User Manual

Adaptive Wavelet Collocation Method (AWCM)

Code Structure and Setup

(alpha release)

August 25, 2006

1 Code Structure

1.1 Overall Code Organization

Adaptive wavelet collocation method (AWCM) solver consists of two parts:

1. elliptic solver and
2. time evolution solver.

The elliptic solver can be used either to solve general elliptic problems of the type $\mathcal{L}\mathbf{u} = \mathbf{f}$ or as a part of initial condiitons, where a linear system of PDEs is solved during each grid iteration instead of prescribing $\mathbf{u}$ analytically. The adaptive grid refinement procedure provides a way to obtain the solution (initial conditions) on an optimal (compressed) grid. The pseudocode for the iterative global elliptic solver is shown in Algorithm 1. Note that if this part of the code is used for initial conditions, analytical expression for u is provided instead of solving elliptic problem. The pseudocode for the time evolution problem is shown in Algorithm 2.

---

initial guess $(m = 0)$**:** $\mathbf{u}_{\mathbf{k}}^{m}$ and $\mathcal{G}_{\geq}^{m}$
while $m = 0$ **or** $m > 1$ **and** $\left[\mathcal{G}_{\geq}^{m} \neq \mathcal{G}_{\geq}^{m-1} \text{ or } \|\mathbf{f}^{J} - \mathbf{L}\mathbf{u}_{\geq}^{J}\|_{\infty} > \delta_{\epsilon}\right]$
  $m = m + 1$
  perform forward wavelet transform for each component of $\mathbf{u}_{\mathbf{k}}^{m}$
  for all levels $j = J : -1 : 1$
    create a mask $\mathcal{M}$ for $|d_{\mathbf{l}}^{\mu,j}| \geq \epsilon$
  end
  extend the mask $\mathcal{M}$ with adjacent wavelets
  perform the reconstruction check procedure
  construct $\mathcal{G}_{\geq}^{m+1}$
  if $\mathcal{G}_{\geq}^{m+1} \neq \mathcal{G}_{\geq}^{m}$
    interpolate $\mathbf{u}_{\mathbf{k}}^{m}$ to $\mathcal{G}_{\geq}^{m+1}$
  end if
  solve $\mathcal{L}\mathbf{u} = \mathbf{f}$ using Local Multilevel Elliptic Solver.
end

Algorithm 1. Global Elliptic Solver.

---

initial conditions $(n = 0)$**:** $\mathbf{u}_{\mathbf{k}}^{n}$ and $\mathcal{G}_{\geq}^{n}$
while $t_n < t_{\max}$
  $t_{n+1} = t_n + \Delta t$
  integrate the system of equations using Krylov time integration to obtain $\mathbf{u}_{\mathbf{k}}^{n+1}$
  perform forward wavelet transform for each component of $\mathbf{u}_{\mathbf{k}}^{n+1}$
  for all levels $j = J : -1 : 1$
    create a mask $\mathcal{M}$ for $|d_{\mathbf{l}}^{\mu,j}| \geq \epsilon$
  end
  extend the mask $\mathcal{M}$ with adjacent wavelets
  perform the reconstruction check procedure
  construct $\mathcal{G}_{\geq}^{n+1}$
  if $\mathcal{G}_{\geq}^{n+1} \neq \mathcal{G}_{\geq}^{n}$
    interpolate $\mathbf{u}_{\mathbf{k}}^{n+1}$ to $\mathcal{G}_{\geq}^{n+1}$
  end if
  $n = n + 1$
end

Algorithm 2. Time Evolution Solver.

1.2 Code Categories and Files

The codes consist of the FORTRAN and Matlab files. FORTRAN code saves results in terms of active wavelet coefficients, while Matlab files are set up to read output files from the FORTRAN code, perform inverse wavelet transform, and visualize the results.

The FORTRAN files can be divided into the followqing categories:

Case Files:

***user_case*.f90**
***user_input*.inp**

where the *user_case* is the name of a specific case set up by user that can be located in any directory, while *user_input* in the user defined input file. Note that the same case can have multiple input files.

Core Files:

wlt_3d_main.f90
wavelet_3d.f90
wavelet_filters.f90
elliptic_solve_3d.f90
time_int_cn.f90
time_int_krylov_3d.f90

This category inculdes all the core files for the Adaptive wavelet collocation method. We don't need to modify any part of these files.

Data-Structure files:

wavelet_3d_wrk.f90
wavelet_3d_wrk_vars.f90

Shared Variables Files:

shared_modules.f90
wavelet_3d_vars.f90
elliptic_solve_3d_vars.f90
wavelet_filters_vars.f90
io_3d_vars.f90

This category includes all the variable only modules, *i.e.* these modules contain no functions or subroutines.

Supplementary Utility Files:

input_files_reader.f90
read_init.f90
read_data_wray.f90
io_3d.f90
util_3d.f90
default_util.f90
vector_util.f90
endienness_big.f90
endienness_small.f90
reverse_endian.f90

Supplementary FFT Package Files:

fft.f90
fftpacktvms.f90
fft.interface.temperton.f90
fft_util.f90 spectra.f90
ch_resolution_fs.f90

These supplementary files are located in subdirectory FFT and, if necessary, can be used for extracting statistics in homogeneous directions.

Supplementary LINPACK files:

d1mach.f
derfc.f
derf.f
dgamma.f
dgeev.f
dgels.f
dqage.f
dqag.f
dtrsm.f
dum.f
fort.1
gaussq.f
needblas.f
r1mach.f
zgetrf.f
zgetri.f

Visualization Files:

c_wlt_3d.m
c_wlt_3d_movie.m
c_wlt_inter.f90
inter3d.m
mycolorbar.m
mycontourf.m
c_wlt_3d_isostats.m
vor_pal.m

These are are used to visualize the output results. All these files are contained in subdirectory post_process.

2 Case Files

This section will attempt to provide some familiarization with the more intricate details of setting up your own case. The code is equipped with some test cases that can be used as a reference for setting up a new case. Each test case has a set of subroutines that must be present in order for the code to function properly. If one of the functions does not apply in your case, it must remain present, but you can leave the contents (other than variable definitions) blank. For example, if you are using periodic boundary conditions, you can leave the details in the user_algebaic_BC function blank, but do not delete the function itself. Each case should have its own separate directory with its own case files and **results/** output directory. Two case files are needed for each case: casename.f90 and casename.inp, where casename is the name of the particular case your are running, *e.g.*, case_elliptic_poisson or case_small_vortex_array. Casename.f90 will be the actual FORTRAN code that is compiled in the source directory. Casename.inp will contain any variables that are not hard coded into the .f90 file. The casename.f90 file contains all the subroutines you see below followed by a brief description of what they do. There are two modules that are needed for memory and variable allocation specific two your case and the rest are functions or subroutines.

Modules

Subroutine Name	Description
user_case_db	Module created for database memory allocation.
user_case	This is the actual case module that contains all of the functions and subroutines that are used in this file.

Subroutines

Subroutine Name	Description
user_setup_pde	Sets up the number of variables that are integrated or interpolated on the initial time step and any subsequent step.
user_exact_soln	Stores the exact solution in memory.
user_initial_conditions	Defines the initial conditions if they can be determined analytically.
user_algebaic_BC	Sets conditions on boundaries
user_algebaic_BC_diag	Determines the diagonal term for calculating boundary conditions.

user_rhs_BC	Sets the right hand side for the boundary conditions.
Laplace	Sets up the Laplacian.
Laplace_diag	Sets up the diagonal terms for the Laplacian
user_rhs	Sets the right hand side of the PDE being solved
user_Drhs	Sets the Jacobian of the right hand side of the PDE.
user_Drhs_diag	Sets up the diagonal terms for the Jacobian when the Crank-Nicolson time integration method is used.
user_project	Used for Crank-Nicolson time integration to get projections of variables that are needed for integration, but not actually integrated with the actual solver.
user_chi	This is your boundary that you define using Brinkman penalization (**Not implemented yet**).
user_stats	Any user specified statistics that need to be calculated to analyze new data can be calculated here.
user_cal_force	Used to calculate lift and drag on any bodies in the flow (**Not implemented yet**).
user_read_input	Any user defined variables defined in the casename.inp file are read through this subroutine.
user_additional_vars	Where any additional variables are calculated.
user_scalar_vars	Any scalar non field parameters that need to be calculated can be done here.
user_scales	Used to override the default scales routine.
user_cal_cfl	Override default cfl condition and use user created condition.

2.1 CaseFile.f90 Details:

Now that we've taken a general look at how the case files are structured, let's look at a specific case and explain how it is set up. The compressible.f90 case is set up to solve the Euler equations for a two dimensional explosion. The initial conditions are

$$\begin{aligned} \rho_{ins} &= 1.0 \quad \rho_{out} = 0.125 \\ u_{ins} &= 0.0 \quad u_{out} = 0.0 \\ v_{ins} &= 1.0 \quad v_{out} = 0.0 \\ p_{ins} &= 1.0 \quad p_{out} = 0.1 \end{aligned}$$

where the subscripts **ins** and **out** denote the inside and outside of a circle of radius 1.0 in a 6.0 x 6.0 domain in the $x-y$ plane. For simplicity we will only solve the euler equations, but to prevent shock formation a smooth discontinuity will be used instead of a full discontinuity. A hyperbolic solver is being developed, but in the mean time it is best to implement the full Navier-Stokes equations to model compressible flow for long time periods. We will also make the boundaries periodic as an introductory case. Now let's look at each module and subroutine one by one.

2.1.1 MODULE user_case_db

This is a temporary solution for memory allocation in the database. Simply make sure that n_var_db is at least as big as the number of variables you will be using.

2.1.2 MODULE user_case

This is the main module that the entire case file is built into. Any variables that you want to be available globally throughout the entire module should be defined in this section. Below where it says "case specific variables," the variable n_var_pressure is declared. As of now, this parameter must always be declared because the code is set up to solve both compressible and incompressible flows. Following are some non-dimensional parameters **Re** and **gama** as well as initial condition parameters **delta** and **radius**. As you will see, these variables are imported from the compressible.inp file in the user_read_input subroutine.

2.1.3 SUBROUTINE user_setup_pde

In this routine we set the global parameters the main code uses to know how many equations it is integrating, interpolating, saving and how many variables there is an exact solution for. The number of variables that are integrated n_integrated is set to $\mathbf{dim}+\mathbf{2}$ since we are solving for ρ, $\rho\mathbf{e}$, and the momentum equations in **dim** dimensions. In the compressible.inp file $\mathbf{dim}=\mathbf{2}$. Leave n_time_levels and n_var_time_levels as they are. They will soon be phased out. Any additional variables that are not integrated, but you want calculated at

each time step set as n_var_additional. As an example, n_var_additional is set to 1 so that we can calculate the pressure and save it at each time step. This is totally unnecessary as pressure can be calculated during the post processing step, but it is pure for demonstration.

The next step is to fill in the variable names according to how we arrange our variables. We will fill in the variables depending upon how many dimensions there are starting with density as the first variable, x-momentum as the second and so on.

The grid does not need to adapt to all 5 variables. Pressure (the 5^{th} variable) is simply found using the 4 conserved variables. Obviously we will only need the grid to adapt to the 4 conserved variables, so we set n_var_adapt to n_var_adapt(1:n_integrated,0) and n_var_adapt(1:n_integrated,1) to .TRUE. The 0 index indicates adaptation when setting up the initial condition and 1 index adapting at each time step afterwards. We also need the grid to interpolate all of the integrated variables as time moves along as well. Adaptation means that the grid will evolve based upon a given variable. Interpolation is what needs to happen in order to add a previous time step to the next. In this case, it is not necessary to interpolate the pressure so we have set the first 4 variables to be interpolated both at the initial condition and all time steps thereafter.

Need more explanation on the different between adaptation and interpolation.

We want to save all 5 variables at each write step, so we have set n_var_save as true for all variables n_var.

For the time being, please ignore the time level counters and leave them the way they are. These will most likely be phased out in the future.

There is no exact solution for the problem we're solving so we have set n_var_save to zero.

If you find that you need more points due to a a certain variable, you can create a scaleCoeff < 1.0 to add more points to that variable. If it is left as 1.0 it will scale the solution in its default manner according to your error threshold parameter ϵ.

2.1.4 SUBROUTINE user_exact_soln

This is where we would calculate an exact solution if we had one. In our case we do not, but if you did you would enter the exact solution into the array u in the same order you defined in the setup_pde routine. For example, $u(1 : nlocal, 1) = ...$ for the density ρ, $u(1 : nlocal, 2) = ...$ for the x-momentum and so on. Even though it is called u, it is not the same u that is used to numerically calculate the solution. The main code takes this u and stores it as the exact solution somewhere else.

2.1.5 SUBROUTINE user_initial_conditions

If you know the analytical initial condition, it is set up here. Instead of using the sign() function for our initial condition, we have used the heavy-side function tanh() in conjunction

with a width parameter delta to ensure a smooth initial condition. We also decided to save the pressure at the initial condition as well even though it is a bit redundant.

2.1.6 SUBROUTINE user_algebaic_BC

Algebraic boundary conditions are set by solving the general equation

$$\mathcal{L}\mathbf{u} = \mathbf{f}.$$

There are three main subroutines used to solve this equation. The first is user_algebaic_BC. This subroutine sets up the Ł**u** portion of the equation.

2.1.7 SUBROUTINE user_algebaic_BC_diag

Since the equation above is solved iteratively, the diagonal terms of Ł need to be provided. user_algebaic_BC_diag becomes this term.

2.1.8 SUBROUTINE user_algebaic_BC_rhs

To finish setting up the equations that need to be solved, user_algebaic_BC_rhs is the right hand side of the above equation **f**.

2.1.9 FUNCTION Laplace

2.1.10 FUNCTION Laplace_diag

2.1.11 FUNCTION user_rhs

Since this manual uses the compressible case to demonstrate its features, we must assume that not all readers are familiar with the governing equations and briefly introduce. We will then reconstruct them in a more general form that is easily implemented into the user_rhs function.

In order to solve for the flow we write the equations in terms of the conserved quantities ρ, ρu, ρv and the total energy ρe. We can write them in vector notation as follows

$$\frac{\partial \mathbf{U}}{\partial t} = -\frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} - \frac{\partial \mathbf{G}(\mathbf{U})}{\partial y}$$

where U, $F(U)$ and $G(U)$ are defined as

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + P \\ \rho uv \\ (\rho e + P)u \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + P \\ (\rho e + P)v \end{bmatrix}.$$

If we rewrite these equations in terms of the U vector quantities $[U_1, U_2, U_3, U_4]$, $F(U)$ and $G(U)$ become

$$\mathbf{F} = \begin{bmatrix} U_2 \\ \frac{U_2^2}{U_1} + P \\ \frac{U_2 U_3}{U_1} \\ (U_4 + P)\frac{U_2}{U_1} \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} U_3 \\ \frac{U_2 U_3}{U_1} \\ \frac{U_3^2}{U_1} + P \\ (U_4 + P)\frac{U_3}{U_1} \end{bmatrix}$$

and our normalized pressure is

$$p = (\gamma - 1)\left(U_4 - \frac{1}{2}\frac{U_2^2 + U_3^2}{U_1}\right).$$

In the function user_rhs, we need to supply the right hand side of the main governing equation. The left hand side is the time derivative that we're integrating and the right hand side is everything else. This is supplied to the function user_rhs. As you can see in the function user_rhs, we defined our $\mathbf{F}(\mathbf{U})$ with an extra index of length dim. For the 2-D case this will account for our $\mathbf{F}(\mathbf{U})$ and $\mathbf{G}(\mathbf{U})$. $\mathbf{F}(\mathbf{U})$ is calculated just as the equations above demand. When we call the subroutine c_diff_fast, it calculates the derivatives in dim number of dimensions regardless of whether or not you need that many derivatives for that term. For example, even though the first derivative call has a first derivative output named dux, it really contains both the information for the x and y direction derivatives. If you look down where the user_rhs output variable is created, you'll see that the difference between dux(ie,:,1) and duy(ie,:,2) lies in the final index. An index of 1 corresponds to the derivative in the x-direction, 2 in the y-direction and 3 in the z-direction. This will later be changed so that you can choose whether or not you want to calculate both derivatives or not. The do loop sorts through the four different equations for the conserved variables and creates the right and side exactly as is needed.

2.1.12 FUNCTION user_Drhs

Since the Crank-Nicolson and Krylov are implicit iterative time integration solvers, we need to provide a first order perturbation to help the iterative solver converge more quickly. The user provides this perturbation to the main code through the function user_Drhs. Since the right hand side has already been defined, finding the perturbation is simply a matter of applying the linearized theory. If we define our RHS as $\mathcal{L}(\mathbf{U})$ then our DRHS can be found by

$$DRHS_i = \left.\frac{\partial \mathcal{L}_i(\mathbf{U})}{\partial U_j}\right|_{\mathbf{U_p}} U_j'$$

where the Jacobian is evaluated at the previous time step U_p and $U_i^{'}$ is the perturbation term. Implementing this equation into user_Drhs is fairly simple. Since $\mathcal{L}(\mathbf{U})$ is a linear operator, we can pull the spatial derivative out and re-write our $\mathbf{F}^{'}(\mathbf{U_p})$ and $\mathbf{G}^{'}(\mathbf{U_p})$ as

$$\mathbf{F}^{'} = \begin{bmatrix} U_2^{'} \\ -\left(\frac{U_{2p}}{U_{1p}}\right)^2 U_1^{'} + 2\left(\frac{U_{2p}}{U_{1p}}\right) U_2^{'} + P^{'} \\ -\left(\frac{U_{2p}U_{3p}}{U_{1p}^2}\right) U_1^{'} + \frac{(U_{2p}U_3^{'} + U_{3p}U_2^{'})}{U_{1p}} \\ (U_4^{'} + P^{'})\frac{U_{2p}}{U_{1p}} + (U_{4p} + P_p)\left(\frac{U_2^{'}}{U_1^{'}} - \frac{U_{2p}}{U_{1p}^2}U_1^{'}\right) \end{bmatrix} \mathbf{G} = \begin{bmatrix} U_3^{'} \\ -\left(\frac{U_{2p}U_{3p}}{U_{1p}^2}\right) U_1^{'} + \frac{(U_{2p}U_3^{'} + U_{3p}U_2^{'})}{U_{1p}} \\ -\left(\frac{U_{3p}}{U_{1p}}\right)^2 U_1^{'} + 2\left(\frac{U_{3p}}{U_{1p}}\right) U_3^{'} + P^{'} \\ (U_4^{'} + P^{'})\frac{U_{3p}}{U_{1p}} + (U_{4p} + P_p)\left(\frac{U_3^{'}}{U_1^{'}} - \frac{U_{3p}}{U_{1p}^2}U_1^{'}\right) \end{bmatrix}$$

and the pressures P_p and $P^{'}$ are

$$P_p = (\gamma - 1)\left(U_{4p} - \frac{1}{2}\frac{U_{2p}^2 + U_{3p}^2}{U_{1p}}\right)$$

$$P^{'} = (\gamma - 1)\left(U_4^{'} + \frac{1}{2}\frac{(U_{2p}^2 + U_{3p}^2)}{U_{1p}^2}U_1^{'} - \frac{U_{2p}U_2^{'} + U_{3p}U_3^{'}}{U_{1p}}\right).$$

Once the equations are constructed we enter them into user_Drhs in the same fashion as user_rhs and the DRHS assignment is complete. If there is a mistake made when entering the equations into DRHS it may still converge, but it may take a few more iterations. Otherwise, the solver will continue to iterate without reaching its error threshold criteria indefinitely.

2.1.13 FUNCTION user_Drhs_diag

When using the Crank-Nicolson technique, a diagonal term needs to be created from our DRHS. The diagonal term is found by letting all $U_i^{'} \rightarrow 0$ for all terms besides the term you are looking for $U_{j\neq i}$. Each term in $\mathbf{F}(\mathbf{U})^{'}$ is much simpler than in the original DRHS. Not only do we set all other $U_i^{'}$ to zero, we set $U_{j\neq i}$ equal to 1 as well to find the diagonal derivative coefficients. The flux functions $\mathbf{F}(\mathbf{U})^{'}$ are exactly this situation. To demonstrate, the fluxes before setting the $U_i^{'}$ terms to zero is

$$\mathbf{F}_{\mathbf{diag}}^{'} = \begin{bmatrix} 0 \\ (3-\gamma)\frac{U_{2p}}{U_{1p}}U_2^{'} \\ \frac{U_{2p}}{U_{1p}}U_3^{'} \\ \gamma\frac{U_{2p}}{U_{1p}}U_4^{'} \end{bmatrix} \mathbf{G}_{\mathbf{diag}}^{'} = \begin{bmatrix} 0 \\ \frac{U_{3p}}{U_{1p}}U_2^{'} \\ (3-\gamma)\frac{U_{3p}}{U_{1p}}U_3^{'} \\ \gamma\frac{U_{3p}}{U_{1p}}U_4^{'} \end{bmatrix}$$

where the pressure has been substituted into the equations already. The equations are now in the form of some function of un-primed terms multiplied by prime terms. Note when the

derivative is taken, no chain rule is applied as in analytical case, since when spatial derivatives are taken using finite difference, the whole nonlinear term in front of $U_i^{'}$ is multiplied by corresponding weight. Thus, we simply replace $U_i^{'}$ by diagonal weight corresponding to the differential operator. In the case when the term is written in algebraic from, $U_i^{'}$ is simply replaced by unity.

Note that this is only for the Crank-Nicolson method. When using the Krylov integrator this section does not need to be defined because it is not used.

2.1.14 SUBROUTINE user_project

This routine is mostly used for incompressible flows where the pressure needs to be tracked, but not integrated. The Crank-Nicolson method is used with this projection step to ensure that the velocity vector **v** remains divergent free. The small_vortex_array case demonstrates this feature more clearly.

2.1.15 FUNCTION user_chi

If there is to be an obstacle in the flow using Brinkman penalization, you would determine your χ function here. If it is equal to 1.0 it is inside the boundary and if it is equal to zero then it is outside. Note that this function is not fully implemented yet.

2.1.16 SUBROUTINE user_stats

After results of a new time step are available, you may want to generate some statistics based on those results. This function serves that purpose and is called from the main code. User_stats allows you to make these calculations without touching the main code.

2.1.17 SUBROUTINE user_cal_force

If an obstacle has been defined in the user_chi function, the lift and drag are calculated here.

2.1.18 SUBROUTINE user_read_input

In the subroutine setup_pde we defined our variables *Re*, *gama*, *delta* and *radius*. This is where we read them into the module from the compressible.inp input file. Each variable is read in separately with its own input_real command. The input_real command is only used to input real type numbers. If you want to read an integer you would use input_integer. The parameter 'stop' tells the code that if that variable is not read properly, it will stop the code execution. If you look in the compressible.inp file, you can see how to set up your own variables just like these.

2.1.19 SUBROUTINE user_additional_vars

Any additional variables that aren't integrated but are stored in the main storage array u are calculated here. In our case, we are storing pressure in the n_var_pressure portion of u that we created in setup_pde. Again, normally it would be better to calculate pressure in the post processing step, but we are storing it here as an example.

2.1.20 SUBROUTINE user_scalar_vars

If there are any scalar (non-field) parameters defined at the beginning of the module that need to be calculated, they should be handled here.

2.1.21 SUBROUTINE user_scales

In the situation that you wish to override the default scales routine in the main code, you would do that here.

2.1.22 SUBROUTINE user_cal_cfl

If you wish to override the default cfl condition you would do that here. The subroutine get_all_local_h returns the grid spacing for all points on the active grid. We then determine the cfl for each grid point and ultimately select the largest as cfl_out.

2.2 CaseFile.inp Details

The input file is where all of the variables that are not hard coded into your case file are stored. As seen in the previous section, any variables you need are read using an input_real or input_integer statement. For the compressible case, we ran a 2-D euler equation simulation. Inside if the compressible.inp file you will see all of the parameters that are needed to run this case.

The first several variables pertain to input output and initialization parameters. file_gen is whatever you want the output in the results subdirectory to be saved as. It is possible to restart your simulation at any given output time by setting IC_restart = T (true) and specifying which output time with IC_restart_station. If you wanted to restart from a different file you can set IC_from_file = T and specify the format and IC_filename. The output can be formatted text or binary (unformatted). Unformatted is the preferred format due to its speed and smaller file size.

The domain size is specified with coord_min and coord_max. We have specified three different dimensions in the case that dimension is set to 3 then the input file is ready to use. In the case of our 2-D run, it simply ignores the extra dimension. In order to avoid problems at the boundaries, it is sometimes necessary to specify a coord_zone_min and coord_zone_max.

Although this example uses periodic boundary conditions, these parameters should usually be set outside of the actual domain as seen in this example.

The overall solution error is controlled with the parameter ϵ. The code has been set up to allow the use of two different ϵ's depending on your particular situation. Eps_init can be used as a separate initial thresholding parameter for eps_adapt_steps number of steps before using the regular eps_run parameter. In our case there is no difference between the two.

Setting a threshold parameter ϵ is only effective on its own if all integrated variables are normalized to the same scale. This is usually not the case and a scaling method Scale_Meth needs to be specified to keep the relative error in check. In our compressible case, all the equations are normalized with a mean value of zero so choosing Linf error as the scaling method is not a bad choice. However, in other situations it may be better to use an L2-norm error or possibly a RMS scaling method. In some situations, the time scales can change very rapidly which causes your scales to rapidly vary as well. Since this can sometimes be problematic, a temporal filter has been provided to slow the speed at which these scales change scl_fltwt. This parameter has the range of [0, 1] where zero corresponds to no filtering and 1 corresponds to being completely based upon historical scales.

In every simulation, the minimum and maximum grid size needs to be specified by setting the minimum and maximum levels of resolution, J_MN and J_MX, as well as the initial M_vector. In our case the M_vector is [4, 4] and J_MN and J_MX are 2 and 4. The minimum grid size is $M\_vector * 2^{J\_MN}$ (16x16) and the maximum grid size is $M\_vector * 2^{J\_MX}$ (64x64). In our case J_MX is set relatively small because it is a test case and we want to limit the level of resolution for computational reasons.

The boundaries are set to be periodic and the grid is said to be uniform in all directions. Although our grid is adaptive, the grid spacing on each level of resolution remains constant throughout the domain, hence the grid is called uniform.

The parameters i_h and i_l specify the order in which boundaries are stored in the u array and what type of boundary conditions will be imposed, either algebraic or evolutionary. The order of the wavelets used are specified with N_predict and N_update. These parameters determine how many points are used on either side of the point being interpolated. In our case they are 2, so we are using 4th order wavelets. N_diff is the number of points used on each side of a centered difference scheme to calculate the derivatives. Since N_diff is 2, we are using a 4th order central differencing scheme. Generally it is wise to make all three of these parameters the same.

The next two parameters IJ_ADJ and ADJ_type deal with the fine details of how the code chooses to pick points. It's generally good to leave IJ_ADJ as 1 for all three levels. If ADJ_type is set to 0 it will use less points and run faster, but if convergence becomes an issue, set this parameter to more conservative. This alone may solve a convergence issue when faced with one.

If non-periodic boundary conditions are used, BNDzone should normally be set to true.

It will use the parameters coord_zone_min and coord_zone_max when this is on. BCtype specifies the type of boundary conditions used if custom ones are not needed. The time integration method is chosen in this example to be the Crank-Nicolson scheme. This option is not yet implemented.

The next several variables all involve temporal aspects of the code. The beginning and ending of the simulation are specified with t_begin and t_end. The initial time step used to start off the run is dt. The minimum and maximum time steps are set with dtmax and dtmin. Dtmin can serve as a stability indicator in that many times the time step will become smaller and smaller as the solution begins to diverge. If dt drops below dtmin it is usually because there is a problem. The parameter t_adapt can be used to make all time steps below t_adapt of the same size and once t is greater than t_adapt, the time step is free to change with the solution. The maximum and minimum cfl conditions are set with cflmax and cflmin.

In the compressible case example, we needed the variables gama, delta and radius. They are set here for no reason in particular. Note that the order in which variables are declared is not important.

The variable Zero_Mean is for cases in which you want the mean value of the first 1:dim variables to remain zero. The next several variables are for the Krylov time integrator and the elliptic solver. It is best not to touch these.

If you wish to insert an object into the flow using Brinkman Penalization you would set the obstacle variable to T. Its location, size and movement directions can all be specified using the various different variables. You would probably create a few of your own variables and then make sure to define that objects domain in the user_chi subroutine. The rest of the variables are other integration parameters that should normally be left alone.

3 Compilation and Execution

In order to have one makefile for different computer architectures, the makefile is setup using gmake. The casename.f90 is compiled with the rest of the code in the source directory by providing the compiler flag with the same name, *e.g.* "gmake CASE=casename". Once the code is compiled, it is run with the input argument "casename". As of now the exact syntax used to compile is "gmake DB=db_wrk CASE=[path of case]/casename wlt_3d". Make sure to add the wlt_3d at the end so the compiler will know what to do.

Once the code has compiled, the executable file will be found in the case directory. The code is executed by using the input file as a command line argument for the executable, *e.g.* "./casename.out casename.inp". The output files will be saved in the **results/** subdirectory.

In order to view the output files in MATLAB, the post-process file c_wlt_inter.f90 needs to be compiled. From the source directory compile all the needed files by "gmake DB=db_wrk inter". Once this is done, you are ready to visually view the output using MATLAB.

4 Data Visualization

Once everything has been compiled and executed we can plot the data using the provided Matlab routines. It is important to compile the c_wlt_inter.f90 file prior to data visualization. In the compressible case directory, there should be a file showme3D.m. This program is run within Matlab to visualize your data. This program simply runs the more general routine c_wlt_3d.m routine which contains many different input parameters. These parameters are explained in the showme3d.m file, but we will look at them a bit as well.

If you open the showme3D.m file in the compressible case directory you will notice a line where the POSTPROCESS_DIR variable is defined. Make sure to modify that path to the path of you own post_process directory.

Further down all the arguments used to call the plotting function c_wlt_3d are briefly explained. At the bottom, a call to this function is shown with each argument defined for you. The first argument in this list is the file name 'euler_test.'. You will find this matches the name in the compressible.inp file. The j_range is the range of resolution that Matlab will use to plot your solution. The bounds argument determines the domain that will be plotted.

The figure is set to plot the solution using a contour plot. If you would like to see the grid, you must set the figure type to 'grid'. The plot_comp variable is the component of the solution that you want to plot. This is defined in the compressible.f90 code. The station number is which output step you are plotting. In this case, it is set to 20, which is the final output time for this case.

References

Goldstein, D. E. & Vasilyev, O. V., 2004, Stochastic coherent adaptive large eddy simulation method, *Phys. Fluids*, **16**(7), 2497–2513.