User Guide

Curvilinear Navier-Stokes (CNS) C++ Code

Bobby Arthur
Department of Civil and Environmental Engineering
Environmental Fluid Mechanics Laboratory
Stanford University

October 18, 2013

Contents

Chapter 1

Code Structure

1.1 Introduction

This curvilinear Navier-Stokes (CNS) C++ code is also known as "cns-koltakov" or, more affectionately, "Sergey's Code," after its author, Sergey Koltakov. It is the third version of the original Fortran code written by Yan Zang at Stanford in the early 1990's (Zang et al., 1994). The original code was serial (one processor). The second version was updated by Anqing Cui to run with multiple processors using MPI, and is known as PCUI (Cui and Street, 2001). This latest version of the code is basically a translation of PCUI to C++ using an object-oriented framework, which allowed for the addition of a moving grid (Koltakov and Fringer, 2012). Over the years, these various versions of the code have been used to study a wide variety of geophysical flows.

The code is based on the fractional step method of Zang et al. (1994) and includes the LES model of Zang et al. (1993). It solves the incompressible Navier-Stokes equations on a generalized curvilinear grid with a rigid lid, employing a semi-implicit time integration scheme with Adams-Bashforth for the explicit terms and Crank-Nicholson for the implicit terms. Additionally, it uses the QUICK and SHARP schemes for advection of momentum and scalars, respectively, and solves the pressure Poisson equation with a multigrid method.

1.2 File Summary

The code is contained in the directory `/cns-koltakov`, which contains the following files:

`Makefile`
: Used to compile and run the code (see Section 1.3).

`parameters.dat`
: The main input file for the code, which contains the most commonly adjusted variables. The file is read at runtime, so the user may redefine variables without recompiling the code.

`parameters.h`
: Defines all the parameters for the code, including physical variables, boundary conditions, and multigrid parameters. Also allows the user to turn different modules (e.g. scalar advection, moving grid, etc.) on or off, run predefined cases (see Chapter 3), or change the bottom bathymetry. In order to modify `parameters.dat`, the function `Set_Parsable_Values()` must be modified here.

`test.cpp`
: The main loop of the code.

`navier_stokes_solver.h/.cpp`
: Contains the time-stepping scheme for the momentum equations, including, most importantly, the `Predictor()` and `Corrector()` schemes. Also where all arrays containing physical variables are initialized. Additionally, the scalar solve, background potential energy, post-processing, and other similar functions are called here.

`pressure.h`
: Contains the multigrid pressure Poisson equation solver.

`scalar.h`
: Updates scalar quantities after the velocity has been updated.

`convection.h`
: Contains the advection schemes used in the code, specifically, QUICK for momentum advection and SHARP for scalar advection.

`universal_limiter.h`
: Contains the limiter used in the above advection schemes.

`tridiagonal_solver.h`
: Contains functions used to solve tridiagonal systems with periodic or non-periodic boundary conditions. Used both in the implicit part of the momentum and scalar solutions, as well as in the smoothing operation in the multigrid solver.

`curvilinear_grid.h/.cpp`
: Creates the grid and calculates the necessary metric quantities. Note that a custom grid (stored as a binary file) can be used if `read_grid_from_file` is set to `true` and `grid_filename` is specified in `parameters.h`.

`mpi_driver.h/.cpp`
: Initializes and controls parallel implementation using MPI. Also defines binary output functions.

`metric_quantities.h`
: Defines the metric quantities that are calculated in `curvilinear_grid.cpp`.

`array_Xd.h/.cpp`
: Classes that define all array operations in the code, where `X` is a number 1-3.

`potential_energy.h`
: Calculates the background potential energy and related quantities. Note that kinetic energy and dissipation are calculated in `navier_stokes_solver.cpp`.

`turbulence.h`
: Contains the turbulence model, which is not fully implemented (yet).

`curvilinear_moving_grid.h/moving_grid_engine.h`
: Implementation of the moving grid.

`interpolant.h`
: Used with the moving grid to interpolate physical variables from the old to new grid.

`data_aggregator.h`
: Aggregates a time-series of physical variables.

`parameter_file_parser.h`
: Reads the input file `parameters.dat`.

1.3 Getting Started

The code uses the `git` version control system, and is stored in an online repository on `github`. To download the code onto your local machine, use the following command:

```
$ git clone https://github.com/barthur1/cns-koltakov.git
```

This will create the `/cns-koltakov` directory, which includes all the necessary files for the code, in your current directory. Use `git` to back-up your own version of the code, and to manage your own revisions locally. Check `http://www.git-scm.com` for a tutorial and full documentation for `git`.

To begin using the code, it must first be compiled with the Intel C++ Compiler (`icc`) and the MPICH compiler (`mpicc`). A free version of the Intel C++ Compiler can be found at `http://software.intel.com/en-us/non-commercial-software-development`, while MPCIH can be found at `http://www.mpich.org/downloads/`. Note that MPICH must be compiled on your machine with the Intel compiler `icc` flag, see the MPICH documentation for specifics.

Once the compilers have been downloaded, make sure that the path to `mpicc` is specified correctly in the Makefile next to `CC=`. Then, the code can be compiled with `make test`. Note that the `-DNDEBUG` flag should be removed during development to make use of existing `assert` statements in the code. Once compiled, the code can be run with `make runX`, where

`X` is the number of processors. Note that this number must agree with what is specified in `parameters.dat`. That is, `X = num_cpu_x*num_cpu_y*num_cpu_z`.

Chapter 2

Code Output

2.1 Binary Output

The code output is contained in binary files that correspond to a specific processor. Data is output for the variables specified in `parameters.h` at the frequency defined by `save_timestep_period` in `parameters.dat`. These files will be stored in `output_dir` defined in `parameters.dat`. The files are organized by variable and processor. For example, the velocity output from the 0th processor will be stored in `velocity.0`.

2.2 Loading and Viewing Output Data

The `/cns-koltakov/mfiles` directory contains MATLAB scripts to view data from binary output directly, or to load `.mat` files which can be used in your own scripts. The files included in this directory are as follows.

`cns_viz.m`
: Plots 2D slices of any physical variable or 3D isosurfaces of density and vorticity.

`cns_load.m`
: Loads physical variables into matrices that are stored in a `.mat` file.

`cns_load_energy.m`
: Loads a time series of energy quantities and saves a `.mat` file.

`load_binary_X.m`
: Functions that load data directly from binary output files and return a 3D (x,y,z) matrix for a given time step.

`calculate_binary_X.m`
: Functions that calculate other quantities (e.g. vorticity) using data from binary output files.

2.3 Restarts

The code is able to save data for restarts if a nonzero value for `restart_timestep_period` is given in `parameters.dat`. This will save restart files at the specified period named `restart_tX.0`, where `X` is the time step. To run a restart job, specify `restart_timestep` and `restart_dir` (where the restart files from the old run were stored) in `parameters.dat`. Make sure to change `output_dir` so that your old data is not overwritten. Then, run the job as usual with `make runX`. Note that `max_timestep` still starts from 0, not from the restart time step.

Chapter 3

Examples

The following example cases have been set up in the code. Going through them provides a good introduction to different parameters and variables, as well as how to view output using `cns_viz.m`.

3.1 Internal Seiche

To run the internal seiche test case, set `internal_seiche` to `true` in `parameters.h`. This will initialize the density field with

$$\frac{\rho}{\rho_0} = 1 - \frac{\Delta\rho}{2\rho_0}\tanh\left(\frac{2(z + H/2 - a\cos(\frac{2\pi}{\lambda}x)}{\delta}\tanh^{-1}(\alpha)\right), \tag{3.1}$$

as shown in Figure 3.1. Try running this case with a domain size of 1×0.25×1 m, a grid of size 32×8×32, and a time step of 0.1 s. The initial density profile for this case is shown in Figure 3.1

3.2 Lid-driven Cavity

To run the lid-driven cavity test case, set `lid_driven_cavity` to `true` and `scalar_advection` to `false` in `parameters.h`. This will initialize a horizontal velocity $u = 0.1$ m/s on the top of the domain. Also set `molecular_viscosity` to 10^{-3} m^2/s in `parameters.dat`. Over time, circulation will develop in the domain. Try running this case with a domain size of 1×0.25×1 m (giving $Re = uL/\nu = 100$), a grid of size 32×8×32, and a time step of 0.1 s. After 300 time steps, or 30 s, the flow should look like Figure 3.2.

3.3 Lock Exchange

To run the lock exchange test case, set `lock_exchange` to `true` in `parameters.h`, and make sure `scalar_advection` is set back to `true`. The domain will be initialized with dense fluid

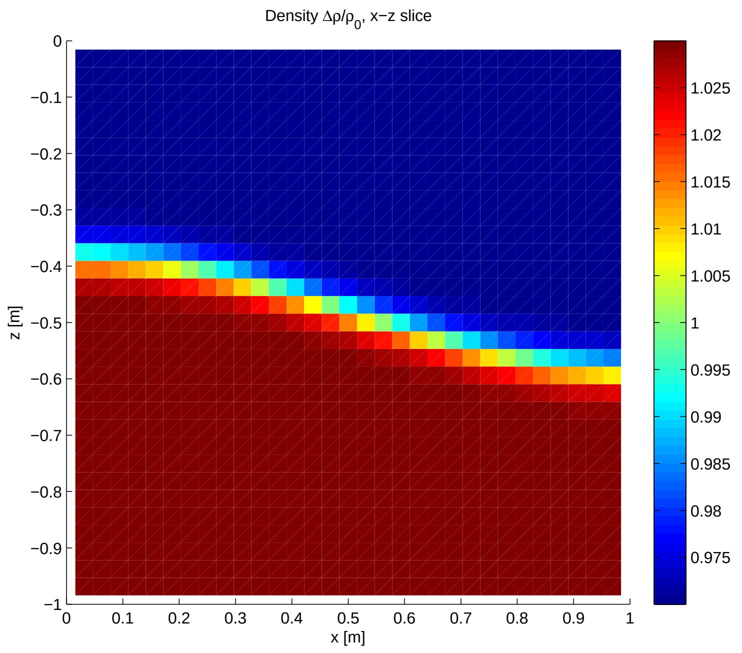

Figure 3.1: Initial density profile for internal seiche case, plotted using `cns_viz.m`.

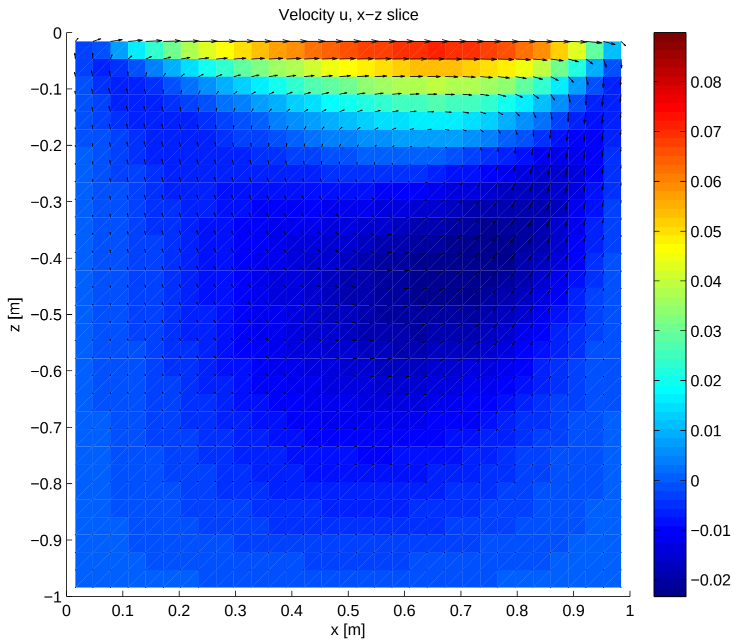

Figure 3.2: Horizontal velocity with velocity vectors for lid-driven cavity case after 300 time steps, or 30 s, plotted using `cns_viz.m`.

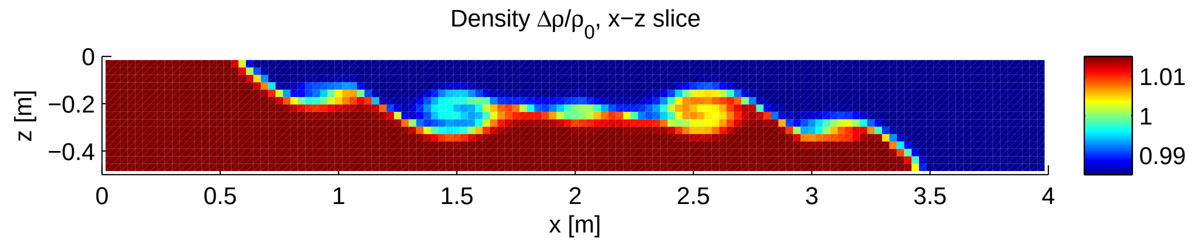

Figure 3.3: Density structure of lock exchange after 800 time steps, or 80 s, plotted using `cns_viz.m`.

on the left side and heavy fluid on the right. Over time, density currents will develop and propagate in both directions. Try running this case with a domain size of 4×0.25×0.5 m, a grid of size 128×8×16, $\nu = 10^{-6}$ m^2/s, a time step of 0.1 s, and a free-slip bottom boundary condition. After 800 time steps, or 80 s, the flow should look like Figure 3.3. Check out the shear instabilities that develop at the interface!

3.4 Breaking Internal Wave

Two types of internal wave cases can be run in the code: solitary waves and progressive waves. To run a solitary wave, set `solitary_wave` to `true` in `parameters.h`. To run a progressive wave, set `progressive_wave` to `true` in `parameters.h`. For both cases, set `variable_fixed_depth` to `true` in `parameters.h`, and specify the start of the slope `x_s`, the slope value `slope`, and the interface depth (h_1) `upper_layer_depth` in `parameters.dat`. Try running both cases with a domain size of 3×0.25×0.5 m, a grid of size 512×16×32, $\nu = 10^{-6}$ m^2/s, a time step of 0.01 s, and a no-slip bottom boundary condition. Set the slope to start at $x = 1$ m and the slope itself to be 0.218. Also specify $\alpha = 0.99$, $\delta = 0.1$ m, $\Delta\rho/\rho_0 = 0.03$, $\rho_0 = 1000$ kg/m^3, and $h_1 = 0.3$ m.There are additional parameters for each case that can be specified in `parameters.dat`. For the solitary wave, set the amplitude `a` to 0.1 m and the wavelength `Lw` to 0.7 m. For the progressive wave, set `forcing_amp` to 0.05 and `forcing_period` to 10 s.

Note that the solitary wave case uses closed boundaries, and is created by an initial gaussian in the interface on the left (west) side of the domain. The progressive case, however, uses

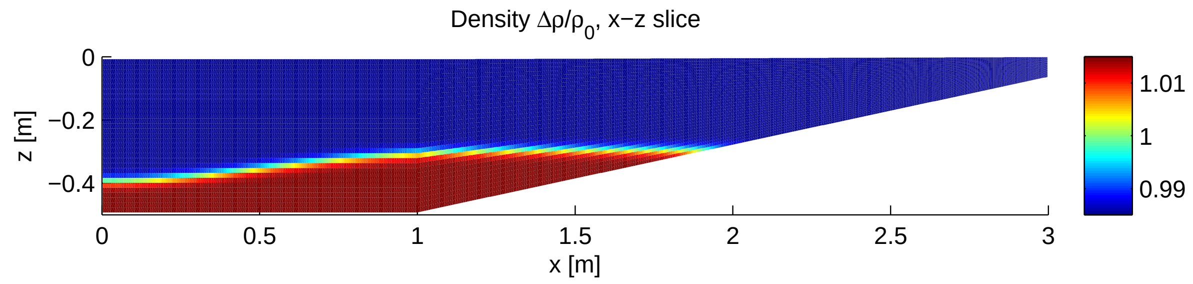

Figure 3.4: Initial density profile of solitary wave case, plotted using `cns_viz.m`.

an open boundary condition, where the horizontal velocity `u.x` and horizontal volume flux `U_xi` are specified on the west boundary. These fluxes must conserve volume in the domain by having a net zero flow. Look at `Set_Progressive_Wave_BC()` in `navier_stokes_solver.cpp` and `Quick_Velocity_Flux_Update()` in `convection.h` to see how this boundary condition is implemented.

The initial condition for the solitary wave run can be see in Figure 3.4, while Figure 3.5 shows a progressive wave after 1200 time steps, or 12 s. Run each case yourself to see the wave breaking on the slope! Note that this example has a much larger grid size than the previous examples; you will need to use multiple processors to get a reasonable runtime.

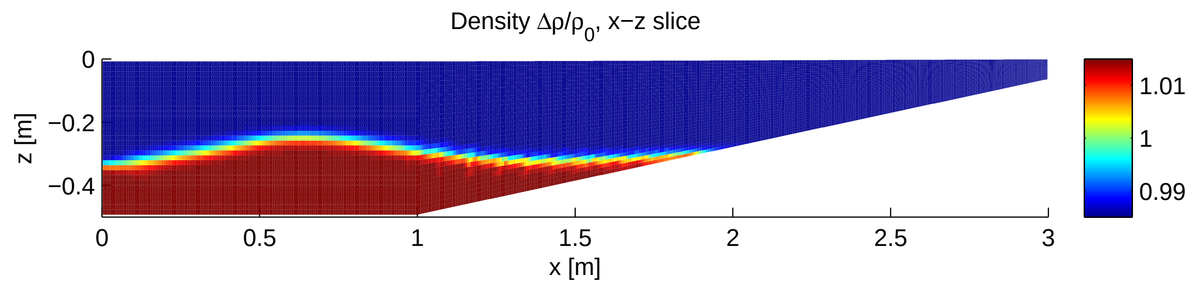

Figure 3.5: Density profile of progressive wave case after 1200 time steps, or 12 s, plotted using `cns_viz.m`.

Bibliography

Cui, A. and Street, R. L. (2001). Large-eddy simulation of turbulent rotating convective flow development. *J. Fluid Mech.*, 447:53–84.

Koltakov, S. and Fringer, O. B. (2012). Moving grid method for numerical simulation of stratified flows. *Int. J. Numer. Meth. Fl.*

Zang, Y., Street, R. L., and Koseff, J. R. (1993). A dynamic mixed subgrid-scale model and its application to turbulent recirculating flows. *Physics of Fluids A: Fluid Dynamics*, 5:3186.

Zang, Y., Street, R. L., and Koseff, J. R. (1994). A non-staggered grid, fractional step method for time-dependent incompressible navier-stokes equations in curvilinear coordinates. *J. Comput. Phys.*, 114(1):18–33.