

Getting Started with *AstroBEAR*

October 4, 2006

1 Preliminary Install

BEARCLAW is a Fortran-scripted software package for solving time-dependent partial differential equations. BEARCLAW links against several freely available libraries which must be installed before the program can be run; these include HDF4, HDF5, and MPI. Additionally, the `pgplot` plotting library and `openDX` visualization tool are useful for visualizing the results of a calculation and are recommended installs. In particular, the supplemental visualization tool `bear2fix` requires the `pgplot` libraries. The steps to download and install these libraries are given in this section.

BEARCLAW has been developed primarily using the Intel (Fortran & C++) compiler suite: <http://www.intel.com/cd/software/products/asm-na/eng/compilers/284264.htm> It is assumed in this section that the Intel compilers are installed under `/usr/local/intel/fce/9.0/` for the Fortran 95 compiler and `/usr/local/intel/cce/9.0/` for the C/C++ compiler.

While the code should be compatible with any Fortran 95 compiler, the following instructions for building the supporting libraries that require Fortran 95 language bindings will use the Intel compilers. In general, it is not possible to link Fortran 9x module files between different compilers. It is possible, however to link Fortran 77 library objects compiled using the GNU compilers with most commercial Fortran 95 compilers. Libraries that only require Fortran 77 will, therefore, be compiled using the standard GNU compilers.

In the instructions below, the `homedir` string should be replaced by the user's full home directory (e.g. `/home/username`) and all of the commands printed below should be run in this directory unless otherwise noted.

- **Environment variables**

Several additions are required to the `~/bashrc` file, which are listed here. Note that you must “`source ~/bashrc`” or login under a new shell for these changes to take effect.

```
source /usr/local/intel/fce/9.0/bin/ifortvars.sh
source /usr/local/intel/cce/9.0/bin/iccvars.sh

export LD_LIBRARY_PATH=/usr/local/pgplot_intel_9.0:/usr/local/hdf5_intel_9.0/lib/:
$LD_LIBRARY_PATH
export PATH=homedir/bear2fix:/usr/local/bin:/usr/local/lam-intel_9.0/bin:$PATH
export BEARCLAW=homedir/bearclaw
export PGPLOT_DIR=/usr/local/pgplot_intel_9.0

export DXHOST=localhost
ulimit -s unlimited
```

- **HDF4** (to be compiled with gcc and f77 language bindings) There are several possible output formats in BEARCLAW; HDF4 is standard, as is HDF5 (given below). The following steps will install the HDF4 library on your system.

```
wget ftp://ftp.ncsa.uiuc.edu/HDF/HDF/HDF_Current/src/HDF4.2r1.tar.gz
tar -xzvf HDF4.2r1.tar.gz
cd HDF4.2r1
export LDFLAGS=-lm # The HDF4 package needs help finding math library
./configure --enable-fortran --prefix=/usr/local/hdf
make
# As root:
make install
```

- **HDF5** (to be compiled with f9x bindings using the intel compilers)

```
export FC=ifort;export F77=ifort;export F9X=ifort
export CXX=icpc;export CXXCPP="icpc -E"
export CC=icc;export CPP="icc -E"
wget ftp://ftp.ncsa.uiuc.edu/HDF/HDF5/current/src/hdf5-1.6.5.tar.gz
tar -xzvf hdf5-1.6.5.tar.gz
cd hdf5-1.6.5
./configure --prefix=/usr/local/hdf5_intel_9.0 --enable-fortran
make
# As root:
make install
```

- **lam-mpi** (to be compiled with f9x bindings using the intel compilers) The LAM-MPI library is required for parallel jobs (and therefore is not required if parallelization is not desired).

```
export FC=ifort;export F77=ifort;export F9X=ifort
export CXX=icpc;export CXXCPP="icpc -E"
export CC=icc;export CPP="icc -E"
wget http://www.lam-mpi.org/download/files/lam-7.1.2.tar.gz
tar -xzvf lam-7.1.2.tar.gz
cd lam-7.1.2
./configure --prefix=/usr/local/lam-intel_9.0 --with-rsh="ssh -x" --with-fc=ifort
make
# As root:
make install
# Make a sim link for "mpif90"
cd /usr/local/lam-intel_9.0/bin
ln -s mpif77 mpif90
```

- **pgplot** (to be compiled with gcc and f77 language bindings) As noted before, **pgplot** and **openDX** are plotting and visualization tools which are highly recommended installs.

```
wget ftp://ftp.astro.caltech.edu/pub/pgplot/pgplot5.2.tar.gz
tar -xzvf pgplot5.2.tar.gz
chmod -R a+rX pgplot
# As root:
mkdir /usr/local/pgplot
cd /usr/local/pgplot
cp ~/pgplot/drivers.list .
# EDIT drivers.list to include the drivers: PS, VPS, CPS, & VCPS
~/pgplot/makemake ~/pgplot linux g77_gcc
```

Sidenote: If your system uses the newer **gfortran** instead of **g77**, edit the Makefile to reflect this.

Also: if necessary, the location of 64-bit X11 libraries on x64 systems can be set via

```
LIBS=-L/usr/X11R6/lib64 -lX11
MOTIF_LIBS=-lXm -lXt -L/usr/X11R6/lib64 -lX11
ATHENA_LIBS=-lXaw -lXt -lXmu -lXext -L/usr/X11R6/lib64 -lX11
TK_LIBS=-L/usr/lib -lgtk -ltcl -L/usr/X11R6/lib64 -lX11 -ldl
```

```
# Now you can proceed to make the library
make
make clean
```

- **openDX** (to be compiled with gcc)


```
wget http://opendx.npaci.edu/source/dx-4.4.0.tar.gz
tar -xzvf dx-4.4.0.tar.gz
export CPPFLAGS=-Ihomedir/hdf/include/
export LDFLAGS=-L/usr/local/hdf/lib/
cd dx-4.4.0
./configure --prefix=/usr/local --with-hdf \
--x-includes=/usr/local/hdf/include:/usr/X11R6/include \
--x-libraries=/usr/local/hdf/lib:/usr/X11R6/lib64 --libdir=/usr/lib64
make
# As root:
make install
```

At this point, the environment should be setup to allow the download and install of BEARCLAW.

2 Downloading the Code via SVN, Compiling the Code

Subversion (SVN) is a code repository and version control system for *AstroBEAR*. It is used to automatically track changes and updates to the code, and to store the code in a central location. The user's primary interaction with it is to initially download the entire code package from the repository, to download any updates as they become available, and to upload any important changes he or she has made. This section is designed to illustrate this process.

Initial Download To initially download (“check out”) a copy of the *AstroBEAR* code and the visualization tool *bear2fix*, from a your home directory type

```
svn co https://data-store/svn/orda/bearclaw1/dev bearclaw
svn co https://data-store/svn/orda/bear2fix/dev bear2fix
```

The last word on each line is the relative path that SVN will use to install (so if you were sitting in `/home/username/`, the above would make two directories `/home/username/bearclaw/` and `/home/username/bear2fix/`). Note that whatever path you choose should be reflected in the `$BEARCLAW` environmental variable in `~/bashrc`.

This initial checkout will create a directory tree that should be complete and nearly ready to be compiled. After this initial checkout, you should only need to “update” your working version (see below) rather than checkout the entire code again.

Compiling Compiling the code involves selecting the “Makefile” appropriate to your computing environment and choosing between employing debugging or optimization. Optimization is turned on by default, and assuming that the code is being run on the Orda cluster all that is required to compile the code is to navigate to `bearclaw/contrib/astro` and type

```
make xbear
```

(for a single-processor executable) or

```
make mpibear
```

(for a multiple-processor executable).

[Details concerning manipulation of makefiles to be added.]

After this is done, there should exist a working executable (`xbear` or `mpibear`) in `bearclaw/contrib/astro`. (Note that compiling the visualization tool is directly analogous, by running `make bear2fix` in the `bear2fix` directory.)

Setting up and running a job is addressed in the next section. The remainder of this section gives additional details about the SVN repository which will become important later and are presented here for completeness.

M	File has been modified
!	File is missing (moved or deleted w/o involving SVN, see below)
?	File is ignored by SVN (added w/o involving SVN, or created when code was compiled)
	File is the same as when it was checked out

Table 1: Common SVN status symbols. For other symbols and additional information, see {the SVN quick reference card}. *[Add hyperlink to svn-refcard-1.pdf]*

More Information on SVN As mentioned above, the repository uses automatic version control. In essence, this means that whenever someone “checks in” a new copy of a file or files to the repository, the repository automatically assigns a new revision to these files and notes any changes.

One can see the actual changes for any revision, as well as who authored those changes and when, by viewing the repository online,

```
https://data-store/cgi-bin/viewcvs/viewcvs.cgi/?root=bearclaw1
https://data-store/cgi-bin/viewcvs/viewcvs.cgi/?root=bear2fix
```

(The same may be accomplished on the command-line but is cumbersome and not color-coded.)

If a situation arises where you require a version of the code other than the current one, you may checkout a different revision N (where N can be 1 to the current revision number) via

```
svn co -r N https://data-store/svn/orda/bearclaw1/dev bearclaw
```

Within the directory structure that was created by the initial SVN checkout, a user may modify files as desired. SVN can compare the existing files **to those that were originally checked out**, and can tell you which have been modified via the status command,

```
svn st -v
```

which will return a list of filenames and symbols, the most common of which are given in Table 1.

The only way to have the repository recognize that you want a file to be deleted or added is to include SVN in the process, for example

```
svn add myfile.f90
svn mv file.f90 oldfile.f90
```

And so forth. This can cause confusion, for example, when trying to delete a file—failing to prepend `rm` with `svn` will make the repository think the file mysteriously disappeared, and so it will restore it on the next update. In order to have the file be deleted *in the repository*, `svn rm` must be used.

Using the `-v` flag can be contrasted to using `-u`, which compares the files in your directory **to the current repository versions**. Note that this command, as well as most others, may be performed on the entire working copy/directory structure/module or on a single file, e.g.

```
svn st -u arf.f90
```

will tell you whether arf.f90 is up to date.

To update your files,

```
svn up
```

will go through and **only** update files that have not been modified. For files that have been modified by you **and** changed in the repository, it may try to merge them, or will report a conflict and demand attention. Files you have modified which have not been changed in the repository will be left alone.

Once you have modified or added a file that is important to have on the repository, you may commit/check-in your changes via

```
svn ci -m ‘‘Check-in details (keep in quotations).’’
```

—OR—

```
svn ci
```

If you have the environmental variable \$EDITOR or \$SVN_EDITOR set (e.g., in ~/.bashrc, “export SVN_EDITOR=nano”), doing the latter will open that text editor and allow you to give more complete details about the commit, whereas the former is useful probably only for one-line comments. (Note that if neither of those variables is set, when you try to do the latter SVN will complain and not do the commit.) Finally, if you try to do a commit with an out-of-date working copy, SVN will not do it and will tell you you’re out of date, necessitating an update before the commit.

3 Running a Job

Hopefully by now you have compiled both versions of the code, so that both `xbear` and `mpibear` are sitting happily in `bearclaw/contrib/astro`. Now it is time to put those executables to work.

First, we need a place to run them. If you did not receive this documentation through a subversion download, open a terminal, navigate to a directory of your choosing (for instance, `/home/username/`), and type:

```
svn co https://data-store/svn/orda/doc Documentation
```

This will create the folder `Documentation/`, which contains all of the “.tex” files used to create this documentation as well as a folder called `Documentation/firstrun`. Navigate to that folder (`Documentation/firstrun`) now. This is where you will run *AstroBEAR* for the first time.

But not quite yet. First, take a look at the files in `firstrun/` (using the command `ls`). You will see the following list:

```
TABLES bear.data domain.data host.def out physics.data template.data
```

The files with the “.data” suffix as well as the directory `out/` must be in every folder where you intend to run *AstroBEAR*. These files are the input of your run. Their contents will be discussed a little later. The directory `out/` is where *AstroBEAR* will record the data it computes. It must be named “out”. Once again, every directory in which you intend to run `bearclaw` must contain these

files, with one caveat. The file `template.data` contains the physical parameters for a specific “toy” problem designed for this documentation. It will be replaced by similar data files named after the problem modules for which they contain parameters (for example, `envelope.data`). Writing your own problem module will be discussed in a subsequent section. For now, you will be simulating a steady shock running over a clump.

Speaking of that, it’s now time to run your first simulation.

Running a Single Processor Simulation To run a single processor job, you will use the `xbear` executable. Copy that file to the current directory by typing:

```
cp (location of bearclaw directory)/bearclaw/contrib/astro/xbear .
```

Now find a node of the cluster on which to run your job. Our cluster does not currently have any front end job scheduling, so you have to search around for a node with a free processor. Use the command `ssh ordaN`, where N is the number of the node you want to explore (1-12), to log into a given cluster node. Then use the command `top` to list what jobs are running on that particular node. If someone is already using the processors on that node, move on to the next one. Remember, though, that all of our nodes are dual processor. Since you are only running a single processor job, that means that you can run your job with no loss of performance on the same machine as someone else running a single processor job (or as a job that you have already started). In fact, it is preferable to try and stack two single processor jobs to a node, as it leaves other nodes free for people running multiprocessor jobs.

Once you find a node on which you want to run, beginning the run is easy. In fact, you could just type `./xbear` and your simulation would start running without looking back. However, it is recommended that you use the command `nohup` to ensure that your run keeps going even if your connection to the cluster terminates or you accidentally close the terminal. The command `nohup` will by default divert your console output from this simulation to a file called `nohup.out` unless you pipe the output to a different file. It is usually a good idea to give this file a descriptive name, such as `firstrun.out`. You can then use the command `tail -f` to view your console output as it is generated. (The `-f` option “follows” the end of the file as it is created). Putting all this together, a typical sequence of commands to start a single processor job would look like:

```
nohup ./xbear > firstrun.out &
tail -f firstrun.out
```

Now your simulation is running in the background while the console out put is being followed by the `tail` command in the foreground. If you need to use the terminal for something else, you can always suspend the `tail` command and restart it when you are done, without harming your simulation.

As you might imagine, the cluster gets quite full of jobs named `xbear` or `mpibear`. However, the `xbear` and `mpibear` executables can be given more descriptive names. For instance, to rename `xbear` as `firstrunbear` and run it, you would type the following at the terminal:

```
mv xbear firstrunbear
nohup ./firstrunbear > firstrun.out &
tail -f firstrun.out
```

Now, when you use the command `top` on the node on which you are running, you will see your job listed as `firstrunbear`, instead of as `xbear`. This is especially helpful if you need to manually kill the simulation and you cannot remember its process id.

Running a Parallel Simulation Running a parallel (or multiprocessor) simulation works much the same as a single processor simulation, except that you will be using the `mpibear` executable. Therefore, copy `mpibear` to the `firstrun` directory using:

```
cp (location of bearclaw directory)/bearclaw/contrib/astro/mpibear .
```

(If you do not want to overwrite the data produced by the single processor run, rename the `out` directory and then make a new, empty directory called "out".)

The first step is to find multiple processors to run on, which means finding nodes without jobs running on them. Do this using the same method used above for single processor jobs. Since our cluster is composed of dual processor nodes, you will almost always run parallel jobs using a multiple of two processors. For instance, to run a four processor job, you need to find two free nodes with two open processor each. Once you have found two free nodes, you need to record them in the file `host.def`. If you use your favorite text editor (for instance `nano`) to open `host.def` (`nano host.def`), you will see the following:

```
#orda1
#orda2
#orda3
#orda4
#orda5
#orda6
#orda7
#orda8
#orda9
#orda10
#orda11
#orda12
#ordadisk
```

To select the two nodes you found, delete the `#` before their names, exit and save the file. For instance, to use nodes `orda6` and `orda8`, your `host.def` file should look like:

```
#orda1
#orda2
#orda3
#orda4
#orda5
orda6
#orda7
orda8
#orda9
```

```
#orda10
#orda11
#orda12
#ordadisk
```

Now, before you execute the `mpibear` command, make sure you are logged into one of the nodes you selected and type `lamboot host.def`. This will setup the communication between the processors that you are intending to use. It is vital to include the argument `host.def` in your command, otherwise `lamboot` will assume that you want to run a multiple processor job all on the one node to which you are currently logged in. Consequently, when you ask it for four processors, it will create four instances of `mpibear` which will all fight for time on the single node, slowing your simulation tremendously.

The syntax for telling a job to use multiple processors is simple. Before the executable file, type `mpirun -np N`, where N is the number of processors you intend to use. In this case, you want four processors. Following suit with the single processor case, the sequence of commands to start a parallel job would look like:

```
lamboot host.def
nohup mpirun -np 4 ./mpibear > firstrun.out &
tail -f firstrun.out
```

If you use `top` on one of the nodes on which you are running, you should see two instances of `mpibear`. As with `xbear`, you can rename `mpibear` as something more descriptive so as not to get your job confused with others, in which case all the instances of your run will bear the descriptive name. If you need to terminate your parallel run, or if one but not all of the `mpibear` instances terminate unexpectedly, `lamboot` has a handy inverse call `wipe`. Executing the command `wipe host.def` from one of the nodes running your parallel job will terminate all of the instances of `mpibear` running on those nodes.

Viewing your simulation At this point, you might want to skip to the next section to learn how to see what your simulation is doing. Once you have visualized your first simulation, return to this point to learn how to modify that simulation.

Manipulating the Input Files As mentioned before the files `bear.data`, `domain.data`, `physics.data`, and `template.data` are the input files for *AstroBEAR*. They define both the physical parameters and the computational options for your simulation; as such, they must appear in every directory in which you intend to run *AstroBEAR*. These datafiles are usually well commented with short descriptions for each of the parameters they contain. The following subsection describes some of the most important parameters and so that you may begin modifying (playing) with the template simulation.

As you begin to change the simulation parameters and run new simulations, it is recommended that you do not continue to run in the same directory as your first simulation. Instead, it is good practice to create a new directory everytime you reconfigure the parameters. To do this, merely create a new directory (such as `Documentation/secondrun`), copy the “.data” files into your new directory, create a new out directory, modify the “.data” files to taste, and repeat the steps for running your simulation. Not only does this ensure you do not

accidentally overwrite a previous simulation, it creates a record of each simulation you run in your directory structure.

The “.data” files you are about to explore are merely text files formatted to be read by the *AstroBEAR* executable. As such, you can view their contents easily with your favorite inline text editor. For instance, to view `template.data` with `nano`, type `nano template.data`.

- **template.data:** This file describes all the parameters specific to this problem of running a shock over a clump. Its contents are mostly self-explanatory. Note that the author chose to use computational units instead of physical units. This is because the module was not written to produce any particular physical situation, so it is easier to use computational units instead of physical units.
- **bear.data:** This file describes all the parameters related to the actual computation in the simulation. The file is divided into multiple sections. *AstroBEAR* has the ability to run a simulation with multiple root domains, each with its own size and parameters defining its AMR structure. This is a useful ability to have when one wants to run a simulation with a very large domain, but does not require the same resolution in every part of this domain. The first section deals with parameters that will be true across all of these domains. The second section, and all subsequent sections, define the parameters unique to each individual domain. Most of the simulations you will be running require only one root domain, so you will have only two sections.

Many of the parameters in this file are well described by the comments in the file. The following are some of the most common uses of these parameters, and some of the most useful for new users.

- *Changing the size and/or resolution of your simulation:* The size of your simulation depends on `xlower`, `xupper`, `ylower`, and `yupper`. If you had more than one root domain, these parameters would correspond to the beginning and end of the particular domain defined in this section. However, since you only have one root domain, `xlower` and `ylower` should always be 0.0, and `xupper` and `yupper` should correspond to the length of the domain in the horizontal and vertical direction respectively. These parameters are in computational units. To find out what one computational unit corresponds to in physical units, you need to calculate the length scale for this simulation. You can do this by hand, but most simulations do this for you and output it (with the label `lscale`) at the beginning of the run. (See the section about writing your own application.) Therefore, it is probably easier to start a simulation, stop it after you know what scaling it is using, and then change these parameters to fit the size of the domain that you choose.

The resolution of your simulation depends on the following parameters: `mx`, `my`, `mglobal(1)`, `mglobal(2)`, and `MaxLevel` (which appears in both the `Global parameters` section and the `Grid 1` section). For the time being, you should always have `mglobal(1) = mx` and `mglobal(2) = my`. (Again, this would not be the case with more than one root domain.) The parameters `mx` and `my` correspond to the number of grid cells on the coarsest level grid in the horizontal and vertical directions, respectively. The parameter `MaxLevel` defines how many levels of refinement (including the root level) the AMR system will use. It should be the same in both the the “`global parameters`” section and the “`grid`

1” section. For each level of refinement, the AMR system will take a cell flagged for refinement on the previous level and divide it in half along the direction begin refined. So, for `MaxLevel = 3`, the AMR system might divide a root level grid cell once, then divide each subsequent cell once to make four cells. The root level of refinement counts as level one, so the root level of refinement will have a resolution of `mx` by `my`. The resolution on the finest level is therefore given by:

$$\text{mx} * 2^{(\text{MaxLevel}-1)} \text{ by } \text{my} * 2^{(\text{MaxLevel}-1)}$$

One useful way to measure your resolution is as the maximum number of cells per computational unit (which can be calculated by dividing the above formula by the number of computational units in the domain in the appropriate direction). It is not necessary to have an integer number of cells per computational unit, though it is easier on the user sometimes. It is also not necessary to make `mx` and `my` such that this number is the same in both directions, in which case your cells will merely be rectangular but not square.

- *Domain decomposition:* Domain decomposition (located at the end of the first set of parameters in the `Grid 1` section) divides the root domain into several sections, or grids, for a multiprocessor run. These different grids can then be divided among the different processors so that the work load can be better balanced. Were it not for domain decomposition, all of the computation on the root level grid would have to be done on one processor. On higher levels of refinement, the AMR system automatically defines the grids, or regions that have been refined from the level below, and balances them among the processors.

Domain decomposition is defined in `bear.data` by a sequence of integers, separated by spaces, that represent the number of decompositions in the dimension corresponding to their position in the sequence. For a two dimensional simulation, the first number is the number of times the domain will be decomposed in the first (horizontal) direction, and the second number is the number of times the domain will be decomposed in the second (vertical) direction. In this example, `bear.data` contains the sequence `2 2`, which means there will be a total of four domain decompositions, one in each quadrant of the domain. You should always have at least as many domain decompositions as processors you intend to use for your multiprocessor simulation. Also, the number of cells in each direction (`mx` and `my`) must be divisible by the number of decompositions along that direction. In this case, that means that the number of cells in each direction must be even.

The final purpose of domain decomposition is to force a better load balancing of areas in the domain that will require the most processing. In this template simulation, there is a clump placed (initially) in the center of the domain. One fourth of this clump lays in each of the four quadrants created by the domain decomposition described above. This means that when the AMR system creates refined grids, it will have to create four grids on the next level of refinement in each of the four quadrants, instead of one grid that contains the whole clump. Therefore, you can have four processors working on the computation surrounding the clump instead of just one.

- *Boundary conditions:* The parameters `mthbc(1)`, `mthbc(2)`, `mthbc(3)`, `mthbc(4)` determine how the simulation behaves at its left, right, bottom and top boundaries, respectively. There are three types of boundary conditions that may used. The extrapolating

boundary condition treats the edge of the domain as if it is open and allows material to flow off the domain freely. To use this boundary condition, set the appropriate parameter to 1. The periodic boundary condition essentially wraps the domain around on itself. It translates material that encounters one edge of the simulation to the opposite edge with its physical properties intact. To use this condition, set the appropriate two opposing boundary parameters to 2. The reflecting boundary condition perfectly reflects material that encounters the edge of the domain back into the domain. This condition is useful if you want to simulate a problem that is symmetric along one or more of the domain's boundaries. For instance, a circumstellar environment with the star placed in the lower left corner of the domain would use reflective boundary condition along the left and bottom boundaries. To use this condition, set the appropriate parameter to 3.

- *Changing the runtime and number of output frames of your simulation:* The run time for your simulation is defined by the parameter `tfinal`. It is given in computational units. To find the physical time corresponding to one computational unit, you must calculate the time scale for your simulation. As with `lscale`, it is easier to do this by letting the simulation do it for you. When the simulation begins running, it will output a parameter called `runtimesec` which is the number of seconds simulated per unit of computational time.

The number of frames your simulation will output is given by `nout`. If you have the flag `WriteFrameAtSpecifiedTimes` set to false, you may write slightly fewer frames than requested by `tt nout`. This is because, with `WriteFrameAtSpecifiedTimes` set to false, the simulation will step through time in a way such that the last time step before it outputs a frame steps past the time at which it should have output the frame to maintain a constant spacing in time between the frames output. This will usually not be noticeable in your visualization of the simulation, and it allows the simulation to run more fluently than if it tried to step to exactly the time at which it should be outputting a frame. As a user, it is your choice as to how to set this parameter.

- *Restarting your simulation:* Any simulation that has written a frame can be restarted from the state it was in at that frame with different parameters. To do this, change the `Restart` flag to T and select the frame at which you want to restart via `RestartFrame`. If you choose to change `tfinal`, be sure to change `nout` accordingly to maintain the same ratio of frames to computational unit of time.

- **physics.data:** This file contains the parameters that determine which physical principles are applied and how they are applied within the simulation.

- *Source terms:*

- *Viscosities:*

- **domain.data:** Most of the parameters in this file are obsolete. Despite appearances, the `xLowerDom`, `xUpperDom`, `yLowerDom`, and `yUpperDom` are irrelevant in determining the domain size. The only parameters of general import are declarations at the end of the file, which set the problem module you are using for your simulation. For more information about this, see the section about writing your own problem module.

4 Visualizing

With AstroBEAR, there are two packaged methods of visualizing your data. These are `bear2fix` and a visualization package in MatLab.

- **bear2fix** Bear2fix is a simple to use highly customizable package written in Fortran. To download bearfix, simply obtain it from the SVN server by using these commands:

```
mkdir bear2fix
svn co https://data-store/svn/orda/bear2fix/dev bear2fix
```

This creates a directory called 'bear2fix' and puts the latest revision of bear2fix into it. To make bear2fix, simply navigate to the bear2fix directory, and type 'make'. To use bear2fix, create a symbolic link to the data directory you wish to process, and name the link 'out', or place the executable bear2fix in your run directory. The bear2fix program looks for a directory named 'out' with the data files, and if there is neither a link nor a directory, the program will terminate. It is also very important that the out directory specified has the entire contents of the run, because it needs files like "gd.data" to proceed. To create a symbolic link, use this notation:

```
ln -s source_directory_or_file symbolic_link_name
```

To execute bear2fix:

```
./bear2fix
```

The program prompts the user for a choice:

```
select data set (Enter -1 for all frames, -2 to specify a range or -3 to use defaults)
```

If '-1' is selected, the program will process all frames.

If '-2' is selected, the program will prompt for what frames:

```
what frames? (set end frame=-1 to select all frames after begin frame)
begin frame, end frame?
```

Here, the beginning and end frames are specified, separated by a space. For example, to process frames 3 through 10, use:

```
3, 10
```

The results of selecting '-3' will be discussed a little further on.

The next prompt is:

```
select coarsening ratio
```

This interpolates the finest level of AMR refinement onto a coarser grid. When set to 1, the entire grid is at the finest AMR resolution.

Bear2fix then asks how the data will be processed:

What application?

1=hydro astro

2=mhd astro

3=hydro lab (LLE)

This changes how the data dumps are processed (Units, etc..). The various data fields are different for Hydro and MHD. Also, Hydro astro presents the data in n/cc as opposed to g/cc for hydro lab (LLE).

What operation?

1=write fixed grid hdf

2=write fixed grid fits

3=plot log10 density (2D only)

4=plot cut

5=plot other field

6=Schlieren

7=user coded

This selects how the data is processed and visualized. Option 7 is highly customizable.

Option 1 This option writes output data in HDF as though it were a fixed grid at the finest AMR resolution.

Option 2 This option is the same as 1, but in FITS (Flexible Image Transport System).

Option 3 This option produces a PostScript file of the density in Log10. The script ps2png.pl can be used to convert the PS files to PNG images for easier viewing.

Option 4 Plot Cut will produce a line graph of whatever data field is chosen at the points specified. This will prompt for a cut position, and just input it as requested:

```
cut position x1,y1,x2,y2
```

After the cut position is specified, the program prompts for the field:

```
cut field
```

Simply put in the desired field number for the cut.

Take log10 (T/F)?

This asks if the data field should be in log10, which is useful for a density cut. Use T or F for True or False. Use ps2png.pl to convert the ps to png.

Option 5 With this option, any data field can be visualized. Simply input the field number. Use ps2png.pl to convert the ps to png.

Option 6 This produces a "Schlieren" plot (Black and White). Use ps2png.pl to convert the ps to png.

Option 7 This option can be changed to suit the needs of any particular application. These are options that usually aren't needed in the general case or aren't used on a regular basis. There are a multitude of options, and these can be selected by editing the bear2fix.f90 file, and either uncommenting the desired option or program a custom subroutine. Be sure to comment out the other options that are undesired. There are multiple options already included in the bear2fix.f90, like plot slicing, vorticity, and dumping the data files to binary. These options are located in the bear2fix.f90 after the 7 cases as a block of commented out code.

After selecting what operation to complete, the program may prompt for the hotbox level:

```
max hotbox level (-1 for no hotboxes)
```

Simply input the level of desired hotboxes, which is useful when trying to make sure that the code is refining areas that need refinement. It will draw boxes around the areas that were refined, up to the level specified. To be sure to see all hotboxes, set it to a large number like 999. To see no hotboxes, input -1.

Also, the program may prompt for how to scale the frames:

```
What frame scaling?  
0=relative  
1=absolute  
2=zeroed  
3=user specified
```

Relative scaling makes each frame's scaling independent of the others, meaning that if there are great changes in the values of your data, the color bar and numerical equivalent will shift.

Absolute scaling makes the program read each frame for the maximum and minimum values before writing any output. This makes sure that all frames have the same max and min, and the colorbar will not shift in any way.

Zeroed scaling makes the program read each frame for the maximum value, and scales that against a minimum of 0.

User Specified means that the user picks maximum and minimum values, which are prompted for next.

Enter: minvalue,maxvalue

Simply enter the desired minimum and maximum values.

If in the very beginning, -3 was selected for the defaults, without editing the sourcecode itself, the program would select the following:

1. Hydro Astro as the application
2. Coursening Ratio of 1.
3. Write Log10 Density (2D only) as the operation
4. Hotbox level -1, meaning no hotboxes.

The only input required would be the frame scaling, which can be selected at the prompt.

Various applications of BearClaw may use a different version of HDF. To change between HDF4 and HDF5, Bear2fixIO.f90 must be edited. Under the heading '! fixed grid variables' at the very beginning of the code file, there is an option called:

```
INTEGER,PARAMETER :: HDF4=4,HDF5=5
INTEGER,PARAMETER :: OUTPUT_FORM=HDFx
```

Where the HDFx is either HDF4 or HDF5. Change the x to the desired version number (either 4 or 5), and compile the program.

For using ps2png.pl, simply use:

```
./ps2png.pl LOCATION_OF_PS_FILES/*.ps
```

- **MATLAB** is the base program for another set of visualization tools for Bearclaw.

In Matlab, the path of the visualization tool directory (BEARCLAW_DIR/matlab) must be added to the path for matlab. Goto File -> Set Path, and add the directory. Be sure to clock save, or else the changes won't stay. To launch the tool, at the matlab command prompt:

viz

This lanuches the XBEAR Visualization tool. Make the directory of the data you wish to process the "Current Directory" in Matlab. This brings you to a simple to use program that can do quite a bit. However, it is slower than bear2fix. There are multiple buttons and pull down menus to choose from:

File Format: This suite only supports AMRClaw and HDF4 data formats.

File Type: Choose whether the data files are formatted or unformatted.

Plot Type: Choose what type of plot, either Pseudo-Color, Contour, Schlieren, Scatter Plot, Grey-Scale, or Jet Colormap. This only changed what the data looks like.

Plot Vectors: This allows either velocity or magnetic field vectors to be plotted, with multiple options.

Streamlines: This allows streamlines of either velocity or magnetic field to be plotted, with multiple options.

X-Y Cut: This allows a plot-cut at the specified x and y locations.

Colorbar On/Off: This allows the colorbar to be displayed or not displayed on the graph.

Advanced: This can change the number of levels plotted for the Plot, Grid Edges, and Grid Lines.

Component of q to plot: This allows the manual selection of a datafield to be displayed. Use this or the pull down box next to it, "Choose Variable"

Choose Variable: This allows the selection of set datafields to graph, like Log(Density), Velocity, Mach Number, Pressure, Temperature, Log(Temp), or User Defined. Use this or "Component of q to plot".

Set Colorbar Scaling: Here, the scaling of the colorbar can be set, to see the data however is best/must useful.

Figure Size: The size of the figure can be changed, either Normal, Large, or User Defined.

Axes: Change the size of the axes based on user preference.

When using the tool to visualize data, and the run used more than a 1x1 root grid decomposition, be sure that the Axes are accurate, as sometimes they will only address a small piece of the overall grid. Just change the axes to be however is desired, or to be the width and length of the domain in computational units (From bear.data).

There are also methods to save the figures and to create movies out of them. These are the "Save Image" and "Save" buttons.

Click "Plot Frame" to plot the frame based on the options. Click "Plot Next Frame" to advance the same plot by one frame. This suite has the capability to plot up to 4 frames, by clicking on "Plot Multiple Frames". This has many of the same options as the single frame version, and only need to have up to 4 data fields selected to plot. This also has the ability to advance by one frame and save images. It produces up to 4 plot windows that appear as a 2x2 grid on the screen.

5 Writing Your Own Application

The simulation which was set up and run in Section 3 and visualized in Section 4 was an example of an *AstroBEAR* application module, written to simulate a specific environment (in this case, a simple shock/clump interaction). This section will discuss how a module is constructed in general and how a newly constructed module connects to the rest of the code.

The comments contained in this section are designed to give the user an understanding of the process on a broad level, so that each of the three sections in the module is motivated by an understanding of how it fits within the rest of the module and the code overall. The module itself, in complement to this document, contains much more detailed information on the actual mechanics of an application module.

The source code for all the existing *AstroBEAR* modules are contained in `bearclaw/contrib/astro/`. For the example module, the actual source file is `bearclaw/contrib/astro/template/template.f90`.

5.1 Module Description

As mentioned above, a module is essentially broken up into three parts, or subroutines. Two of these parts are called when the simulation begins, and the third is called at every time step in the simulation:

Scaling Subroutine (`scaleTemplate`) The scaling subroutine is used in conjunction with the data files to establish a relationship between the computational units of the program and the physical units that are to be simulated. It is called by `setprob` at the same time that `physics.data` and `bear.data` are being read. Depending on how the existing data files are used—*i.e.*, if the environment is entirely described by the data files, and the specifics of the simulation are hard-coded into the source file—this subroutine may contain very little code.

Initialization Subroutine (`initTemplate`) The initialization subroutine is called immediately after the “raw” simulation domain has been created, via `qinit`. The subroutine is used to describe the initial physical parameters (density, momentum, internal energy, magnetic field) of every point in that domain where they are intended to deviate from the “ambient.” In other words, before the subroutine is run the domain looks like Fig. 1, and once the subroutine is done (*i.e.*, at Frame 0) it looks like Fig. 2.

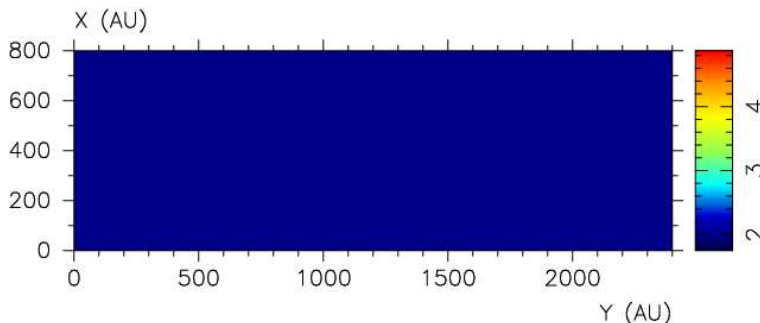


Figure 1: An Uninitialized Simulation Domain; no data for ρ , v , B , etc. for anything but the ambient in the grid.

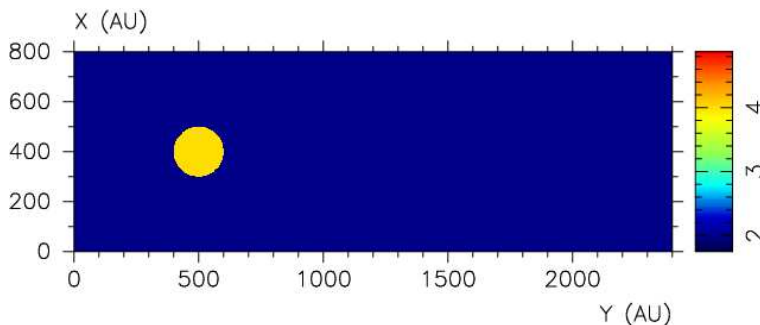


Figure 2: An Initialized Domain, as specified in `initTemplate`.

Before-Step Subroutine (`b4stepTemplate`) The before-step subroutine is called before each time-step of the simulation, so that the driving mechanisms specified by the user can be

reapplied at each step. The ordinary physics going on in the simulation (advection, gravity, etc.) are of course handled by the code—this subroutine directs behavior like a specified rate of flow of material from a boundary into the domain. In contrast, if the initialization routine specified, say, an initial momentum for a clump and the before-step routine was empty, the clump would react as if it was given an initial impulse and would evolve accordingly.

5.2 Connection to Code

When a new application module is written, the user must let the compiler know that it exists. This involves adding lines to certain files in the `astro` directory (`bearclaw/contrib/astro`) to mimic the following, appearing with their approximate linenumbers:

<i>Filename</i>	<i>Line</i>	<i>Line contents</i>
<code>i_qinit.f90</code>	50	<code>IF (lTemplate) CALL qinitTemplate(Info)</code>
<code>i_setprob.f90</code>	100	<code>IF (lTemplate) CALL scaleTemplate</code>
<code>problem.f90</code>	50	<code>USE Template</code>
	240	<code>IF (lTemplate) CALL b4stepTemplate(Info)</code>
<code>Makefile</code>	40	<code>\$(BEARCLAW_ASTRO)/template/template.f90 \</code>
	60	<code>\$(BEARCLAW_ASTRO)/template/template.o \</code>
<code>globaldeclarations.f90</code>	25	<i>Add lTemplate to LOGICAL list.</i>
	100	<i>Add lTemplate to namelist.</i>

6 Troubleshooting

In the course of running a simulation, there are a few common errors which may arise.

Bad CFL The Courant, Freidricks, Levy (CFL) number specified in `bear.data` under the item `cflv(2)` relates to the acceptable speed of propagation of information in the simulation. Therefore, when variable-timestepping is allowed (which it is normally), the CFL number sets an upper-limit on the timestep that the code is allowed to take. Essentially, a smaller CFL forces a smaller timestep—it typically has a value of 0.4 or 0.5. The initial timestep is also given in `bear.data` under the item `dtv(1)`.

Occasionally, this initial timestep will be such that the code may mistakenly assume a much-too-large timestep for the subsequent step, resulting in CFL numbers that may range from about 1 to greater than 10^{12} . Since the maximum allowable CFL number, `cflv(1)`, is 1, this may cause the simulation to stop at the very beginning of a run.

Increasing or decreasing the initial timestep by a factor of 10 or 100 will in many cases fix this problem. If it does not, it's possible that the relationship between the initial timestep and the length of the simulation in computational time units or the physical time are disparate and need to be checked. If these relationships are acceptable, then there probably is a problem with how the environment is being initialized.

Another Issue Another solution. (a work in progress)