MANUAL FOR PADE DISK CODE

KARIM SHARIFF WITH ROUTINES BORROWED FROM ALAN WRAY

(1) The `pade` code simulates protoplanetary disks in cylindrical coordinates (r, z, ϕ). Currently the compressible inviscid/viscous hydrodynamic equations have been implemented. It is a finite-difference code and the compact 4th-order standard Padé scheme is used for spatial differencing. Padé differentiation is known to have spectral-like resolving power. Numerical boundary schemes (for the non-periodic direction) are chosen that lead to collapsing sum derivatives to ensure conservation.

The 4th order Runge-Kutta is used for time advancement. A more accurate version of the FARGO technique (compared Masset's original implementation) for eliminating the time-step restriction imposed by Keplerian advection has been implemented. Capturing of shocks that are not too strong is done using artificial bulk viscosity.

The code can be compiled in serial or parallel (mpi) mode.

(2) Additional capabilities that would be desirable are (i) Shearing box. (b) Particles. A strong desire on the part of others to use the code would motivate me to put these in.

(3) Parallelization is achieved using the pencil data structure in which one of the complete directions is in the processor memory. We say that an array is in z space, for example, when an array has all of z and a portion in ϕ and r. Of course, at a given time, each processor can have arrays in any of the three spaces. Going from one space to another is done using transpose routines which I borrowed from Alan Wray's stellar box code.

Most of the work is done in z space (long part of brick along z and short parts in r and ϕ. A typical dimensioning of the flowfield array `q` in z space is:

```
real(8), dimension(sr:er, sphi:ephi, nz, ndof) :: q
```

where `sr` and `er` are the starting and ending indices in r that the processor has. Similarly `sphi` and `ephi` are the starting and ending indices for the ϕ direction. The processor has all of z. `ndof` is the number of "degrees of freedom", i.e., the number of flow variables which is currently 5.

(4) The distribution file should be named `pade_x.x.tar`, where `x.x` is the version number. To untar the distribution file type

```
tar -xvf pade_x.x.tar
```

This will create a sub-directory called `PADE_x.x`, where `x.x` is the version number. This directory contains:

(a) All the `.f90` source files, `Makefile`, and `input_file` which is the (namelist) input file you will use for specifying run parameters.
(b) The directory `PADE_x.x` also contains a sub-directory called `COMPILED_MODULES` where `.mod` files created by the fortran compiler will sit.
(c) The sub-directory `DOC` which contains this manual.
(d) A simple unix command file `clean_run` which clears your run directory of any `.dat`, `.tec`, save, and restart files.
(e) A sample PBS script file `pade.pbs`.

(5) To compile the code into a serial executable (`pade_mpi`) you will need a fortran 90 compiler. Currently the compiler options in the makefile are for gfortran and ifort. To compile the code into a parallel executable (`pade_mpi`) you will need, in addition, mpi. OPENMPI and MPICH are open source offerings and are easily installed on your machine. Make sure that you use the `--enable_fortran` and correct `--prefix` option (for the location of the library) at the configure step of the MPI installation.
(6) The code uses an FFT to implement the corrected Fargo method for Keplerian advection. In the makefile you can either choose the Rogallo fft (which the code comes supplied with) or the FFTW library.
(7) If you choose FFTW, you will need to download it from the web. I usually download FFTW in a sub-directory in my home directory, for example,

```
~/Applications/fftw-3.3.7
```

Then to install FFTW type, for example,

```
./configure --enable-fortran --prefix /home5/kshariff/Applications/fftw-3.3.7
make
make install
```

Note that you will need in `Makefile` to point to the correct `lib` and `include` directories for FFTW.
(8) Currently, for simplicity of the coding, we do not allow the number of processors to be changed between restarts. This can be changed in the future by copying the appropriate code from Alan Wray's solar code.
(9) To compile the code, edit `Makefile` and make sure that near the top
(a) The setting of the variables `fortran` and `mpi_fortran` near the top correctly reflect the invocation of fortran90 and mpi fortran90 on your system.
(b) The paths for the FFTW include and lib directories are properly set.

Then type

```
make pade
```

or

```
make mpi_pade
```

to create the serial and/or parallel versions of the executable, respectively. You can also compile the code in debug mode. This will enable output into stdout and

a run time check on array bounds being exceeded. To do this, start with a clean source directory (if needed) and use `mode = debug`:

```
make clean
make pade mode=debug
```

or

```
make clean
make pade_mpi mode=debug
```

Note: Debug mode incurs an overhead and should not be used for production runs.

(10) To run the code, create a run directory somewhere since lots of output files may be generated. Copy

```
input_file
```

into the directory. In the input file you can choose a run type. Currently the run types are:

(a) Test of a vertically propagating acoustic wave (z-dependence only).
(b) Test of vertical hydrostatic equilibrium (z-dependence only).
(c) Test of homentropic solid body rotation (r-dependence only).
(d) Vertical shear instability.
(e) Vertically integrated disk (r and ϕ dependence).
(f) Viscous Taylor-Couette flow which I used to test the viscous coding.
(g) Two vortices for the FARGO paper.

Make sure that the directory `PADE_x.x` is in your path. Then invoke the executable by typing

```
pade
```

or (assuming you want 32 processors)

```
mpirun -np 32 mpi_pade
```

for the serial or parallel executables, respectively.

(11) Before running in parallel you should ensure that your choice for the number of grid points and number of processors can be perfectly packed into pencils. To do this run the code `part_tool` which stands for "partition tool." To create the executable `part_tool` type `make part_tool` in the `TOOLS` directory.

(12) Most likely, the current run types will not suit your purpose. In this case you can set up/modify `user_application.f90` to suit your purpose and invoke it using `run_type = 0` in `input_file`. You can also get further hints for setting-up a user application by looking at existing application subroutines such as

```
app_vsi_3D.f90
```

You will see that the basic steps in an application subroutine are the following.

(a) Read some run parameters needed from the namelist file `input_file`.

(b) `call initialize` with the appropriate arguments. In case of a restart, `call initialize` will cause the restart file to be read. For a fresh start it will also generate the mesh.
(c) Assign the initial field in the q array (for fresh start).
(d) Set up a time stepping loop. The main ingredients in this loop will be:
(e) `call rk4`, the fourth-order time stepping subroutine.
(f) Call routines for plotting output at regular intervals. You can invoke existing plotting output routines which can be found in `plotting_output.f90`. Or you can write your own.
(g) `call finalize`. This will write a save file named `save`, and finalize mpi for an mpi run.
(h) Degree of freedom indices defined in `module dof_indices` are
`integer, parameter :: irho = 1, rmom = 2, zmom = 3, amom = 4, ener = 5`
`ener` is the internal energy $\rho c_v T$. We use the internal energy instead of the total energy for a reason related to the FARGO method and explained in our FARGO paper. To use the above indices `use dof_indices` in your subroutine.

(13) At the end of a run that completed successfully or the code itself aborted (rather than the system aborting the run), a file called `return_status.dat` is written. This file contains only one line with a 0 (normal return) or 1 (abnormal return; refer to `stdout` for the error message.

This allows resubmitting PBS jobs.