INSTRUCTIONS FOR RUNNING PADÉ

KARIM SHARIFF
JUNE 20, 2024

1. Basic description

PADÉ simulates protoplanetary disk hydrodynamics 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. The z direction can be periodic or non-periodic. The 4th order Runge-Kutta method is used for time advancement. A more accurate version of the FARGO technique for eliminating the time-step restriction imposed by Keplerian advection has been implemented. Capturing of shocks that are not too strong can be done by activating using artificial bulk viscosity.

2. Test and application subroutines

There are several application subroutines and test case subroutines that come with the code. You can list them by going into the `/Src` directory and typing `ls -1 app*` giving

```
app_euler1d_tests.f90
app_homentropic_solid_body_rotation_test.f90
app_hydrostatic_test.f90
app_pade_diff_test.f90
app_single_vortex_fargo_test.f90
app_taylor_couette.f90
app_vortex_pair.f90
app_vsi_3D.f90
app_user.f90
```

In the main directory of the distribution is a sample namelist input file called `input_file`. One selects the application/test to be run by setting the value of `i_run_type`. The input data for each application/test follows in different sections.

Note: The following subroutines are very simple 1D tests and run either in serial mode or with mpi and a single processor:

```
app_euler1d_tests.f90
app_homentropic_solid_body_rotation_test.f90
app_hydrostatic_test.f90
app_pade_diff_test.f90
```

The subroutine `app_user.f90` is a placeholder for a user to write his/her own application. It is populated with the basic elements that every application will typically have. The user can modify it to suit his/her purposes.

To make a run requires an executable called `pade` and `input_file`. We next describe how to set up the Makefile so as to create an executable.

3. Setting up the makefile

(1) Begin by editing `/Src/Makefile`. It begins with a section that must be set-up by the user. The first variable that needs to be set is `parallel`. Set it equal to `yes` to generate an mpi code or `no` to generate a serial code. Note: The name of the final executable will be `pade` for both cases.
(2) You can set the rest of the variables equal to `no`. However, read their descriptions in case you need them (especially for debugging) in the future.
(3) The next thing you need to do is specify how to invoke your fortran90 compiler and your mpi fortran. The variables you will need to set are `fortran_compiler` and `mpi_fortran`. You can do this for the different hosts you use as indicated in the conditional statements. Note that certain things (mostly parameters for run time checks) are set depending on whether `fortran_compiler` is `gfortran` or `ifort`. If you use the different compiler then you will have to add the options for your compiler.
(4) The code uses an FFT to implement the corrected Fargo method for Keplerian advection. In the makefile you can either set the variable `fft` equal to `fftw` to use the FFTW library, or set `fft = rogallo` if you are too lazy to install the FFTW library. The Rogallo fft comes supplied with the code. For production runs I recommend that you use FFTW since I used the Rogallo FFT only very early on during code development and cannot guarantee its correctness. If you set `fft = fftw` then you will need to specify `fftw_include_path` and `fftw-library-path`.
(5) That's it.

4. Compiling the code

(1) The executable is called `pade` and is created in `/Src` by issuing `make pade` in `/Src`.
(2) For future use, note that there are three optional command line variables you can give to `make` for debugging/timing purposes. These variables are `print`, `checks`, and `transpose_timing`.
   - `make pade print=yes` causes extremely verbose output to be written stdout about what it is doing. If you wish to make use of this feature in your own user application, you should enclose your print statements as follows:

```
#ifdef debug_print
   if (my_node .eq. 0) print *, ' node 0: Sample debugging output'
#endif
```

- `make pade checks=yes` causes compiler run-time checks to be enabled. Currently, only the options for the gfortran and ifort compilers have been implemented.
- `make tranpose_timing=yes` causes information to be output about the cpu time taken by transpose routines.

5. Running the shock-tube test case

If you have successfully compiled the code, go to the directory `/Test_cases/Shock_tube`. There you will find `input_file`, the input file for this test case. If `/Src` is in your path, then you can simply type `pade` to run the test case serially or `mpirun -np 1 pade` to run it with one cpu. At the end you will see output files for the pressure, density, and velocity at $t = 0, 0.15, 0.45$, and 0.60, for instance `rho_t_0000.6000_step_001026.dat` for the density at $t = 0.60$. Each file has three columns, x, the value of the field variable, and the exact solution. You can plot the variable using your favorite xy plotter. If you want to clean up the directory, leaving only `input_file`, you can type `clean_run` which executes a shell script in `Shell_scripts` provided it is in your path.

6. Running the axisymmetric VSI (vertical shear instability) test case

- We will run this test case with multiple processors so make sure that you set `parallel = yes` in /Src/Makefile if you have previously set `parallel = no`. If you need to recompile the code in parallel mode, go to `/Scr` and type

  ```
  make clean
  make pade
  ```

- The resolution for this run is $360 \times 256 \times 1$ ($n_r \times n_z \times n_\phi$) and let us plan to use 4 cpus. We know that the grid will evenly divide into 4 cpus but to make sure we can run the fortran90 code `/Tools/partition_tool`. To do this, go into `/Tools` and type `make partition_tool`. The `Makefile` in the `/Tools` directory invokes `gfortran` so you may need to edit the Makefile in case you are using a different fortran90 compiler.
- Assuming that `make partition_tool` we can run it as follows:

  ```
  >> partition_tool
   enter nr, nz, nphi, num_nodes--->360 256 1 4
    ***** Found a solution *****
    ng1 =            4  mr =           90
    ng2 =            1  mphi =            1
    mz_r =           64  mz_phi =          256
  ```

  The output tells us that the processor layout will be 4×1 (`ng1` $\times$ `ng2`), i.e., the group size along one direction is 4 and along the other is 1.
- To run the test case go to `/Test_cases/Axisymmetric_VSI` and type

  ```
  mpirun -np 4 pade
  ```

The code should run 10,000 steps up to $t = 10.96$.

- Let us run it for another 10,000 steps. To do this go into `input_file` and set `restart = .true.` and `perturb = .false.`. Next, we want to rename (or copy) the last "save" file into a restart file:

  ```
  cp mpi_save_version2_0010000 mpi_restart_version2
  ```

  For fun we will change the number of processors to 8. You are allowed to change the number of processors provided your choice results in a valid partitioning.

  ```
  mpirun -np 8 pade
  ```

- The code should now have run up to step 20,000 and $t = 21.05$. 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.

7. Writing your own application subroutine

The source file `app_user.f90` is a place holder for you to write your own application. It is populated with the main elements that any application must have. You can also start with a current application subroutines (whose source file name is prefixed with `app_`) that is closest to what you would like to accomplish.

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

(1) Read some run parameters from the namelist file `input_file`.
(2) Call a sequence of set up routines. All set up routines can be found in `set_up_routines.f90`. All of these three must be called in this sequence:

  ```
  call set_up_domain_mesh_and_partition
  call set_up_thermal_parameters
  call set_up_boundary_conditions
  ```

(3) Activate certain features by calling optional "activate" subroutines. These routines are to be found `activate_routines.f90`. Here is a complete list:

  ```
  call activate_gravity
  call activate_fargo
  call activate_pade_filter
  call activate_artificial_pressure
  call activate_viscosity
  call activate_plotting_shift
  ```

(4) Assign the initial field in the q array (for fresh start).
(5) Set up a time stepping loop. The main ingredients in this loop will be:
  (a) `call rk4`, the fourth-order time stepping subroutine.
  (b) 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.

(6) `call terminate_with_save`. This will write a "save" file of the flow field which can be used as restart file for continuing the run. and finalize mpi for an mpi run. It will also create the file `return_status.dat` with the status code you indicate. I used it so a PBS script can read it and resubmit the job in case the run completed successfully.

(7) 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 application subroutine.

NASA Ames Research Center
Email address: Karim.Shariff@nasa.gov