CHAPTER 10 Troubleshooting

Mistakes made in setting up a problem for CFL3D are (hopefully) diagnosed by the code's error checks. The error messages, which are printed to unit 11 should be clear enough so that the user can easily identify and remedy the problem. This chapter contains a few general tips based on common user errors.

Unfortunately, it is sometimes the case with CFL3D (as with any CFD code) that the code blows up, even though the user has apparently done everything right. For example, if the code tries to calculate a negative square root, it will bomb with a floating point error. What to do in such cases is never clear and is often more of an art form than a science. Listed below are some things to try and/or look for when experiencing difficulty with CFL3D (when it blows up for no easily-apparent reason). These limited suggestions for what to try when problems occur are based in part on the code-developers' experiences and in part on what is heard from the user community:

1. When the code gives the message "negative volume" or blows up (end-of-file) when trying to read the grid, check the following:
   (a) Be sure the grid is written correctly, using the right-hand rule (see "The Right-Hand Rule" on page 67) for both x, y, z and i, j, k. Either CFL3D format or PLOT3D/TLNS3D format may be used. See "Grid File" on page 65 for a description of the two formats. Remember to set **ngrid** > 0 for CFL3D format and **ngrid** < 0 for PLOT3D/TLNS3D format.
   (b) Check the grid near the point where the negative volume is indicated. Make sure there are no grid lines that cross or are positioned incorrectly.
   (c) When running on a workstation, be sure to use the necessary precision for arrays (many turbulent grids are too fine to use single precision).
   (d) Be consistent with the precision between the grid and the code (i.e., the grid must be created in double precision if the CFL3D code is "made" that way).
2. Check the grid. It seems that 95% of the time, the problem is with the grid. Too much stretching too quickly, poor "quality", badly-skewed cells, etc. can all cause problems. Also be sure that the grid *and all its coarser levels* are of a reasonable size (when using multigrid, a coarser grid level should not be *too* coarse).
3. Be sure that the correct direction (either z or y) is "up" according to the type of grid being used. See "Grid File" on page 65.

4. If problems occur when starting a run for a very large grid, debugging on a coarser grid is recommended. Unfortunately, running on a coarser grid level using the **mseq** flag still requires a large percentage of the full amount of memory. Because this can be frustratingly slow for very big grids, the use of the following tools, found in the **`Tools`** subdirectory (see “The Code and Supplementary Files” on page 9), is recommended:

`everyotherp3d.f` –	reads a PLOT3D grid and writes out a coarser (every other grid point) grid
`v5inpdoubhalf.f` –	reads a CFL3D input file and creates a new input file applicable for either half or double the number of grid points

   After employing these tools to create a smaller grid and the corresponding input file, rerun `precfl3d` on the new input file and then recompile CFL3D and debug the problem on the coarser grid. The required memory will be substantially smaller than that necessary for the original grid.

5. Be sure to save a copy of the original code *before* making any modifications. If an error *is* made, the revision can be compared with the original version to aid with debugging. If additional subroutines are added, be sure they are added correctly to the makefile. If they are merely “tacked on” to the beginning of line 2 in the makefile, then they will not necessarily be recompiled correctly when `cfl1.h`, etc. are changed.
6. Peruse the output files early in the computation to make sure that they make sense. For example, if part of the grid is in motion, is it moving at the correct speed and in the expected direction? Do the results “look” as expected? Valuable CPU time can be saved by not running erroneous calculations for hundreds of iterations.
7. Check to make sure the boundary conditions are implemented correctly. This includes all 1-to-1 and patched interfaces. For 1-to-1 boundaries, look for “mismatch” in the output files for the “geometric mismatches” that may be a source of error. These numbers should all be close to zero. Small mismatches may be acceptable, but large mismatches (O(1)) most likely indicate that one of the segments on the block boundary has been specified incorrectly, or possibly backwards. For patched or overlapped interfaces, it is quite a bit more difficult to assure valid communication stencils. Check all outputs from ronnie for patched grids and from MaGGiE for overlapped grids.
8. Try lowering the CFL number significantly initially and allowing it to “creep” up as the solution progresses. Unless the grid is *really* bad, the CFL number should not have to be set below 0.1 (**dt** = -0.1). Sometimes it may be necessary to *remain* at low CFL numbers for particularly difficult problems. The optimum CFL number to run at is generally around 5 (**dt** = –5.0), for most problems on decent grids.
9. Make sure the executable is appropriate for the input file, particularly if any changes have been made to it *or* to the grid since the previous run. Run `precfl3d` to make sure the dimensioning is correct in the `cfl*.h` files. If `precfl3d` changes them, CFL3D must be recompiled.
10. Try employing mesh sequencing. Many people have found this to be beneficial for tough problems! For a description of mesh sequencing, see “Mesh Sequencing” on page 134.

11. Peruse all output files (there are a lot of them!) for any clues as to what might be wrong.
12. Try restarting from a less difficult, converged case on the same grid. For example, run and converge the configuration at **alpha** = 0 first; then restart at the higher, more difficult angle of attack (or Mach number or whatever).
13. When *really* desperate, try running with first order in space for awhile (**nitfo** = **ncyc**), then restart with third order from that. Or, try using second order (**rkap0** = -1) rather than third order (**rkap0** = +0.333333).
14. If multigrid is being used, try turning multigrid off for a short time. If multigrid is *not* being used, turn it *on*! In general, for steady-state problems, multigrid should definitely be used. See “Multigrid” on page 125 for instructions.
15. Be sure to use the latest version of the code (as of November 1996, Version 5.0).
16. There only seems to be sporadic success when varying **idiag**, **ifds**, or **iflim**. In general, if trouble arises when the default values are used for these variables, then trouble will occur even if they are altered. There are of course exceptions. For example, often hypersonic flow cases run more successfully with flux-vector splitting (**idiag** = 0). Also, sometimes, if a particular configuration is marginally stable (i.e., on the brink of going unsteady), then flux-vector splitting, which has more inherent dissipation than flux-difference splitting, may yield a steady solution while flux-difference splitting may go unsteady. What this means, however, is unclear, since it may be that the real physics of the flow *should* go unsteady. Presumably in such a case, if the grid is refined extensively, even flux-vector splitting should probably go unsteady.
17. There is little recent experience with this, but experimentation can be made with **ngam** (type of multigrid cycle), **mitL** (number of iterations on each multigrid level), and **mglevg** (the number of multigrid levels).
18. Another feature that is rarely used, but could be experimented with is residual correction and/or smoothing with multigrid. Set **issc** and/or **issr** to 1. (Zero is the default.) These parameters have been known, in isolated instances (particularly for hypersonic cases), to cause a “bombing” solution to work. In fact, some users turn residual correction and smoothing *on* as default. This is *not* recommended however because, for the majority of cases the code developers have seen, these parameters seem to hurt more than help. It probably depends a lot on the type of configuration or case being run.
19. Try running a different turbulence model that is more robust (Baldwin-Lomax and Spalart-Allmaras are probably the most robust of the models). Then restart the case from this converged solution using the desired model.
20. Monitor `cfl3d.turres`. Make sure there are either no or relatively few `nneg` values. Make sure the residual of the turbulent equations is not going *up* gradually over time rather than down. If there are problems, try setting `factor` lower in the appropriate turbulent model subroutine (such as subroutine `spalart` when using the Spalart-Allmaras turbulence model).
21. Sometimes the order of the indices in the grid can make a difference, since the code performs the approximate factorization (AF) in a particular order. Experience has shown that it is usually best to let the *primary* viscous direction (if there is one), such

as the direction normal to the wing surface, be the k direction. Eventually, at steady-state, the answer should be the same regardless of the index directions, but how *well* the code converges to the steady state can unfortunately sometimes be influenced by the choice.

22. When restarting and switching turbulence models, it is sometimes necessary, at least temporarily, to lower the CFL number or time step to get the new solution going. After a time, the CFL number or time step may then be “bumped up” to the desired level.
23. Hypersonic cases can often be difficult to start. Some suggestions are:
    (a) Start with a very low CFL number (on the order of 0.1) and run for a while, then later ramp it up.
    (b) Use mesh sequencing.
    (c) Use flux-vector splitting (**ifds** = 0), at least in the beginning.
    (d) Restart from a previous similar solution.
    (e) Experiment with varying **mitL** (number of iterations on each multigrid level).
24. If the residuals near patched boundaries are high (particularly when the grid sizes are very different near the patch), try replacing calls to `int2` with calls to `int3` instead (`int3` employs gradient limiting on the patch stencil).