COMPUTING HELIOSEISMIC WAVE INTERACTIONS

Shravan M. Hanasoge

Max-Planck-Institut für Sonnensystemforschung
Department of Geosciences, Princeton University

This document describes in some detail a code that can be used to compute the interactions of waves with magnetic flux tubes, sound-speed and damping perturbations, and study the wave field in the presence of multiple/single sources or anomalies thereof. The computational method is described in detail in Hanasoge et al. (2007a,b); Hanasoge & Duvall (2007); Hanasoge (2007a,b); Hanasoge et al. (2010); we solve the linearized MHD and Euler equations in 3D Cartesian geometry. The derivatives are computed using sixth-order compact finite differences (in all three directions) or FFTs in the horizontal directions and an optimized second-order RK time stepping scheme is implemented. The verification and validation are discussed in detail in the above references.

1. FILES AND COMPILATION

The code is written in a mixture of Fortran 90 and Fortran 77. The following files are required for successful compilation:

- Basic parameters (size, output directory etc.) - `params.i`
- Name says all; controls i/o streams, when to time evolve etc. - `driver.f90`
- Initializing the code, setting up the background model etc. - `initialize.f90`
- Subroutines to compute norms, i/o etc. - `all_modules.f90`
- The physics kernel: initializing the system to be solved, and sponge-boundary equations - `physics.f90`
- Convolutional-Perfectly-Matched-Layer boundary-conditioned velocity equations - `pml.f90`
- CPML/ Sponge boundary-conditioned equations in displacement - `displacement.f90`
- Derivative routines and parallel transposes of arrays - `derivatives.f90`
- 2D routines: MHD / quiet and PML / sponge - `physics2d.f90`
- Time stepping algorithm and controlling the de-aliasing rate - `step.f90`
- Compute the derivative along the first dimension - `dbyd1.f`
- File associated with `dbyd1.f` - `tridag.f`
- Compute the derivative along the second dimension - `dbyd2.f`
- File associated with `dbyd2.f` - `mtridag.f`
- One to bind them all - `makefile`

The dependencies of files on each other are listed in `makefile`. The construction of sophisticated makefiles is beyond the abilities of the author; consequently, the makefile can perform only very crude and simple tasks:

- Typing `make` on the Linux prompt will compile the file
- `make clean` will rid the directory of the `.o, .mod` files and the executable - this is to allow for fresh recompilation.

One must inspect the `makefile` in order to ensure that the correct Fortran compiler (`FC=`) has been identified. The flags (`FFLAGS =`) should call for the highest degree of optimization (`-O3` in the case of `ifort`) allowed by the compiler to ensure best performance. The `FFTW` and `CFITSIO` libraries are required for compilation; the former in the event that the horizontal derivatives be computed using FFTs and the latter to read and output files in the FITS Astronomical format. These libraries may be easily downloaded from the internet; a mere Google search for these two libraries will produce a number of relevant links. The MPI parameters file, `mpif.h` should preferably be placed in the same directory as the code; if not, an appropriate link to its location must be provided in the file `initialize.f90`, i.e. in the command: `INCLUDE 'location/mpif.h'`, where `location` is the path to the `mpif.h` file. The name of your executable may be altered at will; simply change the `COMMAND =` in `makefile` to reflect your choice.

Lastly, because of differences in the OpenMPI (eg. IBM Machines) and MPT (Message Passing Toolkit - for SGI Altix) standards, I've created two different versions of the `initialize.f90` file, titled `initialize_mpt.f90` and `initialize_openmpi.f90` and of the `all_modules.f90` file - `all_modules_mpt.f90` and `all_modules_openmpi.f90`. Depending on the system architecture and software please rename the files appropriately - i.e., if you are on an OpenMPI system, rename `initialize_openmpi.f90` and `all_modules_openmpi.f90` as `initialize.f90` and `all_modules.f90` respectively.

2. PARALLEL ALGORITHM

Because the sizes of the problems of study are rather large, a parallel implementation is unavoidable. The parallelism is concordant with the Message Passing Interface (MPI) standards v1.2 and OpenMPI. The central variables of evolution are $\rho, v_x, v_y, v_z, p, b_x, b_y, b_z$ (fluctuations in density, three components of velocity, pressure, and three components of magnetic field respectively). Each variable is an array of dimensions (n_x, n_y, n_z), where the three components are the number of grid points in directions (x, y, z) respectively. The algorithm involves dividing the arrays into chunks of the data among the processors as equitably as possible. Denoting the number of processors by n_P, the storage pattern is determined such that each processor contains approximately $(n_x, \lfloor n_y/n_P \rfloor, n_z)$ sized chunks of the original array. Thus the parallelism is only in one direction; this makes the book-keeping much easier when interchanging information between processors. It is also viable when the problem is not too computationally expensive. Thus for very large sizes, i.e. when $n_x > 700$ or so, a parallel algorithm that involves domain decomposition along multiple directions may be more efficient. In any case, the derivatives in the (x, z) directions may be computed in-processor while for the y gradient, the arrays must be reconfigured such that all the y points for a given (x, z) are in the same processor. The machinery required for the export and reception of information between processors is constructed in the subroutine `Initialize_all`. The total number of processors (i.e. = how many requested) involved in the calculation is called `numtasks`. When the MPI job is initialized, each processor is assigned a unique number between `0` and `numtasks-1`, denoted by `rank` in the code. The `rank = 0` process takes on the role of root. The MPI process is set off by a call to the subroutine `Initialize_all`.

Once the MPI variables have been initialized, the domain decomposition is performed; essentially arrays of sizes (nx, dim2(rank), nz) are created locally in each processor. `nx` and `nz` correspond to the array dimensions in the `x,z` directions respectively. The `y` dimension is decomposed into bits of sizes `dim2(rank)`. The `dim2` part refers to the fact that the decomposition is along the second dimension; a process with assignation `rank` stores `(nx, dim2(rank), nz)` sized arrays. In the `y` dimension, each process knows only of the existence of indices spanning the range `[1,dim2(rank]` - clearly it is not possible to proceed further without determining what these indices map on to in the uncorrupted (non-decomposed) `y` grid. Thus we introduce another variable `ystart(rank)` which indicates the starting index of access of the process `rank` along the undivided `y` axis. And since the size of the decomposed second dimension stored in the process `rank` is `dim2(rank)`, the range `[ystart(rank), ystart(rank) + dim2(rank)]` denotes the indices along the undivided `y` axis that `rank` stores. Note that the values of the coordinate axes (i.e. the `x(i),y(j),z(k)`) are defined on each processor.

The `x,y` co-ordinates are normalized by their horizontal side lengths such that both lie in the range [0,1]. The `z` co-ordinate is the normalized solar radius, $r/R_\odot$, meaning that $\mathbf{z}_{\rm photosphere} = 1$, with $\mathbf{z} < 1$ referring to the interior and $\mathbf{z} > 1$, the atmosphere. When incorporating inhomogeneous source distributions or localized sound-speed/damping perturbations, the coding of these must be performed in parallel space. Note that each processor has access to all the `x` and `z` grid points but only a limited number of `y` points. Denoting the running indices in these directions by `i,j,k`, here is an example of a Fortran `do`-loop that computes the spherical radial coordinate `r(i,j,k)` in parallel space:

```
do k=1,nz
  do j=1,dim2(rank)
    do i=1,nx
     r(i,j,k) = (x(i)**2.0 + y(j+ystart(rank)-1)**2.0 + z(k)**2.0)**0.5
    enddo
  enddo
 enddo
```

Thus on each local processor, the size of the radial coordinate array `r` is `(nx,dim2(rank),nz)`, but its values reflect the fact that different processors store differing values of the `y` coordinate. Here's another example that shows how to suppress wave source excitation over a circular region of radius approximately 10 Mm:

```
   do j=1,dim2(rank)
    do i=1,nx
     tempxy=(((x(i)-0.5)*400)**2.0+((y(j+ystart(rank)-1)-0.5)*400)**2.0)**0.5
     vr(i,j,:)=vr(i,j,:)/(1.0+exp((8.0-tempxy)*1.5))
    enddo
   enddo
```

In this case, the horizontal sides of the cube are 400 Mm each, and the perturbation is located at the center of the computational domain - which is why the term `tempxy = ((x(i) - 0.5)**2.0 + .....` This do-loop may be placed after the forcing function, `vr` is read in. Note that the directive to read in the forcing function `vr` is in `driver.f90`, the central controller of i/o streams from the code. The third dimension of the forcing

function is time; so in the above `do` loop, the sources are suppressed over the entire simulation duration. Lastly, a manner of introducing sound-speed perturbations is demonstrated:

```
z_cent = 1.0
z_spread = 20.
do k=1,nz
 do j=1,dim2(rank)
   do i=1,nx
     tempxy = ((x(i)-0.5)*200.0)**2.0 + ((y(j+ystart(rank)-1) - 0.5)*200.)**2.0
     c2(i,j,k) = c2(i,j,k)*(1.0 + 0.1*exp(-((1.0 -z(k))*Rsun*10.0**(-8.0)/z_cent)**2.0 &
        - tempxy/z_spread**2.0) )
    enddo
   enddo
 enddo
```

This is a slightly more complicated case but no different in principle from the other two. `c2`, as suspected, refers to the sound speed square; the 0.1 factor in front of the exponential term implies a 10% increase in the squared sound speed. The term `(1.0 -z(k))*Rsun * 10.0 **(-8.0)` (`Rsun` is expressed in cgs units) converts the solar radius normalized `z` into mega-meters, with the photospheric level now defined as the zero point. The `tempxy` term represents the cylindrical radius squared, with the $r = 0$ point the horizontal center of a 200×200 Mm^2 box. The vertical extent of the perturbation is approximately `z_cent = 1.0` Mm and the horizontal size is given by `z_spread = 20` Mm (or scales like these terms). Also look at the file `examples` for a list of perturbations and the method used to code them in.

3. THE VARIABLES AND IMPORTANT SUBROUTINES

The code starts from the `driver.f90` file. The first subroutine to be called is `Initialize_all` which sets up the MPI variables, data types and communication protocols. The following variable arrays (amongst others) are allocated:

- `a` - dimensions of (nx, dim2(rank), nz, nvar) - stores all the system state (`nvar` variables)
- `temp_step` - same dimensions as `a` -work space for the time evolution scheme (`physics, step.f90`)
- `scr` - same dimensions as `a` - the right hand side work space for the time evolution scheme (`physics, step.f90`)
- `rho, v_x, v_y, v_z, p, bx, by, bz, xi_x, xi_y, xi_z` - pointers to the different variables - they point to sequential parts of the `temp_step` array
- `RHScont, RHSv_x, RHSv_y, RHSv_z, RHSp, RHSb_x, RHSb_y, RHSb_z, RHSxi_x, RHSxi_y, RHSxi_z` - point to different parts of `scr` depending on the equations being solved
- `p0, rho0, c2, box, boy, boz, v0_x,v0_y,v0_z` - background variables (self explanatory)
- `vr, forcing` - forcing functions, the former has dimensions of `nx,dim2(rank),nt` while the latter, `nx,dim2(rank)`. `vr` contains the excitation function sampled every minute (read in by `driver.f90`),

whereas the latter is the function interpolated to the current time level through the application of sixth-order Lagrange polynomials.

Each of the background variables is normalized: `rho0, c2` by their maximum value in the domain (termed in the code as `dimrho, dimc`), `p0` by `dimrho * dimc**2.0`, the horizontal co-ordinates by the length of the side, background magnetic field by $\sqrt{\mathbf{4} * \pi * \mathbf{dimrho} * \mathbf{dimc} * *\mathbf{2}}$, velocities (background and oscillatory) by `dimc`, and finally the time coordinate by `Rsun/dimc`. When the system is started off at a non-zero initial state, they must be read into the appropriate parts of the variable `a`. For example, if starting with an initial density condition, say `rhotemp`, (over the background, of course), the following procedure is to be adopted:

- `a(:,:,:,1) = rhotemp(:,:,:)`
- For values of k=1,8 (depending on whether the simulation is magnetic or not), `a(:,:,:,k)` refers to `rho, v_x, v_y, v_z, p, bx, by, bz` respectively (for equations in velocity).
- For values of k=1,6, `a(:,:,:,k)` refers to `xi_x, xi_y, xi_z, v_x, v_y, v_z` respectively (for equations in displacement).
- If background variables such as velocity are programmed directly in the code (as opposed to being read in), they must be normalized by `dimc`.

Next, `MP_initialize_RHS` and `Initialize_step`, functions that arrange the machinery to compute the right hand sides and perform the time evolution, are called by the `driver.f90` file. The forcing function is subsequently read in and the time stepping is started. Before each evolutionary step (achieved by calling the subroutine `step`), the forcing function `vr` is interpolated to the current time level. At each step, information regarding the wall clock time/ cpu time and the stability of the system are reported. The stability is quantified crudely by computing the L_2 norm of the vertical velocity variable `v_z`. The RMS value at each timestep is stored in the file `rms_hist` placed in the local simulation directory. Plotting these values will tell you whether the calculation is stable or not. `driver.f90` is a good place to implement perturbations (look in `examples` for directions), extract and read in data.

Thus far I have religiously adopted the FITS FORMAT of binary file storage. I use the following subroutines (located in `all_modules.f90`) to read and write in files:

- `call readfits(filename, array-name, size of third dimension)` will read from `filename` (full location must be given of course) into an array called `array-name`. The dimensions of the data (in normal space) are expected to be `nx, ny, nthird`, where `nthird` is an input to the `readfits` call. Thus to read a forcing function of size `(nx, ny, nt)` from the directory `/tmp20/shravan/forcing.fits` into the array `vr`, we would use the following command: `call readfits('/tmp20/shravan/forcing.fits', vr, nt)`.
- `call writefits_3D(filename, array-name, size of third dimension)` - same functionality as the `readfits` command. If say we wanted to output the data at a series of heights from index 192 to 221, we would then use: `call writefits_3D('/tmp20/shravan/data.fits', a(:,:,192:221,4),30)`. This outputs the `v_z` variable at all horizontal locations and at the specific heights stated.
- `call write_out_full_state` - this outputs the full 3D state of the system (all relevant variables)
- `call write_out_slice (grid_point)` - this is a subroutine that outputs all relevant variables at the radial grid point specified.

The files contain some basic documentation to assist in code interpretation. It's mostly simple.

Lastly, for more advanced manipulation, the subroutines `transpose_3D_y` and `inv_transpose_3D_y` can be called to perform data transposes across the processors. For example, if one wished to access all the `y` grid points on the same processor (normally they are distributed across the processors), then the command `call transpose_3D_y(input, output)` would effect this in the output array `output`; subsequently, if one wished to then revert to the original distribution, invoke `call inv_transpose_3D_y(input,output)` to accomplish this (`output` has the requisite arrangement).

4. FORCING FUNCTION

Generally the easiest way to devise forcing functions is to go into spectral space (k_x, k_y, ω), choose a Gaussian distributed random number for each coefficient and multiply the resultant array by a function of frequency, $F(\omega)$ such that a solar-like variation of power with frequency is obtained. Keeping in mind the limitations on computer memory, the cadence of the forcing function should be about a minute. Note that because the damping in the simulation is not in the least solar-like, the final balance of power tilts in favour of the higher frequencies. The lack of appropriate frequency dependent damping means that low mode-mass high-frequency waves now abound blissfully. Consequently, ensure that the forcing function drives the lower frequencies harder (not too low, otherwise the travel times turn out strange). For now, I've put in a number of forcing functions that can be downloaded from the website.

5. SIMULATION PARAMETERS AND PERFORMANCE

The file `params.i` contains a list of important parameters that may be changed based on the requirements of the simulations; the grid size, grid dimensions, i/o type and locations, wall clock time of the simulation, the restarting time index (if the simulation had been paused) etc. The `params.i` file contains documentation to assist in choosing the parameters of interest.

Although I haven't really tested the performance of the code carefully, I have found reasonable linear scalability for a smallish number processors (< 128). Note that there is a balance between computational effort and inter-processor communication that determines the optimum. A rule of thumb that works for the most part is to make sure that $\lfloor n_y/n_P \rfloor \geq 2$, so that the computation to communication ratio does not become too small. The communication costs depend on the latency and asymptotic bandwidth of the network and must be estimated experimentally.

6. ADDITIONAL BELLS, WHISTLES, & CAVEATS

Now there is an option to use one of two types of boundary conditions: the standard sponge and the new, fancy Convolutional Perfectly Matched Layer (Hanasoge et al. 2010). You can study MHD, flows, and quiet Sun problems using either condition. I found the MHD equations are remarkably hard to stabilize; there are deep holes in our understanding of how to limit the Alfvén speed and interpret the equations correctly. The primary cause for instabilities in MHD runs I found were the z derivatives of the horizontal fields in $\mathbf{j}_0 = \boldsymbol{\nabla} \times \mathbf{B}_0$: I have now set them to zero - and this is not an unreasonable assumption, especially in the atmosphere. I also smoothly reduce the $\mathbf{j}_0$ to zero in the last 10 grid points. These may be found in

the initializing subroutine in `physics.f90`. I must warn the user that I cannot guarantee convergence with arbitrary magnetic fields. An excellent way to check things is to do a 2D run - an enormously faster and cheaper method of testing the setup. Note that in order to run in 2D mode, you have to set $n_y = 1$.

I have also included the equations for displacement - these are decidedly cheaper for flows but the expense is almost identical for the MHD/quiet. All in all, 16 options (systems of equations + 2D/3D + boundary conditions) exist.

7. ACKNOWLEDGEMENTS

The HD and flow part of the code was written sometime towards the end of 2006. Aaron Birch must be thanked for excellent ideas on how to build reasonable forcing functions and his help with pinpointing temporal aliasing issues. The MHD extension was put in over the period 2007-2008; several discussions with Keiji Hayashi and Robert Cameron helped greatly. Many thanks to Hamed Moradi for spending time on the code and helping with troubleshooting issues. The Columbia support staff at NASA Ames have assisted in sorting out several bugs and in generously extending my computational quota. A significant part of the software was developed on Stanford machines and subsequently, the Max-Planck group for helioseismology; many thanks to Keh-Cheng Chu, Brian Roberts, and Michel Bruns for technical support. This code would not have been written without Phil Scherrer's strong support over the years. The funding to allow for a revision of this software came from a DLR grant whose PI is Laurent Gizon.

REFERENCES

Hanasoge, S. M. 2007a, ArXiv e-prints, 712

—. 2007b, PhD thesis, Stanford University

Hanasoge, S. M., Couvidat, S., Rajaguru, S. P., & Birch, A. C. 2007a, ArXiv e-prints, 707

Hanasoge, S. M., & Duvall, Jr., T. L. 2007, Astronomische Nachrichten, 328, 319

Hanasoge, S. M., Duvall, Jr., T. L., & Couvidat, S. 2007b, ApJ, 664, 1234

Hanasoge, S. M., Komatitsch, D., & Gizon, L. 2010, ArXiv e-prints