MFC: User's guide

S. H. Bryngelson, K. Schmidmayer, V. Coralic, J. C. Meng, T. Colonius

Division of Engineering and Applied Science,
California Institute of Technology

K. Maeda

Center for Turbulence Research,
Stanford University

Change log:

- Added section 7.2, including naming conventions for serial output files (and equation numbers)
- Spencer: Removed most of Install section to defer to `README.md`, which we keep up to date.

Contents

1 Preliminary

This document provides instructions for installation and configuration of MFC (Multi-component flow code), a CFD-framework for simulation of compressible, multi-component fluid flows. The reader is also pointed to a separate journal publication, Bryngelson et al. (2019) for a scientific backgrounds and overview of MFC. MFC is licensed under the GNU GPLv3.

2 Source code

2.1 Documentation

The source code, located in the `src/` directory, contains three components: `pre_process/`, `simulation`, and `post_process`. These codes are all documented via Doxygen, which can be located at `https://mfc-caltech.github.io`.

2.2 Naming conventions

The Fortran files `*.f90` in the source code directories utilize the naming conventions found in table 1.

Variable	Description
*_sf	Scalar field
*_vf	Vector field
*_pp	Physical parameters
*[K,L,R]	WENO-reconstructed cell averages
*_avg	Roe/arithmetic average
*_cb	Cell boundary
*_cc	Cell center
*_cbc	Characteristic boundary conditions
*_cons_*	Conservative
*_prim_*	Primitive
gm_*	Gradient magnitude
*_ndqp	Normal direction Gaussian quadrature points
*_qp	Cell-interior Gaussian quadrature points
un_*	Unit-normal
dgm_*	Curvature (derived gradient magnitude)
*_icpp	Initial condition patch parameters
*_idx	Indices of first and last (object)
cont_*	Continuity equations
mom_*	Momentum equations
E_*	Total energy equation
adv_*	Volume fraction equations
*_id	Identifier
dflt_*	Default value
orig_*	Original variable
q_*	Cell-average conservative or primitive variables
q[L,R]_*	Left[right] WENO-reconstructed cell-boundary values
dq_*	First-order spatial derivatives
*_rs	Riemann solver variables
*_src	Source terms
*_gsrc	Geometric source terms
[lo,hi]_*	Related to TVD options
*_IC	Inter-cell
*_ts	Time-stage (for time-stepper algorithm)
wa_*	WENO average
crv_*	Geometrical curvature of the material interfaces

Table 1: Code variables

3 Installation

Please see the `README.md` file.

4 Configure Python master script

MFC uses a python-based interface. Python master script, `/master_script/m_python_proxy.py`, contains a dictionary of input parameters and scripting function that interconnect procedures' execution. Low-level access to the portable batch system (PBS) and Slurm Workload Manager (Slurm) is also included through additional dictionary definitions and provides the MFC with parallel run capabilities.

The user must prepare a separate python input file (`input.py`) for each case. The master script receives the MFC component name (`pre_process/simulation/post_process`), the case dictionary, the MFC location, and the engine configuration from the input file. The scripting function (located in m_python_proxy.py) compiles the source code for the selected component (if not completed already) and writes the component's Fortran input file. A batch file will also be generated if the parallel engine is used. If this is the case, the component's executable will be executed via the submitted PBS/Slurm batch file. Otherwise, it runs the executes the component directly in the command-line.

Configurations of the job scheduler are often enviornment-dependent, and thus should be specified in the scripting function f_create_batch_file.py in by the user when the MFC is installed in a new environment. Examples are shown below, although configurations can vary and should be modified as needed.

PBS example:

```
def f_create_batch_file(comp_name, case_dict, mfc_dir): # ----------------------

    # Enabling access to the PBS dictionary
    global pbs_dict


    # Setting the location of the batch file
    file_loc = comp_name + '.sh'


    # Opening and obtaining a handle for it
    file_id = open(file_loc, 'w')


    # Populating Batch File ==================================================
    file_id.write( \
                                                                           \
        # Script interpreter
        '#!/bin/sh' + '\n' \
                                                                           \
        # Account to be charged for the job:
        '#PBS -A [account name]' + '\n' \
                                                                           \
        # Name of the queue to which the job should be submitted:
        '#PBS -q ' + str(pbs_dict['queue']) + '\n' \
                                                                           \
        # Name of the job to be submitted to the scheduler:
        '#PBS -N ' + comp_name + '\n' \
                                                                           \
        # Node(s) and processor(s) per node (ppn) for job:
        '#PBS -l nodes=' + str(pbs_dict['nodes']) \
               + ':ppn=' + str(pbs_dict[ 'ppn' ]) + '\n' \
                                                                           \
        # Maximum amount of time to commit to the execution of the job:
        '#PBS -l walltime=' + str(pbs_dict['walltime']) + '\n' \
                                                                           \
        # Declare the job rerunable (y) or non-rerunable (n)
        '#PBS -r n' + '\n' \
                                                                           \
```

```
    # Output standard output and error in a single file
    '#PBS -j oe'
                                                                  \
    # Notify by email when job begins (b), aborts (a), and/or ends (e):
    '#PBS -m bae' + '\n' \
    '#PBS -M ' + str(pbs_dict['mail_list']) + '\n' \
                                                                  \
    # Total number of processor(s) allocated for job execution
    'num_procs=$(cat $PBS_NODEFILE | wc -l)' + '\n' \
                                                                  \
    # Moving to the case directory
    'cd $PBS_O_WORKDIR' + '\n' \
                                                                  \
    # Setting up the output file's header information:
    'echo MFC ' + basename(getcwd()) \
                         + ': $PBS_JOBNAME.o${PBS_JOBID:0:7}' + '\n' \
    'echo Description: $PBS_JOBID executed on $num_procs ' \
               + 'processor\'(s)\'. The' + '\n' + 'echo ' \
               + '\' \' command-line output ' \
               + 'information may be found below.' + '\n' \
    'echo Start-date: `date +%D`' + '\n' \
    'echo Start-time: `date +%T`' + '\n' \
    'echo' + '\n' + 'echo' + '\n' \
    'echo \'================================ Terminal Output ' \
        + '================================\'' + '\n' + 'echo' + '\n' \
                                                                  \
    # Starting the timer for the job execution
     't_start=$(date +%s)' + '\n' \
                                                                  \
    # Executing job:
    'mpirun ' \
                          + mfc_dir + '/' + comp_name \
                          + '_code' + '/' + comp_name + '\n' \
    # Stopping the timer for the job
    't_stop=$(date +%s)' + '\n' + 'echo' + '\n' \
                                                                  \
    # Setting up the PBS output file's footer information
    'echo \'================================================' \
        + '================================\'' + '\n' \
    'echo' + '\n' + 'echo' + '\n' \
    'echo End-date: `date +%D`' + '\n' \
    'echo End-time: `date +%T`' + '\n' + 'echo' + '\n' \
    'echo Total-time: $(expr $t_stop - $t_start)s' + '\n' \
                                                                  \
    # Removing the input file
    'rm -f ' + comp_name + '.inp' + '\n' \
                                                                  \
    # Removing the batch file
    'rm -f ' + comp_name + '.sh' )
# END: Populating Batch File ==============================================


# Closing the batch file
file_id.close()
```

```
    # Giving the batch file the permission to be executed
    cmd_status = Popen('chmod +x ' + comp_name + '.sh', shell=True, stdout=PIPE)
    output, errors = cmd_status.communicate()
# END: def f_create_batch_file ------------------------------------------------
```

Slurm example:

```
def f_create_batch_file(comp_name, case_dict, mfc_dir): # ----------------------

    # Enabling access to the PBS dictionary
    global pbs_dict


    # Setting the location of the batch file
    file_loc = comp_name + '.sh'


    # Opening and obtaining a handle for it
    file_id = open(file_loc, 'w')


    # Populating Batch File ====================================================
    file_id.write( \
                                                                            \
        # Script interpreter
        '#!/bin/sh' + '\n' \
                                                                            \
        # Account to be charged for the job:
        '#SBATCH -A [account name]' + '\n' \
                                                                            \
        # Name of the queue to which the job should be submitted:
        '#SBATCH -p ' + str(pbs_dict['queue']) + '\n' \
                                                                            \
        # Name of the job to be submitted to the scheduler:
        '#SBATCH -J ' + comp_name + '\n' \
                                                                            \
        # Node(s) and processor(s) per node (ppn) for job:
        '#SBATCH --nodes=' + str(pbs_dict['nodes']) + '\n' \
        '#SBATCH --ntasks-per-node=' + str(pbs_dict['ppn']) + '\n' \
                                                                            \
        # Constrain allocated nodes to single rack for best code efficiency:
        '#SBATCH --switches=1' + '\n' \
                                                                            \
        # Maximum amount of time to commit to the execution of the job:
        '#SBATCH -t ' + str(pbs_dict['walltime']) + '\n' \
                                                                            \
        # Output standard output and error in a single file
        '#SBATCH -o ' + comp_name + '.o%j' + '\n' \
        '#SBATCH -e ' + comp_name + '.o%j' + '\n' \
                                                                            \
```

```
        # Notify by email when job begins (b), aborts (a), and/or ends (e):
        '#SBATCH --mail-type=all' + '\n' \
        '#SBATCH --mail-user=' + str(pbs_dict['mail_list']) + '\n' \
                                                                        \
        # Setting up the output file's header information:
        'echo MFC ' + basename(getcwd()) \
                            + ': $SLURM_JOB_NAME.o$SLURM_JOB_ID' + '\n' \
        'echo Description: $SLURM_JOB_ID executed on $SLURM_NTASKS ' \

                       + 'processor\'(s)\'. The' + '\n' + 'echo ' \
                       + '\' \' command-line output ' \
                       + 'information may be found below.' + '\n' \
        'echo Start-date: `date +%D`' + '\n' \
        'echo Start-time: `date +%T`' + '\n' \
        'echo' + '\n' + 'echo' + '\n' \
        'echo \'================================ Terminal Output ' \
            + '===============================\'' + '\n' + 'echo' + '\n' \
                                                                        \
        # Starting the timer for the job execution
         't_start=$(date +%s)' + '\n' \
                                                                        \
        # Executing job:
        'mpirun ' \
                                + mfc_dir + '/' + comp_name \
                                + '_code' + '/' + comp_name + '\n' \
        # Stopping the timer for the job
        't_stop=$(date +%s)' + '\n' + 'echo' + '\n' \
                                                                        \
        # Setting up the PBS output file's footer information
        'echo \'================================================' \
            + '===============================\'' + '\n' \
        'echo' + '\n' + 'echo' + '\n' \
        'echo End-date: `date +%D`' + '\n' \
        'echo End-time: `date +%T`' + '\n' + 'echo' + '\n' \
        'echo Total-time: $(expr $t_stop - $t_start)s' + '\n' \
                                                                        \
        # Removing the input file
        'rm -f ' + comp_name + '.inp' + '\n' \
                                                                        \
        # Removing the batch file
        'rm -f ' + comp_name + '.sh' )
    # END: Populating Batch File ============================================

    # Closing the batch file
    file_id.close()

    # Giving the batch file the permission to be executed
    cmd_status = Popen('chmod +x ' + comp_name + '.sh', shell=True, stdout=PIPE)
    output, errors = cmd_status.communicate()
# END: def f_create_batch_file ------------------------------------------------
```

5 How to run

MFC can be run by navigating to a case directory and executing the appropriate Python input file. Example Python input files can be found in the `example_cases` case directories and they are called `input.py`. Their contents, and a guide to filling them out, are the subject of section 6. The MFC can be executed as

```
# python input.py pre_process
```

This will generate the `restart_data` directory that contains initial flow field and grid data files in a binary data format. Then

```
# python input.py simulation
```

will read the data files and execute the flow solver. The last (optional) step is to post treat the binary data files and output Silo-HDF5 database for the flow variables via

```
# python input.py post_process
```

This will generate `silo_hdf5` that contains the database. This requires installation of Silo and HDF5, as described in section **??**.

6 Python input file

Python input file `input.py` defines dependencies and logistics, and input parameters for each simulation case. In this section, details of the input file and how to edit it are described. The user can also leverage the example input files as necessary.

6.1 Dependencies and Logistics

To specify dependencies and logistics, users are required to specify the directory of MFC and computational engine. If the parallel engine is chosen, the python input file automatically generates a batch job-script file and submits it to a job-schedule in the given environment.

Example of the Dependencies and Logistics:

```
#!/usr/bin/env python3

# Dependencies and Logistics ==========================

# Command to navigate between directories
from os import chdir

# Command to acquire directory path
from os.path import dirname

# Command to acquire script name and module search path
from sys import argv, path

# Navigating to script directory
if len(dirname(argv[0])) != 0: chdir(dirname(argv[0]))

# Adding master_scripts directory to module search path
mfc_dir = '[MFC directory]'; path[:0] = [mfc_dir + '/master_scripts']
```

```
# Command to execute the MFC components
from m_python_proxy import f_execute_mfc_component

# Serial or parallel computational engine
engine = 'parallel'
#engine = 'serial'
# ========================================================
```

MFC is optimized to work with a parallel engine. Nevertheless, if serial engine is specified, MFC can be executed without using a job scheduler.

6.2 Input parameters

There are multiple sets of parameters that must be specified in the python input file:

1. Job scheduler parameters (see table 2).
2. Computational domain parameters (see table 3).
3. Patch parameters (see table 4).
4. Fluid material's parameters (see table 5)
5. Simulation algorithm parameters (see table 6).
6. Formatted database and structure parameters (see table 7).
7. (Optional) Acoustic source parameters (see table 8).
8. (Optional) Ensemble-averaged bubble model parameters (see table 9).

Items 7 and 8 are optional sets of parameters that activate the acoustic source model and ensemble-averaged bubble model, respectively. Definition of the parameters is described in the following subsections.

6.2.1 Job-scheduler parameters

Parameter	Type	Description
`case_dir`	String	Case script directory
`run_time_info`	Logical	Output run-time information
`nodes`	Integer	Number of nodes
`ppn`	Integer	Number of cores
`queue`	String	Queue name
`walltime`	Time	Maximum run time
`mail_list`	String	Information sent to this email

Table 2: Job-scheduler parameters

Table 2 lists the job-scheduler parameters. The parameters are used to configure the batch file that is submitted to a parallel job scheduler.

`case_dir` specifies the directory where the python input file is located.

`run_time_info` generates a text file that includes run-time information including the CFL number(s) at each time-step.

nodes and ppn specify the number of node and the number of cores per node used in parallel run. The total number of processors used is thus given as nodes $\times$ ppn.

queue and walltime define the queue name and the maximum run time of the job. They must be consistent with specific queue rules that are defined in a computer cluster/environment in that MFC is installed.

6.2.2 Computational domain parameters

Parameter	Type	Description
x[y,z]_domain%beg[end]	Real	Beginning [ending] of the $x[y,z]$-direction domain
stretch_x[y,z]	Logical	Stretching of the mesh in the $x[y,z]$-direction
a_x[y,z]	Real	Rate at which the grid is stretched in the $x[y,z]$-direction
x[y,z]_a	Real	Beginning of the stretching in the negative $x[y,z]$-direction
x[y,z]_b	Real	Beginning of the stretching in the positive $x[y,z]$-direction
cyl_coord	Logical	Cylindrical coordinates (2D: Axisymmetric, 3D: Cylindrical)
m	Integer	Number of grid cells in the x-coordinate direction
n	Integer	Number of grid cells in the y-coordinate direction
p	Integer	Number of grid cells in the z-coordinate direction
dt	Real	Time step size
t_step_start	Integer	Simulation starting time step
t_step_stop	Integer	Simulation stopping time step
t_step_save	Integer	Frequency to output data

Table 3: Computational domain parameters

Table 3 lists the computational domain parameters. The parameters define the boundaries of the spatial and temporal domains, and their discritization that are used in simulation.

x[y,z]_domain%beg[end] define the spatial domain in $x - y - z$ Cartesian coordinates: $x \in$[x_domain%beg, x_domain%end]; $y \in$[y_domain%beg, y_domain%end]; $z \in$[z_domain%beg, z_domain%end].

m, n, and p define the number of finite volume cells that uniformly discritize the domain along the x, y, and z axes, respectively. Note that the actual number of cells in each coordinate axis is given as $m[n,p]+1$. For example, m=n=p= 499 discretizes the domain into 500^3 cells. When the simulation is 2D/axi-symmetric or 1D, it requires that p= 0 or p=n= 0, respectively.

stretch_x[y,z] activates grid stretching in the $x[y,z]$ directions. The grid is gradually stretched such that the domain boundaries are pushed away from the origin along a specified axis.

a_x[y,z], x[y,z]_a, and x[y,z]_b are parameters that define the grid stretching function. When grid stretching along the x axis is considered, the stretching function is given as:

$$x_{cb,stretch} = x_{cb} + \frac{x_{cb}}{a_x}\Bigg[\log[\cosh(\frac{\texttt{a\_x}(x_{cb}-\texttt{x\_a})}{L})] + \log[\cosh(\frac{\texttt{a\_x}(x_{cb}-\texttt{x\_b})}{L})] - 2\log[\cosh(\frac{\texttt{a\_x}(\texttt{x\_b}-\texttt{x\_a})}{2L})]\Bigg], \quad (1)$$

where x_{cb} and $x_{cb,stretch}$ are the coordinates of a cell boundary at the original and stretched domains, respectively. L is the domain length along the x axis: $L =$x_domain%end$-$x_domain%beg. Crudely speaking, x_a and x_b define the coordinates at which the grid begins to get stretched in the negative and positive directions along the x axis, respectively. a_x defines the smoothness of the stretching. Stretching along the y and z axes follows the same logistics. Optimal choice of the parameters for grid stretching is case-dependent and left to the user.

`cyl_coord` activates cylindrical coordinates. The domain is defined in x-y-z cylindrical coordinates, instead of Cartesian coordinates. Domain discritization is accordingly conducted along the axes of cylindrical coordinates. Wnen `p`$=0$, the domain is defined on x-y axi-symmetric coordinates. In both Coordinates, mesh stretching can be defined along the x- and y-axes. MPI topology is automatically optimized to maximize the parallel efficiency for given choice of coordinate systems. Meng (2016)

`dt` specifies the constant time step size that is used in simulation. The value of `dt` needs to be sufficiently small such that the Courant-Friedrichs-Lewy (CFL) condition is satisfied.

`t_step_start` and `t_step_end` define the time steps at which simulation starts and ends, respectively. `t_step_save` is the time step interval for data output during simulation. To newly start simulation, set `t_step_start`$=0$. To restart simulation from k-th time step, set `t_step_start`$=k$.

6.2.3 Patch parameters

Parameter	Type	Description
`num_patches`	Integer	Number of initial condition geometric patches
`num_fluids`	Integer	Number of fluids/components present in the flow
`geometry`*	Integer	Geometry configuration of the patch (see table 11)
`alter_patch(i)`*	Logical	Alter the `i`-th patch
`x[y,z]_centroid`*	Real	Centroid of the applied geometry in the x[y,z]-direction
`length_x[y,z]`*	Real	Length, if applicable, in the x[y,z]-direction
`radius`*	Real	Radius, if applicable, of the applied geometry
`smoothen`*	Logical	Smoothen the applied patch
`smooth_patch_id`*	Integer	A patch with which the applied patch is smoothened
`smooth_coeff`*	Real	Smoothen coefficient
`alpha(i)`*	Real	Volume fraction of fluid `i`
`alpha_rho(i)`*	Real	Partial density of fluid `i`
`pres`*	Real	Pressure
`vel(i)`*	Real	Velocity in direction `i`

Table 4: Patch parameters.

*These parameters should be prepended with `patch_icpp(j)%` where `j` is the patch index.

Table 4 lists the patch parameters. The parameters define the geometries and physical parameters of fluid components (patch) in the domain at initial condition. Note that the domain must be fully filled with patche(s). The code outputs error messages when an empty region is left in the domain.

`num_patches` defines the total number of patches defined in the domain. The number has to be a positive integer.

`num_fluids` defines the total number of fluids defined in each of the patches. The number has to be a positive integer.

`patch_icpp(j)%geometry` defines the type of geometry of j-th patch by using an integer from 1 to 13. Definition of the patch type for each integer is listed in table 11).

`x[y,z]_centroid`, `length_x[y,z]`, and/or `radius` are used to uniquely define the geometry of the patch with given type. Requisite combinations of the parameters for each type can be found in is listed in table 11).

patch_icpp(j)%alter_patch(i) activates alternation of patch(i) with patch(j). For instance, in a 2D simulation, when a cylindrical patch(2) is immersed in a rectangular patch(1),
patch_icpp(1)%geometry= 3;
patch_icpp(2)%geometry= 2;
patch_icpp(2)%alter_patch(1)=TRUE.

smoothen activates smoothening of the boundary of the patch that alters the existing patch. When smoothening occurs, fluids of the two patches are mixed in the region of the boundary. For instance, in the aforementioned case of the cylindrical patch immersed in the rectangular patch, smoothening occurs when patch_icpp(2)smoothen=TRUE. smooth_coeff controls the thickness of the region of smoothening (sharpness of the mixture region). The default value of smooth_coeff is unity. The region of smoothening is thickened with decreasing the value. Optimal choice of the value of smooth_coeff is case-dependent and left to the user.

patch_icpp(j)alpha(i), patch_icpp(j)alpha_rho(i), patch_icpp(j)pres, and patch_icpp(j)vel(i) define for j-th patch the void fraction of fluid(i), partial density of fluid(i), the pressure, and the velocity in the i-th coordinate direction. These physical parameters must be consistent with fluid material's parameters defined in the next subsection. See also adv_alphan in table 6.

6.2.4 Fluid material's parameters

Parameter	Type	Description
gamma	Real	Stiffened-gas parameter Γ of fluid
pi_inf	Real	Stiffened-gas parameter Π_∞ of fluid
Re(1)*	Real	Shear viscosity of fluid
Re(2)*	Real	Volume viscosity of fluid

Table 5: Fluid material's parameters. All parameters should be prepended with fluid_pp(i)% where i is the fluid index.

*Parameters that work only with model_eqns=2.

Table 5 lists the fluid material's parameters. The parameters define material's property of compressible fluids that are used in simulation.

fluid_pp(i)%gamma and fluid_pp(i)%pi_inf define Γ and Π as parameters of i-th fluid that are used in stiffened gas equation of state.

fluid_pp(i)%Re(1) and fluid_pp(i)%Re(2) define the shear and volume viscosities of i-th fluid, respectively. When these parameters are undefined, fluids are treated as inviscid. Details of implementation of viscosity in MFC can be found in Coralic (2015).

6.2.5 Simulation algorithm parameters

Parameter	Type	Description
bc_x[y,z]%beg[end]	Integer	Beginning [ending] boundary condition in the x[y,z]-direction (negative integer, see table 10)
model_eqns	Integer	Multicomponent model: [1] Γ/Π_∞; [2] 5-equation; [3] 6-equation
alt_soundspeed*	Logical	Alternate sound speed and $K\nabla\cdot\boldsymbol{u}$ for 5-equation model
adv_alphan	Logical	Equations for all N volume fractions (instead of $N-1$)
mpp_lim	Logical	Mixture physical parameters limits
mixture_err	Logical	Mixture properties correction
time_stepper	Integer	Runge–Kutta order [1–5]
weno_vars	Integer	WENO reconstruction on [1] Conservative; [2] Primitive variables
weno_order	Integer	WENO order [1,3,5]
weno_eps	Real	WENO perturbation (avoid division by zero)
char_decomp	Logical	Characteristic decomposition
mapped_weno	Logical	WENO with mapping of nonlinear weights
null_weights	Logical	Null WENO weights at boundaries
mp_weno	Logical	Monotonicity preserving WENO
riemann_solver	Integer	Riemann solver algorithm: [1] HLL*; [2] HLLC; [3] Exact*
avg_state	Integer	Averaged state evaluation method: [1] Roe averagen*; [2] Arithmetic mean
wave_speeds	Integer	Wave-speed estimation: [1] Direct (Batten et al. 1997); [2] Pressure-velocity* (Toro 1999)
commute_err†*	Logical	Commutative error correction via cell-interior quadrature
split_err†*	Logical	Dimensional splitting error correction via cell-boundary
reg_eps*	Real	Interface thickness parameter for regularization terms
flux_lim*	Integer	Choice of flux limiter: [1] Minmod; [2] MC; [3] Ospre; [4] Superbee; [5] Sweby; [6] van Albada; [7] van Leer.
tvd_rhs_flux*	Logical	Apply TVD flux limiter to intercell fluxes outside Riemann solver
tvd_riemann_flux*	Logical	Apply TVD flux limiter to cell edges inside Riemann solver
tvd_wave_speeds*	Logical	TVD wave-speeds for flux computation inside Riemann solver

Table 6: Simulation algorithm parameters.

*Options that work only with model_eqns= 2.
†Options that work only with cyl_coord=FALSE.

Table 6 lists simulation algorithm parameters. The parameters are used to specify options in algorithms that are used to integrate the governing equations of the multi-component flow based on the initial condition. Models and assumptions that are used to formulate and discritize the governing equations are described in Bryngelson et al. (2019). Details of the simulation algorithms and implementation of the WENO scheme can be found in Coralic (2015).

bc_x[y,z]%beg[end] specifies the boundary conditions at the beginning and the end of domain boundaries in each coordinate direction by a negative integer from -1 through -12. See table 10 for details.

model_eqns specifies the choice of the multi-component model that is used to formulate the dynamics of the flow using integers from 1 through 3. model_eqns=1, 2, and 3 correspond to Γ-Π_∞ model (Johnsen, 2008), 5-equation model (Allaire et al., 2002), and 6-equation model (Saurel et al., 2009), respectively. The difference of the two models is assessed by (Schmidmayer et al., 2019). Note that some code parameters are only compatible with 5-equation model.

alt_soundspeed activates the source term in the advection equations for the volume fractions, $K\nabla \cdot \underline{\mathrm{u}}$, that regularizes the speed of sound in the mixture region when the 5-equation model is used. The effect and use of the source term are assessed by Schmidmayer et al. (2019).

adv_alphan activates the advection equations of all the components of fluid. If this parameter is set false, the void fraction of N-th component is computed as the residual of the void fraction of the other components at each cell:

$$\alpha_N = 1 - \sum_{i=1}^{N-1} \alpha_i, \tag{2}$$

where α_i is the void fraction of i-th component. When a single-component flow is simulated, it requires that adv_alphan=TRUE.

mpp_lim activates correction of solutions to avoid a negative void fraction of each component in each grid cell, such that $\alpha_i > \varepsilon$ is satisfied at each time step.

mixture_err activates correction of solutions to avoid imaginary speed of sound at each grid cell.

time_stepper specifies the order of the Runge-Kutta (RK) time integration scheme that is used for temporal integration in simulation, from the 1st to 5th order by corresponding integer. Note that time_stepper=3 specifies the total variation diminishing (TVD), third order RK scheme (Gottlieb and Shu, 1998).

weno_vars specifies the choice of state variables that are reconstructed using a WENO scheme by an integer of 1 or 2. weno_vars=1 and 2 correspond to conservative variables and primitive variables, respectively.

weno_order specifies the order of WENO scheme that is used for spatial reconstruction of variables by an integer of 1, 3, and 5, that correspond to the 1st, 3rd, and 5th order, respectively.

weno_eps specifies the lower bound of the WENO nonlinear weights. Practically, weno_eps$< 10^{-6}$ is used.

char_decomp activates projection of the state variables onto characteristic fields prior to WENO reconstruction.

mapped_weno activates mapping of the nonlinear WENO weights to the more accurate nonlinear weights in order to reinstate the optimal order of accuracy of the reconstruction in the proximity of critical points (Henrick et al., 2005).

null_weights activates nullification of the nonlinear WENO weights at the buffer regions outside the domain boundaries when the Riemann extrapolation boundary condition is specified (bc_x[y,z]%beg[end]$= -4$).

mp_weno activates monotonicity preservation in the WENO reconstruction (MPWENO) such that the values of reconstructed variables do not reside outside the range spanned by WENO stencil (Balsara and Shu, 2000; Suresh and Huynh, 1997).

riemann_solver specifies the choice of the Riemann solver that is used in simulation by an integer from 1 through 3. riemann_solver=1,2, and 3 correspond to HLL, HLLC, and Exact Riemann solver, respectively (Toro, 2013).

avg_state specifies the choice of the method to compute averaged variables at the cell-boundaries from the left and the right states in the Riemann solver by an integer of 1 or 2. avg_state=1 and

2 correspond to Roe- and arithmetic averages, respectively.

`wave_speeds` specifies the choice of the method to compute the left, right, and middle wave speeds in the Riemann solver by an integer of 1 and 2. `wave_speeds`=1 and 2 correspond to the direct method (Batten et al., 1997), and indirect method that approximates the pressures and velocity (Toro, 2013), respectively.

`commute_err` activates WENO reconstruction of the cell-averaged variables at the cell-interior Gaussian quadrature points, following the two-point, fourth order Gaussian quadrature rule (Titarev and Toro, 2004).

`split_err` activates numerical approximation of the left or right cell-boundary integral-average of the given variables by getting the arithmetic mean of their WENO-reconstructed values at the cell-boundary Gaussian quadrature points, following the two-point, fourth order Gaussian quadrature rule (Titarev and Toro, 2004). When `commute_err` and `split_err` are set `TRUE` and the 5th-order WENO is used, the global order of accuracy of the spatial integration of the governing equations becomes fourth order (Coralic and Colonius, 2014).

`reg_eps` specifies the magnitude of interface regularization for two-component flows that prevents diffusion of the phase interface (Tiwari et al., 2013). The default value of `reg_eps` is unity. When `reg_eps` is undefined, interface regularization is not used. Details of implementation and assessment are addressed in Meng (2016); Schmidmayer et al. (2019).

`flux_lim` specifies the choice of flux limiter that is used in simulation by an integer from 1 through 7 as listed in table 12. When `flux_lim` is undefined, flux limiter is not applied. Details of the limiters and their implementations in MFC can be found in Meng (2016).

`tvd_rhs_flux` activates a specified flux limiter to inte-rcell fluxes outside Riemann solver.

`tvd_riemann_flux` activate a specified flux limiter to cell edges inside the Riemann solver. `tvd_rhs_flux` and `tvd_riemann_flux` are mutually exclusive.

`tvd_wave_speeds` activates the use of the TVD wave speeds for flux computation inside the Riemann solver when `tvd_riemann_flux` is set `TRUE`.

6.2.6 Formatted database and structure parameters

Parameter	Type	Description
`format`	Integer	Output format. [1]: Silo-HDF5; [2] Binary
`precision`	Integer	[1] Single; [2] Double
`parallel_io`	Logical	Parallel I/O
`cons_vars_wrt`	Logical	Write conservative variables
`prim_vars_wrt`	Logical	Write primitive variables
`fourier_decomp`	Logical	Apply a spatial Fourier decomposition to the output variables
`alpha_rho_wrt(i)`	Logical	Add the partial density of the fluid `i` to the database
`rho_wrt`	Logical	Add the mixture density to the database
`mom_wrt(i)`	Logical	Add the `i`-direction momentum to the database
`vel_wrt(i)`	Logical	Add the `i`-direction velocity to the database
`E_wrt`	Logical	Add the total energy to the database
`pres_wrt`	Logical	Add the pressure to the database
`alpha_wrt(i)`	Logical	Add the volume fraction of fluid `i` to the database
`gamma_wrt`	Logical	Add the specific heat ratio function to the database
`heat_ratio_wrt`	Logical	Add the specific heat ratio to the database
`pi_inf_wrt`	Logical	Add the liquid stiffness function to the database
`pres_inf_wrt`	Logical	Add the liquid stiffness to the formatted database
`c_wrt`	Logical	Add the sound speed to the database
`omega_wrt(i)`	Logical	Add the `i`-direction vorticity to the database
`schlieren_wrt`	Logical	Add the numerical schlieren to the database
`fd_order`	Integer	Order of finite differences for computing the vorticity and the numerical Schlieren function [1,2,4]
`schlieren_alpha(i)`	Real	Intensity of the numerical Schlieren computed via `alpha(i)`
`probe_wrt`	Logical	Write the flow chosen probes data files for each time step
`num_probes`	Integer	Number of probes
`probe(i)%x[y,z]`	Real	Coordinates of probe `i`
`com_wrt(i)`	Logical	Add the center of mass of fluid `i` to the database
`cb_wrt(i)`	Logical	Add coherent body data of fluid `i` to the database

Table 7: Formatted database and structure parameters

Table 7 lists formatted database output parameters. The parameters define variables that are outputted from simulation and file types and formats of data as well as options for post-processing.

`format` specifies the choice of the file format of data file outputted by MFC by an integer of 1 and 2. `format`=1 and 2 correspond to Silo-HDF5 format and binary format, respectively.

`precision` specifies the choice of the floating-point format of the data file outputted by MFC by an integer of 1 and 2. `precision`=1 and 2 correspond to single-precision and double-precision formats, respectively.

`parallel_io` activates parallel input/output (I/O) of data files. It is highly recommended to activate this option in a parallel environment. With parallel I/O, MFC inputs and outputs a single file throughout pre-process, simulation, and post-process, regardless of the number of processors used. Parallel I/O enables the use of different number of processors in each of the processes (i.e. simulation data generated using 1000 processors can be post-processed using a single processor).

`cons_vars_wrt` and `prim_vars_wrt` activate output of conservative and primitive state variables into the database, respectively.

`[variable's name]_wrt` activates output of the each specified variable into the database.

`schlieren_alpha(i)` specifies the intensity of the numerical Schlieren of i-th component.

`fd_order` specifies the order of finite difference scheme that is used to compute the vorticity from the velocity field and the numerical schlieren from the density field by an integer of 1, 2, and 4. `fd_order`=1, 2, and 4 correspond to the first, second, and fourth order finite difference schemes, respectively.

`probe_wrt` activates output of state variables at coordinates specified by `probe(i)%x[y,z]`.

`com_wrt(i)` activates output of the center of mass of i-th fluid component into the database.

`cb_wrt(i)` activates output of the coherent body mass of i-th fluid component in the domain into the database.

6.2.7 (Optional) Acoustic source parameters

Parameter	Type	Description
`Monopole`	Logical	Acoustic source
`num_mono`	Integer	Number of acoustic sources
`Mono(i)%pulse`	Integer	Acoustic wave form: [1] Sine [2] Gaussian [3] Square
`Mono(i)%npulse`	Integer	Number of pulse cycles
`Mono(i)%support`	Integer	Type of the spatial support of the acoustic source : [1] 1D [2] Finite width (2D) [3] Support for finite line/patch
`Mono(i)%loc(j)`	Real	j-th coordinate of the point that consists of i-th source plane
`Mono(i)%dir`	Real	Direction of acoustic propagation
`Mono(i)%mag`	Real	Pulse magnitude
`Mono(i)%length`	Real	Spatial pulse length

Table 8: Acoustic source parameters.

Table 8 lists acoustic source parameters. The parameters are optionally used to define a source plane in the domain that generates an acoustic wave that propagates in a specified direction normal to the source plane (one-way acoustic source). Details of the acoustic source model can be found in Maeda and Colonius (2017).

`Monopole` activates the acoustic source.

`num_mono` defines the total number of source planes by an integer.

`Mono(i)%pulse` specifies the choice of the acoustic wave form generated from i-th source plane by an integer. `Mono(i)%pulse`=1, 2, and 3 correspond to sinusoidal wave, Gaussian wave, and square wave, respectively.

`Mono(i)%npulse` defines the number of cycles of the acoustic wave generated from i-th source plane by an integer.

`Mono(i)%mag` defines the peak amplitude of the acoustic wave generated from i-th source plane with a given wave form.

`Mono(i)%length` defines the characteristic wavelength of the acoustic wave generated from i-th source plane.

`Mono(i)%support` specifies the choice of the geometry of acoustic source distribution of i-th source plane by an integer from 1 through 3:
`Mono(i)%support`= 1 specifies an infinite source plane that is normal to the x-axis and intersects with the axis at x =`Mono(i)%loc(1)` in 1-D simulation.
`Mono(i)%support`= 2 specifies a semi-infinite source plane in 2-D simulation. The i-th source

plane is determined by the point at [Mono(i)%loc(1), Mono(i)%loc(2)] and the normal vector [cos(Mono(i)%dir), sin(Mono(i)%dir)] that consists of this point. The source plane is defined in the finite region of the domain: $x \in [-\infty, \infty]$ and $y \in$[-mymono_length/2, mymono_length/2].
Mono(i)%support= 3 specifies a semi-infinite source plane in 3-D simulation. The i-th source plane is determined by the point at [Mono(i)%loc(1), Mono(i)%loc(2), Mono(i)%loc(3)] and the normal vector [cos(Mono(i)%dir), sin(Mono(i)%dir), 1] that consists of this point. The source plane is defined in the finite region of the domain: $x \in [-\infty, \infty]$ and $y, z \in$[-mymono_length/2, mymono_length/2].

6.2.8 (Optional) Ensemble-averaged bubble model parameters

Parameter	Type	Description
bubbles	Logical	Ensemble-averaged bubble modeling
bubble_model	Integer	[1] Gilmore; [2] Keller–Miksis
polytropic	Logical	Polytropic gas compression
thermal	Integer	Thermal model: [1] Adiabatic; [2] Isothermal; [3] Transfer
R0ref	Real	Reference bubble radius
nb	Integer	Number of bins: [1] Monodisperse; [> 1] Polydisperse
Ca	Real	Cavitation number
Web	Real	Weber number
Re_inv	Real	Inverse Reynolds number
mu_l0*	Real	Liquid viscosity (only specify in liquid phase)
ss*	Real	Surface tension (only specify in liquid phase)
pv*	Real	Vapor pressure (only specify in liquid phase)
gamma_v†	Real	Specific heat ratio
M_v†	Real	Molecular weight
mu_v†	Real	Viscosity
k_v†	Real	Thermal conductivity

Table 9: Ensemble-averaged bubble model parameters. These options work only for gas-liquid two component flows. Component indexes are required to be 1 for liquid and 2 for gas.

* These parameters should be pretended with patch index 1 that is filled with liquid: fluid_pp(1)%.
† These parameters should be pretended with patch indexes that are respectively filled with liquid and gas: fluid_pp(1)% and fluid_pp(2)%.

Table 9 lists the ensemble-averaged bubble model parameters.'

bubbles activates the ensemble-averaged bubble model.

bubble_model specified a model for spherical bubble dynamics by an integer of 1 and 2. bubble_model=1 and 2 correspond to the Gilmore and the Keller-Miksis equations, respectively.

polytropic activates polytropic gas compression in the bubble. When polytropic is set FALSE, the gas compression is modeled as non-polytropic due to heat and mass transfer across the bubble wall with constant heat and mass transfer coefficients based on (Preston et al., 2007).

thermal specifies a model for heat transfer across the bubble interface by an integer from 1 through 3. thermal=1, 2, and 3 correspond to no heat transfer (adiabatic gas compression), isothermal heat transfer, and heat transfer with a constant heat transfer coefficient based on Preston et al. (2007), respectively.

R0ref specifies the reference bubble radius.

nb specifies the number of discrete bins that define the probability density function (PDF) of the bubble radius.

Ca, Web, and Re_inv respectively specify the Cavitation number, Weber number, and the inverse Reynolds number that characterize the offset of the gas pressure from the vapor pressure, surface tension, and liquid viscosity when the polytropic gas compression model is used.

mu_l0, ss, and pv, gamma_v, M_v, mu_v, and k_v specify simulation parameters for the non-polytropic gas compression model. mu_l0, ss, and pv correspond to the liquid viscosity, surface tension, and vapor pressure, respectively. gamma_v, M_v, mu_v, and k_v specify the specific heat ratio, molecular weight, viscosity, and thermal conductivity of a chosen component. Implementation of the parameterse into the model follow Ando (2010).

7 Flow visualization and data output

Post-processed database in Silo-HDF5 format can be visualized and analyzed using VisIt (Childs et al., 2012). VisIt is an open-source interactive parallel visualization and graphical analysis tool for viewing scientific data. Versions of VisIt after 2.6.0 have been confirmed to work with the MFC databases for some parallel environments. Nevertheless, installation and configuration of VisIt can be environment-dependent and are left to the user. Further remarks on parallel flow visualization, analysis and processing of MFC database using VisIt can also be found in Coralic (2015); Meng (2016).

7.1 Procedure

After post-process of simulation data (see section 5), a folder that contains a silo-HDF5 database is created, named silo_hdf5. silo_hdf5 includes directory named root, that contains index files for flow field data at each saved time step. The user can launch VisIt and open the index files under /silo_hdf5/root. Once the database is loaded, flow field variables contained in the database (see section 6.2.6) can be added to plot.

As an example, figure 1 shows the iso-contour of the liquid void fraction (alpha1) in the database generated by example case 3D_sphbubcollapse. For analysis and processing of the database using VisIt's capability, the user is encouraged to address VisIt user manual[1].

[1] https://wci.llnl.gov/simulation/computer-codes/visit/manuals

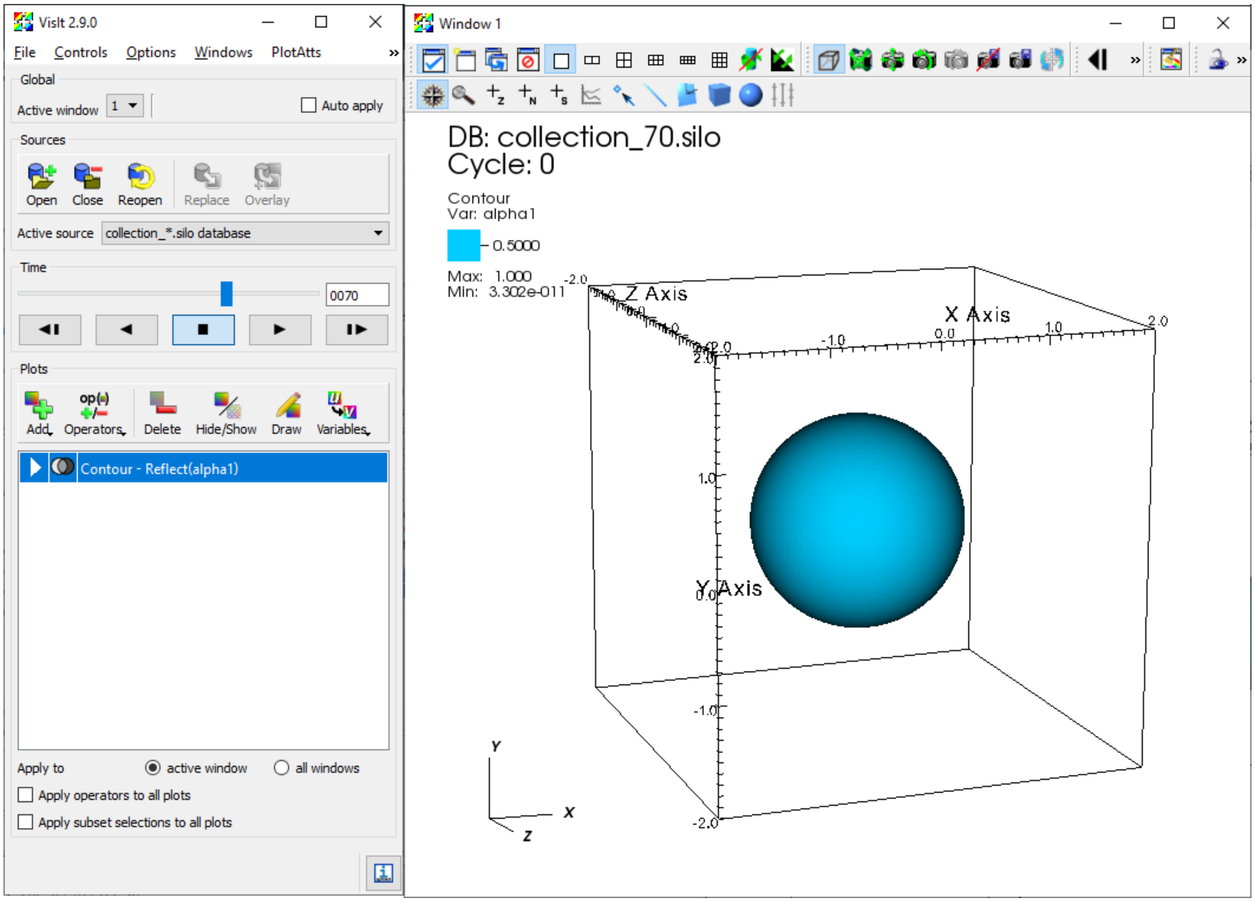

Figure 1: Iso-contour of the liquid void fraction (`alpha1`) in the database generated by example case `3D_sphbubcollapse`.

7.2 Serial data output

If `parallel_io = F` then MFC will output the conservative variables to a directory `D/`. If multiple cores are used ($\texttt{ppn} > 1$) then a separate file is created for each core. If there is only one coordinate dimension (`n = 0` and `p = 0`) then the primivative variables will also be written to `D/`. The file names correspond to the variables associated with each equation solved by MFC. They are written at every `t_step_save` time step. The conservative variables are

$$\{(\rho\alpha)_1, \dots, (\rho\alpha)_{N_c}, \rho u_1, \dots, \rho u_{N_d}, E, \alpha_1, \dots, \alpha_{N_c}\}, \tag{3}$$

and the primitive variables are

$$\{(\rho\alpha)_1, \dots, (\rho\alpha)_{N_c}, u_1, \dots, u_{N_d}, p, \alpha_1, \dots, \alpha_{N_c}\}, \tag{4}$$

where N_c are the number of components `num_fluids` and N_d is the number of spatial dimensions. There are exceptions: if `model_eqns = 3`, then the six-equation model appends these variables with the internal energies of each component. If there are sub-grid bubbles `bubbles = T`, then the bubble variables are also written. These depend on the bubble dynamics model used. If `polytropic = T` then the conservative variables are appended by

$$\{n_b R_1, n_b \dot{R}_1, \dots, n_b R_{N_b}, n_b \dot{R}_{N_b}\} \tag{5}$$

where n_B is the bubble number density and N_b are the number of bubble sizes (see matching variable in the input file, `Nb`). The primitive bubble variables do not include n_B:

$$\{R_1, \dot{R}_1, \ldots, R_{N_b}, \dot{R}_{N_b}\}. \tag{6}$$

Acknowledgements

This work was supported in part by multiple past grants from the US National Institute of Health (NIH), the US Office of Naval Research (ONR), and the US National Science Foundation (NSF), as well as current NIH grant number P01-DK043881 and ONR grant numbers N0014-17-1-2676 and N0014-18-1-2625. The computations presented here utilized the Extreme Science and Engineering Discovery Environment, which is supported under NSF grant number CTS120005. K.M. acknowledges support from the Funai Foundation for Information Technology via the Overseas Scholarship.

References

Allaire, G., Clerc, S., and Kokh, S. (2002). A five-equation model for the simulation of interfaces between compressible fluids. *Journal of Computational Physics*, 181(2):577–616.

Ando, K. (2010). *Effects of polydispersity in bubbly flows*. PhD thesis, California Institute of Technology.

Balsara, D. S. and Shu, C.-W. (2000). Monotonicity preserving weighted essentially non-oscillatory schemes with increasingly high order of accuracy. *Journal of Computational Physics*, 160(2):405–452.

Batten, P., Clarke, N., Lambert, C., and Causon, D. M. (1997). On the choice of wavespeeds for the hllc riemann solver. *SIAM Journal on Scientific Computing*, 18(6):1553–1570.

Bryngelson, S. H., Schmidmayer, K., Coralic, V., Meng, J. C., Maeda, K., and Colonius, T. (2019). Mfc: An open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver. *arXiv preprint arXiv:1907.10512*.

Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Harrison, C., Weber, G. H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C., Bethel, E. W., Camp, D., Rübel, O., Durant, M., Favre, J. M., and Navrátil, P. (2012). VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372.

Coralic, V. (2015). *Simulation of shock-induced bubble collapse with application to vascular injury in shockwave lithotripsy*. PhD thesis, California Institute of Technology.

Coralic, V. and Colonius, T. (2014). Finite-volume weno scheme for viscous compressible multicomponent flows. *Journal of computational physics*, 274:95–121.

Gottlieb, S. and Shu, C.-W. (1998). Total variation diminishing runge-kutta schemes. *Mathematics of computation of the American Mathematical Society*, 67(221):73–85.

Henrick, A. K., Aslam, T. D., and Powers, J. M. (2005). Mapped weighted essentially non-oscillatory schemes: achieving optimal order near critical points. *Journal of Computational Physics*, 207(2):542–567.

Johnsen, E. (2008). *Numerical simulations of non-spherical bubble collapse: With applications to shockwave lithotripsy*. PhD thesis, California Institute of Technology.

Maeda, K. and Colonius, T. (2017). A source term approach for generation of one-way acoustic waves in the euler and navier–stokes equations. *Wave Motion*, 75:36–49.

Meng, J. C. C. (2016). *Numerical simulations of droplet aerobreakup*. PhD thesis, California Institute of Technology.

Preston, A., Colonius, T., and Brennen, C. (2007). A reduced-order model of diffusive effects on the dynamics of bubbles. *Physics of Fluids*, 19(12):123302.

Saurel, R., Petitpas, F., and Berry, R. A. (2009). Simple and efficient relaxation methods for interfaces separating compressible fluids, cavitating flows and shocks in multiphase mixtures. *journal of Computational Physics*, 228(5):1678–1712.

Schmidmayer, K., Bryngelson, S. H., and Colonius, T. (2019). An assessment of multicomponent flow models and interface capturing schemes for spherical bubble dynamics. *arXiv preprint arXiv:1903.08242*.

Suresh, A. and Huynh, H. (1997). Accurate monotonicity-preserving schemes with runge–kutta time stepping. *Journal of Computational Physics*, 136(1):83–99.

Thompson, K. W. (1987). Time dependent boundary conditions for hyperbolic systems. *Journal of computational physics*, 68(1):1–24.

Thompson, K. W. (1990). Time-dependent boundary conditions for hyperbolic systems, ii. *Journal of computational physics*, 89(2):439–461.

Titarev, V. A. and Toro, E. F. (2004). Finite-volume weno schemes for three-dimensional conservation laws. *Journal of Computational Physics*, 201(1):238–260.

Tiwari, A., Freund, J. B., and Pantano, C. (2013). A diffuse interface model with immiscibility preservation. *Journal of computational physics*, 252:290–309.

Toro, E. F. (2013). *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer Science & Business Media.

A Boundary conditions

	#	Description
Normal	-1	Periodic
	-2	Reflective
	-3	Ghost cell extrapolation
	-4	Riemann extrapolation
	-5	Slip wall
Characteristic.	-6	Non-reflecting subsonic buffer
	-7	Non-reflecting subsonic inflow
	-8	Non-reflecting subsonic outflow
	-9	Force-free subsonic outflow
	-10	Constant pressure subsonic outflow
	-11	Supersonic inflow
	-12	Supersonic outflow

Table 10: Boundary conditions.

The boundary condition supported by the MFC are listed in table 10. Their number (#) corresponds to the input value in `input.py` labeled `bc_x[y,z]%beg[end]` (see table 6). The entries labeled "Characteristic." are characteristic boundary conditions based on Thompson (1987) and Thompson (1990).

B Patch types

#	Name	Dim.	Smooth	Description and required parameters
1	Line segment	1	N	Requires `x_centroid` and `x_length`.
2	Circle	2	Y	Requires `x[y]_centroid` and `radius`.
3	Rectangle	2	N	Coordinate-aligned. Requires `x[y]_centroid` and `x[y]_length`.
4	Sweep line	2	Y	Not coordinate aligned. Requires `x[y]_centroid` and `normal(i)`.
5	Ellipse	2	Y	Requires `x[y]_centroid` and `radii(i)`.
6	Vortex	2	N	Isentropic flow disturbance. Requires `x[y]_centroid` and `radius`.
7	2D analytical	2	N	Assigns the primitive variables as analytical functions.
8	Sphere	3	Y	Requires `x[y,z]_centroid` and `radius`.
9	Cuboid	3	N	Coordinate-aligned. Requires `x[y,z]_centroid` and `x[y,z]_length`.
10	Cylinder	3	Y	Requires `x[y,z]_centroid`, `radius`, and `x[y,z]_length`.
11	Sweep plane	3	Y	Not coordinate-aligned. Requires `x[y,z]_centroid` and `normal(i)`.
12	Ellipsoid	3	Y	Requires `x[y,z]_centroid` and `radii(i)`.
13	3D analytical	3	N	Assigns the primitive variables as analytical functions.

Table 11: Patch geometries

The patch types supported by the MFC are listed in table 11. This includes types exclusive to one-, two-, and three-dimensional problems. The patch type number (#) corresponds to the input value in `input.py` labeled `patch_icpp(j)%geometry` where `j` is the patch index. Each patch requires a different set of parameters, which are also listed in this table.

C Flux limiter

#	**Description**
1	Minmod
2	MC
3	Ospre
4	Superbee
5	Sweby
6	van Albada
7	van Leer

Table 12: Flux limiter

The flux limiters supported by the MFC are listed in table 12. Each limiter can be specified by specifying the value of `flux_lim`. Details of their implementations can be found in Meng (2016).