"Hello, disk!" with the PENCIL CODE

Wlad Lyra
Uppsala University

1 Introduction

This file is intended to be a guide on how to setup a simulation using the PEN-CIL CODE. It will tour through the configuration files, then through the start and finally run parameters. We choose a simple 2D problem that is conceptually simple and can at the same time, highlight some of the nuances of the code better than a 1D problem. The problem we choose is to set up a differentially rotating accretion disk around a central star of mass M. We make the approximations of isothermality, laminarity (as opposed to turbulence), and axis-symmetry, as well as ignoring the disk's self-gravity. Shock terms are also ignored.

The equations to solve are the continuity and Navier-Stokes equations in an inertial frame

$$\frac{D\rho}{Dt} = -\rho \boldsymbol{\nabla} \cdot \boldsymbol{u}, \tag{1}$$

$$\frac{D\boldsymbol{u}}{Dt} = -\frac{1}{\rho}\boldsymbol{\nabla} p - \frac{GM}{r^3}\boldsymbol{r} + \nu \nabla^2 \boldsymbol{u} \tag{2}$$

where ρ and $\boldsymbol{u}$ are the density and velocity of the gas, p is the pressure, G is the gravitational constant and ν is the kinematic viscosity. The operator $D/Dt = \partial/\partial t + \boldsymbol{u} \cdot \boldsymbol{\nabla}$ represents the advective derivative.

We write cylindrical coordinates as (s,ϕ,z) and spherical coordinates as (r,ϕ,θ), where θ is the polar angle and ϕ the azimuthal angle. The z direction is perpendicular to the midplane of the disk.

The Navier-Stokes equation are in fact three equations, one for each component of the velocity field. In cylindrical coordinates and under the conditions of azimuthal symmetry ($\partial/\partial\phi$=0) and initial centrifugal balance $u_r(t_0)$=0, they read

$$\ddot{s} - s\dot{\phi}^2 = -\frac{1}{\rho}\frac{\partial p}{\partial s} - \frac{GM}{r^3}s \tag{3}$$

$$s\ddot{\phi} = \nu\left(\frac{\partial^2 s\dot{\phi}}{\partial s^2} + \frac{1}{s}\frac{\partial s\dot{\phi}}{\partial s} - \frac{\dot{\phi}}{s}\right) \tag{4}$$

$$\ddot{z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - \frac{GM}{r^3}z \tag{5}$$

The first equation gives the condition for initial centrifugal balance ($\ddot{s}$=0)

$$\Omega^2 = \Omega_{\mathrm{K}}^2 + \frac{1}{s\rho}\frac{\partial p}{\partial s} \tag{6}$$

where we substituted $\Omega_{\mathrm{K}} = GM/r^3$ for the Keplerian angular frequency and $\Omega = \dot{\phi}$ for the true (pressure-corrected) angular frequency.

The second one explicits that viscosity is continuously depriving the disk of angular momentum and making it accrete into the central star. Although, as ν is small, this effect takes many orbits to settle into a steady inflow.

The third equation gives the condition for hydrostatic equilibrium ($\ddot{z}$=0)

$$\frac{1}{\rho}\frac{\partial p}{\partial z} = -\Omega_{\mathrm{K}}z \tag{7}$$

This last equation will be ignored in this problem, but we state it for the sake of completeness.

Our job is to code this initial condition.

2   Initial Conditions

As the equations show, we have a density and a velocity equation to solve, involving gravity, viscosity and pressure. We therefore need the files density.f90, hydro.f90, gravity_r.f90, viscosity.f90 along, of course, with a suitable equation of state. Your src/Makefile.local should therefore look

```
###                      -*-Makefile-*-
### Makefile for modular pencil code -- local part
### Included by 'Makefile'
###
MPICOMM   =     nompicomm
HYDRO     =     hydro
DENSITY   =     density
GRAVITY   =     gravity_r
```

The ideal gas equation of state and viscosity modules are switched on by default. Your start.in file should contain, in this order, init_pars, eos_init_pars, hydro_init_pars and grav_init_pars.

By inspecting start.f90, you see that these variables are initialized by first calling gravity, then velocity, then density. This occurs because hydro needs to know the gravity field to initialize the centrifugal equilibrium. The pressure gradient is also needed to ensure centrifugal balance, but this term can be added to the velocity field a posteriori, in the density module itself.

So, let's start from the beginning, setting the gravity field.

2.1   Gravity

We are interested in the gravitational field of a point mass M centered at r=0, generating a Newtonian acceleration of

$$g = -\frac{GM}{r^2}\hat{r}. \tag{8}$$

2

The only parameter one needs is the product GM, which in the code is expressed by the constant g0.

This acceleration, however, can lead to numerical problems, since it has a singularity in the origin. A modified version usually used in numerical schemes is the acceleration generated by a potential $\Phi = -GM(r^n + r_0^n)^{-1/n}$

$$g = -\boldsymbol{\nabla}\Phi = -GMr^{n-1}(r^n + r_0^n)^{-\frac{n+1}{n}}\hat{r}. \tag{9}$$

This gravity in the code is called smoothed-newton. The unsmoothed Newtonian is called no-smooth. While no-smooth just needs g0, smoothed-newton also needs the exponent n and the peak radius r_0. In the code, these constants are n_pot and r0_pot.

We will use the no-smooth gravity, with g0=GM=1. So, your grav_init_pars should look

```
&grav_init_pars
    ipotential='no-smooth'
    g0=1.
/
```

So, what does this option do? As it can be seen in the code, initialize_gravity first register the variable gg as a *global* variable.

```
call farray_register_global('gg',iglobal_gg,vector=3)
```

The gravity field to be calculated will be stored in this global variable in start time and will not be calculated again in run-time. In order words, the gravity field set by gravity_r is *always* static.

Next, the user has the option to choose if the distance to use in setting the gravity field will be spherical (r) or cylindrical (s). Physical gravity is always spherical, but the cylindrical "limit" is useful for studying some processes where the vertical direction has limited meaning. In our 2D case, they have the same value. The cylindrical version of gravity can be activated by switching on lcylindrical_gravity=T in grav_init_pars. As it can be seen, the gravity field is calculated in the subroutine get_gravity_field. This routine will set the vertical gravity to zero if lcylindrical_gravity is switched on.

2.2 Velocity

Now we have to initialize the velocity field. Here we have to ensure that the velocity field will centrifugally balance the radial gravity. From the radial momentum equation (Eq. 6)

$$s\Omega^2 = \frac{GM}{r^3}s + \frac{\boldsymbol{\nabla}p}{\rho} \tag{10}$$

and $u_\phi = s\Omega$. In the hydro module we will just initialize the gravitational term. The pressure gradient term will be corrected later, in the density module. Choose the option centrifugal-balance for inituu. The subroutine centrifugal_balance will take the gravitational acceleration from the gravity module

and calculate Ω (called `OO` in the code). Then the velocity is set for different coordinate systems. For cylindrical coordinates, $u_r=0$ and $u_\phi=s\Omega$. For Cartesian $u_x=-y\Omega$ and $u_y=x\Omega$.

In the subroutine `centrifugal_balance` you will see

```
call acceleration(g_r)
OO=sqrt(-g_r/rr_cyl)
f(:,m,n,iux) = f(:,m,n,iux) - y(m)*OO
f(:,m,n,iuy) = f(:,m,n,iuy) + x*OO
f(:,m,n,iuz) = f(:,m,n,iuz) + 0.
```

As you can see, this routine is a general routine to code centrifugal balance for any gravitational potential. The case `ipotential='smoothed-newton'` is dealt with as well. But in this case, other parts of the code must also be aware of the values of `n_pot` and `r0_pot`. Through the code, the variable `rsmooth` is used instead of `r0_pot` (which is internal to the gravity module) and `n_pot=2` is assumed. If you choose to use smoothed Newtonian gravity in a global disk, the centrifugal_balance subroutine stops the code if `rsmooth` is different than `r0_pot` and if `n_pot` is not equal to 2.

So, the only thing needed here for the `start.in` file is

```
&hydro_init_pars
   inituu='centrifugal-balance'
/
```

Now we turn to the pressure term. It depends, of course, in the chosen equation of state of the gas.

2.3 Equation of State

The equation of state for accretion disks is not a subject of much attention in the literature. Usually, one assumes an ideal gas equation of state

$$p = \rho c_s^2 \tag{11}$$

with a radially dependent sound speed that does not evolve in time. That is the *locally isothermal approximation*. Physically it means that the disk not only is in radiative equilibrium, but can also efficiently radiate any excess energy so that no heating ever occurs.

Such approximation is carried out by using the switch `llocal_iso=T` in the `init_pars field` of the `start.in` file. The sound speed you want is

$$c_s^2 = c_{s0}^2 s^{-q_T} \tag{12}$$

The constants you need are then c_{s0} and q_T. These are called `cs0` and `ptlaw` in the code, and are start parameters of the Equation of State module.

We need to know the unit of velocity $[u]$ to set the sound speed. As we set $GM=1$, its dimension of length^3 time^{-2}, sets a constraint on the product of these two units. By choosing a unit of length $[x]=s_0$, the unit of time follows from this as being the inverse of the angular frequency at s_0

$$[t] = \sqrt{\frac{GM}{s_0^3}} = \Omega_0^{-1}, \tag{13}$$

which gives an orbital period $P = 2\pi$ at s_0.

The unit of velocity
$$[\boldsymbol{u}] = [x]/[t] = \Omega_0 s_0 \tag{14}$$

is therefore the local Keplerian speed at s_0. The quantity s_0 is set at `init_pars` as `r_ref`. We set it to 1. Ω_0 and the unit of velocity are therefore also 1. We will assume that, at this position, the Mach number is 20. The constant `cs0` is therefore 0.05. We will also choose $q_T=1$, so that the sound speed falls with the square root of the (cylindrincal) distance. Your eos_init_pars should then look

```
&eos_init_pars
   cs0=0.05
   ptlaw=1.
   gamma=1.
/
```

The pressure gradient that this equation of state yield is

$$
\begin{align}
f_p &= -\rho^{-1}\boldsymbol{\nabla}p \tag{15} \\
&= -\rho^{-1}\left(\rho\boldsymbol{\nabla}c_s^2 + c_s^2\boldsymbol{\nabla}\rho\right) \tag{16} \\
&= -c_s^2\left(\boldsymbol{\nabla}\ln c_s^2 + \boldsymbol{\nabla}\ln\rho\right) \tag{17} \\
&= -c_s^2\left(\boldsymbol{\nabla}\ln T + \boldsymbol{\nabla}\ln\rho\right)
\end{align}
$$

where T is the temperature. The logarithm of the squared sound speed and the logarithm of the temperature are the same quantity.

As the sound speed profile, the sound speed and the gradient of the logarithmic temperature can be set as global variables. The idealgas file of the Equation of State module sets the global arrays for these 4 quantities (temperature gradient is a vector). They will be initialized when the Density module calls the `local_isothermal` subroutine. Choose `initlnrho='local-isothermal'` in `density_init_pars`. You see in the subroutine `local_isothermal_density` that the first thing that it does is to get the `ptlaw` variable from the equation of state module and set the thermodynamical quantities with it.

The subroutine `set_thermodynamical_quantities` is only used for the locally isothermal approximation. It will stop the code if entropy or temperature is used, or if γ is not 1. It will then set the sound speed as a power law

```
call power_law(cs20,rr_cyl,ptlaw,cs2)
f(:,m,n,iglobal_cs2)= cs2
```

as well as the associated temperature gradient

```
gslnTT=-ptlaw/(rr_cyl**2+rsmooth**2)*rr_cyl
f(:,m,n,iglobal_glnTT )=gslnTT*x /rr_cyl
f(:,m,n,iglobal_glnTT+1)=gslnTT*y(m)/rr_cyl
f(:,m,n,iglobal_glnTT+2)=0.
```

Next, the subroutine corrects for the azimuthal velocity by the temperature gradient term in the pressure force.

```
corr=gslnTT*cs2
tmp1=(f(:,m,n,iux)**2+f(:,m,n,iuy)**2)/rr_cyl**2
tmp2=tmp1 + corr/rr_cyl

f(:,m,n,iux)=-sqrt(tmp2)*y(m)
f(:,m,n,iuy)= sqrt(tmp2)*x
```

So now the only thing left is to set the density field.

2.4 Density

After setting the thermodynamical quantities, the code sets the density field. We want a power-law density profile, typical of accretion disks with constant mass accretion rate

$$\rho = \rho_0 s^{-q} \tag{18}$$

In the PENCIL CODE, all initial conditions for density are coded as logarithmic densities. If the switch `ldensity_nolog` is switched on, the exponential of the coded initial condition is taken in the end of `init_lnrho`. So, we have to code

$$\ln \rho = \ln \rho_0 - q \ln s \tag{19}$$

The quantity ρ_0 is 1 by default. The exponent q is called `plaw` and is zero by default (constant density). We will use these defaults and work with linear density.

Your density_pars_init should then be

```
&density_init_pars
    initlnrho='local-isothermal'
    ldensity_nolog=T
/
```

The subroutine `local_isothermal_density` will then set the density as a power law

```
lnrhomid=log(rho0)-.5*plaw*log(rr_cyl**2+rsmooth**2)
f(:,m,n,ilnrho) = lnrhomid
```

rsmooth is the same parameter that avoids singularities in the gravity and velocity fields in the center of the grid. In our case it is set to zero, so Eq. 18 is recovered. The next line of the subroutine calls another subroutine to correct for the density gradient term in the pressure gradient. This is very similar to what was done to correct the temperature gradient. It calculates the density gradient,

```
call grad(f,ilnrho,glnrho)
```

multiplies it by the squared sound speed

6

```
   gslnrho=(glnrho(:,1)*x(l1:l2) + glnrho(:,2)*y(m))/rr_cyl
   corr=gslnrho*f(l1:l2,m,n,iglobal_cs2)
```

and applies this correction to the angular velocity according to Eq. 6

```
   tmp1=(f(l1:l2,m,n,iux)**2+f(l1:l2,m,n,iuy)**2)/rr_cyl**2
   tmp2=tmp1 + corr/rr_cyl
!
   f(l1:l2,m,n,iux)=-sqrt(tmp2)*y( m )
   f(l1:l2,m,n,iuy)= sqrt(tmp2)*x(l1:l2)
```

Okay, all the equations are set. Let's take a look at the boundary conditions.

3   Boundary Conditions

To start the code, we just have to set the box, choose the appropriate boundary conditions and set the resolution in cparam.local.

As we are solving in a Cartesian grid, what we will do is to embed a cylinder inside the box and only solve the equations inside it. Everything outside of it will be frozen (the time-derivatives will be set to zero). We can set the outer boundary of this cylinder at s_{ext} =2.5. If we also set the end of the Cartesian box at 2.5, the ghost cells will lead to instabilities in the boundary. So let's give some room to avoid it and set the edge of the Cartesian box at 2.6

```
   xyz0 = -2.6,-2.6,-1., !  first corner of box
   xyz1 = 2.6, 2.6, 1., !  second corner of box
```

The "real" boundaries of the box then do not matter. We can set them all to periodic.

```
   lperi = T , T , T
```

Now, the disk will also need an inner boundary. Why? Because we are using non-smoothed Newtonian gravity, that contains a singularity at the origin, that's why. Let's avoid it by placing the inner boundary at, say, s_{int} =0.4

```
   lcylinder_in_a_box=T
   r_int=0.4,                ! radius of interior cylindrical boundary
   r_ext=2.5,                ! radius of exterior cylindrical boundary
```

We are then done with start.in It should look

```
!               -*-f90-*-(for Emacs)
!
!  Initialisation parameters
!
&init_pars
   ip=14                  ! debugging parameter
   xyz0 = -2.6,-2.6,-1!  first corner of box
   xyz1 = 2.6, 2.6, 1,!  second corner of box
   lperi = T , T , T, !  periodic direction?
```

```
    r_int=0.4,          !  radius of interior cylindrical boundary
    r_ext=2.5,          !  radius of exterior cylindrical boundary
    lcylinder_in_a_box=T
    lwrite_ic=T
    llocal_iso=T
    r_ref=1.0
/
&eos_init_pars
    cs0=0.05
    ptlaw=1., gamma=1.
/
&hydro_init_pars
    inituu='centrifugal-balance'
/
&density_init_pars
    initlnrho='local-isothermal'
    ldensity_nolog=T
/
&grav_init_pars
    ipotential='no-smooth'
    g0=1.
/
```

Next we choose the resolution. At least 320 points in x and y will be needed to damp numerical instabilities in the inner edge of the disk. The line

```
!  MGLOBAL CONTRIBUTION 4
```

is needed because of the extra 4 thermodynamical variables the local isothermal equation of state requires (sound speed + 3 components of the temperature gradient)

```
! -*-f90-*- (for emacs) vim:set filetype=fortran:  (for vim)
! cparam.local
!
! Local settings concerning grid size and number of CPUs.
! This file is included by cparam.f90
!
! MGLOBAL CONTRIBUTION 4
!
    integer, parameter ::  ncpus=1,nprocy=1,nprocz=ncpus/nprocy,nprocx=1
    integer, parameter ::  nxgrid=320,nygrid=320,nzgrid=1
```

Done. Now just run make and start.csh

4 Running

The run parameters are much simpler. The main thing involves the freezing boundaries used. After evolving the dynamical equations, we set the time derivatives of all variables to zero in the region outside s_{int} and s_{int}. To avoid numerical instabilities due to this abrupt jump from frozen to evolving regions, we have to apply a buffer zone to the derivatives of the variables. This is done

by setting a width within which the motion will be damped until it reaches the frozen boundary.

Set this width to 0.05 in the inner and 0.1 in the outer boundary. The parameters fshift_int and fshift_ext are flags that specify if the width occurs before (-1) or after (1) the boundary. It must of course be external in the inner boundary and internal in the outer one. So fshift_int=1 and fshift_ext=-1

```
!

   wfreeze_int = 0.05
   wfreeze_ext = 0.1
   fshift_int=1
   fshift_ext=-1
!
```

The effect of this can be seen in equ.f90. After the equations are solved and df is fully calculated, the following terms apply the damping and freezing for the inner boundary:

```
pfreeze_int = quintic_step(p%rcyl_mn,rfreeze_int,wfreeze_int,SHIFT=fshift_int)
if (lfreeze_varint(iv)) df(l1:l2,m,n,iv) = pfreeze_int*df(l1:l2,m,n,iv)
```

The external freezing is similar,

```
pfreeze_ext = 1-quintic_step(p%rcyl_mn,rfreeze_ext,wfreeze_ext,SHIFT=fshift_ext)
if (lfreeze_varext(iv)) df(l1:l2,m,n,iv) = pfreeze_ext*df(l1:l2,m,n,iv)
```

And what quintic_step does is

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   function quintic_step_mn(x,x0,width,shift)
!
!  Smooth unit step function with quintic (smooth) transition over [x0-w,x0+w].
!
!  Version for 1d arg (in particular pencils).
!
!  09-aug-05/wolf:  coded
!
use Cdata, only:  tini
!
      real, dimension(:)  ::  x
      real, dimension(size(x,1)) ::  quintic_step_mn,xi
      real ::  x0,width
      real, optional ::  shift
      real ::  relshift=0.
!
      if (present(shift)) then; relshift=shift; else; relshift=0.; endif
      xi = (x-x0)/(width+tini) - relshift
      xi = max(xi,-1.)
      xi = min(xi, 1.)
      quintic_step_mn = 0.5 + xi*(0.9375 + xi**2*(-0.625 + xi**2*0.1875))
```
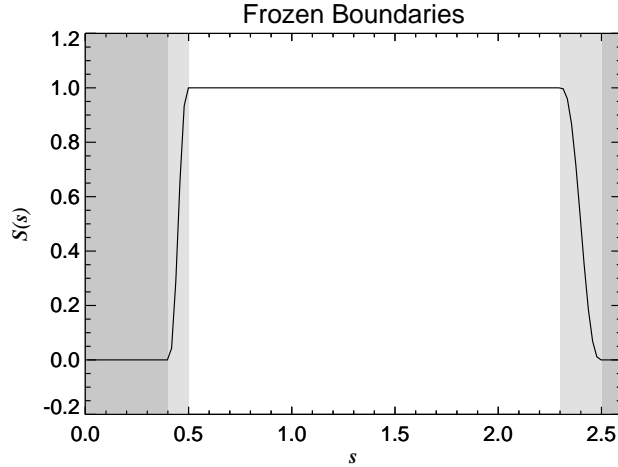
9

Figure 1: Damping and freezing profile to be applied to the derivatives of the variables.

Combined, these two functions result in the freezing profile shown in Fig. 4. That function multiplies the derivatives of the variables. We simply have to specify that we want both internal and external freezing for density and velocities by setting lfreeze_uint=T,lfreeze_uext=T and lfreeze_lnrhoint=T lfreeze_lnrhoext=T

Now, the last thing needed is dissipation. For density, we can use upwinding (lupw_rho=T)

For velocity, a simple viscosity with constant ν as in Eq. 2 will work well. For the choice of sound speed and length, 10^{-5} should do. The run.in file should then be

```
!                         -*-f90-*-   (for Emacs)
!  Run parameters!
!
&run_pars
   ip=14,
   nt=50, it1=5, isave=100, itorder=3, ialive=1
   cdt=0.4,
   cdtv=0.4,
   dtmin=1e-6,
   dsnap=6.2831 !VARN for every complete orbit at s0
   dvid=1.  !video slices more often
!
   wfreeze_int = 0.05
   wfreeze_ext = 0.1
   fshift_int=1
   fshift_ext=-1
!
/
&eos_run_pars
```

10

```
/
&hydro_run_pars
    lfreeze_uint=T,lfreeze_uext=T
/
&density_run_pars
    lupw_rho=T
    lfreeze_lnrhoint=T
    lfreeze_lnrhoext=T
/
&grav_run_pars
/
&viscosity_run_pars
    ivisc='nu-const'
    nu=1e-5
/
```

Now run and relax!