Rayleigh
Notes on Building and Running the Code

Nick Featherstone
feathern@gmail.com

1. Required Libraries

Rayleigh requires that a few external libraries are installed before you can compile it:

1. BLAS
2. LAPack
3. FFTW3
4. MPI

If you are using Intel's 2012 (or later) compiler, the Math Kernel Library (MKL) is included and provides interfaces to FFTW, LAPack, and BLAS routines. Examples of MKL and non-MKL compiles are included in the Makefiles directory. Specifically, Makefile_Nick provides an example for linking items 1-3 and compiling with gfortran. Makefile_LCD, Makefile_Pleiades, and Makefile_Discover illustrate complilation using modern Ifort and MKL.

2. Building the code

Before building Rayleigh, you will likely need to copy one of the existing makefiles and modify it for your own machine. Each makefile refers to another file called object_list. That file should not be modified and is intended to be machine-independent. Place the new makefile in the Makefiles directory, and name it Makefile_machinename, where machinename indicates the machine you are using.

Once you have a makefile, building the code is straightforward. Within the rayleigh_dev directory, you will find an executable script named *build_rayleigh.* To run this script, type

```
./build_rayleigh machinename
```

This build script copies all relevant source files to the build directory, along with the relevant Makefile from the *Makefiles* directory. Each Makefile is tagged with a machine name – same same machine name you would use in the command above. For example, to build the code on my personal laptop, I type

```
./build_rayleigh Nick
```

so that Makefile_Nick is copied to the build directory. To build the code on Pleiades, I type:

```
./build_rayleigh Pleiades ,
```

and Makefile_Pleiades is copied to the build directory. Following these copies, the script executes a “make rayleigh” command, and an exectuable named *rayleigh* is built within the build directory.

Build_rayleigh also accepts three optional arguments following the machine name. Each of these is assigned to an environment variable named RAYLEIGH_OPT1, RAYLEIGH_OPT2, and

RAYLEIGH_OPT3. These variables may be seen by the Makefile and can be used to control the flow of the makefile (using debugging flags instead of optimized flags, for instance).

As an example, running

```
./build_rayleigh Pleiades debug
```

would compile the code with Intel's debugging flags (see Makefile_Pleiades).

Note: Depending on how your environment is set up, you might run into issues with the build script trying to use the Makefile. If that's the case, run the build_script (so that it copies the source files over), and then just move to the build directory and make the code directly from the command line (*make rayleigh*).

Preparing to run the Code

Each simulation run using Rayleigh should have its own directory. The code is run from within that directory, and code output is stored in various subdirectories. This means that, wherever you create your simulation directory, you need to have sufficient space (/nobackupp on Pleiades, for example) to store the output.

After you create your run directory, you will want to copy (cp) or softlink (ln -s) the executable from rayleigh_dev/build to your run directory. I like to soft-link, so that the executable is always up-to-date. Next, you should copy the script named make_dirs to your run directory. Execute this script from within your run directory. This will create the various subdirectories needed by the code (will discuss what these are later).

Finally, you need a main_input file. This file contains all the information that describes how your simulation is run. Rayleigh always looks for a file named main_input in the directory that it is launched from. For now, copy the main_input_hydro file from the input_examples/anelastic_benchmarks directory to your run directory. Rename it to main_input. This input file is setup to run the anelastic benchmark from Jones et al. (2011, Icarus 216, 120).

Running the Code

We will examine the main_input file in detail shortly but, first, we want to just run the code. While an OpenMP/MPI hybridization is about 30% complete, Rayleigh is currently parallelized using pure MPI and a 2-D domain decomposition. The pure MPI bit means that if you have any OpenMP environment variables set for FFTW, LAPack, or BLAS, you should probably disable them. The 2-D domain decomposition bit means that we envision the MPI Ranks as being distributed in rows and columns. The number of MPI ranks within a row is *nprow* and the number of MPI ranks within a column is *npcol.* When Rayleigh is run with N MPI ranks, the following relation must be satisfied:

$$N = \text{npcol} \times \text{nprow} \quad .$$

If this relation is not satisfied , the code will complain and exit.

If you look at the main_input file, you will see that it is divided into Fortran namelists. The first namelist is problemsize_namelist. Within this namelist, you will see a place a specify nprow and npcol. Edit the makefile so that nprow and npcol agree with the N you intend to use. The O(0) effect on scalability is the number of messages sent per iteration. This means that nprow and npcol should be as close to one another in value as possible. The O(1) effect is cache latency. This means that if nprow and npcol cannot be equal, then nprow should be largest. When nprow increases, fewer Legendre polynomial matrices have to be pulled into/out of memory during each iteration. In summary:

1. N = nprow x npcol.
2. nprow and npcol should be equal.
3. If they cannot be equal, increase nprow first.

Once you have picked N, nprow, and npcol, you are ready to run the code. This will probably be using either mpiexec or mpirun. To run the code, you just type (or put in your job script):

```
mpiexec -np N ./rayleigh
```

Alternatively, you can specify nprow and npcol at run time (as opposed to main_input). This is useful because you can modify the job script without modifying the main_input file. In this case, suppose X and Y were the values for nprow and npcol, then you would type/insert into your jobscript:

```
mpiexec -np N ./rayleigh -nprow X -npcol Y
```

X and Y overwrite whatever values of nprow and npcol were specified in main_input.

If everything worked, you should see the code run for 10 time steps and print some timing information before exiting.

MPI Environment Variables

It is usually a good idea to set the following variables in your MPI job scripts – otherwise you may not have enough buffer space set aside for MPI. This can cause slowdowns and/or hangups as MPI tries to allocated sufficient buffer space on the fly.

```
setenv MPI_BUFS_PER_PROC 64
setenv MPI_BUFS_PER_HOST 256
```

The main_input File

Rayleigh simulations are controlled by the main_input file. We'll go over a few of the important options in that file here.

problemsize namelist

Rmin and rmax specify the inner and outer radius of the spherical shell (rmin should never be zero). N_r and n_theta specifiy the number of radial and latitudinal colocation points.

N_*r and n_theta should always be even.*

numerical_controls namelist

Chebyshev set to .true. Set this to .false. to use finite-differences in radius.

physical_controls namelist

Rotation and/or magnetism can be switched on and off here.

temporal_controls namelist

max_time_step -- The time step taken is controlled by the CFL condition. Set this to enforce an even more stringent time step. time_step = min(CFL delta_t, max_time_step).

max_iterations – the maximum number of time steps for which you want to integrate the simulation (usually set this to something big).

check_frequency – Number of time steps to take between checkpoint dumps.

output namelist

The output namelist controls governs the output of different diagnostics. Diagnostics are computed in physics/Diagnostics.F90. At the top of that file, you will see a menu of available diagnostics and their corresponding quantity codes. The different diagnostics are computed in the subroutine PS_Output.

Each diagnostic quantity may be output in 6 different formats in Rayleigh

Shell Slices:	Lat/Lon slices through the simulation of predefined radial indices
Azimuthal Averages:	Radius/Latitude profiles of diagnostics averaged in longitude (phi).
Shell Averages:	Radial profiles of diagnostics averaged over the sphere at each radius
Global Averages:	Diagnostics averaged over the full spherical shell
Shell Spectra:	Spherical harmonic spectra of diagnostics taken at various radial indices
Full 3D:	Full 3-D dumps of the specified diagnostic.

All outputs types share three common namelist elements:
output_values : The quantity codes for the diagnostics that are to be output this way
output_frequency: The number of time steps taken between dumps of this output type
output_nrec: The number of dumps (individual timesteps) stored in the output file

Shell slices and shell spectra use an additional namelist element:
output_levels: radial indices (1 through n_r) of the slices or spectra to be output

Each output-type is stored in a directory (made by make_dirs) with a similar name. Files are numbered by the time step of the last output in the file. For example, suppose
globalavg_frequency = 3
globalavg_nrec = 2

Time steps 3 and 6 would be stored in a file named G_Avgs/00000006 and time steps 9 and 12 would be stored in file named G_Avgs/00000012

Boundary_Conditions_Namelist

The boundary conditions are set up for fixed entropy (T_top and T_Bottom) at the upper and lower boundaries. For now, anything that says T or Tvar should be interpretted as the thermal variable, which is entropy if you are stratified. If you're running a Boussinesq system, you can think of tvar as

temperature.   You can alternatively run with fixed flux at the top. To do so, you have first turn off fix_tvar_top amd then turn on fix_dtdr_top:
fix_tvar_top = .false.
fix_dtdr_top = .true.

Initial_Conditions_Namelist

See the comments in main_input_hydro, but two important things. init_type controls the hydro init and magnetic_init_type controls the initialization of the magnetic stream functions.

init_type, magnetic_init_type = -1	: restart from checkpoint
init_type,magnetic_init_type = 7	: random thermal and magnetic perturbuations
init_type = 6	: hydro anelastic benchmark init

Test Namelist

Leave this alone

Reference_Namelist

This controls the setup of the reference state.   See comments.   For the moment, all you can really do is set up a polytropic atmosphere.

Non-dimensional Namelist

Leave this alone except for angular_velocity. This is where the rotation rate is set – will move this later.

Transport Namelist

This is where the transport coefficients nu, kappa, and eta are specified. By default, these coefficients are constant in radius. Alternatively, they can be made to vary as a function of density. To change nu so that nu ~ 1/sqrt(density), you would add:
nu_type = 2
nu_power = -0.5d0

nu_top, kappa_top, and eta_top determine the value of these coefficients at the outer boundary.

Reading & Plotting Output

The rayleigh_dev directory contains a subdirectory named reading_routines, which in turn contains idl and python subdirectories. The easiest thing to do is download the sample_output tarball from the website, untar it, and copy all the .py routines into that directory. There are four examples of plotting that demonstrate how to read in the output:
plot_spectrum.py
plot_shell.py
plot_shellavg.py
plot_gavg.py
plot_azavg.py is only partially functional at this time..

Load Balancing: Choosing nprow and npcol

When running rayleigh, it is important to consider the load balancing. Radial levels are distributed across across the process columns. This means that N_r = n x npcol for some integer n. Ideal n and npcol would be factors of N_r, but the code will run even if they are not (but npcol must be no greater than N_R)

Spherical harmonic modes are distributed in low-m, high-m pairs across a process row. This means that, ideally, (lmax+1)/2 = m x nprow for some integer m.

The relation between lmax and n_theta is that n_theta = 3/2 x (lmax+1).

Here's a concrete example, since maybe that's confusing. Suppose N_R = 128 and n_theta = 192. This means that lmax+1 = 128. If we are running with 256 cores, a good choice is nprow = npcol = 16. If we run with 512 cores, good choices are nprow = 32, npcol = 16 and nprow=16,npcol = 32. Though the first choice is better.

You can increase nprow as high as lmax + 1, but the load balancing bad in that case. It's much better to keep nprow at (lmax+1)/2 or less. In practice, the communication time grows to outweigh the workload somewhere around ¼ of the maximum number of cores for all but the largest problems. That's still a lot of cores though. You should be able to run this sample case pretty efficiently at 1024 cores, for instance, with npcol = 64 and nprow = 32.