SOMA Documentation

Javier Villarreal

09/27/2021

1 Introduction

The purpose of this document is to keep track of the project to port the C++ SOMA code to Fortran.

2 C++ Code Architecture

The C++ code is broadly arranged in the following way:

1. [**Complete**] Reads in simulation parameter text files and defines some constants.
2. [**Complete**] Constructs the objects that hold the variables: `Domain`, `SimFluid`, and `Approximator`
3. [**Compete**] Reads in geometry-based data files to appropriate object variables.
4. [**Redundant**] Enforces boundary conditions and calculates derivatives on initial data.
5. Defines parameters for the genetic algorithm optimization code.
6. (optional) Reads initial values from text files and re-calculates BC's and derivatives.
7. Prints initial data to output text files.
8. Loops over time steps. (SOMA proper)
9. Prints latest values to output text files.

In the SOMA step, the code splits into one of two modes, explicit (using Runge-Kutta) or implicit (using RBF addition). Different mechanisms within the code based on convergence criteria or iteration counts switch the code between one mode or the other. Each one will be explained in its own section.

The Fortran implementation will slightly modify the code architecture. The optional step of reading flow conditions from a previous run overwrites the boundary conditions and derivatives, and they have to be calculated again. Instead, the option to read in data will happen *before* those steps, to avoid redundancy.

3 Pre-Processing

3.1 Input Data

Geometry data used to run the simulation is currently created on a case by case basis using Matlab codes to generate .txt files that are then read in by SOMA. SimulationValues.txt defines the configuration of the simulation, and Sizes.txt contains metadata used to properly allocate arrays.

- SimulationValues.txt (Mach, AOA, Re)
- Sizes.txt (# of domain, body, farfield, cloud, ghost, extrapolation, total nodes)

The geometry data files themselves are:

- x,y.txt (node coordinates)
- DX,DY.txt (DQ coefficients)
- EC.txt (extrapolation coefficients)
- Jd,Jb,Jf.txt (domain, body, farfield node indices)
- nxb,nyb.txt (body node unit normal vectors)
- nxf,nyf.txt (farfield node unit normal vectors)
- s11,s12,s21,s22.txt (Flow tangency matrices)

Most of the data files are read by the code into variables holding "carbon copies" of the data, with a few exceptions:

- The data in SimulationValues and Sizes are stored in individual variables for each number, rather than vectors.
- The total number of nodes is equal to the number of domain nodes plus the number of farfield nodes plus the number of ghost points mutiplied by the number of body nodes. The body nodes themselves are already included within the domain nodes.
- The normal vectors, which are created in separate files by components (e.g. nxb and nyb are the x- and y- components of the normal vectors), are read by the code into arrays containing the entire vectors, `b_normal` and `f_normal`, where each row corresponds to a node and the columns correspond to [x y].
- The matrices used to enforce flow tangency boundary conditions are stored in the array `slip`, where each row corresponds to a node, and the columns correspond to [s11 s12 s21 s22]

Note: The structure of the normals and matrices were defined to reflect the text files to maintain readability, but it means the code will loop over those matrices as in

```
do node=1,n
    use array(node,:)
end do
```

which is slower than looping over the right-most index due to how Fortran stores data in memory. Eventually, the text files and arrays should be restructured for optimization.

3.2 Data architecture

[LEGACY] C++ code implementation

The current task is to create all the variables necessary to hold the simulation data. So far, the only existing variables are those used to read in text file data.

Most of the C++ variables were defined as members in classes. The classes are

- Data
- FlowVars
- InternalBoundary
- Approximator
- Time
- ExternalBoundary
- SimFluid
- Domain
- GeneticAlgorithm

Only some classes are instantiated in the main code, though, with some only existing within other classes as members. The data structure is roughly

```
Data is never instantiated, serves as an "abstract" class
    Q[4] => air.[r,u,v,e]
Domain simRegion
    InternalBoundary body
    ExternalBoundary edge
SimFluid air
    FlowVars r,u,v,e
    Time t
Approximator solver
    GeneticAlgorithm ga
    dom => simregion
    air => air
```

where `=>` denotes pointer variables

A full listing of the classes and their respective class member variables (not member functions) can be found in `C++_classes.txt`. Of course, there are also variables defined within the scope of certain functions. Those are listed in `C++_functions.txt`.

Fortran implementation

The most important flow variables to fully describe compressible fluid flow are the primitive variables: ρ (density), u (x-velocity), v (y-velocity), E_t (total energy), and some dependent variables, like p (pressure), T (temperature), μ (dynamics viscosity), k (thermal conductivity). Because the code is non-dimensionalized, though, thermal diffusivity is instead represented by the Prandtl number, which is constant for calorically perfect gases.

All the variables listed above have been defined, but rather than using a complex structure like that found in the C++ code, the variables and their derivatives are currently defined in simple arrays. In addition, an array was included for shear stress tensors that was not previously recorded in the C++ code.

3.3 Initialization

For the most part, the initial values of the simulation are defined in the Fortran code config.f90, (with the exceptions of Mach number, angle of attack, and Reynolds number). As far as the initial values of the flow variables, the initial total velocity is given, and the initial density and temperature. From those, the initial values for total energy, pressure, and viscosity are derived using the equation of state for calorically perfect gases and Sutherland's law for viscosity.

4 Boundary Conditions

Under the C++ code's OOP architecture, boundary enforcement was divided between enforcement on the surface boundary (cylinder, airfoil, etc), and the farfield boundaries. The Fortran code dose not use

such a distinction, so all the boundary condition subroutines are in one module. Boundary conditions are enforced directly on the flow variables, without the use of any special structures or classes.

Here is the general layout of boundary condition algorithms used in the C++ code:

- "Internal Boundary" (surface)
  - Set velocity to zero if viscous flow (no-slip condition)
  - Set density and energy at ghost nodes equal to one-off nodes
  - Set velocities at ghost nodes equal to velocities at one-off nodes if vicsous; and
  - Use slip matrix to set velocities at ghost nodes if inviscid.
- "External Boundary" (farfield)
  - Either set variable to predetermined farfield value; or
  - Extrapolate from nearby values, depending on whether flow is sub- or supersonic and it's an inlet or outlet.

An in-depth explanation of how these bounary conditions are enforced is described in the thesis.