1.

In following subsections, each module will be introduced in detail.

2.1 The partition of the domain

Basically, two aspects need to be handled well in the mesh module, i.e., how to generate a quality partition of the domain, and how to manage it efficiently during the simulation. Mesh generation is a highly nontrivial topic in numerically solving PDEs, and there have been many well known software available for the purpose, such as Gmsh [37], Netgen [5], Tetgen [69]. In AFEPack, there is no such a module for the mesh generation currently. Instead, interface functions are provided in AFEPack for those popular mesh generators, so that the mesh data generated by other software can be read and transferred into the internal mesh file natively supported by AFEPack.

Before introducing the format of the data of the internal mesh in AFEPack, let us review a common format used in the popular mesh generators. For example, in Fig. 2, a tetrahedral mesh consisting of two tetrahedrons is demonstrated. To uniquely determine such a mesh, a simple description given in Table 1 works.

It is noted that for a mesh consisting of the simplex geometries, the format used in Table 1 should be the simplest one, since besides the necessary distribution of the grid points, the topology of the mesh can be deduced clearly from the description of the given highest dimensional information. However, from the efficiency point of view, this format is far from ideal due to the missing of the lower dimensional information, which are useful in designing numerical algorithms.

A hierarchical approach is adopted in AFEPack to describe a given geometry, dimension by dimension. The design is based on the following observations from Fig. 3, in which a decomposition of a tetrahedron from Fig. 2 is demonstrated. It is noted that such a decomposition is based on a unified criterion, i.e., a geometry (nD, $n = 0,1,2,3$) can be

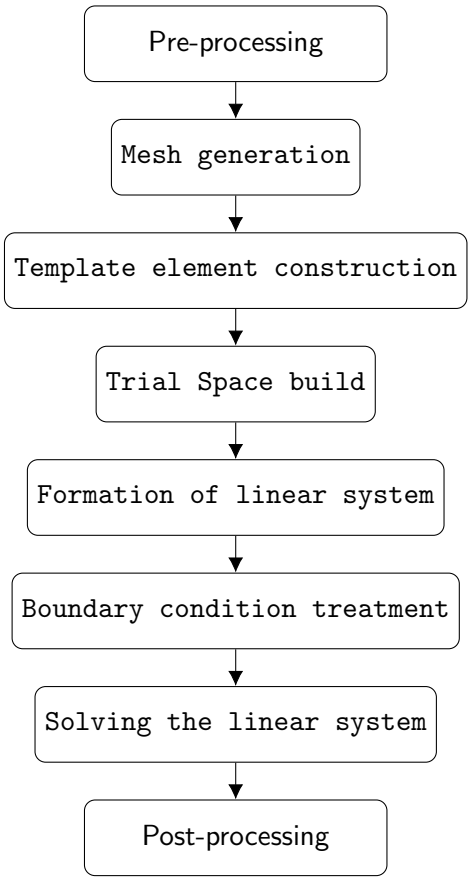

Figure 1: A flowchart for solving a partial differential equation.

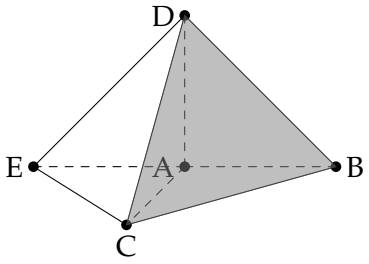

Figure 2: A tetrahedral mesh with 2 tetrahedral elements.

described by its vertices (0D geometries) and boundaries ($(n-1)$D geometries). For instance, in Fig. 3, the 3D tetrahedron ABCD consists of four vertices A, B, C, and D, and four boundaries (triangles) ΔABC, ΔABD, ΔACD, and ΔBCD. For the 2D triangle ΔABC, it consists of three vertices A, B, and C, and three boundaries (line segments) AB, AC, BC. For the 1D line segment AB, it consists of two vertices A and B, and two boundaries

Table 1: Format of mesh data.

5					#There are 5 points in the mesh
0:	0	0	0		#The index, and the coordinate of the point A
1:	1	0	0		#The index, and the coordinate of the point B
2:	0	1	0		#The index, and the coordinate of the point C
3:	0	0	1		#The index, and the coordinate of the point D
4:	-1	0	0		#The index, and the coordinate of the point E
2					#There are 2 tetrahedrons in the mesh
0:	0	1	2	3	#The index the tetrahedron, and the index of its each vertex.
1:	0	2	3	4	

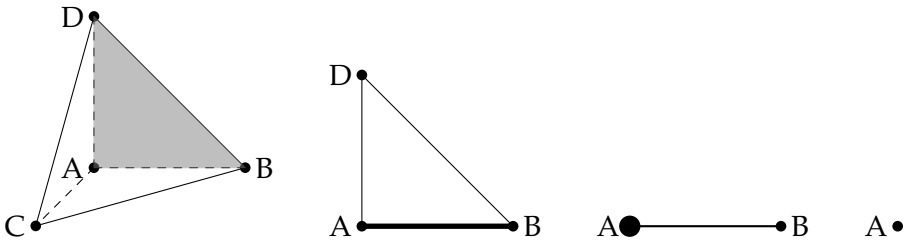

Figure 3: From left to right: a 3D geometry ABCD, a 2D geometry ABD, a 1D geometry AB, and a 0D geometry A.

(points) A and B. Finally, the point A consists one vertex A, and one boundary (point) A. From above descriptions, some redundant information can be seen when we describe 1D and 0D geometries. However, it should be pointed out that with above descriptions, a unified data structure can be designed for managing the geometry in the code. It is noted that, such a design potentially benefits the study of problems in fracture mechanics, in which the crack can appear causing the topological change of the domain.

In AFEPack, an internal data format for the mesh is designed according to above descriptions, based on which the mesh shown in Fig. 2 is given in Table 2. With such a data format, the mesh can be decomposed completely by a dimension by dimension approach, based on which both the flexibility and the efficiency of the algorithm in designing numerical methods for PDEs can be benefited. Although compared with the simple format shown in Table 1 more information have to be stored in this internal format in AFEPack, it is noted that in solving PDEs with finite volume methods, discontinuous Galerkin methods, etc., the information of the mesh in lower dimension is necessary, which means that such information needs to be calculated anyway if the simple format is employed.

Four main classes in AFEPack in Listing 1 are defined to manage the mesh data. It is worth mentioning that due to a unified description for geometries in all dimensions, classes *Geometry* and *GeometryBM* needed not to be templatized.

Table 2: A complete *mesh* data for the mesh shown in Fig. 2 in AFEPack.

5					...				
0	0	0		6				
1	0	0		2	0	4		
0	1	0		2	0	4		
0	0	1		0				
-1	0	0		7				
					2	2	4		
5					2	2	4		
0					0				
1	0				8				
1	0				2	3	4		
0					2	3	4		
1					0				
1	1								
1	1				7				
0					0				
2					3	0	1	2	
1	2				3	3	1	0	
1	2				0				
0					1				
3					3	0	1	3	
1	3				3	4	2	0	
1	3				0				
0					2				
4					3	1	2	3	
1	4				3	5	4	3	
1	4				0				
0					3				
					3	0	2	4	
9					3	7	6	1	
0					0				
2	0	1		4				
2	0	1		3	0	3	4	
0					3	8	6	2	
1					0				
2	0	2		5				
2	0	2		3	2	3	4	
0					3	8	7	5	
2					0				
2	0	3		6				
2	0	3		3	0	2	3	
0					3	5	2	1	
3					0				
2	1	2						
2	1	2		2				
0					0				
4					4	0	1	2	3
2	1	3		4	2	6	1	0
2	1	3		0				
0					1				
5					4	0	2	3	4
2	2	3		4	5	4	3	6
2	2	3		0				
0									
...									

```
template <int DIM> class Point;
class Geometry;
class GeometryBM;
template <int DIM, int DOW=DIM> class Mesh;
```

Listing 1: Four main classes in AFEPack for managing mesh data.

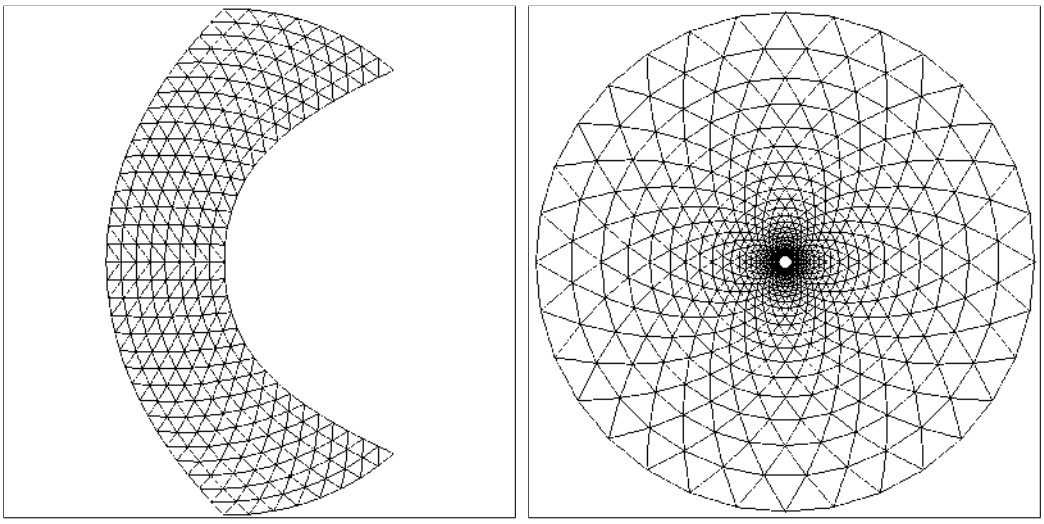

Figure 4: Well designed meshes. Left: for ringleb flow simulation; Right: for circular flow simulation.

Although the mesh generation is an essential module in developing a software for numerically solving PDEs, the task is highly nontrivial. Consequently, except for well designed mesh for certain case, in AFEPack we mainly resort to the third party software for the purpose currently by developing interfaces. For example, DBMesh [34] and EasyMesh [68] are supported by AFEPack in generating two dimensional triangulation of the domain, while it is Gmsh [37] for three dimensional cases. For example, in Fig. 4, two well designed two dimensional meshes are shown for the numerical simulations of Euler equations, while in Fig. 5, meshes generated by Easymesh (top two) and Gmsh (bottom two) are demonstrated. It is noted that for the visualization, the open source software OpenDX [12] is used, which will be introduced later.

Remark 2.1. For the mesh shown in Fig. 2 in 3D, there are 5 modules in the mesh data shown on the left side. The first module contains the number of grid points as well as the coordinate for each point. In each one of the rest four modules, a unified format is adopted for describing geometries in 0D, 1D, 2D, and 3D. More specifically, besides the number of geometries, the information of each geometry includes the index of the geometry, the number of vertex/vertices as well as the index of each vertex, the number of boundary/boundaries as well as the index of each boundary, and the boundary mark of

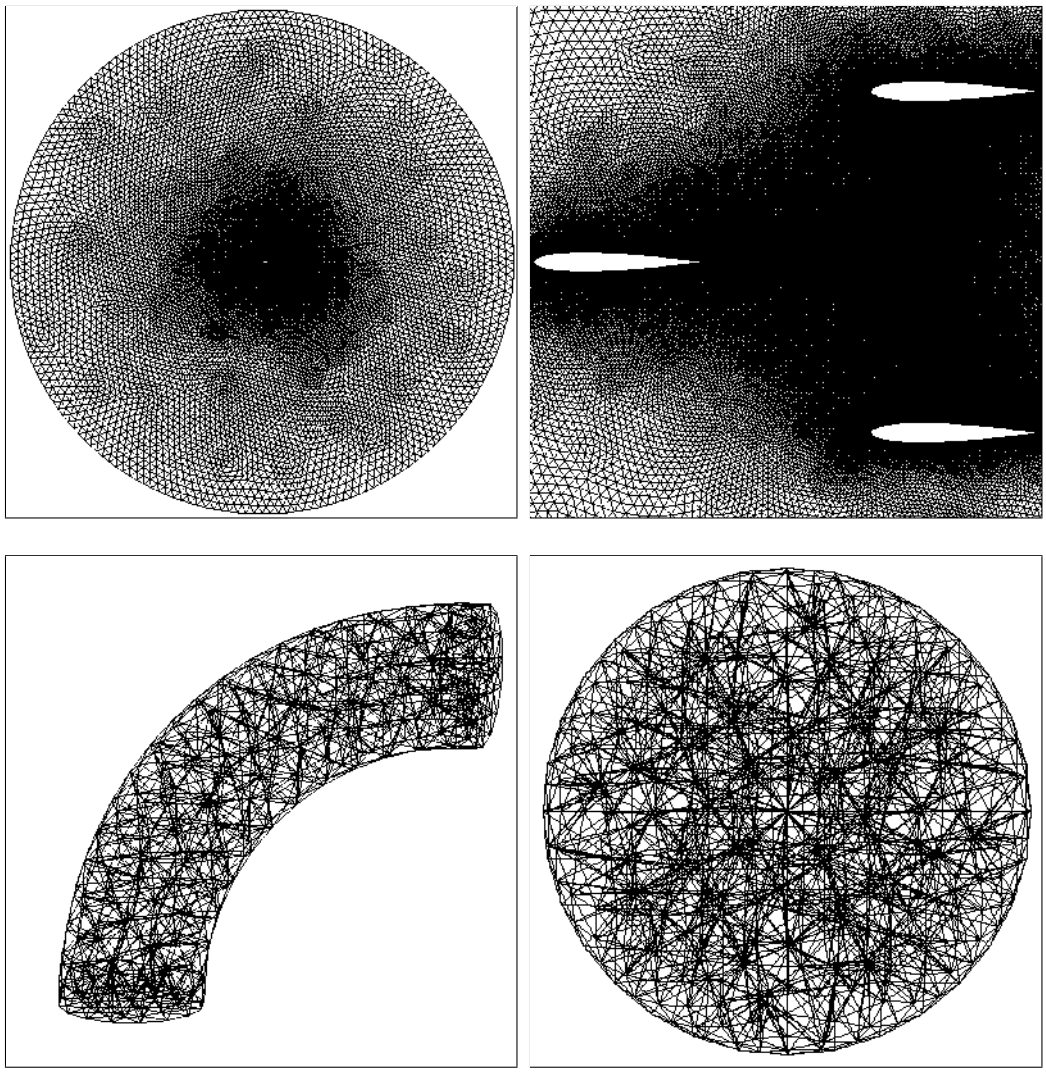

Figure 5: Meshes generated from third party software. Top: the meshes generated by EasyMesh for a domain including three naca0012 airfoils (left: the whole mesh, right: mesh grids around airfoils); Bottom: the meshes generated by Gmsh for a twist cylinder (left) and a ball(right).

the geometry. It is clear that by building the mesh in a dimension by dimension strategy, all information of a given mesh can be resolved completely by a recursive approach.

2.2 The generation of the approximate space

A finite dimensional space V_h needs to be built for obtaining an approximation solution u_h through solving (2.3). For the convenience of the following discussion, let us use $\mathcal{T}^h := \{K_i^h\}_{i=0}^{N_\mathcal{T}}$ to denote the mesh, where K_i^h denotes the i-th geometry element in the

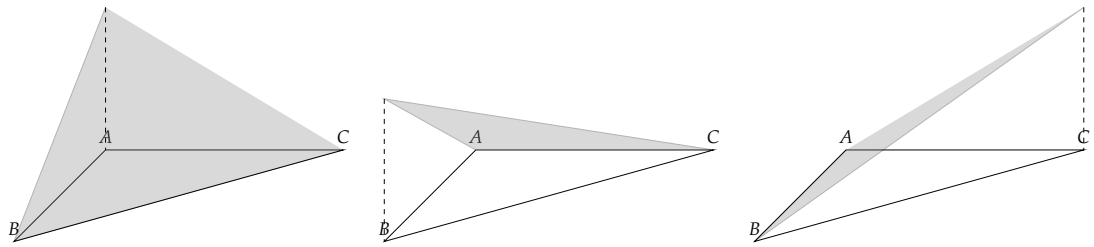

Figure 6: Shape functions of a linear polynomial space defined on a triangle.

mesh, and $N_{\mathcal{T}^h}$ denotes the total number of the geometry elements of the mesh.

We restrict ourselves to the finite dimensional space V_h with piece-wise linear approximation, for the introduction of related modules in AFEPack. Following the definition given by Ciarlet in [25], in each geometrical element K_i, a finite element can be defined by

Definition 2.1. The 3-tuple $(K,\mathcal{P},\mathcal{N})$ is called a finite element, with

- K: a closed and bounded domain with nonempty interior and piece-wise smooth boundary;
- $\mathcal{P}$: a finite dimensional space of shape functions;
- $\mathcal{N}=\{N_1,N_2,N_3,\cdots,N_k\}$: a basis for the space $\mathcal{P}$.

A simple case for the above definition is the finite element defined on a simplex domain (triangle in two dimensional case, or a tetrahedron in three dimensional case), with linear polynomials as shape functions. In this case, the vertices of the simplex can be chosen as the nodal variables to build a conforming finite element space. Fig. 6 shows shape functions defined in a triangle element. For a given mesh $\mathcal{T}$, a finite element space V_h can be constructed by $V_h:=\{(K_i^h,\mathcal{P},\mathcal{N})\}_{i=1}^{N_\mathcal{T}}$.

In coding a finite element method, a classical approach for building a finite element space is to resort to a template element, for handling the basis functions and the setup of the numerical integration in each finite element. The idea is that the information in each finite element is obtained by the information in a template finite element and a Jacobian transformation between two elements. In AFEPack, this strategy is realized in a well designed manner. More specifically, a folder entitled *template* is created in AFEPack, in which several sub-folders entitled, for example *triangle* and *tetrahedron*, are included. In every sub-folder, four kinds of files are provided for the construction of the template element, i.e., the geometry information, the coordinate transformation, the distribution of degrees of freedom, and basis functions. For instance, in the folder *triangle*, the geometry information of a template triangle element is given in a file entitled *triangle.tmp_geo*, in which the geometry of the template triangle and the Gauss quadrature information are given. In files *triangle.crd_trs* and *triangle.crd_trs.c*, the affine maps between the template

element and the finite element are defined. In a sub-folder entitled *triangle.1*, the linear polynomial approximation is defined, for which the total number of degrees of freedom, as well as the location of each degree of freedom is described in *triangle.1.tmp_dof*, and the related basis functions are defined and managed in *triangle.1.bas_fun* and *triangle.1.bas_fun.c*. Similarly, the quadratic polynomial approximation is defined in the sub-folder *triangle.2*. So far, triangle and quadrilateral templates in two dimension, and tetrahedron and hexahedron templates in three dimension are available in AFEPack.

Remark 2.2. In solving stationary nonlinear problems and time-dependent problems, the system of linear equations needs to be reformed for many times. In AFEPack, two template structures shown in Listing 2 are defined to store information of the finite element for the reuse, as long as the finite element space keeps unchanged. Furthermore, quantities such as the least square system in reconstructing the variation of the solution locally can also be stored in a class inherited from the template class *ElementAdditionalData*.

Remark 2.3. Recently, Zhan and Hu has successfully introduced a novel realization of a tetrahedral spectral element method based on AFEPack, based on which a solver for the Kohn-Sham equation was proposed. This tetrahedral spectral element method can serve as a high order method for general PDEs. Please refer to [77] for the introduction of the method, and [9] for the source code.

```
template <int DIM>
struct GeometryAdditionalData
{
  public:
  int n_quadrature_point;
  double volume;
  Point<DIM> bc;
  std::vector<double> Jxw;
  std::vector<Point<DIM> > q_point;
};

template <typename value_type, int DIM>
struct ElementAdditionalData : public GeometryAdditionalData<DIM>
{
  public:
  std::vector<std::vector<value_type> > basis_value;
  std::vector<std::vector<std::vector<value_type> > > basis_gradient;
};
```

Listing 2: Two template structures for reusing information.

2.3 The formation of the system of linear equations

To generate the stiff matrix in (2.4), the integral $\int_\Omega \nabla u_h \cdot \nabla v d\mathbf{x}$ in (2.3) will be calculated in an element by element strategy, i.e.,

$$\int_\Omega \nabla u_h \cdot \nabla v d\mathbf{x} = \sum_{K_k^h \in \mathcal{T}^h} \int_{K_k^h} \nabla u_h \cdot \nabla v d\mathbf{x}. \tag{2.5}$$

Following our above discussion, for an element-wise linear polynomial approximation in a triangle element, the approximation u_h in the k-th element K_k^h can be expressed by

$$u_h|_{K_k^h} = u_h^{(0),l}\phi_k^{(0)} + u_h^{(1),m}\phi_k^{(1)} + u_h^{(2),n}\phi_k^{(2)},$$

where $\phi_k^{(i)}$, $i = 0,1,2$ are three basis functions in the element K_k^h, while $u_h^{(0),l}$ denotes the 0-th nodal variable in the element K_k^h. It is noted that the superscript l here denotes the global index of this nodal variable, which is used to deliver the entry from the element stiff matrix to the global stiff matrix.

By taking test functions the same to trial functions, in the element K_k^h we finally get the following element stiff matrix

$$\begin{matrix} \quad l & \quad m & \quad n \end{matrix}$$
$$\begin{bmatrix} \int_{K_k^h} \nabla\phi_k^{(0)} \cdot \nabla\phi_k^{(0)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(0)} \cdot \nabla\phi_k^{(1)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(0)} \cdot \nabla\phi_k^{(2)} d\mathbf{x} \\ \int_{K_k^h} \nabla\phi_k^{(1)} \cdot \nabla\phi_k^{(0)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(1)} \cdot \nabla\phi_k^{(1)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(1)} \cdot \nabla\phi_k^{(2)} d\mathbf{x} \\ \int_{K_k^h} \nabla\phi_k^{(2)} \cdot \nabla\phi_k^{(0)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(2)} \cdot \nabla\phi_k^{(1)} d\mathbf{x}, & \int_{K_k^h} \nabla\phi_k^{(2)} \cdot \nabla\phi_k^{(2)} d\mathbf{x} \end{bmatrix} \begin{matrix} l \\ m \\ n \end{matrix},$$

then each entry of the above matrix will be added into the entry of the global matrix according to the indices l,m and n.

In the above element stiff matrix, the numerical integration is needed for the calculation of each entry, which can be done trivially in AFEPack with the well designed finite element, and template class *ElementAdditionalData* shown in Listing 2.

In AFEPack, the above process is realized in a template class *BilinearOperator*, in which a member function *getElementMatrix* is defined for the calculation of the element stiff matrix. It is noted that such a member function is declared with the keyword *virtual*, which allows users to redefine this function to handle their models.

Due to the locality of basis functions defined in each finite element, besides the diagonal entry, there would be only several nonzero entries in each row of the global stiff matrix. A typical distribution of nonzero entries in a stiff matrix is shown in Fig. 7. In existing software, a popular choice for storing and operating such sparse matrices is the compressed row storage (CRS) format, in which the total number of nonzeros in each row, indices of columns of nonzeros in each row, as well as those nonzero entries, are stored in three vectors, as a simple description of this format, please refer to Fig. 8 for one

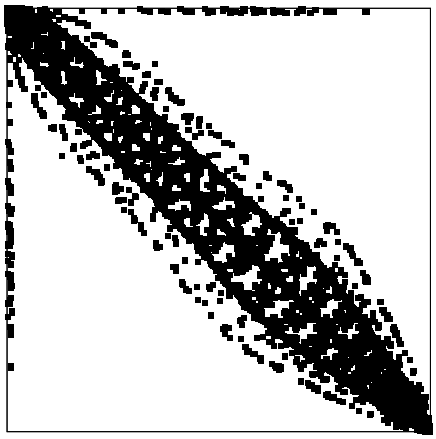

Figure 7: The pattern of nonzero entries in a stiff matrix. Each black square stands for a nonzero entry.

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

$$[0,2,5,8,11,13]$$

$$[0,1,0,1,2,1,2,3,2,3,4,3,4]$$

$$[-2,1,1,-2,1,1,-2,1,1,-2,1,1,-2]$$

Figure 8: The format of row compressed storage. Top: a 5×5 matrix; Bottom: three one dimensional vectors in CRS for the above matrix.

simple example. With CRS format, the matrix vector multiplication can be implemented trivially, which benefits the development of many classical solvers for the system of linear equations and eigenvalue problems, such as conjugate gradient methods, multigrid methods, Krylov subspace methods, Lanczos methods, Arnoldi methods.

Due to its well developed linear algebra module, we developed the linear algebra module in AFEPack based on the template classes *SparseMatrix* and *Vector* from deal.II. Hence, besides the fundamental operations such as matrix vector multiplication can be used directly, well-developed solvers from software such as PETSc and Trilinos can also

be used through API provided in deal.II. In addition, in AFEPack, a module of the linear algebra is under developed, in which calculations for both dense matrix and sparse matrix are supported. It is noted that the application program interface (API) for the Linear Algebra PACKage (LAPACK) is provided, so that the functions from LAPACK can be used to handle the operation for the dense matrix. The operation for the sparse matrix based on CRS format is also supported internally. Some preliminary results show that such a module works very well. The module will be available in the forthcoming version of the library.

2.4 Boundary conditions

Boundary conditions are essential components in a PDE model making the model well-posed. There are three basic kinds of boundary conditions, i.e., the Dirichlet boundary condition specifying the value of the unknown variable, the Neumann boundary condition specifying the derivative of the unknown variable, as well as the Robin boundary condition which is a combination of previous two. So far, the Dirichlet boundary condition can be handled automatically in AFEPack, by using the code shown in Listing 3, in which the instantiation *boundary* of the template class *BoundaryFunction* is initialized as a Dirichlet boundary, with the boundary mark 1 and the evaluation given by the function u, while the instantiation *boundary_admin* of the template class *BoundaryConditionAdmin* is initialized with the finite element space *fem_space*. After the *boundary* is added in the *boundary_admin*, the function *apply* is called to revise the system of linear equations *stiff_matrix* * *solution* = *right_hand_side* according to the given Dirichlet boundary conditions. Users can change the second and the third parameters in the declaration of the instantiation *boundary* to make the code fitting their own models.

```
1 BoundaryFunction<double,DIM> boundary(BoundaryConditionInfo::DIRICHLET, 1, &u);
2 BoundaryConditionAdmin<double,DIM> boundary_admin(fem_space);
3 boundary_admin.add(boundary);
4 boundary_admin.apply(stiff_matrix, solution, right_hand_side);
```

Listing 3: Code in AFEPack for handling Dirichlet boundary conditions.

The study on boundary conditions is very important in both theoretical and numerical investigation for PDEs. The treatment of boundary conditions could be quite complicated, due to the complex geometry of the domain, physical constraints from practical problems, etc. Particularly, in FEM discretization, Dirichlet boundary conditions requires the modification of both matrix and right-hand side after assembling process. This boundary condition can be imposed following the procedure in Listing 3. In contrast, the Neumann and Robin boundary conditions could be integrated into the weak form and handle in the assembling process, where fruitful functions are provided in the package to operate matrices and vectors for implementing these boundary conditions.

2.5 Solving the system

Finally, we arrive at solving the system of linear equations (2.4). For solving the system derived from the Poisson equation, there is a well developed algebraic multigrid (AMG) solver in the package. The restriction and prolongation operations in the AMG solver are designed following [21, 27], while the classical Gauss-Seidel iteration is used as the smoother.

In the package, the code shown in Listing 4 is used for solving the system of linear equations, in which an instantiation *solver* of the class *AMGSolver* is declared, and then initialized with the matrix *stiff_matrix*. The function *lazyReinit* is used here to construct the restriction and prolongation operators, as well as a series of coarsened systems of linear equations. It is noted that the algorithm in this function is a simple one, which only depends on the sparsity pattern of the matrix. In the package, there is also a function called *reinit* in the class *AMGSolver* for the same purpose, with more complicated algorithm following [21,27]. As a comparison, the solver initialized by the function *reinit* will cost more time for the initialization, but it will solve the system faster, than the one initialized by the function *lazyReinit*. In the last step, the function *solve* is called to obtain the solution of the system of linear equations. In this function, the first two parameters are the solution vector and right hand side vector, respectively. The third one is a user defined tolerance, so that the implementation of this function will be terminated if the norm of the residual vector caused by the solution is less than this tolerance. The last parameter denotes the maximum number of the AMG iteration steps. It is used to terminate the implementation when the norm of the residual vector is still greater than the given tolerance, but the maximum iteration number is reached.

```
1 AMGSolver solver;
2 solver.lazyReinit(stiff_matrix);
3 solver.solve(solution, right_hand_side, 1.0e−08, 200);
```

Listing 4: Code for solving the system of linear equations with an AMG solver.

It is noted that besides being a solver for the system of linear equations, the AMG method in the package is also designed as a preconditioner for other solvers, such as preconditioned conjugate gradient method. A class *AMGPreconditioner*, inherited from the class *AMGSolver*, is provided in the package for the purpose.

2.6 The visualization of the numerical result

The visualization of numerical results plays an important role in the scientific computing, which helps researchers to understand the structure and/or dynamics of the physical process described by the governing equation. There have been many mature visualization tools in the market, for instance, the open source software such as OpenDX [12], ParaView [6], and the commercial software such as Tecplot [54].

The above mentioned tools are all supported by the package, which can be found in the template class *FEMFunction*. One example of outputting numerical results is given in

Listing 5, in which the function *writeOpenDXData* is called to output the file supported by the OpenDX, then the L_2 error of the numerical solution is calculated by using the function *L2Error* defined in the namespace *Functional* in the package. It is noted that the second parameter in the function is the exact solution of the PDE, while the third parameter is the algebraic accuracy used in the numerical integration.

```
1 solution.writeOpenDXData("u.dx");
2 double error = Functional::L2Error(solution, FunctionFunction<double>(&u), 3);
3 std::cout << "L2 error = " << error << std::endl;
```

Listing 5: Code for outputting numerical results.

To show the effectiveness of the package, we demonstrate the code of solving a Poisson equation in Example 2.1.

Example 2.1. A well-posed two dimensional Poisson equation is given by

$$\begin{cases} -\Delta u(\mathbf{x}) = f(\mathbf{x}), & \text{in } \Omega, \\ u(\mathbf{x}) = g(\mathbf{x}), & \text{on } \partial\Omega, \end{cases} \tag{2.6}$$

where $\Omega = (0,1)\times(0,1)$. Using the exact solution $u(\mathbf{x}) = \sin(\pi x)\cos(3\pi y)$ with $\mathbf{x} := [x,y]$, two functions $f(\mathbf{x})$ and $g(\mathbf{x})$ would be $f(\mathbf{x}) = 10\pi^2 u(\mathbf{x})$ and $g(\mathbf{x}) = u(\mathbf{x})|_{\partial\Omega}$, respectively.

A complete code for solving the above equation is given in Appendix B, in which the code is organized following the flowchart shown in Fig. 1. In Fig. 9, the numerical solution obtained with 1907201 degrees of freedom is shown by using OpenDX, while in Table 3 the desired convergence rate of numerical solutions obtained from a series of

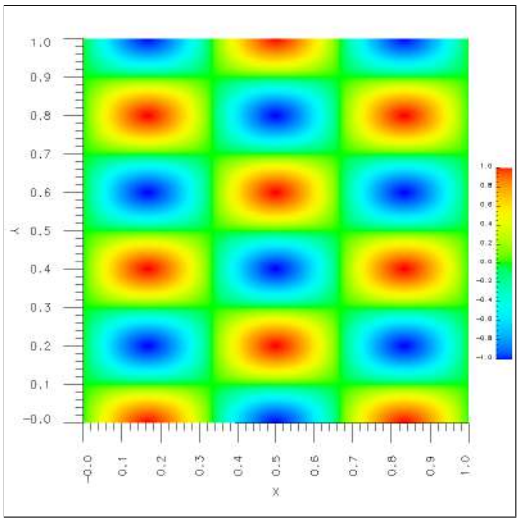

Figure 9: The numerical result of the Poisson equation in Example 2.1, shown by OpenDX.

Table 3: Convergence of numerical solutions on a series of successively refined meshes.

No. of DOFs	L_2 error	Conv. order
506	2.68e-02	-
1941	6.81e-03	1.98
7601	1.71e-03	1.99
30081	4.28e-04	2.00
119681	1.07e-04	2.00
477441	2.68e-05	2.00
1907201	6.70e-06	2.00
7623681	1.67e-06	2.00

successively refined meshes is shown successfully.Moreover, an example code for solving diffusion equation is provided in Appendix C for further investigation of numerical simulations via AFEPack.

So far the design of the package has been briefly introduced based on a model PDE. They are the mesh adaptivity and parallel computing modules in the package which make the AFEPack attractive in both scientific and engineering computing. In following two sections, the design and applications of two modules will be introduced in detail, respectively.

3 The mesh adaptivity

In many cases, it can be observed that the solution becomes highly nontrivial, only locally somewhere in the domain. Towards an efficient simulation of such problems, a predesigned nonuniform mesh could partially resolve the issue. However, it generally can not be used for temporal problems due to unknown dynamics of the solution.

Mesh adaptivity is an effective strategy to handle the above issue, for both stationary nonlinear and temporal problems. The idea is to dynamically adjust the mesh grids (either by the local refinement of mesh grids, or by moving the mesh grids) according to certain criterion, to guarantee that the solution can always be represented in a quality finite dimensional space. Based on the adjustment strategies, there have been two classical approaches for the mesh adaptivity, i.e.,

- The *h*-refinement: In this approach, geometry elements in a mesh would be locally refined or coarsened. The feature of this approach is that the position of each existing grid point would not change during the mesh adaptivity process. However, the topology of the mesh grids will change with the implementation of the local refinement.
- The *r*-refinement (moving mesh): In this approach, grid points in a mesh would be redistributed based on certain criterion. The feature is that the total number of grid

points in a mesh, as well as the topology of the mesh would not be changed, during the movement of grid points.

Three main issues need to be resolved well in designing a quality adaptive mesh method, i.e., the quality error indication for guiding the remeshing, the efficient implementation of the remeshing, and the quality representation of the numerical solution on the new mesh.

In the package, both the h-refinement and the r-refinement approaches are supported. In following subsections, the design of two approaches will be introduced in detail, respectively.

3.1 On h-adaptive mesh methods

In the package, two concepts, i.e., the hierarchy geometry tree (HGT) and the geometry forest (GF), are proposed for managing the mesh, based on which the implementation of the remeshing and the solution representation can be resolved well.

To understand the philosophy behind the HGT and the GF, let us imagine that we are taking a bird's eye view of an island, which is completely covered by a number of trees. With an assumption that each tree exclusively covers a part of this island, we actually have a coarse partition of the island. Further, with an assumption that the region occupied by a tree is completely filled by its non-overlapping leaves, a fine partition of the island is available. With above assumptions, the purpose of h-adaptive mesh methods can be understood as that, to control the growth of those trees so that less leaves (geometry elements) can be used for better covering the island, according to its geography (exact solution).

In the package, tree data structures are employed for the realization of h-adaptive mesh methods to realize the above idea. In the following, we restrict the discussion into the category of simplex geometries, i.e., triangle elements in two dimensional cases and tetrahedral elements in three dimensional cases. Let us take the two dimensional case as an example to describe the detail.

For two dimensional cases, a quadtree is employed in the package to manage the mesh refinement. A quadtree is a tree data structure, in which each internal node has exactly four children. This property perfectly matches the regular refinement of a triangle that by connecting three middle points of edges of the triangle, four subtriangles are generated. In Fig. 10, for a square domain $ABCD$, a process of the mesh refinement as well as the growth of corresponding quadtrees is demonstrated. Initially, there are two triangles, i.e., ΔABC and ΔACD in the mesh (Fig. 10, top left), which correspond to two nodes in the top level of two quadtrees. Then, five middle points, i.e., E,F,G,H and I, are introduced, based on which the mesh is uniformly refined, and there are totally eight triangles in the refined mesh (Fig. 10, top middle). These eight triangles correspond to eight nodes in the second level of two quadtrees. Finally, among these eight triangles, it is found that triangles ΔAIH and ΔBFE need to be further refined. Hence, the middle points J,K,L and M,N,O are introduced for the purpose for two triangles, respectively

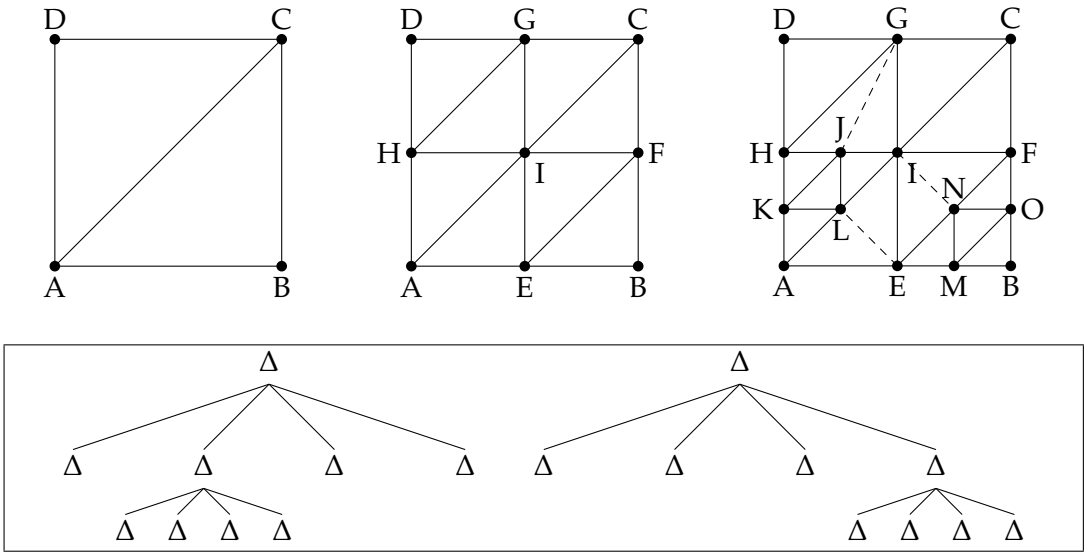

Figure 10: The correspondence between the mesh refinement and the growth of the geometry tree. Bottom: two geometry trees with three levels; Top: the left and the middle meshes consist of nodes (geometries) locating at the first and the second levels in below geometry trees, respectively, while the right mesh consists of all leaf nodes of geometry trees.

(Fig. 10, top right). In two quadtrees, four children are generated for each corresponding node, and the collection of all leaf nodes corresponds to the nonuniform mesh.

Before the nonuniform mesh shown in Fig. 10 (top right) can be used in building the finite element space, the issue related to the hanging point needs to be resolved well. A natural approach is to split the triangle with a hanging point into two triangles. For example, by connecting the vertex I and the hanging point N in Fig. 10, two new triangles, ΔENI and ΔFIN, would be generated, and the hanging point issue is well resolved. However, such a behavior is not consistent with property of a quadtree that every node would generate four children. In the package, a concept called twin-triangle is introduced to handle the hanging point. The idea behind is to treat the hanging point as a vertex of the geometry, by designing a basis function for it. It is done by logically splitting the triangle, e.g., ΔEFI, into two triangles, e.g., ΔENI and ΔFIN. Then basis functions are designed for two triangles, respectively. Finally, basis functions from two triangles are organized well to express the finite element for the twin-triangle ΔEFI.

To realize the *h*-adaptive mesh method, in the package two template classes are defined, i.e., *HGeometryTree, IrregularMesh*. In *HGeometryTree*, a template class *HGeometry* is defined to help to build the hierarchy tree, while in *IrregularMesh*, a template class *HElement* is defined to construct the irregular mesh. It is noted that to facilitate the visiting of nodes in the hierarchy tree, several iterators are provided in these template classes. For example, in *IrregularMesh*, the following two kinds of iterators are provided.

```
RootIterator beginRootElement(); RootIterator endRootElement()
ActiveIterator beginActiveElement(); ActiveIterator endActiveElement();
```

Listing 6: Iterators provided in *IrregularMesh*.

A typical code snippet for implementing the *h*-adaptivity in the package is given in Listing 7. The process can be briefly described as follows. First of all, a vector called *indicator* is declared and initialized using current mesh. Such a vector is filled by certain error estimation. Then an instantiation of the template class *MeshAdaptor*, e.g., *mesh_adaptor*, is declared and initialized by the current irregular mesh. The parameter *convergenceOrder* indicates the convergence order of the numerical method, the parameter *refineStep* indicates the times allowed for implementing the local refinement of mesh grids, while the parameter *tolerance* indicates the tolerance for implementing the refinement or coarsening of geometry elements. Please refer to [58] for more details of above parameters. The member function *semiregularize* provided in the template class *IrregularMesh* is designed to prevent the situation that more than one hanging point appeared on a single edge of a triangle, and that hanging points appear on more than one edge in a single triangle. In either case, besides the twin-triangle element, more special triangle elements need to be designed, which would introduce the accuracy and stability issues due to the possible small acute angle. In the member function *semiregularize*, the above issue is resolved by further regular refinement of the geometry element, i.e., once more than one hanging point is detected in a single triangle, the triangle will be regularly refined one time. Please refer to Fig. 11 for above descriptions. Such an operation will be implemented until that every triangle in the mesh has at most one hanging point. Although more triangle elements will be generated during the process, the quality of the mesh can be guaranteed, which benefits the numerical accuracy and stability. Finally, in the member function *regularize* of the template class *IrregularMesh*, all leaf nodes in hierarchy trees will be collected and all vertices, edges, as well as triangles will be indexed, so that a regular mesh can be obtained through the member function *regularMesh* of the template class *IrregularMesh*. It is noted that a finite element space now can be built based on this regular mesh.

```
Indicator<DIM> indicator(regular_mesh);
... /// the generation of indicator
MeshAdaptor<DIM> mesh_adaptor(irregular_mesh);
mesh_adaptor.convergenceOrder() = 1.;
mesh_adaptor.refineStep() = 1;
mesh_adaptor.setIndicator(indicator);
mesh_adaptor.tolerance() = 5.0e−03;
mesh_adaptor.adapt();
irregular_mesh.semiregularize();
irregular_mesh.regularize(false);
RegularMesh<DIM>& regular_mesh = irregular_mesh.regularMesh();
... /// the build of new finite element space
```

Listing 7: A code snippet for *h*-adaptivity.

In the second line of Listing 7, the code for generating the vector *indicator* is needed, in which certain error estimation method would play an important role. Two kinds of meth-

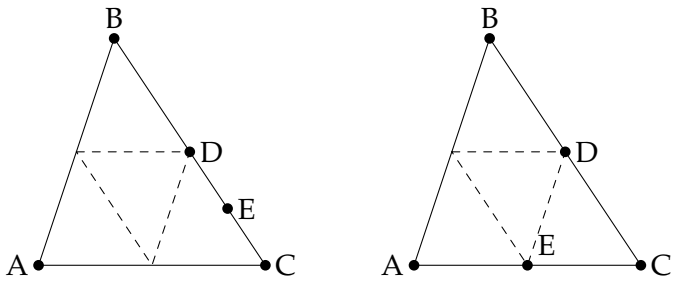

Figure 11: Two typical cases that there are more than one hanging points in a single triangle ΔABC.

ods are popular in the market. One is the so-called feature based methods, in which the error indicator is designed from the experience of the user. For example, to capture the shock structure in a simulation in compressible fluid dynamics, the gradient of the pressure based on numerical solutions would be a good choice for the purpose. The other one is the so-called *a posteriori* error estimation, a technique for generating reliable error information based on numerical solutions. Classical methods include residual-based error estimation, reconstruction-based error estimation, dual weighted residual method, etc. In the package, the error estimation module is designed independently, which provides a convenient platform to researchers for their study in error estimation.

3.2 On *r*-adaptive mesh methods

Besides *h*-adaptive mesh methods mentioned in previous subsection, in the package, *r*-adaptive mesh methods are also provided. The feature of *r*-adaptive mesh methods is that the total amount of the mesh grids as well as the topology of the mesh grids would not change during the simulation, only the distribution of those grid points will be adjusted according to certain criterion.

The realization of *r*-adaptive mesh methods in the package is based on a class of harmonic maps [60,61], which can be briefly summarized as follows.

First of all, for two compact Riemann manifolds Ω and Ω_c, with d_{ij} and $r_{\alpha\beta}$ two metric tensors in certain local coordinates $\mathbf{x}$ and $\vec{\xi}$, respectively. With a map $\vec{\xi}=\vec{\xi}(\mathbf{x})$ from Ω to Ω_c, its energy is given by

$$E(\vec{\xi})=\frac{1}{2}\int_{\Omega}\sqrt{d}d^{ij}r_{\alpha\beta}\frac{\partial\xi^{\alpha}}{\partial x^{i}}\frac{\partial\xi^{\beta}}{\partial x^{j}}d\mathbf{x}, \tag{3.1}$$

where $d=det(d_{ij})$ and $d^{ij}=(d_{ij})^{-1}$, with the standard summation convection. The map $\vec{\xi}$ is called a harmonic map if it is an extremum of (3.1), which can be found by solving

$$\frac{1}{\sqrt{d}}\frac{\partial}{\partial x^{i}}\sqrt{d}d^{ij}\frac{\partial\xi^{k}}{\partial x^{j}}+d^{ij}\Gamma^{k}_{\beta\gamma}\frac{\partial\xi^{\beta}}{\partial x^{i}}\frac{\partial\xi^{\gamma}}{\partial x^{j}}=0, \tag{3.2}$$

with $\Gamma^k_{\beta\gamma}$ the Christoffel symbol of the second kind. Restricting the discussion in the Euclidean space, we have $\Gamma^k_{\beta\gamma}=0$, and the Euler-Lagrange equation (3.2) becomes

$$\frac{\partial}{\partial x^i}G^{ij}\frac{\partial \xi^k}{\partial x^j}=0, \tag{3.3}$$

with $G^{ij}=\sqrt{d}d^{ij}$.

The *r*-adaptive mesh method works as follows. With the numerical solution obtained from current mesh, a monitor function $M=(G^{ij})^{-1}$ is designed. Then the Euler-Lagrange equation (3.3) is solved, whose solution is a new distribution of mesh grids in the domain Ω_c. Subsequently, the displacement of each grid point in Ω_c affinely maps back to the domain Ω, giving the displacement of the corresponding grid point in Ω. Finally, a solution update is implemented to represent the solution on the new mesh.

Based on the above algorithm, in the package two classes are provided for the implementation of *r*-adaptive mesh methods in two and three dimensions, respectively, i.e., *class MovingMesh2D* and *class MovingMesh3D*. In these two classes, several virtual member functions need to be realized by users, see Listing 8 for 3D case.

```
class MovingMesh3D : public Mesh<3,3>
{ ...
 std::vector<float> mon;
  ...
  virtual void getMonitor();
  virtual void smoothMonitor(u_int step = 1);
  virtual void updateMesh();
  virtual void updateSolution() = 0;
  virtual void outputSolution() = 0;
  virtual void getMoveStepLength();
  ...
}
```

Listing 8: The *class MovingMesh3D*.

Among these virtual functions, three are tightly related to the simulation, i.e., *getMonitor()*, *smoothMointor()*, and *updateSolution()*. In *getMonitor()*, the vector *mon* is filled by the quantity depending on numerical solutions, either by a feature-based approach or by an *a posteriori* error estimation approach. The smoothing operation for the monitor function is necessary to guarantee the quality of the simulation. Although approaches such as the Laplacian smoothing work generally, the special method needs to be designed to handle the specific problem. For example, in [71], a smoothing strategy based on the diffusive mechanism was designed, which became a key on applying *r*-adaptive mesh methods in simulating dendritic growth of crystals. In *updateSolution()*, a solution update algorithm is needed to transfer the numerical solution from the current mesh to the new mesh. Both the numerical accuracy and the implementation efficiency of such an update algorithm are crucial, especially for solving time-dependent PDEs. In [60], a solution update strategy was proposed based on an assumption that after updating the numerical solution,

the profile of the solution should be kept unchanged. In such a strategy, a convection mechanism is utilized based on an observation that with the movement of grid points in Ω, the associated solution on each grid point flows to the new position. Hence, the solution update can be done by solving a convection equation with a well designed velocity. It is this solution update strategy which makes the proposed *r*-adaptive mesh method an independent module in the package, which can be used generally for the given PDEs. In [31], the harmonic map based *r*-adaptive mesh method was successfully extended to solve the problem defined on a sphere.

4 Towards large scale problems

The overall parallel scheme in AFEPack is finished on the architectures with a distributed memory and in a message passing interface (MPI) manner. The key techniques include mesh data preparation, data synchronizing and dynamic load balancing, which will be introduced in detail below, respectively.

Mesh data preparation: During the mesh data preparation region, AFEPack first needs a triangular or a tetrahedron background mesh in two dimensional and three dimensional case, respectively. Then a few rounds of uniform refinement on the background mesh can be applied to obtain a refined mesh according to the number of processes in later simulations. At last, a domain decomposition algorithm based on Hilbert space filling curve (HSFC) to sort the cells in the refined mesh. This decomposition divides the mesh into a group of sub-meshes, as illustrated in Fig. 12 for reference.

Data synchronizing: When the mesh preparation of a simulation is finished, AFEPack assigns each sub-mesh to the corresponding process and numerically solve PDEs in a parallel manner. AFEPack encapsulates MPI by a "shared object" driven data to synchronize mechanism. The data to be transferred are attached on these shared objects and

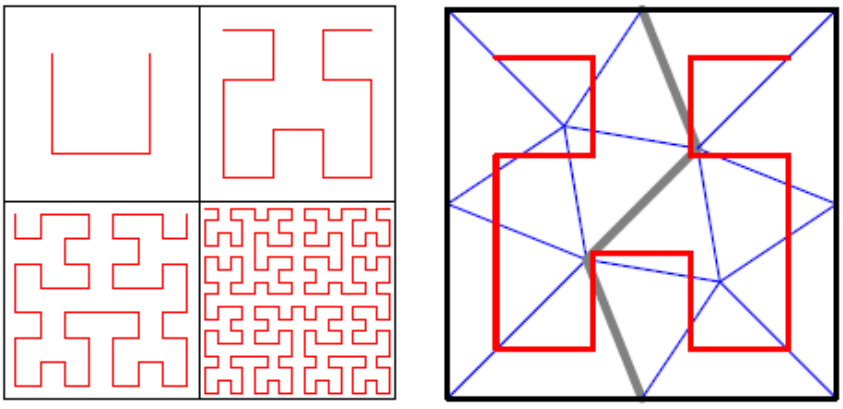

Figure 12: Examples of HSFC (left) and the domain decomposition via HSFC (right).

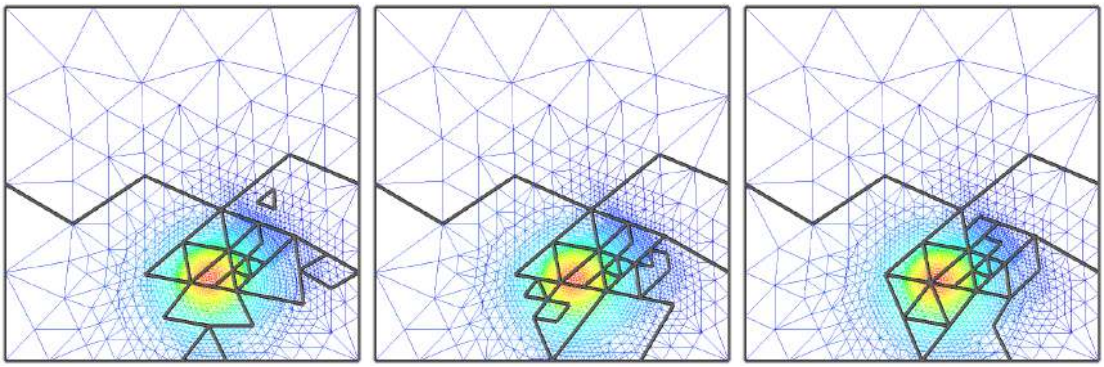

Figure 13: An example of dynamic load balancing for different simulation time.

the shared objects transfer the data to their mirror copies on other processes. The tiny data fragments attached on the shared objects are collected together into stream buffers and then send to destinations based on the data transferring plan. AFEPack defines a transmit map and interface for end-users to synchronize data based on the shared objects, which assumes that the data transferring are bilateral.

Dynamic load balancing: During the parallel simulation, the load imbalance is usually caused due to the h-mesh adaption, which makes the number of mesh cells different from process to process and leads serious computational resource waste. The dynamic load balancing during the simulation can be applied to improve the efficiency. It is already supported by AFEPack to balance the load on the background element level.

When applying the dynamic load balancing, AFEPack first calculates the computational loads of mesh cells in each process, and lumps the loads to the parent geometry recursively. For a given partition, it can quantify the load imbalance by the l_1-load imbalance and l_∞-load imbalance, which is the sum of the idle CPU time and the maximal idle CPU time. When the load imbalance reaches a limit value, a re-partition implementation is accomplished. Since the background cells are sorted based on HSFC, the obtained re-partition is expected to have satisfactory quality.

After the re-partition procedure, every background mesh cell is assigned to a new process. The next step is to move the data from the old process to the new process. In AFEPack, a serialization procedure is adopted to copy the pointer of the hierarchy geometry tree from one process to another process. Then a data migration implementation is followed to migrate the data structures based on the hierarchy geometry tree, such as mesh, finite element space, finite element functions and algebraic system solver.

The above three procedures are the main steps in the parallel scheme of AFEPack, which can maintain the efficiency of parallel computing in a satisfactory level. An illustration of dynamic load balancing is provided in Fig. 13.

5 Applications of AFEPack

So far, the AFEPack has been successfully applied in a variety of scientific and engineering problems, such as computational fluid dynamics, electronic structure calculations, computational micromagnetics. In following subsections, several applications will be introduced briefly to demonstrate the effectiveness of the package.

5.1 Computational fluid dynamics

Computational fluid dynamics has always been an active research area in computational sciences, due to its exuberant vitality in practical applications such as optimal design of the vehicle shape, new energy development, the prevention and prediction of natural disaster. In the following, three examples are described towards applications of the package in computational fluid dynamics.

The steady state solution of Euler equations

As a simplified model of Navier-Stokes equations, Euler equations have been playing an important role in applications in compressible fluid dynamics.

To obtain a steady state solution of the Euler equations, a Newton-GMG solver has been proposed in [62], and developed in [40–43, 45, 65–67], in which the Newton method is used for the linearization of the governing equations, and a geometric multi-grid solver is designed for the solution of the system of linear equations. It is noted that due to the conservation property of solutions, the finite volume method is employed for the spatial discretization, which can be conveniently realized in AFEPack, though the package is originally designed for finite element methods.

In Fig. 14, steady state solutions of Euler equations from an *h*-adaptive finite volume method are demonstrated. Results are obtained from [43], in which a goal oriented *a posteriori* error estimation is designed for the *h*-adaptivity.

The solution of reactive Euler equations

Reactive Euler equations play an important role in the study of detonation phenomenon.In [29], an *h*-adaptive finite volume method was designed for above equations, in which the Strang splitting technique was employed to handle the challenge from the reactive term. In Fig. 15, numerical results from [29] are demonstrated.

Combustion with a multi-component gas model

In the field of combustion research, comprehending the intricate dynamics of combustion processes holds paramount importance for a number of perspectives, encompassing optimizing energy efficiency, mitigating emissions, etc. In this subsection, we simulate a nuclear air blast wave from one kiloton nuclear charge. The computational domain is $[0,400]\times[0,1500]$ in meters, and the burst point is at $(0,50)$m with an initial radius of

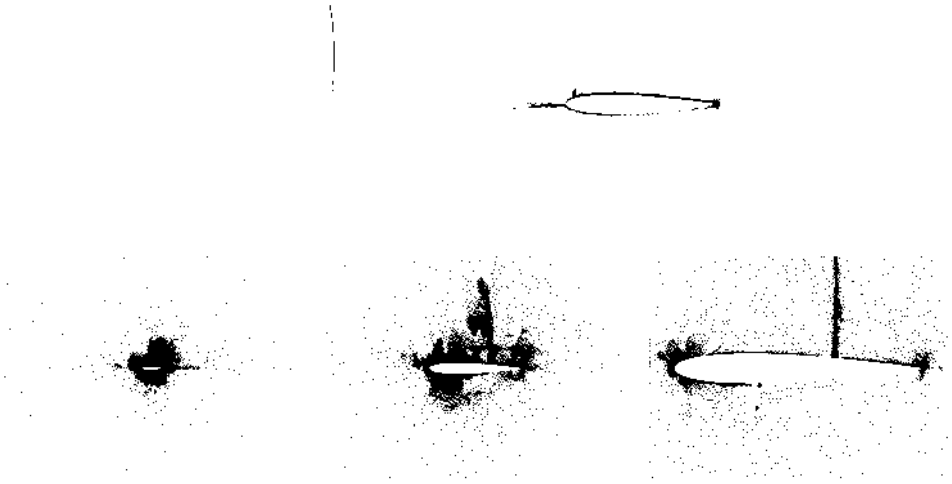

Figure 14: Steady state solutions of Euler equations obtained with an h-adaptive mesh method for the transonic flows around a NACA0012 airfoil, with mach number 0.8 and attack angle 1.25°. Top left: mach isolines around the airfoil; Top right: isolines of x-momentum from the adjoint problem; Bottom: mesh profiles, zoomed in from left to right. Results are from [43].

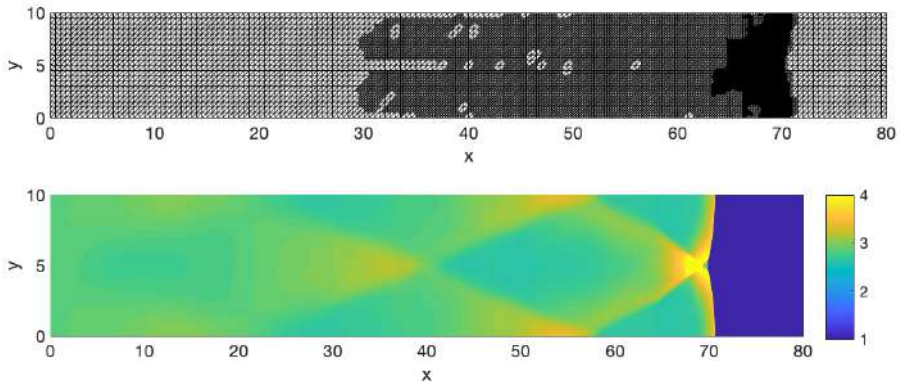

Figure 15: Numerical results when $t\approx 200$ of reactive Euler equations with the configuration given in Example 4.3 in [29]. Top: mesh profile of the results with h-adaptivity; Bottom: the distribution of the pressure.

0.3m. The governing equation of state of this problem is expressed in a cylindrical form, and the initial conditions are

$$[\rho,u,v,p]^T=\begin{cases}\left[618.935,0,0,6.314\times 10^{12}\right]^T, & \sqrt{x^2+(y-50)^2}\leq 0.3,\\ \left[1.29,0,0,1.013\times 10^5\right]^T, & \sqrt{x^2+(y-50)^2}\geq 0.3.\end{cases}$$

The explosion products and air are modeled by the ideal gas equation of states with adiabatic exponents $\gamma=1.2$ and 1.4 respectively. All boundaries are set up by outflow

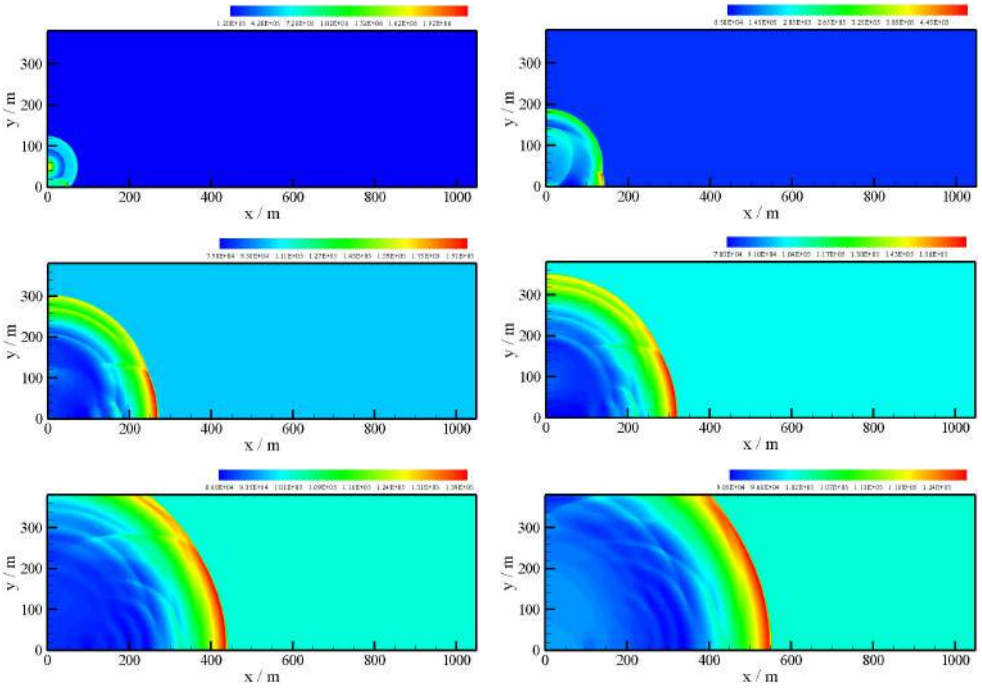

Figure 16: Pressure contours of air blast problem at typical time instants

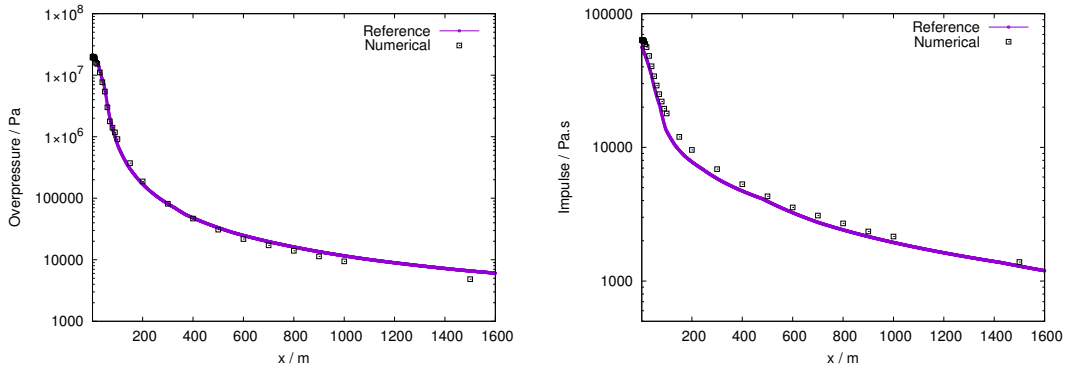

Figure 17: Blast wave parameters of typical radii on the ground.

conditions except for the bottom edge $y=0$ which is a rigid ground. Fig. 16 shows the pressure contours at typical time. When the blast wave produced by the nuclear explosion arrives at the rigid ground, it will be reflected firstly and propagate along the rigid ground simultaneously. When the incident angle exceeds the limit, the reflective wave switches from regular to irregular, and a Mach blast wave occurs. The peak over-pressure and impulse at different radii are shown in Fig. 17, and agree well with the reference data interpolated from the experimental data.

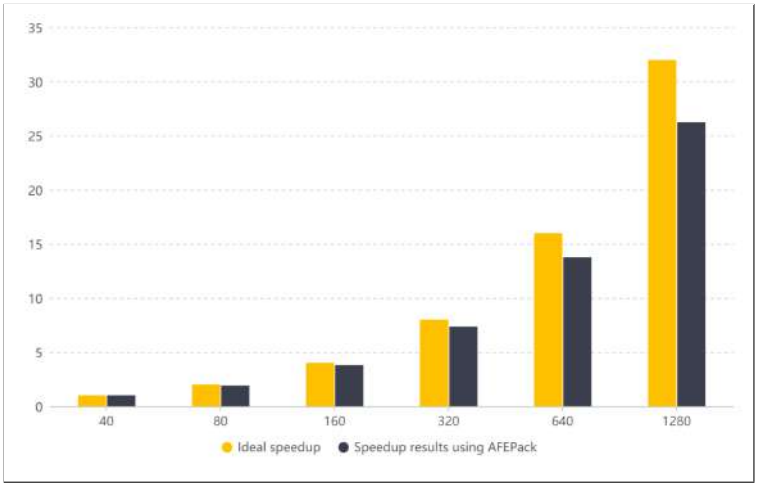

Figure 18: Speedup results in simulations using AFEPack.

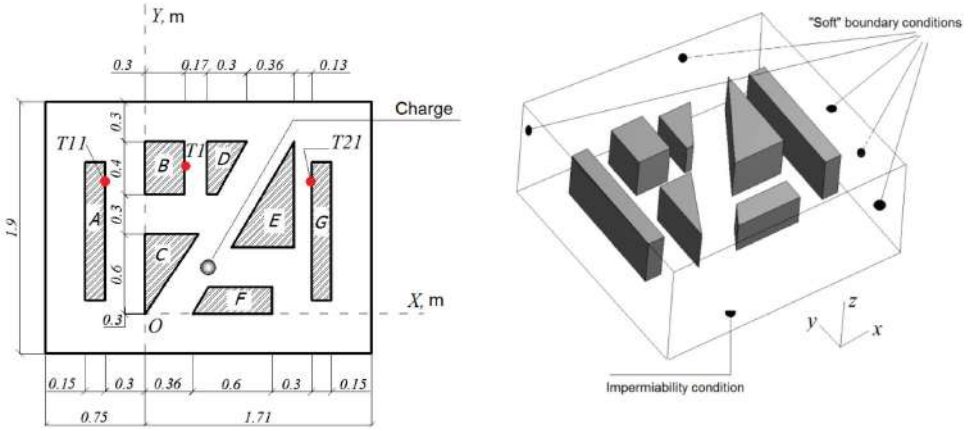

Figure 19: Schematic model of the scaled city buildings.

We also test the parallel behavior of the numerical code in different cores (from 40 to 1280 cores), and the speedup results are given in Fig. 18.

Furthermore, an intense blast wave propagation in a scaled city buildings is simulated. Seven buildings imitated a cityscape, shown in Fig. 19, xf were placed on the plate surface. The heights of the buildings were as follows: $H_A = H_G = 0.45$ m, $H_D = H_F = 0.3$ m, $H_B = H_C = H_E = 0.4$ m. A TNT charge of 16.0 g capacity is detonated at the point located by 0.04 m above the ground between buildings C and F. The distance between adjacent objects is less than or comparable to a linear scale of the buildings.

A full three dimensional numerical simulation is accomplished to simulate the blast wave propagation in the scaled city buildings. The mesh size is about 0.5 cm, and the total number of the mesh is about 80 millions. We use the domain decomposition scheme

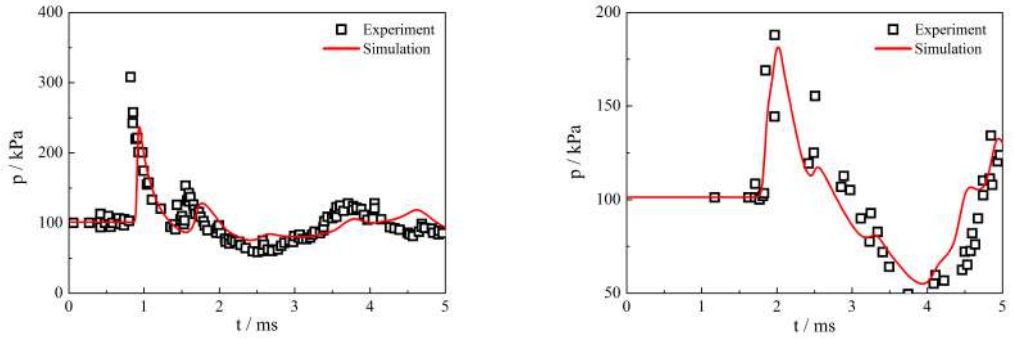

Figure 20: Comparison of pressure history curve between numerical results and experimental data.

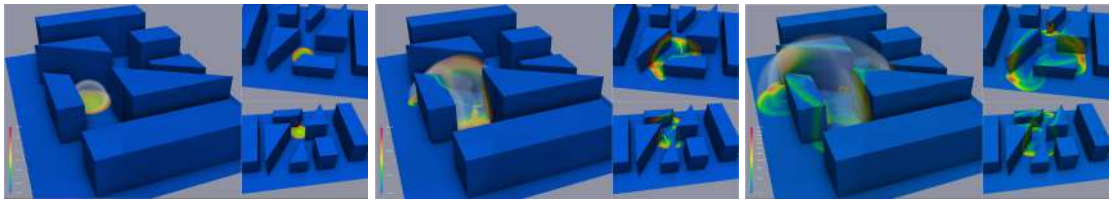

Figure 21: Pressure contours of the computational domain at typical time.

to prepare the parallel mesh data, and 1200 cores is adopted to perform the parallel computation. The comparison between the numerical results and experimental data at the given gauge points, T1 and T21, is shown in Fig. 20, which shows a good agreement between each other. Fig. 21 shows the pressure contours at typical time and reveals that the blast wave will reflect and diffract around the buildings, and finally present a complex wave structures.

5.2 Electronic structure calculations

The Kohn-Sham density functional theory has been one of the most successful approximation models for quantum many-body problems, whose numerical solution has been playing an important role in a variety of application areas such as quantum computational chemistry, nano materials.

In [14], an h-adaptive finite element method was developed for solving the Kohn-Sham equation for the ground state of a given electronic structure. In [16], such an h-adaptive finite element framework was successfully extended for solving the time-dependent Kohn-Sham equation, whose solution can serve the research in photonabsorption spectra of given electronic structures. It is worth mentioning that in [16], the high harmonic generation phenomenon, which is a classical nonlinear phenomenon in

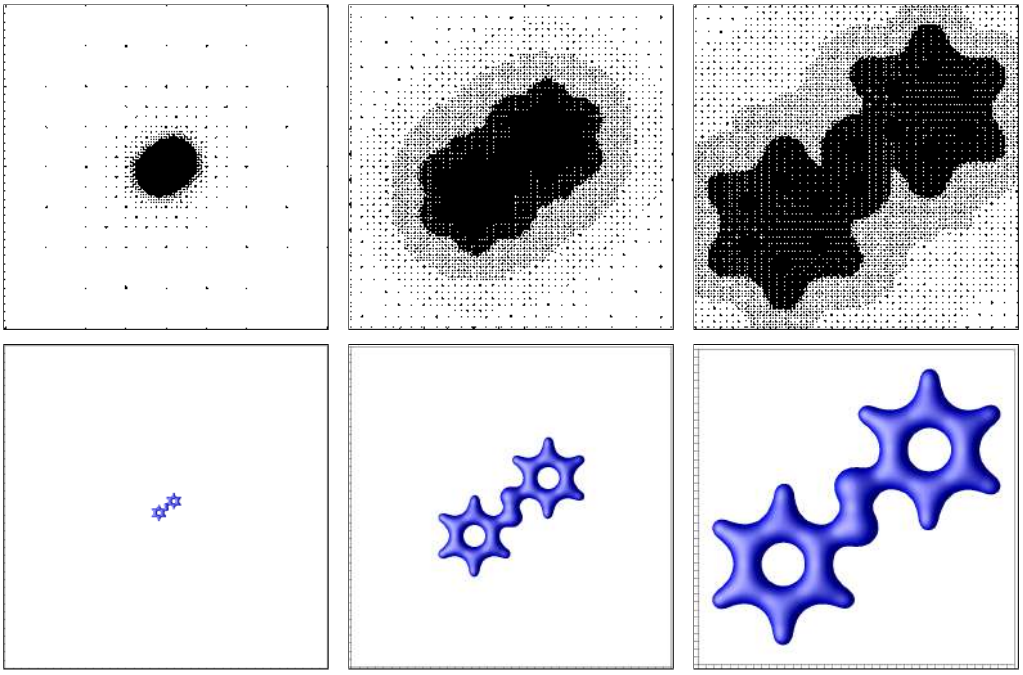

Figure 22: The ground state of an azobenzene molecule $C_{12}H_{10}N_2$. Top: mesh profiles, zoomed in from left to right; Bottom: corresponding isosurfaces of the electron density. Results are reproduced from [56].

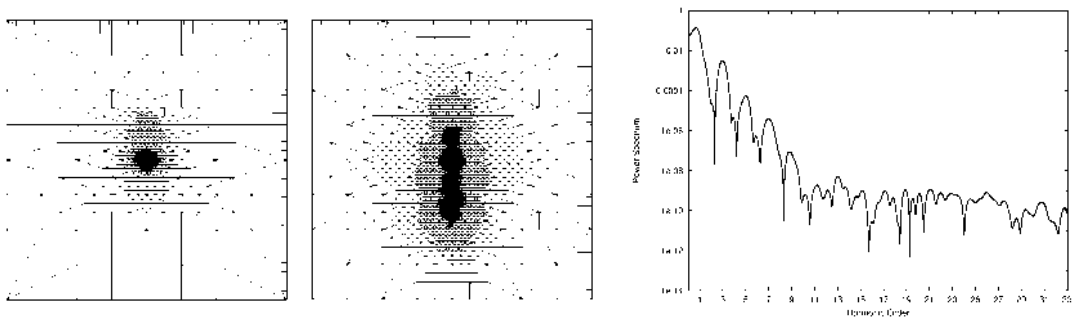

Figure 23: Two slices on the x-y plane of a tetrahedral mesh at $t=400$ (left) and $t=594$ (middle), respectively, as well as the dipole power spectrum of a lithium atom (right). Results are obtained from [16].

quantum optics, was successfully simulated using the h-adaptive mesh method for the first time. The other related works can be found in [13,15,17,18,55,76].

In Fig. 22, numerical solutions of the ground state of an azobenzene molecule $C_{12}H_{10}N_2$ are demonstrated, from which the benefit from the h-adaptive mesh method can be observed obviously. Results are reproduced from [56]. In Fig. 23, numerical solutions of the time-dependent Kohn-Sham equation for a lithium atom are demonstrated,

including two mesh slices at $t=400$ (left) and $t=594$ (middle), respectively. Moreover, the dynamic change of the mesh grids can be seen clearly, as well as the dipole power spectrum of the lithium atom (right), from which the theoretical odd harmonics can be observed successfully.

5.3 Computational micromagnetics

The Landau-Lifshitz-Gilbert equation is a fundamental governing equation in computational micromagnetics, which has been playing an important role in developing the next generation storage device. In [74,75], a finite element method has been developed for the numerical solution of the governing equation.

In Fig. 24, results from [74] on calculating the ground state of the magnetization field in a square thin film with a semi-circle defect are demonstrated, including a tetrahedral mesh with 6778 grid points (left), the magnetization field of the obtained ground state (middle), as well as the convergence of the total energy with the time evolution (right). It is noted that in [74], a prediction for the solution of μMag standard problem #3 is given, which confirms the updated reference value given in the official webpage very well.

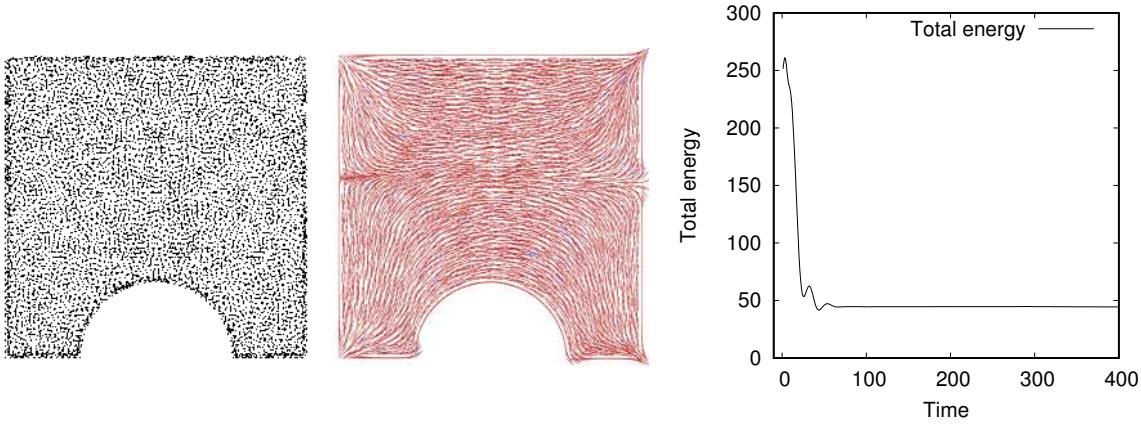

Figure 24: Left: a tetrahedral mesh in a square thin film with a semi-circle defect, totally 6778 grid points; Middle: the magnetization field in the domain when $t=400$; Right: the convergence of the total energy with time evolution. Results are obtained from [74].

5.4 Numerical simulations of reaction-diffusion systems

In [44], numerical simulations for two reaction-diffusion systems, i.e., the Brusselator model and the Gray-Scott model, were studied. The numerical algorithm is based upon a moving finite element method which helps to resolve large solution gradients. High quality meshes are obtained for both the spot replication and the moving wave along boundaries by using proper monitor functions. Several ways for verifying the quality of

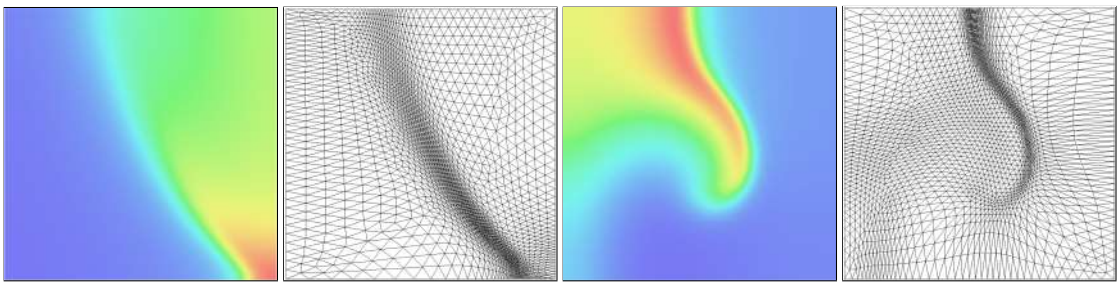

Figure 25: Solutions and corresponding mesh grids of Brusselator model at $t=1$ (left two), and 15 (right two). Please refer to [44] for more details.

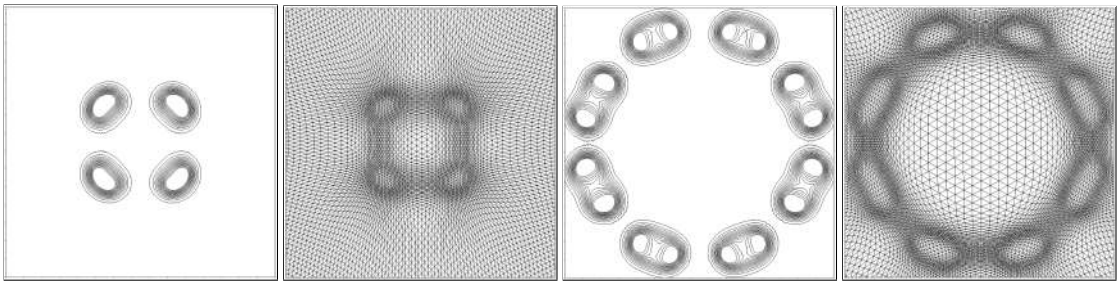

Figure 26: Solutions and corresponding mesh grids of GrayScott model at $t=200$ (left two), and 1500 (right two). Please refer to [44] for more details.

the numerical solutions are also proposed, which may be of important use for comparisons.

In Figs. 25 and 26, numerical solutions and corresponding mesh grids of Brusselator and Gray-Scoot models are demonstrated, respectively. More details can be found in [44] and [73].

5.5 Dendritic growth computations

Dendritic growth stands as a pivotal concern in the realms of pattern formation and metallurgy. In [71], an r-adaptive method has been devised to facilitate efficient simulations of dendritic growth.

Employing meticulously crafted monitor function, the resulting meshes under steady-state simulation conditions are depicted in Fig. 27 and Fig. 28 for 2D and 3D cases, respectively. In Fig. 28, both meshes are sliced at $z=0$, encompassing a total of 6545 degrees of freedom. Evidently, these generated meshes well capture the phase-space variable and dimensionless thermal field.

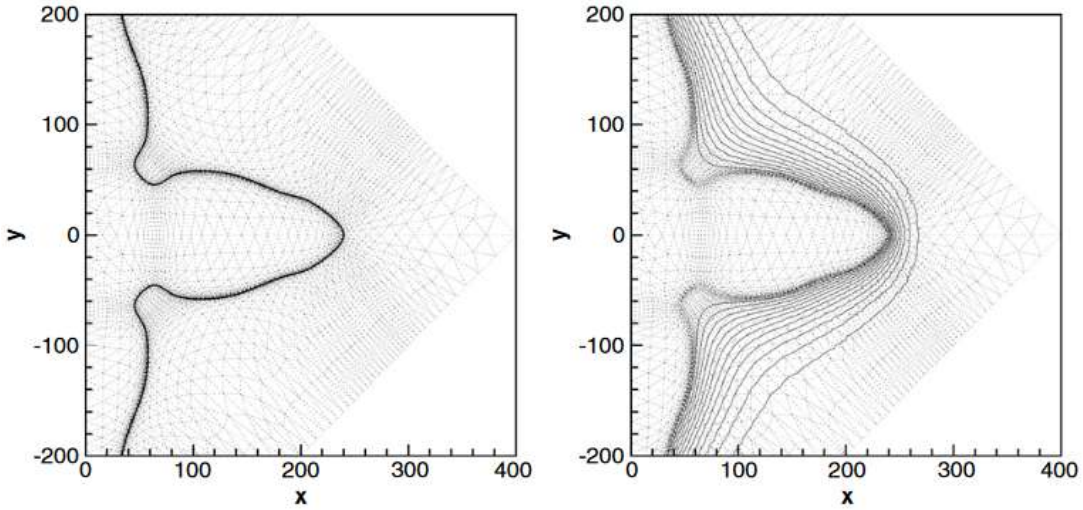

Figure 27: The generated mesh when the simulation is steady in 2D simulation. The solid lines are the contours of phase-space variable (left) and dimensionless thermal field (right). Please refer to [71] for more details.

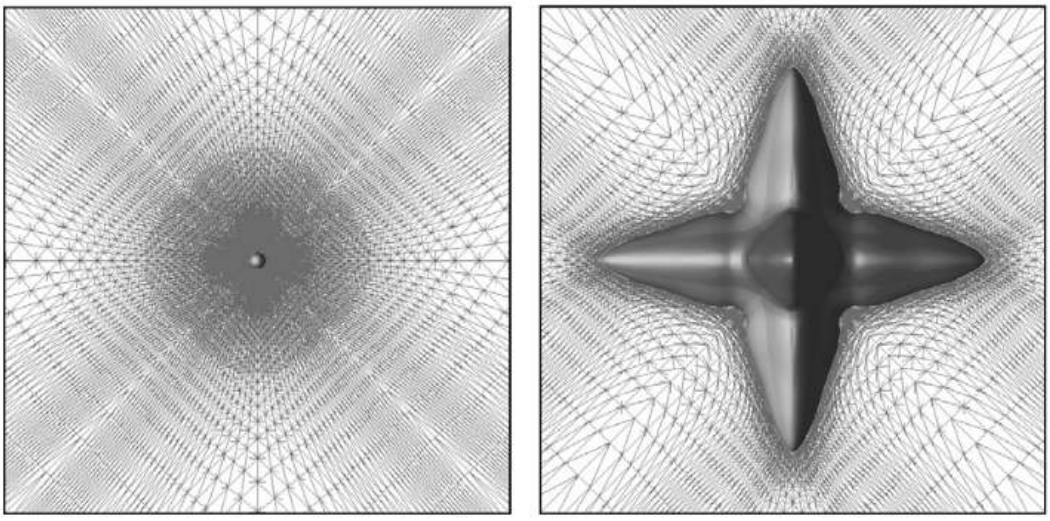

Figure 28: The generated meshes in 3D simulation. The left one is the mesh for the initial value and the right one is the mesh when the simulation is steady. Please refer to [71] for more details.

5.6 Singular problems on a sphere

PDEs defined on a sphere are encountered in numerous practical scenarios, such as polymer rheological analysis, ocean modeling, and global climate simulation. When addressing problems with localized moving singularities, the moving mesh technique becomes

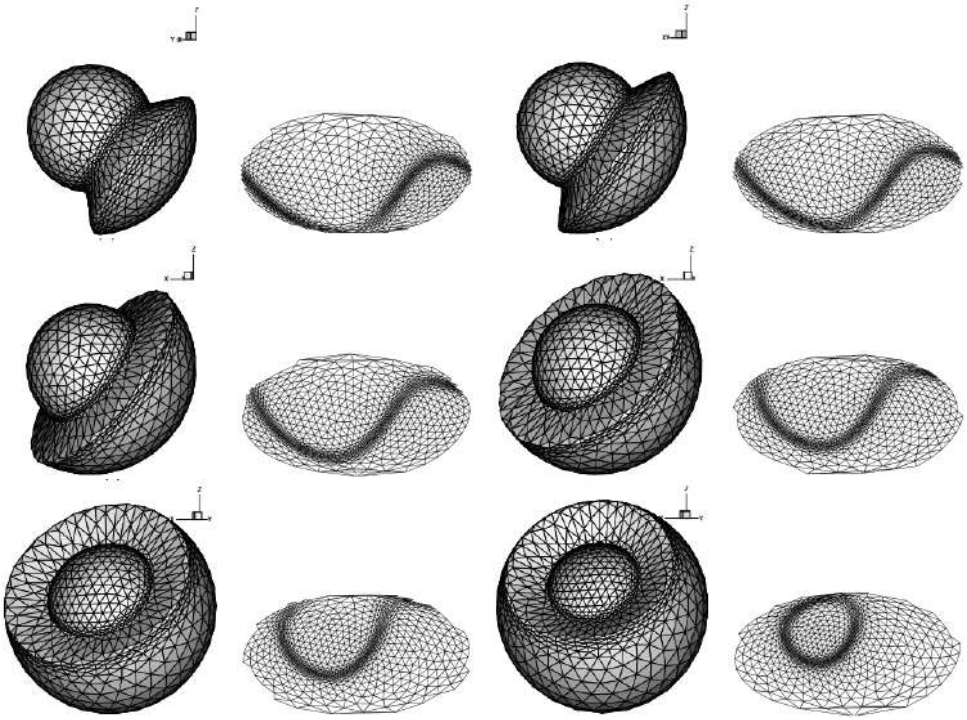

Figure 29: The adaptive meshes on a sphere (column 1,3) and on planisphere (column 2,4) with 800 nodes from $t=0$ to $t=1.5$. Please refer to [31] for more details.

a natural and effective approach. A moving mesh method for singular problems on a sphere was introduced in [31], building upon the perturbed harmonic mapping.

The computational results involving a moving steep front along a specified direction, as detailed in [31], are presented in Fig. 29. In particular, the first and third columns of Fig. 29 depict the moving meshes comprising 800 nodes (about 1600 triangular elements). Meanwhile, the second and fourth columns showcase the corresponding mesh distributions projected onto a planisphere. Notably, the mesh demonstrates commendable adaption within regions characterized by large gradients.

6 Conclusion

In this paper, a systematic introduction of a general-purpose C++ library AFEPack for numerical solutions of partial differential equations was delivered, including the design philosophy of the library, algorithms and data structures used in the discretization of governing equations, numerical linear algebra for the discretized system, as well as the pre-processing and post-processing of the simulations. Two features in the library, i.e., adaptive mesh module and parallel computing module, were also described in detail.

Finally, applications of the library in a variety of research areas, such as computational fluid dynamics, electronic structure calculations, computational micromagnetics, were demonstrated, which showed the potential of the library towards both scientific exploration and engineering projects.

It is worth mentioning that, with the emergence of new hardware and numerical methods for scientific computing, the development of the AFEPack is kept active. For instance, the GPU computing module in the library is under development. Furthermore, several specific-purpose libraries have also been inspired by AFEPack, whose introduction can be found in the forthcoming papers.

Acknowledgments

Z. Cai would like to thank the support from the Academic Research Fund of the Ministry of Education of Singapore (No. A-0008592-00-00). Y. Di would like to thank the support from National Natural Science Foundation of China (Nos. 12271048, 12171042), and Guangdong Key Laboratory (No. 2022B1212010006). G. Hu would like to thank the support from National Natural Science Foundation of China (No. 11922120), The Science and Technology Development Fund, Macao SAR(File/Project nos. 001/2024/SKL and 0082/2020/A2). R. Li would like to thank the support from National Natural Science Foundation of China (Nos. 12201442, 11971041). W. Liu would like to thank the support from BNU-HKBU United International College (No. 202026). F. Yang would like to thank the support from National Natural Science Foundation of China (No. 12201442). H. Zhan would like to thank the support from the China Postdoctoral Science Foundation (No. 2023M740107).

Appendices

A Installation of AFEPack

First of all, the source code can be downloaded from `http://dsec.pku.edu.cn/~rli/software.php#AFEPack`. A typical process for the installation of AFEPack is shown in Listing 9 below.

```
# ./configure
# make
# make install
```

Listing 9: `Installation process of AFEPack.`

It is noted that AFEPack is packed under GNU Build System using Autotools. Using the command *./configure -h* in the terminal will get more information for the installation.

B Sample code for solving the Poisson equation

```
#include <iostream>
#include <fstream>
#include <AFEPack/AMGSolver.h>
#include <AFEPack/Geometry.h>
#include <AFEPack/TemplateElement.h>
#include <AFEPack/FEMSpace.h>
#include <AFEPack/Operator.h>
#include <AFEPack/Functional.h>
#include <AFEPack/EasyMesh.h>

#define PI M_PI
#define DIM 2

double u(const double * p)
{
  return sin(PI*p[0]) * sin(2*PI*p[1]);
};
double f(const double * p)
{
  return 5*PI*PI*u(p);
};

int main(int argc, char * argv[])
{
  EasyMesh mesh;
  mesh.readData(argv[1]);

  TemplateGeometry<DIM> triangle_template_geometry;
  triangle_template_geometry.readData("triangle.tmp_geo");
  CoordTransform<DIM,DIM> triangle_coord_transform;
  triangle_coord_transform.readData("triangle.crd_trs");
  TemplateDOF<DIM> triangle_template_dof(triangle_template_geometry);
  triangle_template_dof.readData("triangle.1.tmp_dof");
  BasisFunctionAdmin<double,DIM,DIM> triangle_basis_function(triangle_template_dof);
  triangle_basis_function.readData("triangle.1.bas_fun");

  std::vector<TemplateElement<double,DIM,DIM> > template_element(1);
  template_element[0].reinit(triangle_template_geometry,
                             triangle_template_dof,
                             triangle_coord_transform,
                             triangle_basis_function);

  FEMSpace<double,DIM> fem_space(mesh, template_element);
  int n_element = mesh.n_geometry(DIM);
  fem_space.element().resize(n_element);
  for (int i = 0;i < n_element;i ++)
    fem_space.element(i).reinit(fem_space,i,0);

  fem_space.buildElement();
```

```
  fem_space.buildDof();
  fem_space.buildDofBoundaryMark();

  StiffMatrix<DIM,double> stiff_matrix(fem_space);
  stiff_matrix.algebricAccuracy() = 4;
  stiff_matrix.build();

  FEMFunction<double,DIM> solution(fem_space);
  Vector<double> right_hand_side;
  Operator::L2Discretize(&f, fem_space, right_hand_side, 4);

  BoundaryFunction<double,DIM>
    boundary1(BoundaryConditionInfo::DIRICHLET, 1, &u);
  BoundaryConditionAdmin<double,DIM> boundary_admin(fem_space);
  boundary_admin.add(boundary1);
  boundary_admin.apply(stiff_matrix, solution, right_hand_side);

  AMGSolver solver(stiff_matrix);
  solver.solve(solution, right_hand_side, 1.0e-08, 20);

  solution.writeOpenDXData("u.dx");
  double error = Functional::L2Error(solution, FunctionFunction<double>(&u), 3);
  std::cerr << "\n L2 error = " << error << std::endl;

  return 0;
};
```

Listing 10: A sample code for solving the Poisson equation by AFEPack.

C Sample code for solving the Diffusion equation

```
#include <stdio.h>
#include <dlfcn.h>

#include <iostream>
#include <fstream>

#include <base/exceptions.h>
#include <lac/vector.h>
#include <lac/sparsity_pattern.h>
#include <lac/sparse_matrix.h>

#include <AFEPack/AMGSolver.h>
#include <AFEPack/Geometry.h>
#include <AFEPack/TemplateElement.h>
#include <AFEPack/FEMSpace.h>
#include <AFEPack/Operator.h>
#include <AFEPack/EasyMesh.h>
#include <AFEPack/Functional.h>

```

```
#define PI M_PI
#define DIM 2
const int N_Ite = 32;
const double T = 1;
const double C = 1;

double u0(const double *p){ return sin(PI*p[0])*sin(PI*p[1]); }
double u0_px(const double *p){ return PI*cos(PI*p[0])*sin(PI*p[1]); }
double u0_py(const double *p){ return PI*sin(PI*p[0])*cos(PI*p[1]); }
double u0_px2(const double *p){ return -PI*PI*u0(p); }
double u0_pxpy(const double *p){ return PI*PI*cos(PI*p[0])*cos(PI*p[1]); }
double u0_py2(const double *p){ return -PI*PI*u0(p); }
double uT(const double *p){ return exp(-C*T)*u0(p); }

double a11_(const double *p){ return p[0]*p[0]; }
double a12_(const double *p){ return p[0]*p[1]; }
double a21_(const double *p){ return p[0]*p[1]; }
double a22_(const double *p){ return p[1]*p[1]; }
double a11_px(const double *p){ return 2*p[0]; }
double a12_px(const double *p){ return p[1]; }
double a21_py(const double *p){ return p[0]; }
double a22_py(const double *p){ return 2*p[1]; }

double f0(const double *p){ // consider exact solution u(x,y,t)=u0(x,y)exp(-Ct)
 return -C*u0(p) - (a11_(p)*u0_px2(p)+(a12_(p)+a21_(p))*u0_pxpy(p)+a22_(p)*u0_py2(p)
 + (a11_px(p)+a21_py(p))*u0_px(p) + (a12_px(p)+a22_py(p))*u0_py(p));
}

class Matrix : public StiffMatrix<DIM, double>
{
 public:
 Matrix(FEMSpace<double,DIM>& sp) :
 StiffMatrix<DIM,double>(sp) {};
 virtual ~Matrix() {};
 private:
 double ts;
 public:
 virtual void getElementMatrix(const Element<double,DIM>& element0,
 const Element<double,DIM>& element1,
 const ActiveElementPairIterator<DIM>::State state);
 void setTimeStep(double timeStep){ ts = timeStep; }
};

void Matrix::getElementMatrix(const Element<double,DIM>& element0,
const Element<double,DIM>& element1,
const ActiveElementPairIterator<DIM>::State state)
{
 int n_element_dof0 = elementDof0().size();
 int n_element_dof1 = elementDof1().size();
 double volume = element0.templateElement().volume();
 const QuadratureInfo<DIM>& quad_info = element0.findQuadratureInfo(algebricAccuracy());
 std::vector<double> jacobian = element0.local_to_global_jacobian(quad_info.quadraturePoint());
```

```
  int n_quadrature_point = quad_info.n_quadraturePoint();
  std::vector<AFEPack::Point<DIM> > q_point = element0.local_to_global(quad_info.quadraturePoint());
  std::vector<std::vector<double> > basis_value = element0.basis_function_value(q_point);
  std::vector<std::vector<std::vector<double> > > basis_gradient = element0.basis_function_gradient(
      q_point);
  for (int l = 0;l < n_quadrature_point;l ++) {
   double Jxw = quad_info.weight(l)*jacobian[l]*volume;
   AFEPack::Point<DIM> q_point = element0.local_to_global(quad_info.quadraturePoint(l));
   double a11 = a11_(q_point), a12 = a12_(q_point), a21 = a21_(q_point), a22 = a22_(q_point);
   for (int j = 0;j < n_element_dof0;j ++){
    for (int k = 0;k < n_element_dof1;k ++){
     elementMatrix(j,k) += Jxw*((a11 * basis_gradient[j][l][0] * basis_gradient[k][l][0]
                                 + a12 * basis_gradient[j][l][0] * basis_gradient[k][l][1]
                                 + a21 * basis_gradient[j][l][1] * basis_gradient[k][l][0]
                                 + a22 * basis_gradient[j][l][1] * basis_gradient[k][l][1]) * ts
                                 // the above corresponds to term \nabla(A \nabla u)
                                + basis_value[j][l] * basis_value[k][l]); // corresponds to term from time
      discretization
     }
    }
   }
  };

  int main(int argc, char * argv[])
  {
   // read mesh info
   EasyMesh mesh;
   mesh.readData("D");

   // construct template element
   TemplateGeometry<DIM> triangle_template_geometry;
   triangle_template_geometry.readData("triangle.tmp_geo");
   CoordTransform<DIM,DIM> triangle_coord_transform;
   triangle_coord_transform.readData("triangle.crd_trs");
   TemplateDOF<DIM> triangle_template_dof(triangle_template_geometry);
   triangle_template_dof.readData("triangle.1.tmp_dof");
   BasisFunctionAdmin<double,DIM,DIM> triangle_basis_function(triangle_template_dof);
   triangle_basis_function.readData("triangle.1.bas_fun");
   std::vector<TemplateElement<double,DIM,DIM> > template_element(1);
   template_element[0].reinit(triangle_template_geometry,
                              triangle_template_dof,
                              triangle_coord_transform,
                              triangle_basis_function);

   // build FEM space
   FEMSpace<double,DIM> fem_space(mesh, template_element);
   int n_element = mesh.n_geometry(DIM);
   fem_space.element().resize(n_element);
   for (int i = 0;i < n_element;i ++) fem_space.element(i).reinit(fem_space,i,0);
   fem_space.buildElement();
   fem_space.buildDof();
   fem_space.buildDofBoundaryMark();
```

```
122
123   // assign matrix for linear system
124   MassMatrix<DIM,double> mass_matrix(fem_space);
125   mass_matrix.algebricAccuracy() = 6;
126   mass_matrix.build();
127   Matrix sp_matrix(fem_space);
128   double dt = T / N_Ite;
129   sp_matrix.setTimeStep(dt);
130   sp_matrix.algebricAccuracy() = 6;
131   sp_matrix.build();
132
133   // prepare boundary info
134   BoundaryFunction<double,DIM> boundary(BoundaryConditionInfo::DIRICHLET, 1, &u0);
135   BoundaryConditionAdmin<double,DIM> boundary_admin(fem_space);
136   boundary_admin.add(boundary);
137
138   // attain initial value by interpolation
139   FEMFunction<double,DIM> solution(fem_space);
140   Operator::L2Interpolate(&u0, solution);
141   double error0 = Functional::L2Error(solution, FunctionFunction<double>(&u0),6);
142   std::cout << "L2 error of uh when t = 0: " << error0 << std::endl;
143
144   // time evolution
145   Vector<double> rhs0(fem_space.n_dof());
146   AMGSolver solver;
147   Operator::L2Discretize(&f0, fem_space, rhs0, 6);
148   for (int i = 1;i <= N_Ite;i ++){
149     Vector<double> right_hand_side(rhs0), tmp(fem_space.n_dof());
150     // contribution from right hand side of equation
151     double t = (i) * dt;
152     right_hand_side *= exp(-C*t) * dt;
153     // contribution from last step
154     mass_matrix.vmult(tmp, (dealii::Vector<double>) solution);
155     right_hand_side.add(1.0, tmp);
156     // solve linear system
157     boundary_admin.apply(sp_matrix, solution, right_hand_side);
158     if (i == 1) solver.reinit(sp_matrix);
159     solver.solve(solution, right_hand_side);
160   }
161
162   // output visualization result and error
163   solution.writeOpenDXData("u.dx");
164   double error = Functional::L2Error(solution, FunctionFunction<double>(&uT), 6);
165   std::cout << "L2 error of uh when t = T: " << error << std::endl;
166
167   return 0;
168 };
```

Listing 11: A sample code for solving the Diffusion equation by AFEPack.

References

[1] Abinit. https://www.abinit.org/.
[2] Clawpack. https://depts.washington.edu/clawpack/.
[3] deal.II. https://www.dealii.org/.
[4] Geoclaw. https://depts.washington.edu/clawpack/geoclaw/.
[5] Netgen. https://ngsolve.org/.
[6] Paraview. https://www.paraview.org/.
[7] PHG. http://lsec.cc.ac.cn/phg/index_en.htm.
[8] Quantum ESPRESSO. https://www.quantum-espresso.org/.
[9] Sem_poisson_solver. https://github.com/hfzhan/SEM_Poisson.git.
[10] SU2. https://su2code.github.io/.
[11] VASP. https://www.vasp.at/.
[12] *Open DX: Paths to Visualization*. VIS Inc, 2001.
[13] Gang Bao, Ricardo Delgadillo, Guanghui Hu, Di Liu, and Songting Luo. Modeling and computation of nano-optics. *CSIAM Transactions on Applied Mathematics*, 1(4):616–638, 2020.
[14] Gang Bao, Guanghui Hu, and Di Liu. An h-adaptive finite element solver for the calculations of the electronic structures. *Journal of Computational Physics*, 231(14):4967–4979, 2012.
[15] Gang Bao, Guanghui Hu, and Di Liu. Numerical solution of the Kohn-Sham equation by finite element methods with an adaptive mesh redistribution technique. *Journal of Scientific Computing*, 55:372–391, 2013.
[16] Gang Bao, Guanghui Hu, and Di Liu. Real-time adaptive finite element solution of time-dependent Kohn–Sham equation. *Journal of Computational Physics*, 281:743–758, 2015.
[17] Gang Bao, Guanghui Hu, and Di Liu. Towards translational invariance of total energy with finite element methods for Kohn-Sham equation. *Communications in Computational Physics*, 19(1):1–23, 2016.
[18] Gang Bao, Guanghui Hu, Di Liu, and Songting Luo. Multi-physical modeling and multi-scale computation of nano-optical responses. In Hongtao Li, Jichun Yang and Eric Machorro, editors, *Recent Advances in Scientific Computing and Applications, Contemporary Mathematics*, volume 586, pages 21–40. American Mathematical Society, 2012.
[19] Gang Bao, Mingming Zhang, Bin Hu, and Peijun Li. An adaptive finite element DtN method for the three-dimensional acoustic scattering problem. *Discrete and Continuous Dynamical Systems – B*, 26(1):61–79, 2021.
[20] Huanying Bian, Yedan Shen, and Guanghui Hu. An *h*-adaptive finite element solution of the relaxation non-equilibrium model for gravity-driven fingers. *Advances in Applied Mathematics and Mechanics*, 13(6):1418–1440, 2021.
[21] M. Brezina, A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge. Algebraic multigrid based on element interpolation (AMGe). *SIAM Journal on Scientific Computing*, 22(5):1570–1592, 2001.
[22] Hongyu Chen, Guanghui Hu, and Defang Ouyang. A numerical study of the distribution of chemotherapeutic drug carmustine in brain glioblastoma. *Drug Delivery and Translational Research*, 2021.
[23] Yun Chen, Bernard Billia, Dian Zhong Li, Henri Nguyen-Thi, Na Min Xiao, Abdoul-Aziz Bogno. Tip-splitting instability and transition to seaweed growth during alloy solidification in anisotropically preferred growth direction. *Acta Materialia*, 66:219-231, 2014.
[24] Xin Bo Qi, Yun Chen, Xiu Hong Kang, Dian Zhong Li, and Tong Zhao Gong. Modeling of coupled motion and growth interaction of equiaxed dendritic crystals in a binary alloy

during solidification. *Scientific Reports*, 7:45770, 2017
[25] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, 2002.
[26] S. Clain, S. Diot, and R. Loubère. A high-order finite volume method for systems of conservation laws—multi-dimensional optimal order detection (mood). *Journal of Computational Physics*, 230(10):4028–4050, 2011.
[27] Andrew J. Cleary, Robert D. Falgout, Van Emden Henson, Jim E. Jones, Thomas A. Manteuffel, Stephen F. McCormick, Gerald N. Miranda, and John W. Ruge. Robustness and scalability of algebraic multigrid. *SIAM Journal on Scientific Computing*, 21(5):1886–1908, 2000.
[28] Sambit Das, Phani Motamarri, Vishal Subramanian, David M. Rogers, and Vikram Gavini. DFT-FE 1.0: A massively parallel hybrid CPU-GPU density functional theory code using finite-element discretization. *Computer Physics Communications*, 280:108473, 2022.
[29] Yana Di, Guanghui Hu, Ruo Li, and Feng Yang. On accurately resolving detonation dynamics by adaptive finite volume method on unstructured grids. *Communications in Computational Physics*, 29(2):445–471, 2020.
[30] Yana Di, Ruo Li, and Tao Tang. A general moving mesh framework in 3D and its application for simulating the mixture of multi-phase flows. *Communications in Computational Physics*, 3(3):582–602, 2008.
[31] Yana Di, Ruo Li, Tao Tang, and Pingwen Zhang. Moving mesh methods for singular problems on a sphere using perturbed harmonic mappings. *SIAM Journal on Scientific Computing*, 28(4):1490–1508, 2006.
[32] Yana Di and Zhijian Yang. Computation of dendritic growth with level set model using a multi-mesh adaptive finite element method. *Journal of Scientific Computing*, 39:441–453, 2009.
[33] Dennis Diehl. *High order schemes for simulation of compressible liquid-vapor flows with phase change*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2007.
[34] A. El-Hamalawi. A 2D combined advancing front-Delaunay mesh generation scheme. *Finite Elements in Analysis and Design*, 40(9):967–989, 2004.
[35] The OpenFOAM Foundation. OpenFOAM. https://openfoam.org/.
[36] Bin Gao, Guanghui Hu, Yang Kuang, and Xin Liu. An Orthogonalization-Free Parallelizable Framework for All-Electron Calculations in Density Functional Theory. *SIAM Journal on Scientific Computing*, 44(3):B723–B745, 2022.
[37] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.
[38] Guanghui Hu. An adaptive finite volume method for 2D steady Euler equations with WENO reconstruction. *Journal of Computational Physics*, 252:591–605, 2013.
[39] Guanghui Hu. A numerical study of 2D detonation waves with adaptive finite volume methods on unstructured grids. *Journal of Computational Physics*, 331:297–311, 2017.
[40] Guanghui Hu, Ruo Li, and Tao Tang. A robust high-order residual distribution type scheme for steady Euler equations on unstructured grids. *Journal of Computational Physics*, 229(5):1681–1697, 2010.
[41] Guanghui Hu, Ruo Li, and Tao Tang. A Robust WENO Type Finite Volume Solver for Steady Euler Equations on Unstructured Grids. *Communications in Computational Physics*, 9(3):627–648, 2011.
[42] Guanghui Hu, Xucheng Meng, and Tao Tang. On robust and adaptive finite volume methods for steady Euler equations. In Christian Klingenberg and Michael Westdickenberg, editors, *Theory, Numerics and Applications of Hyperbolic Problems II*, pages 21–40, Cham, 2018.

Springer International Publishing.
[43] Guanghui Hu, Xucheng Meng, and Nianyu Yi. Adjoint-based an adaptive finite volume method for steady Euler equations with non-oscillatory k-exact reconstruction. *Computers & Fluids*, 139:174–183, 2016. 13th USNCCM International Symposium of High-Order Methods for Computational Fluid Dynamics – A special issue dedicated to the 60th birthday of Professor David Kopriva.
[44] Guanghui Hu, Zhonghua Qiao, and Tao Tang. Moving finite element simulations for reaction-diffusion systems. *Advances in Applied Mathematics and Mechanics*, 4(3):365–381, 2012.
[45] Guanghui Hu and Nianyu Yi. An adaptive finite volume solver for steady Euler equations with non-oscillatory k-exact reconstruction. *Journal of Computational Physics*, 312:235–251, 2016.
[46] Guanghui Hu and Paul Andries Zegeling. Simulating finger phenomena in porous media with a moving finite element method. *Journal of Computational Physics*, 230(8):3249–3263, 2011.
[47] Xianliang Hu, Ruo Li, and Tao Tang. A multi-mesh adaptive finite element approximation to phase field models. *Communications in Computational Physics*, 5(5):1012–1029, 2009.
[48] Ansys Inc. Ansys fluent. https://www.ansys.com/products/fluids/ansys-fluent.
[49] Autodesk Inc. Autodesk cfd. https://www.autodesk.com/products/cfd/overview.
[50] COMSOL Inc. Comsol. https://www.comsol.com/.
[51] Gaussian Inc. Gaussian. https://gaussian.com/.
[52] Q-Chem Inc. Q-chem. https://www.q-chem.com/.
[53] Schrödinger Inc. Jaguar. https://www.schrodinger.com/jaguar.
[54] Tecplot Inc. Tecplot. https://www.tecplot.com/.
[55] Yang Kuang and Guanghui Hu. An adaptive FEM with ITP approach for steady Schrödinger equation. *International Journal of Computer Mathematics*, 95(1):187–201, 2018.
[56] Yang Kuang, Yedan Shen, and Guanghui Hu. An *h*-adaptive finite element method for Kohn-Sham and time-dependent Kohn-Sham equations. *Journal on Numerica Methods and Computer Applications*, 42(1):33–55, 2021.
[57] Ruo Li. *Moving Mesh Methods and Applications*. PhD thesis, Peking University, 2001.
[58] Ruo Li. On multi-mesh *h*-adaptive methods. *Journal of Scientific Computing*, 24:321–341, 2005.
[59] Ruo Li, Qicheng Liu, and Fanyi Yang. A discontinuous least squares finite element method for time-harmonic Maxwell equations. *IMA Journal of Numerical Analysis*, 42(1):817–839, 2021.
[60] Ruo Li, Tao Tang, and Pingwen Zhang. Moving mesh methods in multiple dimensions based on harmonic maps. *Journal of Computational Physics*, 170(2):562–588, 2001.
[61] Ruo Li, Tao Tang, and Pingwen Zhang. A moving mesh finite element algorithm for singular problems in two and three space dimensions. *Journal of Computational Physics*, 177(2):365–393, 2002.
[62] Ruo Li, Xin Wang, and Weibo Zhao. A multigrid block LU-SGS algorithm for Euler equations on unstructured grids. *Numerical Mathematics: Theory, Methods and Applications*, 1(1):92–112, 2008.
[63] Ruo Li, Yanli Wang, and Chengbao Yao. A robust Riemann solver for multiple hydro-elastoplastic solid mediums. *Advances in Applied Mathematics and Mechanics*, 12(1):212–250, 2019.
[64] Ruo Li and Fanyi Yang. A discontinuous Galerkin method by patch reconstruction for elliptic interface problem on unfitted mesh. *SIAM Journal on Scientific Computing*, 42(2):A1428–

A1457, 2020.
[65] Xucheng Meng, Yaguang Gu, and Guanghui Hu. A fourth-order unstructured NURBS-enhanced finite volume WENO scheme for steady Euler equations in curved geometries. *Communications on Applied Mathematics and Computation*, 2021.
[66] Xucheng Meng and Guanghui Hu. A NURBS-enhanced finite volume solver for steady Euler equations. *Journal of Computational Physics*, 359:77–92, 2018.
[67] Xucheng Meng and Guanghui Hu. A NURBS-enhanced finite volume method for steady Euler equations with goal-oriented *h*-adaptivity. *Communications in Computational Physics*, 32(2):490–523, 2022.
[68] Bojan Niceno. Easymesh. https://web.mit.edu/easymesh_v1.4/www/easymesh.html.
[69] Hang Si. Tetgen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2), 2015.
[70] Heyu Wang and Ruo Li. Mesh sensitivity for numerical solutions of phase-field equations using r-adaptive finite element methods. *Communications in Computational Physics*, 3(2):357–375, 2008.
[71] Heyu Wang, Ruo Li, and Tao Tang. Efficient computation of dendritic growth with r-adaptive finite element methods. *Journal of Computational Physics*, 227(12):5984–6000, 2008.
[72] J. A. Weideman and S. C. Reddy. A MATLAB differentiation matrix suite. *ACM Trans. Math. Softw.*, 26(4):465—519, 2000.
[73] Congcong Xie and Xianliang Hu. Finite element simulations with adaptively moving mesh for the reaction diffusion system. *Numerical Mathematics: Theory, Methods and Applications*, 9(4):686–704, 2016.
[74] Lei Yang, Jingrun Chen, and Guanghui Hu. A framework of the finite element solution of the Landau-Lifshitz-Gilbert equation on tetrahedral meshes. *Journal of Computational Physics*, 431:110142, 2021.
[75] Lei Yang and Guanghui Hu. An adaptive finite element solver for demagnetization field calculation. *Advances in Applied Mathematics and Mechanics*, 11(5):1048–1063, 2019.
[76] Lei Yang, Yedan Shen, Zhicheng Hu, and Guanghui Hu. An implicit solver for the time-dependent Kohn-Sham equation. *Numerical Mathematics: Theory, Methods and Applications*, 14(1):261–284, 2020.
[77] Hongfei Zhan and Guanghui Hu. A novel tetrahedral spectral element method for Kohn-Sham model. *Journal of Computational Physics*, 474:111831, 2023.