FSI_T1

Fluid-Structure Interaction Practice Homework

by harryzhou2000 @

School of Aerospace Engineering, Tsinghua University

Note:

if you speak Chinese, my report for coursework is recommended

if you desire to go through some analysis of specific cases. If you want to see the latex formulae properly, see readme.pdf

this document is under work, the contents are incomplete.

Introduction

Origin

This project was a coursework for *the Fundamental of Computational Mechanics* of THU in spring 2021. The course contains both CFD and CSD, and lecturers said I could complete a FSI program in place of both final assignments, as a result this program was born. (It was said FSI projects had seldom been seen in the course apart from one, but I didn't actually get an A for this second record ~~lmao~~)

Basic Purpose

FSI_T1 is basically a c++ (with some template) library providing multiple classes, which enables you to assemble from them a custom
Fluid,

Structural
 or
FSI computation case. Fluid part currently supports only Euler equation, which represents adiabatic and inviscid compressible ideal gas dynamics. Solid part currently supports linear elastic mechanics, including statics and modal-truncation method dynamics. FSI part, correspondingly, is basically meant for moderate structural displacement, typically aeroelastic cases.

The library currently does not have a interface supporting script input or interactive case setting, which means one must construct each single case inside a c++ calling of the relevant classes.

About Project Source Code and Dependencies

What did I code?

All the actual source code files are put in **root** of project directory, all as .hpp or .h files. The cpp files are meant for case building, containing various case sets, and they can be viewed as some kind of 'static scripts' for the program.

The draw-back of not having a script interface and using .cpp as 'script' is that each time you have a new case to compute a re-compiling is needed, which could be rather time consuming.

Dependencies

This project depends on Eigen, which completely is a c++ template library, with no binary file linking. For convenience, I just threw the entire Eigen source code in ./include/, so you won't have to install it. This project also needs c++ standard library and STL to work, which can be handled automatically by compilers mostly.

Numerical Methods and Algorithms

Fluid Methods

To comply with significant movement of boundaries caused by structural displacement, the method of arbitrary Euler-Lagrangian (ALE) description of fluid is adopted. The major differences between ALE and Euler descriptions is that ALE adds some items to Euler relating to the mesh speed[1].

Consider Euler equation for gas dynamics in conservative form:

$$\frac{\partial U}{\partial t}+\frac{\partial F}{\partial x}+\frac{\partial G}{\partial y}+\frac{\partial H}{\partial z}=0$$

$$U=\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ E\end{bmatrix},F=\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ \rho uw\\ u(E+p)\end{bmatrix},G=\begin{bmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ \rho vw\\ v(E+p)\end{bmatrix},H=\begin{bmatrix}\rho w\\ \rho uw\\ \rho vw\\ \rho w^2+p\\ w(E+p)\end{bmatrix}$$

Thus in a ALE control volume:

$$\frac{\partial}{\partial t}\int_{\Omega(t)}UdV+\int_{\partial\Omega(t)}U(u_{\Omega x}n_x+u_{\Omega y}n_y+u_{\Omega z}n_z)d\Gamma+\int_{\partial\Omega(t)}(Fn_x+Gn_y+Hn_z)d\Gamma=0$$

Where $u_{\Omega i}$ are the speed components of the C.V.'s boundaries. The corresponding bold symbols are for vectors in xyz space.

Annotating:

$$\boldsymbol{u}^*=\boldsymbol{u}-\boldsymbol{u}_{\boldsymbol{\Omega}}$$

Thus:

$$\frac{\partial}{\partial t}\int_{\Omega(t)}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ E\end{bmatrix}dV+\int_{\partial\Omega(t)}\left(\begin{bmatrix}\rho u^*\\ \rho u^*u^*+p\\ \rho u^*v^*\\ \rho u^*w^*\\ u^*(E+p)\end{bmatrix}+\begin{bmatrix}\rho v^*\\ \rho u^*v^*\\ \rho v^*v^*+p\\ \rho v^*w^*\\ v^*(E+p)\end{bmatrix}+\begin{bmatrix}\rho w^*\\ \rho u^*w^*\\ \rho v^*w^*\\ \rho w^*w^*+p\\ w^*(E+p)\end{bmatrix}\right)d\Gamma$$
$$+\int_{\partial\Omega(t)}\rho\left(u_x^*n_x+u_y^*n_y+u_z^*n_z\right)\begin{bmatrix}0\\ u_{\Omega x}\\ u_{\Omega y}\\ u_{\Omega z}\\ 0\end{bmatrix}d\Omega=0$$

Thus if using $\boldsymbol{u}^*$ as speed, the flux could be approximated with a common approximate Riemann solver when the approximate field is not C^0. Other items could be handled rather simply.

Finite volume spacial discretion is adopted(FVM). This program uses Roe's approximate Riemann solver[2], which is modified with entropy fix of Harten-Yee[3] for numerical flux. This program conducts 2nd-order reconstruction with Barth-Jesperson limiter[4].

The boundary conditions are built basically in the form of virtual cells. Slip-wall boundary and non-reflecting boundary are implemented. A pressure inlet/outlet boundary is also implemented , but its robustness is yet to be improved.

Time discretion in fluid part is a implicit Euler scheme.

The mesh considering FVM is completely composed of tetrahedra.

Structural Methods

Solid part is conducted with classic linear finite element method(FEM). Using symbols form a text book[5], a linear (small displacement) elastic dynamic problem is discretized as:

$$M\ddot{a} + C\dot{a} + Ka = F$$

Where a is the vector of nodal DOFs, defined as nodal displacement components. The matrices above are distributed in the finite elements as below:

$$K^e = \int_{\Omega e} B^{eT} D B^e dV, \quad M^e = \int_{\Omega e} N^{eT} \rho N^e dV,$$
$$F^e = \int_{\Omega e} f N^e dV + \int_{\partial\Omega e} p N^e d\Gamma$$

Where N^e is a row vector of shape functions(interpolating functions)on the nodes in the finite element, and B^e is a matrix representing linear contributions of the nodal DOFs to the strain field(which has 6 rows if using symmetric 3-D strain tensor, and DOF columns corresponding to DOFs). ρ is just density field, and f , p , are external forces of volume and surface.

In this project, currently only 4-node tetrahedra elements are implemented, which means both integrals in faces or volumes only need one interpolation point. If higher order elements are to be used, the interpolation method should be accurate enough.

A static solver can be found in the program, with LDL decomposition method or PCG method. For FSI purpose, a modal-truncation dynamic solver is implemented. Ignoring matrix C at first, then using a eigen problem:

$$K\Phi = M\Phi\Lambda$$

While demanding that:

$$\Phi^T M\Phi = I$$

Therefore:

$$\Phi^T K\Phi = \Lambda$$

Is diagonal.

The dynamic system becomes:

$$\ddot{x} + \Phi^T C\Phi\dot{x} + \Lambda x = G$$

Where:

$$G = \Phi^T F, \quad \Phi^T x = a$$

If the damping part $\Phi^T C\Phi$ is also diagonal, which is assumed in most CSD computations, the problem is decoupled for each DOF in eigenspace as:

$$\ddot{x_k} + d_k\dot{x_k} + \lambda_k x_k = g_k$$

Where d_k is a modal damper in the eigenspace, and $\sqrt{\lambda_i}$ are frequencies of each mode.

Therefore the discretized problem becomes a set of ODEs that can be separately solved. When the major concern is on vibration rather than delicate propagation of elastic waves, modes with higher frequencies contribute little to major features of the system, so instead of obtaining the whole eigentransformation, modes with n-smallest eigenvalues are solved. As a result, numerical methods solving modes with smallest eigenvalues can be applied here. This project uses the simplest one, the inverse power method.

FSI Methods

The FSI in this project is based on a simple explicit coupling method. The external force for structural model is obtained from the flow in the last coupling time step, and the flow obtains the moving mesh and interface from the structural movement in the last couping time step. If a general implicit method is to be expressed, the condition connecting both problems is the coordinated interface movement and passage of force on the interface, which defines where the overall system should converge. As this project only considers transient problem, only using a explicit method is acceptable.

Due to some problems in my meshing technique, the fluid and structural meshes don't adapt perfectly on the interface, but their mesh densities are similar. So when passing force and displacement, this project conducts simple interpolations between the surfaces(with KNN search). It should be noted that this is a common problem in more complex meshing(such as rotating meshes or overlapping grids), so there must be some better and more robust techniques to perform the interpolation than this project's implementation.

The last problem in FSI is to move the fluid mesh correctly. There are many interpolating methods that moves the mesh well, but this project uses a much simpler but rather time-consuming way. If you view the fluid part as a solid body, and apply displacement boundary conditions on all its outer faces, when the interface moves, the fluid mesh moves correspondingly. When the interface is attached, the movements inside the mesh is also almost continuous. The only problem that arises is that 'stress' distribution in the fluid mesh could be singular around some points of the interface, and is mostly concentrated near the interface. Under the assumption of linear structural response, the elements near the interface are likely to be overlapped or distorted. A simple way of solving this is to set higher value of modulus near the interface, forcing the strain to distribute farther from the interface. Also, setting higher shearing modulus could help abating the distortion.

To set the modulus distribution automatically, a distance field could be applied. However, in this project, as the elements are actually smaller near the interface, the volumes of interface are used to set the modulus.

A Brief Guide To Using This Code

Implementation and Framework

Unstructured Meshes

A good representation of the topology of an unstructured mesh, would be a diagram representing hierarchical connectivity between layers of mesh elements, which is frequently referred to as a *Hasse diagram*. For example, layers could be **vertices** $\Rightarrow$ **edges** $\Rightarrow$ **faces** $\Rightarrow$ **volumes** for a 3-D mesh, while in some certain problems, some intermediary layers could be omitted. The diagram is essentially a DAG whose nodes represent mesh elements. Therefore, the corresponding data structure would be basically a sparse adjacency matrix, or separate matrices between levels of layers. With this kind of information, one can easily find out, for example, the set of vertices sharing edge, face or element with a known vertex, or the set of volumes sharing common faces or nodes with a known volume. These queries naturally take $O(1)$ time with the help of the Hasse diagram.

Normally, the geometry of a mesh is represented as known coordinates of vertices.

When I started this project, it was actually based on some previous work with some rather dumb coding. From the view of a Hasse diagram, my unstructured mesh is only representing relations between volumes and vertices, which is alright with FEM coding, for neither volume and surface integrals nor nodal average in FEM need any more information. In 2nd order FEM, neighboring cells are need, for DOFs are stored with volumes and their adjacency is the same as cellular adjacency. I did not upgrade my data structure to represent one more layer (the face layer), but only added a neighbor searching procedure to the mesh loading method. Apparently, this kind of technique prevents direct extension to higher-order FVM with large stencils. Also, the adjacency representation is not generic, which only supports tetrahedral volumes(cells) with only 4 vertices, which hinders further implementation of higher-order nodal FEM.

In namespace **MeshBasic**, class **gridNeighbour** is defined, which records information about neighboring cells for a single cell, including some precalculated geometric values. Mean, class **TetraNodes** is also defined here to store actual volume to vertices info for a single cell, along with some precalculated geometric information. In the same namespace, I defined class **TetraMesh**, which is a general mesh holder. TetraMesh is able to load meshes and precalculate necessary topological and geometric information (which is primarily for FVM).

I've established some understandings of unstructured mesh, those are recorded below and irrelevant with the current project.

As I truly know little about unstructured mesh management and related data structures, I can only make some assumptions on a nearly 'perfect' implementation of an unstructured mesh framework. This framework should be able to represent a general mesh topology, irrelevant with mesh type, node distribution and dimension. More importantly, the framework should provide methods of obtaining adjacent mesh elements with constant time complexity (with degrees of the graph limited). Also, the frame work should allow users to attach arbitrary form of data onto mesh elements, to represent scalar, vector or tensor field within the mesh geometry, on vertices, faces or volumes. Meanwhile, the framework should enable users to abstract these fields into global vectors and tensors, in cooperation with global discrete operators like sparse matrices. Of course, process-scale parallel is a must, most probably implemented with MPI framework. Thread parallelism and CUDA support are also needed, but message distributing and communication are still the most complicated part of the program.

FEM Solver

In conventional finite-element method, or a common Galerkin method, the trial functions, being identical with the bases of the discrete solution, are considered to be C^0. Therefore the piecewise defined polynomial bases should maintain C^0 continuity on the interfaces of volumes (take 3-D for example). As a result, it is better to consider the discrete DOFs to be set on the vertices, and the bases are interpolation functions that satisfy a kronecker-delta property over the vertices. For this certain project, each volume is a tetrahedron with 4 vertices, and you can easily transform it linearly into a corner of a cartesian box, and using 3 cartesian axes you can easily define first-order bases functions for each vertex. Generally, the basis functions are defined in a normalized coordinate system as polynomials. As the transformation between the normalized space and the geometric space is generally not a linear mapping (basically as curved elements), generally the same set of basis functions are used to interpolate the mapping. Fundamentally, the actual bases are fractions rather than polynomials, but as flat-faced and near flat-faced elements are the majority, the orders of polynomial-based numerical integral are mostly decided with the situation of a linear spacial mapping.

Sadly, in this program you may find no actual numerical integral process, as the variable to integrate are derived form first derivatives of the bases, and they are actually constant in the volumes for linear bases. The only extra value to calculate for each volume is the determinant of the first partial derivatives (or Jacobian matrix) of the spacial mapping (which is also constant in the volume). The determinant is precisely the volume of the tetrahedron.

When discussing a linear elastic static problem, all should be calculated are the discrete linear operator (or stiffness matrix) and the load vector. Using some linear algebra techniques, the program first integrates a local matrix and a local load vector for each volume, and adds them to the global matrix and global load vector. This program applies triplet structure to the sparse matrix, by adding random entries first and doing a sort on the indices in the end, it needs $O(NlogN)$ time for assembly (mainly for sorting). Certainly, taking advantage of the (at least temporarily) static feature of mesh, a preallocation procedure could be added and one can easily build the matrix in CSR form. The CSR preallocation-and-fill paradigm takes $O(Nlog(Degree))$ time, where Degree denotes an average number of non-zeros in a row. This latter kind of implementation cannot be found in the project for it's a few times more complicated.

There are some other problems concerning boundary conditions, which are small modifications in the procedures above (although actually the load vector's non-zero entries are mainly caused by the boundary conditions) concerning reducing DOFs or boundary integration.

When the matrix and vector are produced, the only matters are to solve them. For a time-dependent problem, you need to derive a proper time discretion scheme; for a non-linear problem, you need to update the stiffness matrix at certain times. Nonetheless, the core problem here is to solve the linear system. I just threw them to Eigen's internal template implementation (While being a C++ template library, Eigen is also a great C++ interface and it supports wrapping of various external solvers). For a elliptic problem here with Galerkin discretion, matrices are mostly positive-definite and symmetric, so PCG would be rather favorable. When the problem is not too large, direct methods like LDU decomposition could also be viable (which is extremely helpful in eigenvalue problems for they require a lot of re-solving the matrix). Concerning the sparse eigenvalue problem (for modal analysis in FSI), I simply applied inverse power method, which seems to have some problems in high-rank mode convergence. Should consider switching to some more advanced techniques or just use a well-proven library.

The global vectors are represented as std::vector<T>, for I only apply shared-memory parallelism. All the matrix-related data structures and algorithms are in namespace **SparseMat**, and the **SparseMatGeR** class with relevant solving functions inside. **SolidMaterial** namespace and **ElasSet** class defines some general constitutional properties for the elastic problem. In **FEM** namespace class **FemT4Solver** is defined, which is derived from the **TetraMesh** class (which I now think should become a member rather than father... but inheritance means all the complex data in TetraMesh could be written the same way in TetraMesh member functions...). The **FemT4Solver** class, with **TetraMesh**, imports and manages mesh and problem definition, and assembles the stiffness matrix along with load vector, and provide interface to solve and output.

FVM Solver

Different from FEM, in finite volume method, conventionally the dofs are stored in each volume (and one copy per dof in each volume), which means when representing the distribution of a scalar in the volume, you need to refer to information from other volumes to obtain a polynomial with a order higher than zero. The program only implements a reconstruction of order one, representing the field with piecewise linear functions, for only neighboring volumes are needed. The variables may have discontinuities in the solution, like shock waves for example. A simple linear mapping from discrete DOFs to the reconstructed field definitely generates spurious oscillation near a discontinuity, with an approximation order higher than zero. Therefore a non-linear limiting method is necessary to help the reconstruction remain normal near a discontinuity. This program uses a simple Barth-Jesperson limiter in [4].

A typical FVM procedure first reconstructs a distribution in the volumes, where values on the volume interfaces are not necessarily the same, then numerical fluxes on the interfaces are calculated. A well-known and widely used method of calculating the hyperbolic part of the flux is using an local riemann solver which is mostly rather approximate. With a symmetric way of calculating the hyperbolic flux, generally negative numerical dissipation causes the discrete system to be unstable, while a more 'upwind' flux is often favorable with enough stability while dissipation and dispersion are acceptable. The local riemann solvers provide the only 'upwind' property of a FVM method for hyperbolic PDEs, for the reconstruction procedure does not know the convection direction (or the characteristic lines) and remains central. Without a upwind flux, artificial viscosity would be necessary to save the stability.

A first widely-used approximate riemann solver is Roe's, which is described at length in [2] and [3]. In maths, the Roe's approximate riemann solver is a function mapping the left and right states of a interface into its flux. In a transient problem, the time derivative is a integral of fluxes around a volume, and a corresponding steady problem is thus making time derivative zero. As this program currently only solve the Euler equations of gas dynamics, no other flux parts than the hyperbolic parts represented by Roe's scheme are included.

The discussions above only discretize the differential operators (or equivalent integral operators) of space. Commonly speaking, an implicit time marching method is more robust and stable, especially when discontinuities emerge. A general approach to solve the nonlinear equations in a implicit time step is something like a Newton iteration, which demands to solve a linear system about the equations' first derivative. Apparently, it takes some hard work to derive the Jacobian matrix for the Roe's approximate riemann solver, and solving the non-symmetric global matrix also takes much coding. Therefore, only explicit time marching and a simple fixed-point iteration is applied in a implicit Euler time scheme. The fixed-point implicit method proves to be not very robust when shockwave emerges. More practical implicit solvers are yet to be implemented.

Namespace **RiemannSolver** contains class **RoeSet** describing the properties of fluid and template function **RoeFluxT()** which calculates the output for a Roe's solver for Euler equations of gas dynamics. This template function can be theoretically used in any integer dimension, but 3-D form is the primary concern.

Namespace **UGSolver** contains class **RoeUGsolver** which is derived class from **TetraMesh** (again, it should be a member but the code feels okay and there should be no difference in performance). The class **RoeUGsolver** is responsible for mesh management (as a **TetraMesh**), and is able to import and process problem data, conduct reconstruction and time-evolution, and output its result. The time derivative calculation follows the modifications in ALE description.

Coupling Solver

Classes

Fluid

Structural

FSI

I/O File Convention

Case Building

Compiling

In theory, this project should be included as a header-only c++ library in your project, which means no actual building is needed. However, as the input/output and intermediate files are pretty tricky to produce, so main.cpp is provided for using. mainbkp.cpp provides other useful cases than in main.cpp. To compile, you simply use **g++** to compile the single main.cpp file, or use **make** like instructed below.

If you put all the functions in mainbkp.cpp into main.cpp to compile, when using mingw64 g++ in Windows, the compiler could exit abnormally declaring an error on insufficient resources concerning templates. I can't repeat this problem, please tell me in the issues why this could happen if you know.

The mainSG.cpp is for a 2D structural gird Euler equation gas dynamics solver, sharing some headers. Its case construction appears much more complex.

Make

```
make main.exe #for debug
make mainR.exe #for release
make mainSG.exe #for mainSG debug
make mainSGR.exe #for mainSG release
```

Notes

It was only after I STARTED this document when I find my English being rather amateur, even in my own domains... I even searched for a proper antonym for 'proficient' for use in the previous sentence... So please forgive me if any expression in my text seems strange or erroneous.

References

[1] P, Le, Tallec, et al. Fluid structure interaction with large structural displacements[J]. *Computer Methods in Applied Mechanics and Engineering*, 2001, 190(24-25):3039-3067.

[2] Roe P L . Approximate Riemann solvers, parameter vectors, and difference schemes[J]. *Journal of Computational Physics*, 1981, 43(2):357-372.

[3] Yee H C . Upwind and symmetric shock-capturing schemes. 1987.

[4] Barth T J , Jespersen D C . The design and application of upwind schemes on unstructured meshes[J]. AIAA Aerospace Sciences Meeting, 1989, 0366(13).

[5] 王勖成. 有限单元法[M]. 清华大学出版社, 2003.