LAREXD USER GUIDE

Tony Arber[*]
University of Warwick, Department of Physics, Coventry, CV4 7AL, UK

Version 3.3

Abstract

LareXd are Lagrangian remap codes for solving the MHD equations in 2 or 3D. The code uses a staggered grid and is second order accurate in space and time. The use of shock viscosity and gradient limiters make the code ideally suited to shock calculations. This document describes how to obtain, make and run the *LareXd* codes. The instructions are aimed at code users, not developers, although some references to more detailed numerics are included. The manual will only be updated when there are new physics packages added. Hence this manual applies to all *LareXd* versions starting with a major version number of 3. Internal code changes and optimisations will be documented in the source code itself and through a *changelog.txt* file which will be updated with each new version and included with the source code distribution of *Lare2d*. The *Lare3d* will always track the changes in *Lare2d* so if you only use *Lare3d* you still need to check the manual and changelog in the *Lare2d* distribution.

1 Introduction

The Lagrangian remap code comes in 2D (*Lare2d*) and 3D (*Lare3d*). For convenience when referring to either code this is called the *LareXd* code. A detailed description of the algorithm has been published [1] along with test cases. Note however that the latest version of the code uses a different form of shock viscosity and resistivity to that documented in JCP.

Without going into detail, see [1] for complete algorithm, the main features of the code are:

- Solves the nonlinear MHD equations with user controlled viscosity and resistivity. Optional physics packages are: gravity, the Hall term (only in 2D), partially ionised hydrogen equation of state, Cowling resistivity, parallel thermal conductivity and optically thin radiative losses.
- Splits each timestep into a Lagragrian step followed by a remap onto the original grid. This allows all of the physics to be included into a simple Lagrangian step making it easy to add additional physics. The remap step includes gradient limiters to ensure the correct shock solution.
- Uses a staggered grid to prevent the checkerboard instability and to build conservation laws into the finite difference scheme, i.e. it is a compatible difference scheme in Lagrangian numerical jargon.

[*]E-mail address: T.D.Arber@warwick.ac.uk

- Parallelised via MPI and known to scale linearly.
- Comes with packages for visualisation by IDL, MATLAB, Python and VisIt. [2]

This manual should be sufficient to set up *LareXd* for any initial conditions, run the code and begin analysing the results. In order to do this you must know the core equations that are solved. It is also essential that you know something about the code structure and once the code has run how to load the data into IDL or VisIt. For the more ambitious you will need a good understanding of the Lagrangian step as this is what you will need to change to include additional physics. It is unlikely you will ever need to know how to change the remap step. If editing the core code note that this is always in normalised form so you must understand how this is applied. For examples of the uses of the *LareXd* codes just check out papers that reference [1]. In order that this manual does not get stuck in the fine details of shock theory etc. it has been written in an informal style which deliberately oversimplifies some theoretical points. It's a user manual, not a theoretical treatise. References to grown up theory are at the end.

2 Equations and Normalisation

In order to get used to using the code, and to make the initial description easier, here we concentrate just on standard resistive MHD. In almost all cases the code should be run with shock viscosity but how this is treated is postponed until later to keep the first steps easier. In S.I. units the standard resistive MHD equations are

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v}) \tag{1}$$

$$\frac{\mathrm{D}\mathbf{v}}{\mathrm{D}t} = \frac{1}{\rho}\mathbf{j} \times \mathbf{B} - \frac{1}{\rho}\nabla P \tag{2}$$

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} \tag{3}$$

$$\frac{\mathrm{D}\epsilon}{\mathrm{D}t} = -\frac{P}{\rho}\nabla \cdot \mathbf{v} + \frac{\eta}{\rho} j^2 \tag{4}$$

$$\mathbf{E} + \mathbf{v} \times \mathbf{B} = \eta \mathbf{j} \tag{5}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{j} \tag{6}$$

Where $\mathrm{D}/\mathrm{D}t$ is the advective derivative and all other terms have their usual meaning. Ideal MHD is recovered simply by setting η, the resistivity, to zero. Note in these equations η is resistivity, not magnetic diffusivity, so that $\eta = 1/\sigma$ where σ is the conductivity. Note that the energy equation is written in terms of the specific internal energy density ϵ. There are good numerical reasons for doing this which go beyond the scope of this introduction. Definitions you may find useful for converting between ϵ and the more familiar pressure and temperature are

$$P = \frac{\rho k_B T}{\mu_m}$$

$$\epsilon = \frac{P}{\rho(\gamma - 1)} = \frac{k_B T}{\mu_m(\gamma - 1)}$$

where μ_m is the reduced mass, i.e. the average mass of all particles in the plasma. Hence $\mu_m = m_p$ for neutral hydrogen atoms (m_p is the proton mass) and $\mu_m = 0.5 m_p$ for fully ionised hydrogen.

There are two occasions when you will need to understand the normalisation. The first is to input your variables in dimensionless form. To do this of course you must have chosen a normalisation. For some additional physics packages to work correctly you must tell *LareXd* the numerical values of your normalising length, magnetic field and mass density (more on this below). The additional physics packages that require this are thermal conduction, optically thin radiative losses and the inclusion of neutrals. If you are only running resistive MHD then these are not needed. The second reason you may need to understand the normalisation is that all of the core code works on the normalised equations so that should you want to edit the code, e.g. to change the resistivity, you must do this on normalised equations.

The normalisation is through the choice of normalising magnetic field B_0, density ρ_0 and length L_0. Thus we define dimensionless quantities as

$$\begin{aligned} x &= L_0 \hat{x} \\ \mathbf{B} &= B_0 \hat{\mathbf{B}} \\ \rho &= \rho_0 \hat{\rho} \end{aligned}$$

These three basic normalising constants are then used to define the normalisation for velocity, pressure, time, current density, electric field, temperature, reduced mass and specific internal energy density through

$$\begin{aligned} v_0 &= \frac{B_0}{\sqrt{\mu_0 \rho_0}} \\ P_0 &= \frac{B_0^2}{\mu_0} \\ t_0 &= \frac{L_0}{v_0} \\ j_0 &= \frac{B_0}{\mu_0 L_0} \\ E_0 &= v_0 B_0 \\ T_0 &= \frac{\epsilon_0 \bar{m}}{k_B} \\ \mu_{m0} &= \bar{m} \\ \epsilon_0 &= v_0^2 \end{aligned}$$

so that $\mathbf{v} = v_0 \hat{\mathbf{v}}$, $\mathbf{j} = j_0 \hat{\mathbf{j}}$, $t = t_0 \hat{t}$ and $P = P_0 \hat{P}$ etc. In these equations $\bar{m}$ is the average mass of ions in the plasma, i.e. $\bar{m} = m_p$ for pure hydrogen but $\bar{m} = 1.2 m_p$ is typical for the Solar corona. Thus $\bar{m}$ is actually an input parameter to the code and is set in the `control.f90` file (see later). Applying this normalisation to the ideal MHD equations simply removes the vacuum permeability μ_0. Note if you would like to fix a normalising temperature this means that you must choose L_0, ρ_0, B_0 so that the calculated T_0 gives the required temperature normalisation.

For resistive MHD substitute equation 5 into 3 and apply the normalisation to get

$$\frac{\partial \hat{\mathbf{B}}}{\partial \hat{t}} = \hat{\nabla} \times (\hat{\mathbf{v}} \times \hat{\mathbf{B}}) - \frac{1}{\mu_0 L_0 v_0} \hat{\nabla} \times (\eta \hat{\nabla} \times \hat{\mathbf{B}})$$

Note that the resistivity is not assumed spatially uniform in *LareXd*. This leads naturally to the normalisation of resistivity of

$$\hat{\eta} = \frac{\eta}{\mu_0 L_0 v_0}$$

or $\eta_0 = \mu_0 L_0 v_0$. Since v_0 is the normalised Alfvén speed this means that $\hat{\eta} = 1/S$ where S is the Lundquist number. This resistivity can either be uniformly applied or triggered when the current density exceeds a threshold value. This can be controlled in the `control.f90` file.

Dropping all of the hats on normalised variables the final normalised resistive MHD equations, in Lagrangian form, are

$$\frac{\mathrm{D}\rho}{\mathrm{D}t} = -\rho\nabla\cdot\mathbf{v} \tag{7}$$

$$\frac{\mathrm{D}\mathbf{v}}{Dt} = \frac{1}{\rho}(\nabla\times\mathbf{B})\times\mathbf{B} - \frac{1}{\rho}\nabla P \tag{8}$$

$$\frac{\mathrm{D}\mathbf{B}}{\mathrm{D}t} = (\mathbf{B}\cdot\nabla)\mathbf{v} - \mathbf{B}(\nabla\cdot\mathbf{v}) - \nabla\times(\eta\nabla\times\mathbf{B}) \tag{9}$$

$$\frac{\mathrm{D}\epsilon}{\mathrm{D}t} = -\frac{P}{\rho}\nabla\cdot\mathbf{v} + \frac{\eta}{\rho}j^2 \tag{10}$$

Where ϵ is the specific internal energy density, $P = \rho\epsilon(\gamma - 1)$ and γ is the ratio of specific heats - again this is specified in `control.f90`.

The fact that the thermal pressure is normalised to the reference magnetic pressure means that there is no explicit reference to the plasma β in the code. It does not mean that the code must have a magnetic field. B_0 is the normalising field. It is then perfectly acceptible to set $\mathbf{B} = 0$ as this is really fixing $\hat{\mathbf{B}}$ and has no effect on B_0. The temperature is not needed to solve the resistive MHD equations and as a result no explicit temperature normalisation is needed to solve the resistive MHD equations. However it is often convenient to specify initial conditions in terms of temperature and this must be converted to specific internal energy density before the first timestep.

LareXd is set up with gravity assumed downwards in the z direction in *Lare3d* and the y direction in *Lare2d*. It's trivial to change the direction in the `lagran.f90` file. In normalised units the momentum equation for the z component is therefore given by

$$\frac{\mathrm{D}\mathbf{v}}{Dt} = \frac{1}{\rho}(\nabla\times\mathbf{B})\times\mathbf{B} - \frac{1}{\rho}\nabla P - \hat{g}$$

where $\hat{g} = L_0 g/v_0^2$. In *LareXd* $\hat{g}$ is only a function of z, i.e. it is a 1D array, and it does not change in time. Note also that gravity is defined on the cell vertex, i.e. the same location as all components of velocity. More on this later when the computational grid is defined.

2.1 Shock viscosity

The shock viscosity used in *LareXd* is different to that in Reference [1] and is sketched below along with a reference to the core material. From the user point of view the viscosity is controlled by two parameters. These are `visc1` and `visc2` with the first controlling linear viscosity and hence active on all shock, the second acts mostly at strong shocks. For strong shocks these should both be set to one. However for most solar physics, which only have very weak shocks, it is better to run with smaller values. Typically `visc1=0.1` and `visc2=0` are good starting points. It is best to try these values and then either increase or decrease based on the problem studied. These initial tests can be done on a low-resolution grid while still 'playing' with the problem. For some idealised wave problems it is best to set both to zero for example to avoid wave damping. Experimenting with these parameters is the only way to determine what is best.

The shock viscosity in *LareXd* is based on the edge viscosity formulation in Reference [5]. This is presented first for Euler's equation only and MHD added later. First find the area weighted nodal density

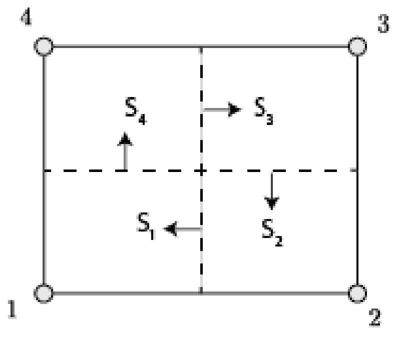

Figure 1: Labels used for edge viscosity.

and sound speed ρ_v and cs_v. Then for each edge of the cell find $cs_i = \min(cs_v^i, cv_v^{i+1})$ where i labels the nodes of the cell as in Figure 1 and is cyclic so for $i = 1, i - 1 = 4$ etc.

Next define the edge density through $\rho_i = 2\rho_v^i \rho_v^{i+1}/(\rho_v^i + \rho_v^{i+1})$. Then on edge i the associated viscous force is

$$\mathbf{f}_i = \rho_i \{\bar{c}_2 \Delta v_i + (\bar{c}_2^2 \Delta v_i^2 + c_1^2 cs_i^2)^{1/2}\}(1 - \psi_i)(\Delta\hat{\mathbf{v}}_i.\mathbf{S}_i)\Delta\mathbf{v}_i \tag{11}$$

with $\bar{c}_2 = c_2(\gamma + 1)/4$ and usually $c_1 = c_2 = 1$. The velocity difference is $\Delta\mathbf{v}_i = \mathbf{v}_i - \mathbf{v}_{i+1}$ and the edge force is only applied if $\Delta\hat{\mathbf{v}}_i.\mathbf{S}_i < 0$, i.e. cell edge compression. In these expressions the median mesh vector $\mathbf{S}_i$ is normal to the median mesh, points anti-clockwise and has magnitude of the distance from the centre of the cell to the edge i. This viscous force is applied to node i and $-\mathbf{f}_i$ is applied to cell $i + 1$. Thus the viscous force applied to node 1, due to the edge viscosity, is $\mathbf{f}_1^{visc} = \mathbf{f}_1 - \mathbf{f}_4$ from this cell. There will be 4 similar contributions to the total viscous force on this node from srounding cells. The compatible heating in the cell due to shock viscosity on its edges is

$$M_z \frac{de}{dt} = -\sum_{i=1}^{4} \mathbf{f}_i.\Delta\mathbf{v}_i \tag{12}$$

The function ψ_i acts as a limiter to turn off viscosity in smooth regions of flow. It is defined through

$$\psi_i = \max\left\{0, \min\left(\frac{r_{Li} + r_{ri}}{2}, 2r_{Li}, 2r_{ri}\right)\right\} \tag{13}$$

with

$$r_{Li} = \frac{\Delta\mathbf{v}_a.\Delta\hat{\mathbf{v}}_i}{\Delta\mathbf{x}_a.\Delta\hat{\mathbf{x}}_i}\frac{|\Delta\mathbf{x}_i|}{|\Delta\mathbf{v}_i|} \quad ; \quad r_{ri} = \frac{\Delta\mathbf{v}_b.\Delta\hat{\mathbf{v}}_i}{\Delta\mathbf{x}_b.\Delta\hat{\mathbf{x}}_i}\frac{|\Delta\mathbf{x}_i|}{|\Delta\mathbf{v}_i|} \tag{14}$$

Here the indexing $\Delta\mathbf{x}_a$ differs from that in Reference [5] as these differences are taken along an index line, not around cell edges. For example for edge 1 $\Delta\mathbf{v}_1 = \mathbf{v}_1 - \mathbf{v}_2$ where the subscripts match the node labelling scheme in Figure 1. If this is cell (ix, iy) then $\Delta\mathbf{v}_a$ is the same difference but for cell $(ix + 1, iy)$ and $\Delta\mathbf{v}_a$ that for cell $(ix - 1, iy)$.

When this viscosity is applied to MHD problems we simply replace the sound speed by an effective fast speed c_f defined through $c_f^2 = c_s^2 + v_A^2$ where v_A is the local Alfvén speed. Often for MHD problems it is better to turn off the $\Delta\hat{\mathbf{v}}_i.\mathbf{S}_i < 0$ and always apply the viscosity on expanding cells as well. There is no good justification for doing this but it does always seem to give better results! If you want to do this it is controlled through the Makefile.

2.2 Boundary Conditions

Boundary conditions can be set in `control.f90`. The preset routines are `periodic`, `open` and `user`. If you specify `user` then you must be sure that what is in `boundary.f90` matches precisely your imposed conditions - take care in getting the grid staggering right. Open boundary works on a simplified far-field characteristic model. For this it is best to test on your problem, if fine then go ahead and use. However if these do not work for you problem, they are not guaranteed to work well for all conditions, then you may have to resort to coding up your own conditions in `boundary.f90` and using `user` as your choice for boundaries.

All choices of boundary also allow damping to be applied near the boundary. This is entirely user configurable and requires that the appropriate boundaries are treated as you prefer in `damp_boundaries` inside `boundary.f90`.

3 Hall MHD

The 2D version *Lare2d* comes with an input deck option to run using Hall MHD. In Hall MHD the only difference in the basic equations is that Ohm's law changes from

$$\mathbf{E} + \mathbf{v} \times \mathbf{B} = \eta \mathbf{j}$$

to

$$\mathbf{E} + \mathbf{v} \times \mathbf{B} - \frac{1}{n_e e}\mathbf{j} \times \mathbf{B} = \eta \mathbf{j}$$

where n_e is the electron number density. This can also be written as

$$\mathbf{E} + \mathbf{v_e} \times \mathbf{B} = \eta \mathbf{j}$$

where v_e is the electron fluid velocity. This seemingly minor change has a dramatic effect on running *Lare2d* due to the dispersion relation for Whistler waves. As a result experience has shown that it is usually best to not have a stretched grid for these simulations and keep the computational cells as square as possible. An example of the application of Hall MHD can be found in [10]. Introducing the Hall term also brings in another dimensionless parameter. This can be seen from making the Hall Ohm's law dimensionless

$$\hat{\mathbf{E}} + \hat{\mathbf{v}} \times \hat{\mathbf{B}} - \frac{m_i B_0}{\mu_0 L_0 e v_0 \rho_0} \frac{\hat{\mathbf{j}} \times \hat{\mathbf{B}}}{\hat{\rho}} = \hat{\eta}\hat{\mathbf{j}}$$

The new parameter in front of the Hall term is the ratio of the characteristic ion inertial depth λ_i to the normalising scalelength L_0. The ion inertial depth can be written in a number of ways, e.g.

$$\lambda_i = \frac{m_i B_0}{\mu_0 e v_0 \rho_0} = \left(\frac{m_i^2}{\rho_0 \mu_0 e^2}\right)^{1/2} = \frac{m_i v_0}{e B_0} = \frac{c}{\omega_{pi}}$$

where c is the speed of light and

$$\omega_{pi} = \left(\frac{n_0 e^2}{m_i \epsilon_0}\right)^{1/2}$$

is the ion plasma frequency. Once the density is normalised, through the mass density, n_0 is known and thus λ_i is not a free parameter. It is however treated as a free parameter in `control.f90` as it is often useful to inflate λ_i, much as one does for the resistivity, for computational reasons. As a result care must be taken when choosing λ_i. In the code you must specify `lambda_i` which is λ_i/L.

If Hall MHD is turned on in `control.f90` then `lambda_i` must be specified inside the initial conditions. Note that like gravity `lambda_i` is a function of space. This is important as there are occasions, such as reflecting wall boundary conditions, when you may need `lambda_i = 0` on the boundary. Note also that `lambda_i` is defined at cell vertices and you need to specify all values in the range `lambda_i(0:nx, 0:ny)`.

4 Partial Ionisation and Cowling Resistivity

Lare3d has an option to include partial ionisation. The equations this solves are documented in [8]. All of this physics is included in the file `neutral.f90`. Key to these routines is the calculation of the neutral fraction ξ_n which is the ratio of the mass density of the neutral gas to the total mass density. For now let's assume that ξ_n is known. ξ_n affects the basic equations in a number of ways. At the simplest level it changes the partial pressures so that now the equation of state is different to that of ideal MHD. There are three options for the equations of state determined by setting `eos_number` in `control.f90`

- `eos_number=EOS_IDEAL`: These have been explained above and are repeated here for easy comparison

$$\begin{aligned} P &= \frac{\rho k_B T}{\mu_m} \\ \epsilon &= \frac{P}{\rho(\gamma-1)} \\ \mu_m &= \bar{m}\hat{\mu}_m \end{aligned}$$

  The reduced mass is controlled through the logical variable `neutral_gas` in `control.f90`. If `neutral_gas = .TRUE.` then $\hat{\mu}_m = 1$, if `neutral_gas = .FALSE.` then $\hat{\mu}_m = 0.5$. Thus setting $\hat{\mu}_m = 1$ gives un-ionised atomic hydrogen while $\hat{\mu}_m = 0.5$ will give fully ionised hydrogen. Setting the value of $\hat{\mu}_m$ is handled in `lareXd.f90`. Note the flag `neutral_gas` is only used for `eos_number=EOS_IDEAL`.

- `eos_number=EOS_PI`: Here the effect of neutrals on the partial pressures is included

$$\begin{aligned} P &= \frac{\rho k_B T}{\mu_m} \\ \epsilon &= \frac{P}{\rho(\gamma-1)} \\ \mu_m &= \frac{\bar{m}}{2-\xi_n} \end{aligned}$$

- `eos_number=EOS_ION`: This includes the full partial pressures and the ionisation energy.

$$\begin{aligned} P &= \frac{\rho k_B T}{\mu_m} \\ \epsilon &= \frac{P}{\rho(\gamma-1)} + (1-\xi_n)\frac{X_i}{\bar{m}} \\ \mu_m &= \frac{\bar{m}}{2-\xi_n} \end{aligned}$$

where X_i is the ionisation energy for hydrogen.

Note that we always assume an ideal gas so the first equation is the same for all options. For an ideal plasma we assume a single ionisation state so that the reduced mass is half the ion mass. If including the effect of neutrals fully one should always use `eos_number=EOS_ION` as the ionisation energy of hydrogen must be taken into account. The second option is therefore never formally correct. It is however retained as the energy equation used in MHD is often missing important sources and sinks and therefore having the option to include this term is sometimes useful for sensitivity studies. This is explained in [8].

All of the above assumes that ξ_n is known. This is specified in `neutral.f90`. In this version of the code ξ_n is determined from a modified Saha equation based on a three level hydrogen model [11]. This is therefore only approximately valid for an averaged quiet solar chromosphere and dubious elsewhere. To change this the routines that need to return, or calculate, a different ξ_n are `get_neutral` and `neutral_fraction` in `neutral.f90`.

As well as affecting the partial pressure the neutral fraction can alter the nature of resistive effects. The dominant term here is the introduction of an anisotropic resistivity so that the Ohm's law becomes $\mathbf{E} + \mathbf{v} \times \mathbf{B} = \eta \mathbf{j}_{\|} + \eta_c \mathbf{j}_{\perp}$ where η_c is the Cowling resistivity which may be orders of magnitude larger than the Spitzer value. Once ξ_n is known η_c can be calculated from the standard formula in [8]. This is done in subroutine `perpendicular_resistivity` in `neutral.f90`.

5 Thermal Conduction

Thermal conduction in *LareXd* is based on the Braginskii thermal conduction in the presence of a magnetic field and is of the form

$$\frac{\partial \epsilon}{\partial t} = \nabla . \left(\left(\kappa \frac{\vec{B}}{B^2 + b_{min}^2} . \nabla T \right) \vec{B} \right) + \nabla . \left(\kappa \frac{b_{min}^2}{B^2 + b_{min}^2} \nabla T \right)$$

where $\kappa = \kappa_0 T^{\frac{5}{2}}$. In the limit $b_{min} \to 0$ this recovers the Braginskii parallel thermal conductivity. Finite b_{min} is used to make the conductivity isotropic when $B = 0$. How this is implemented is described in the Appendix. Note that κ_0 in S.I. units is hard coded to 10^{-11} and then made dimensionless using `B0, L0, RHO0` in `control.f90`. If you want to just pick the thermal conductivity, i.e. make it up and generate unphysical results, then you can do this by changing κ_0 which is set in `SUBROUTINE normalise_transport` in `src/core/normalise.f90`. Depending on your problem you may need to change b_{min} which is set as a constant (parameter) in `src/core/conduct.f90`. Thermal conduction is handled via super-stepping [13]. If the number of sub-steps becomes prohibitively large then you may need to either reduce the hydrodynamic time-step or use a code with an implicit solver.

6 Open Boundaries

Open boundaries are now implemented via far-field Riemann characteristics. This calculates the slow, fast, Alfvén etc. speeds and then projects 1D Riemann invariants into ghost cells. If the wave is outgoing values are propagated from the inside the domain into the ghost cells. If the waves are inward propagating then values are carried in from the far-field. This requires a fixed far-field, i.e. specified values of $\rho, \epsilon, v_x, v_y, B_x, B_y, B_z$. So that the open boundary conditions doesn't affect the initial equilibrium these are set to the ghost cell values as defined in the initial conditions. As a result these open boundaries are only accurate if the magnetic field does not change a lot from the initial field. More accurately the far-field specified at $t = 0$ must be a valid far-field later in the simulation. Otherwise the far-field problem for MHD is not well posed.

Notes: In this version these routines have been tested with simple straight fields and x-point fields and they seem robust. Reflection is typically a few percent but can be as large as 10% in extreme cases. It is best to just try for your problem and see if they work. If the open boundaries do fail then the best option is to use a stretched grid with a damping layer. Most of these problems are to do with the open boundary problem for MHD not being well posed so in all these cases you will be conditioning the solution so caution is needed.

The open boundaries have not been tested with resistivity or thermal conductivity. If this causes problems be sure that resistivity is zero at the boundary. Also the open boundaries do not work with gravity. Open boundaries with gravity should be fine on surfaces of constant potential gravitational potential but not always otherwise. At the lower boundary care must be taken to not let the solution fall through the bottom. Experiment to find the best approach. If all fails use damping regions.

Improvements to the open boundary, if possible, will be released through minor revisions and documented in the open boundary routine source code.

7 Alfvén speed limiter

The file `control.f90` now includes an option to turn on a limiter which reduces the net force used in the Lagrangian step if the Alfvén speed exceeeds some user specified value `va_max`. The background to this is most clearly explained in [15]. There is an option to only apply the limiter to the $\mathbf{j} \times \mathbf{B}$ term. The velocity used to trigger this is in *LareXd* internal units. If you use these flags experiment first to get a feel for how they work and read [15].

8 The Grid

In order to define the initial conditions and understand the location of output arrays in physical space it is vital that all users know the underlying structure of the grid. This is best explained by stepping up through 1D then 2D and finally 3D computational cells. In 1D there are nx computational cells. These are labelled from $ix = 1$ up to $ix = nx$. The variables used in the code are not defined at the same point in a cell. We therefore need to be able to access the location of different parts of the grid to set up the initial conditions. For this xb_i is the position of the right hand boundary of a cell and xc_i the position of the cell centre. This means that the left hand boundary of the computational domain is at xb_0. The width of each cell is dxb_i and the distance between cell centres is dxc_i. In this coordinate system the velocities are all defined at the cell boundaries and all scalars (density, pressure, s pecific internal energy density) are defined at cell centres. This is shown in figure 8. The magnetic field components are defined at different locations for each component. B_x is defined at the cell boundary (xb_i) while the B_y and B_z components are cell centred. Note that this staggering is essential for the accuracy, and conservation properties, of *LareXd*.

One side effect of this staggering that can cause confusion is that to define the initial conditions inside the computational domain more points are needed for velocities and B_x than for the scalars and other magnetic field components. This is because you must define, for example, v_x from $ix = 0$ to $ix = nx$ but only need the density from $ix = 1$ to $ix = nx$. Note that the grid spacing need not be uniform so that dxb_i need not equal dxb_{i+1}. The default setup of the grid is uniform but if the flag `x_stretch=.TRUE.` is set in `control.f90` then the grid will be non-uniform.

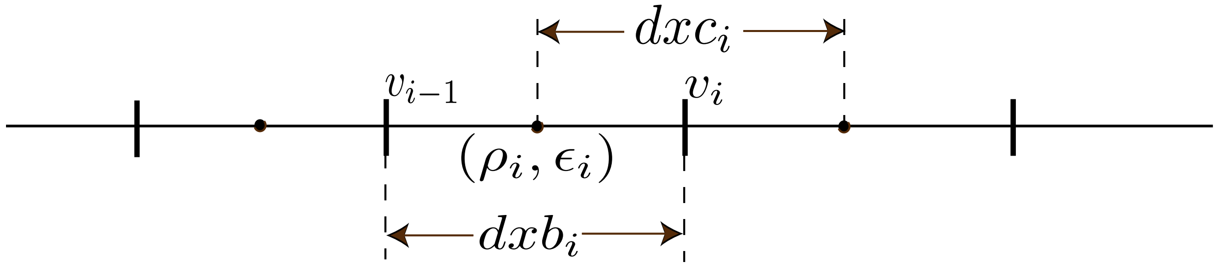

Figure 2: 1D Staggered Grid

In 2D the velocities are defined at cell corners, the scalars and ignorable direction magnetic field at cell centres. The remaining magnetic field components are defined at cell edges. This is shown in figure 3. When analysing output it is OK to define the kinetic energy density as $0.5\rho_{ij}v_{ij}^2$ but the accurate estimate of kinetic energy should make sure that these variables are defined at the same place. In figure 3 the mass density at the vertex (ρ_{ij}^v) is also shown and once defined the kinetic energy density is then $0.5\rho_{ij}^v v_{ij}^2$. To get the vertex density you must average the densities in the four surrounding cells. Since the grid is possibly stretched care should be taken with these averages and it is best to define the vertex density by summing up the total mass in the surrounding four cells and dividing by the total area. This sort of thing is done a lot inside the code but in most cases this isn't important to users of the code. It is flagged here just as a warning for calculating products of output variables which are not defined at the same location. The Hall term is especially sensitive to this and great care should be taken calculating derived quantities on the staggered grid if doing Hall MHD work.

Once you have grasped the 2D grid then 3D is obvious. This is shown in figure 4. Here scalars are volume centred, velocities are defined at cell vertices and magnetic field components are defined as face centred. Gravity is a function only of the z coordinate and is define at the cell vertices, i.e. same location as the velocity.

8.1 Stretched Grids

You can choose to stretch the grid. If you do this the grid can be stretched differently in each of the three directions. The routines for defining the stretched grids are in `setup.F90` and are imaginatively called `stretch_x`, `stretch_y`, and `stretch_z`. Taking `stretch_x` as an example, when this is called the boundaries of the cells xb_i are defined as uniformly spaced from `x_min` to `x_max` as defined in `control.f90`. For reasons to do with the MPI parallelism these are called `xb_global` at this point. The subroutine `stretch_x` is free to redistribute these boundaries however you like but note that if the routine changes the overall size of the domain then you should reset the variable `length_x`. Here's an example of a stretching function

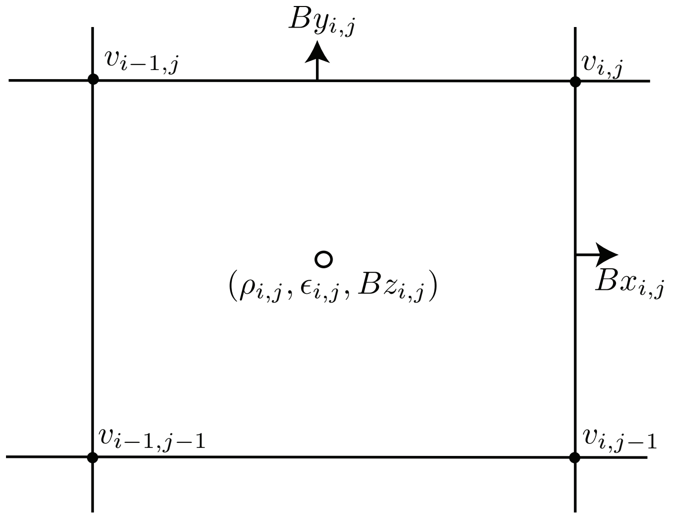

Figure 3: 2D Staggered Grid

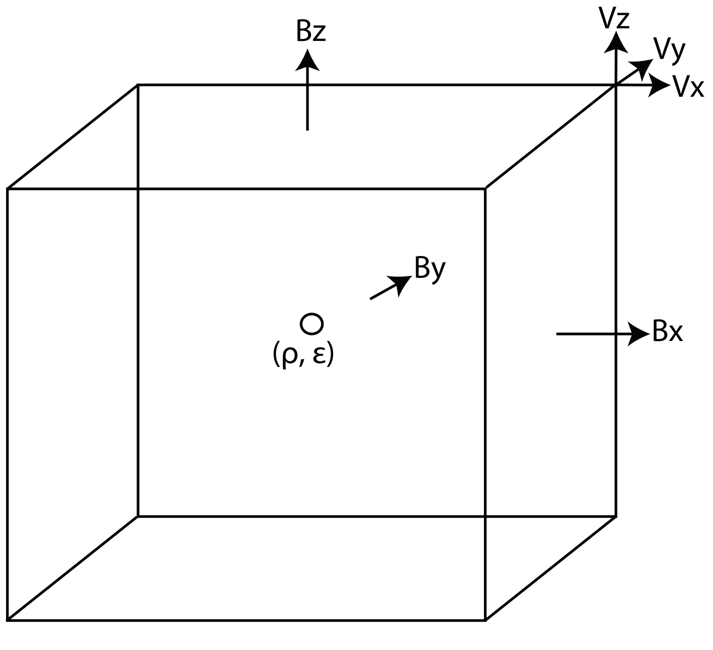

Figure 4: 3D Staggered Grid

```
  SUBROUTINE stretch_x
  ! replace with any stretching algorithm as needed
   REAL(num) :: width, dx, L, f, lx_new
   lx_new = 200.0_num
   ! new total length
   L = length_x / 2.0_num
   ! centre of tanh stretching in unstretched coordinates
   width = length_x / 10.0_num
   ! width of tanh stretching in unstretched coordinates
   f = (lx_new - length_x)/(length_x - L)/2.0_num
   dx = length_x / REAL(nx_global,num)
   dxnew = dx+f*(1.0_num+TANH((ABS(xb_global)-L)/width))*dx
   DO ix = 1, nx_global+2
      xb_global(ix) = xb_global(ix-1) + dxnew(ix)
   ENDDO
   length_x = xb_global(nx) - xb_global(0)
 END SUBROUTINE stretch_x
```

Don't ask me why this function is written this way, I can't remember. All that you need to know is what you want to do and then you're on your own. Good luck! One word of advice: if you stretch the grid so that the cells are far from square or the dxb_i's change by more than 10% from cell to cell then you cannot expect an accurate solution in that region of the computational domain. Do not use a stretched grid with the Hall term as the dispersion relation for Whistler waves means it is very likely that grid scale noise will develop and reflect back off the stretching. As a rule only stretch the grid if you absolutely must and don't overdo the stretching.

9 Code Structure

The code is all stored in the `src` sub-directory. This contains two sub-directories `core, Old`. The files have either a `.f90` or `.F90` extension. The ones with `.F90` are those which contain pre-compiler directives. These are explained below. A brief description of each of these files is set out below, in alphabetical order, for *Lare2d*.

First the files in the highest source code directory `src`. These are the files you may have to setup for each new simulation. For most users you probably won't need need to edit anything else.

- `boundary.f90:` Contains the boundary conditions not set automatically by MPI. If you set any boundary condition to `BC_USER` in `control.f90` (see below) then here is where you must set the boundary conditions you want.

- `control.f90:` This is subdivided into three subroutines: `User_Normalisation` sets the parameters used in the code normalisation, `Control_Variables` sets basic code parameters (see the comments in the code for a full description) and `Set_Output_Dumps` sets information to be included in the output dumps from the code.

- `diagnostics.F90:` Controls the code output, sets the timestep and calculates some diagnostics such as kinetic energy etc. as a function of time.

- `initial_conditions.f90`: The initial conditions!
- `radiative.f90`: Contains the setup of the optically thin radiative losses and a user specified heating function. How to change these is documented in the code itself.

All of the core routines for solving the equations are hidden away inside a `core` sub-directory and in most cases you shouldn't need to touch these files. In case you are curious this is what they all do. When there is a key subroutine that is more likely to need user intervention this will be highlighted.

- `conduct.f90`: Parallel thermal conduction using the standard Braginskii formula. If $B = 0$ then resorts to isotropic conduction. The formula assumes $\ln \Lambda = 18.4$ for no other reason than to make the thermal conductivity $10^{-11} T^{5/2}$ in S.I. units. This module also contains examples for radiative losses and heating functions. The default radiative losses are RTV but these are easy to change.
- `lagran.F90`: All routines associated with the lagrangian step. If you want to change the resistivity it is inside this file in a subroutine called `eta_calc`.
- `lare2d.f90`: The main control routine.
- `mpi_routines.f90`: Setting up the MPI environment and defining area sizes etc.
- `mpiboundary.f90`: All the MPI communication.
- `neutral.f90`: All routines to do with including neutrals, e.g. calculating the neutral fraction and the Cowling resistivity. The calculation of the neutral fraction is based on the modified Saha equation and is only valid for the solar chromosphere.
- `normalise.f90`: Used to normalise transport and physics packages.
- `openboundaries.f90`: The open boundary routines based on farfield characteristics. Anyone who tells you they have an MHD open boundary condition that will work in all circumstances is making it up! Open boundaries for MHD are not well posed problems so use common sense and caution.
- `remap.f90`: Control routine for the x,y,z remaps.
- `setup.F90`: Defines the grid, opens files and gets things ready.
- `shared_data.F90`: Defines all variables that are shared globally.
- `welcome.f90`: A pointless welcome message to make me laugh.
- `x,y,zremap.f90`: The remap routines in each direction.

The `Old` directory contains examples of initial conditions that may be useful to understanding how to set things up. Any examples you have that you feel would be useful to others, or even as a reminder to yourself, can be emailed to Warwick for inclusion. Note that some of these old conditions do not work with the latest *LareXd* as parameters etc. may have changed. Always check before using.

Modules are collections of routines that naturally belong together either because they share common data or actions, e.g. all the lagrangian step routines are grouped together in `lagran.F90`. The headers to all modules are similar. For example the header for `lagran.F90` is

```
MODULE lagran

  USE shared_data
  USE boundary
  USE neutral
  USE conduct

  IMPLICIT NONE

  PRIVATE

  PUBLIC :: lagrangian_step, eta_calc

  ! Only used inside lagran.f90
  REAL(num), DIMENSION(:,:), ALLOCATABLE :: qxy, qxz, qyz
  REAL(num), DIMENSION(:,:), ALLOCATABLE :: qxx, qyy, visc_heat, pressure
  REAL(num), DIMENSION(:,:), ALLOCATABLE :: flux_x, flux_y, flux_z, curlb

CONTAINS
```

The lines with `USE` list all the other modules that are required by this module. `IMPLICIT NONE` means that all variables must be declared, failure to do this will give an error message. Thinking this is a bad idea or a nuisance is a sign of madness - just do it! `PRIVATE` means all the routines below the `CONTAINS` line are private to this module and cannot be called from another module. The exceptions to this are routines listed in the `PUBLIC` line which can be called by other modules. The three lines of type declarations define variables that are global to all routines in this module but invisible to other modules. Below the `CONTAINS` is where all the subroutines go. Note that reals are not defined just as `REAL` but as `REAL(num)`. This means all reals are of type `num` which is defined in `shared_data.F90:`, the top of which is

```
MODULE constants

  IMPLICIT NONE

#ifdef SINGLE
  INTEGER, PARAMETER :: num = KIND(1.0)
  INTEGER, PARAMETER :: num_sz = 4
#else
  INTEGER, PARAMETER :: num = KIND(1.D0)
  INTEGER, PARAMETER :: num_sz = 8
#endif
  INTEGER, PARAMETER :: dbl = KIND(1.D0)

  ! Code dimensions
  INTEGER, PARAMETER :: c_ndims = 2

  REAL(num), PARAMETER :: pi = 3.14159265358979323846264338327950З_num
```

Here we see that `num` is the same kind of number as 1.D0. This means that everything defined such as 1.234_num will be double precision, just as the definition of π (pi) is above. The exception to this is if `DEFINES += $(D)SINGLE` is set in the Makefile in which case the code uses single precision. The advantage of this is only one line needs changing in the Makefile to change from single to double precision. Many people think that c ompilers can take care of this. My experience is that this isn't true and the above procedure guarantees you will get double precision when you want it by only changing one line of the Makefile (where `DEFINES += $(D)SINGLE` is set).

All files with an `F90` extension will first be passed through a pre-compiler. This essentially checks flags set in the Makefile and based on this either includes or removes lines of source code before compiling. This is a bit messy but is worth the trouble. For example inside a big loop you do not want to keep checking conditionals which do not change. These pre-compiler options are always of the form `ifdef ...endif` or `ifndef...endif` to either include or exclude lines of code. Doing things this way just makes the code faster.

10 Setting up the Domain and Control Variables

The control of the number of grid points, domain size and all other initialisation parameters is set in `control.f90`. Since this routine must be edited for each new simulation the documentation for each of the control variables is included in the source code. Careful reading of this manual and those comments ought to be all that is needed to fully understand how to setup and run *LareXd*.

11 Initial Conditions

All that remains to get the code ready to run is to fill the primary arrays. This is done in the file `initial_conditions.f90`. To be complete the 3D initial conditions need to specify the variables everywhere inside the computational domain. Due to the staggered grid these arrays are not all the same size so that the computational domain is defined by

- `rho(1:nx,1:ny,1:nz)`
- `energy(1:nx,1:ny,1:nz)`
- `vx(0:nx,0:ny,0:nz)`
- `vy(0:nx,0:ny,0:nz)`
- `vz(0:nx,0:ny,0:nz)`
- `bx(0:nx,1:ny,1:nz)`
- `by(1:nx,0:ny,1:nz)`
- `bz(1:nx,1:ny,0:nz)`
- `grav(0:nz)`

The simplest possible initial condition is therefore something like

```
SUBROUTINE set_initial_conditions

  vx = 0.0_num
  vy = 0.0_num
  vz = 0.0_num
  bx = 0.0_num
  by = 0.0_num
  bz = 0.0_num
  energy = 1.0_num
  rho = 1.0_num

  grav = 0.0_num

END SUBROUTINE set_initial_conditions
```

Because the code is an MPI code the different processes need to be able to swap information. This is done by each process having a set of 'ghost' cells defined around the real cells. These act as a buffer for the exchange of boundary information through MPI and are how the real boundary conditions are applied. In order to cut down on the number of times that processes have to communicate there are two ghost cells in this buffer zone. As a result the initial conditions should set

- `rho(-1:nx+2,-1:ny+2,-1:nz+2)`
- `energy(-1:nx+2,-1:ny+2,-1:nz+2)`
- `vx(-2:nx+2,-2:ny+2,-2:nz+2)`
- `vy(-2:nx+2,-2:ny+2,-2:nz+2)`
- `vz(-2:nx+2,-2:ny+2,-2:nz+2)`
- `bx(-2:nx+2,-1:ny+2,-1:nz+2)`
- `by(-1:nx+2,-2:ny+2,-1:nz+2)`
- `bz(-1:nx+2,-1:ny+2,-2:nz+2)`
- `grav(0:nz)`

Note that the gravity array has not changed as this is not updated and therefore never needs to be communicated between processes. Pay particular attention to the use, for example, of `xc(ix)` and `xb(ix)`, as different locations are specified.

If Hall MHD is used then you must also specify `lambda_i(0:nz)` in the initial conditions.

12 Probe points

Probe points can also be added to the run by adding any number of lines of the form
`CALL add_probe(xprobe, yprobe)`
where `(xprobe,yprobe)` are the (x,y) coordinates of a point in the simulation domain. Once this is

include the code will output the (x,y) coordinates of the probe point, the velocity and magnetic field components ate that point on each timestep. More variables can be added by editing the `write_probes` routine in `diagnostics.F90`. This data can be read in IDL using `getprobe` which has the same syntax as `getenergy`, see later.

13 Makefile

The code is a Fortran 90 code and I think conforms with the standard so should work on all compilers. Since the code is also parallel you must have MPI installed to compile and run the code, even if you are only going to run on a single processor machine. To compile *LareXd*, simply use the suplied Makefile. Specify a particular compiler with, for example
`make COMPILER=pgi`
Specify debugging flags with
`make MODE=debug`
Alternatively, these options can be specified as environment variables eg. `export COMPILER=gfortran` can be added to `$HOME/.bash_profile`. This will compile the code using the default supplied compiler flags and generate a binary called
`bin/lare`*x*`d`
where *x* is replaced by either 2 or 3. The code can then be run on a workstation using either
`./bin/lare`*x*`d`
or
`mpirun -np` {*nproc*} `./bin/lare`*x*`d`
where {*nproc*} is the number of processors requested. Make sure the data directory you have specified for the output exists before you try to run the code. On a cluster, the code should be run using the normal submission scripts/system for the cluster that you're running the code on, always remembering that the binary file is in the `./bin` subdirectory. The compiler intermediate files go into the `obj` subdirectory.

To remove the compiled binary, and the compiler intermediate files
`make clean`
To compile and install the VisIt plugin, type
`make visit`
This will only work if `visit` is in your PATH. To uninstall the VisIt plugin, type
`make visitclean`
When a new version of VisIt has been installed, type
`make visitclean`
`make visit`
to rebuild the VisIt plugin using the latest version of VisIt. A VisIt plugin built against an old version of the code will not work.

13.1 The standard Makefile

The *LareXd* Makefile is relatively complex and the main working parts should not be changed without a specific reason. However, it is possible to change other parts of the makefile to allow additional features.

```
# Set the compiler flags
```

```
#FFLAGS = -fast
FFLAGS = -O3
```

The `FFLAGS` environment variable is used to control the general command line parameters passed to the Fortran compiler. `-O3` and `-fast` are portable general purpose flags for turning on most optimisations on most machines. `-fast` is usually more agressive than `-O3` and should be removed if any unusual behaviour is noted (see section on debugging).

```
# Set some of the build parameters
TARGET = lare3d
```

`TARGET` controls the name of the output binary file in the `bin` subdirectory. Normally, there is no need to change this.

```
# The following are a list of pre-processor defines which can be added to
# the above line modifying the code behaviour at compile time.

# Uncomment the following line to run with limiters on shock viscosity
#DEFINES += $(D)SHOCKLIMITER

# Uncomment the following line to run in single precision
#DEFINES += $(D)SINGLE

# Uncomment the following line to use first order scheme for resistive update
#DEFINES += $(D)FOURTHORDER
```

The above section controls optional flags which can be passed to the code or compiler. `SINGLE` forces the code to run in single precision. This is much faster and is recommended for preparatory work but the full double precision should always be checked for publications. `FOURTHORDER` runs the resistive routines in fourth order instead of the default first order. Usually first order is fine for diffusive terms but you should check for your problem. In order to allow vector optimisation of the code all `IF` statements were removed from the main loop. This meant that allowing the change between using tensor shock viscosity and simple Wilkins QMONO shock viscosity had to be moved out to the makefile. To run the code with tensor shock viscosity comment this flag out, to run with the Wilkins viscosity uncomment it.

13.2 Further Optimisation

The *LareXd* codes are fairly robust with regard to compiler flags, so you can normally use your favourite optimisations. An example would be

- ifort : -ip -xP (Turn on higher level optimisation)

13.3 Debugging

The *LareXd* codes are also robust with regard to input conditions. Therefore, the most likely cause of a code crash is that the code either has not been given all the data which is needed or you have written over the end of an array when initialising variables. This is most easily checked by adding the `-C` flag into the compiler options of the makefile. This flag is not formally a part of the Fortran 90 standard but is supported by all compilers that have been tested to date. This flag tests for attempts to write outside the bounds of an array and will print the location of the error and the name of the array which causes the problem. Unfortunately the flag turns off all compiler optimisations and so cannot be left on for general use, since optimisation increases the code speed by at least a factor of 3.

14 Common Questions

How do I run several copies of *LareXd* at the same time?

One common use of *LareXd* is to run parameter space studies using a large Beowulf cluster, with several versions of the code in the cluster queue at the same time. This can be achieved by the simply copying the entire *LareXd* directory and compiling and running different versions of the code there. Although this leads to an unnecessary duplication of data, it has the advantage that all the source files used in the run are retained, meaning that it is trivial to determine which run produced the data. An alternative, and the one I use, is to rename the binary generated after compilation, e.g. rename `./bin/lare2d` to `./bin/lare2d1`, then run the renamed code. Be sure that each instance of the binary writes to a different output directory otherwise you will overwrite data from multiple runs.

How do I delete old data that I no longer want?

Unlike earlier versions of *LareXd* and many other codes, the current version does not delete the data when the command
`make clean`
is executed. In order to delete the data, simply type
`rm -rf {Data_Dir}`
where `{Data_Dir}` should be replaced by the name of the directory holding the data that you no longer want.

How do I restart from an old snapshot?

You can specify to restart from any previous snapshot by setting `initial = IC_RESTART` in `control.f90` and then specifying the restart file number with `restart_snapshot`. In previous versions of *LareXd*, additional work was required to maintain the energy diagnostics in the file en.dat. This has now been fixed so that the code correctly appends to en.dat after a restart.

15 Output

15.1 IDL

The IDL routines used for loading the data are all contained within the SDF/IDL and IDL directories. They perform the following functions:

`getdata:` This is a function which returns a structure containing the snapshot data.

`getenergy:` This is a function which returns a structure containing the energy diagnostic data.

The *LareXd* IDL functions and subroutines make use of IDL structures. This has the advantage over just loading the separate arrays in that it allows each snapshot to be stored and passed between functions as one entity which is a lot neater as it removes the need for common blocks. Section 15.1.4 gives some advice for dealing with structures if you've not used them before.

15.1.1 Starting IDL

In order to load in the *LareXd* IDL routines the IDL script "Start.pro" needs to be run. This can either be run from the command line:

```
$ idl Start.pro
```

Or from within IDL using

```
IDL> @Start
```

There is only one script for loading in both the single and double precision data. The `getdata` function will automatically detect the precision and return a structure with the correct precision.

15.1.2 `getdata`

This function loads the snapshot data. The syntax is:

```
IDL>  ds = getdata(snapshot[,/var1,...,wkdir=directory])
```

After running this `ds` will contain the snapshot data structure. `snapshot` is the snapshot number to load, just providing the snapshot number will load all the variable arrays in from the default data directory "Data".

It is possible to just load a selection of the variables rather than all of them, this can be useful with large data files as it saves memory and is quicker. The available variables are listed in Table 1, they are loaded using `/variablename`. There are also two flags which load the vector quantities: `/fields` l oads the three magnetic field components (`bx,by,bz`), `/vel` loads the three velocity components (`vx,vy,vz`). As many variables as required can be listed after the snapshot number, see the examples below.

The `wkdir=` option allows data to be loaded from a directory other than "Data", `directory` should be replaced with a string containing the name of the directory.

Some examples:

```
IDL>  ds = getdata(10)
```

This is the minimum required to load the data, this will load all the variables from snapshot 10 in the default data directory "Data".

```
IDL>  ds = getdata(7,/rho,wkdir="OtherDir")
```

This will load just the density from snapshot 7 in the data directory "OtherDir".

```
IDL>  ds = getdata(0,/rho,/temperature,/magnetic_field)
```

This will load the density, the temperature and the three components of the magnetic field from snapshot 0 (the initial conditions) in the default directory "Data".

To list the contents of a file without loading the data, you can use the `list_variables` routine. The syntax is:

```
IDL>  list_variables,snapshot[,directory])
```

Header variables:
These are all included by default and cannot be individually selected.

filename	String containing the filename of the snapshot file the data was loaded from.
timestep	The simulation time-step that the snapshot was written.
time	The simulation time that the snapshot was written.

Simulation variables:
To load only one variable use `/variable`. Where `variable` is the name of the variable. The name of some common variables is given in the left column.

x	Array containing the cell-centre x coordinates.
y	Array containing the cell-centre y coordinates.
z	(3D only) Array containing the cell-centre z coordinates.
grid	A structure containing information about the grid.
grid.x	Array containing the cell boundary x coordinates.
grid.y	Array containing the cell boundary y coordinates.
grid.z	(3D only) Array containing the cell boundary z coordinates.
rho	Array containing the density. Also loaded by `/fluid`
energy	Array containing the internal energy. Also loaded by `/fluid`
temperature	Array containing the temperature. Also loaded by `/fluid`
pressure	Array containing the pressure. Also loaded by `/fluid`
cs	Array containing the sound-speed. Also loaded by `/fluid`
vx	Array containing the x component of velocity. Also loaded by `/velocity`.
vy	Array containing the y component of velocity. Also loaded by `/velocity`.
vz	Array containing the z component of velocity. Also loaded by `/velocity`.
bx	Array containing the x component of the magnetic field. Also loaded by `/magnetic_field`.
by	Array containing the y component of the magnetic field. Also loaded by `/magnetic_field`.
bz	Array containing the z component of the magnetic field. Also loaded by `/magnetic_field`.
j_par	Array containing the parallel current. Also loaded by `/pip`.
j_perp	Array containing the perpendicular current. Also loaded by `/pip`.
neutral_fraction	Array containing the neutral fraction. Also loaded by `/pip`.
eta_perp	Array containing the perpendicular resistivity. Also loaded by `/pip`.
eta	Array containing the resistivity. Also loaded by `/pip`.

Table 1: List of all the elements of the data structure and a brief description.

15.1.3 `getenergy`

The `getenergy` function has the following syntax:

```
IDL> en = getenergy([wkdir=directory])
```

This will set `en` equal to a structure containing all the energy diagnostics data. There is only one optional argument to this function: `wkdir=` is used to change the directory from which the diagnostics are to be loaded, if no arguments are supplied (i.e. `en = getenergy()`) the default "Data" directory is used. Table 2 lists the elements of the energy diagnostic structure.

points	The number of data points in the diagnostic arrays. All the following arrays are of this size.
time	Array containing the time of each data point.
en_b	Array containing the total magnetic energy.
en_ke	Array containing the total kinetic energy.
en_int	Array containing the total internal energy.
heating_visc	Array containing the total viscous heating as a running total.
heating_ohmic	Array containing the total ohmic heating as a running total.

Table 2: List of all the elements of the energy diagnostics structure.

15.1.4 IDL Structure Tips

This section gives a brief overview of how to use IDL structures. The easiest way of doing this is with an example, to create the data structure:

```
IDL>  ds = getdata(0)
```

To view the elements contained in a structure the `help` command can be used with the `/struct` flag:

```
IDL> help, ds, /struct
** Structure <2015808>, 19 tags, length=3219368, data length=3219364, refs=1:
   FILENAME        STRING    'Data/0000.sdf'
   TIMESTEP        LONG                 0
   TIME            DOUBLE        0.021560115
   DT              DOUBLE        0.021560115
   TIME_PREV       DOUBLE          0.0000000
   VISC_HEATING    DOUBLE          0.0000000
   GRID            STRUCT    -> <Anonymous> Array[1]
   X               DOUBLE    Array[200]
   Y               DOUBLE    Array[200]
   RHO             DOUBLE    Array[200, 200]
   ENERGY          DOUBLE    Array[200, 200]
   VX              DOUBLE    Array[201, 201]
   VY              DOUBLE    Array[201, 201]
   VZ              DOUBLE    Array[201, 201]
   BX              DOUBLE    Array[201, 200]
```

```
    BY                DOUBLE     Array[200, 201]
    BZ                DOUBLE     Array[200, 200]
    TEMPERATURE       DOUBLE     Array[200, 200]
    PRESSURE          DOUBLE     Array[200, 200]
```

Ignore the first line. The rest of the lines are the elements, the variable type, if the variable is an array it gives the array dimensions otherwise it gives the value.

To access the elements use `datastructure.element`, for example to print the filename:

```
IDL> print, ds.filename
Data/0000.sdf
```

Or to contour the temperature:

```
IDL> contour, reform(ds.temperature[40,*,*]), ds.y, ds.z
```

Because the `getdata` function returns the structure a more advanced use is to use the `getdata` function as the data structure, for example if you want to store `vx` slices as a function of time over 10 snapshots:

```
IDL> v = dblarr(81,81,11)
IDL> FOR i=0,10 DO BEGIN v[*,*,i] = (getdata(i,/vx)).vx[*,*,40]
```

Just remember to put the extra brackets round `getdata`!

15.2 VisIt

15.2.1 Installation

VisIt can be downloaded from [2]. This site lists the latest version (2.x) as well as older versions. Installation is a case of extracting the tar ball somewhere convenient. It will create a “visit” directory and place all the files in there. You then need to include the directory “visit/bin” in your path. Once that’s been done type “visit” in the *LareXd* directory to both confirm that it has been set up correctly and also to create a “.visit” directory in your home directory where the plug-in will be installed.

To create the *LareXd* plug-in type

```
$ make visit
```

This will build and install the plug-in.

15.2.2 My First Plot

Data is loaded into VisIt by clicking on “File/Open File...”, in the file window select the *.llld database. To select the snapshot to view use the slider underneath the “Selected files”.

Plots are created by clicking on the word “Plots”. This will open up a menu to select the type of plot required. As an example, using *Lare3d*, once the data is loaded in: clicking on “Pseudocolor/Fluid/Energy”, then click on the “Draw” button. This will create an image in the viewer window. Clicking and dragging on the viewer window ill rotate the cube round. To reset the view click on the button with the camera and the green cross at the top.

At this point the plot needs to be modified with the operators. Selecting “Operators/ThreeSlice” will draw pseudocolor plots on three planes. To adjust the parameters of the plot double click on the name of the plot or the operator in the “Active Plots” list. To change the variable plotted, click on “Variable” while the plot is highlighted and it will redraw the plot using the new variable.

From now on the easiest way to learn how to use VisIt is to experiment.

APPENDIX I: Code details

Thermal Conduction

Thermal conduction in Lare is based on the Braginskii thermal conduction in the presence of a magnetic field and is of the form

$$\rho \frac{\partial \epsilon}{\partial t} = \nabla . \left(\vec{k}.\nabla T \vec{n} \right) + \nabla . \left(\frac{b_{min}^2}{B^2 + b_{min}^2} \kappa \nabla T \right)$$

where $\vec{k} = \kappa \vec{n}$, $\vec{n} = \vec{B}/(B^2 + b_{min}^2)^{1/2}$ and $\kappa = \kappa_0 T^{\frac{5}{2}}$. In the limit $b_{min} \to 0$ this recovers the Braginskii parallel thermal conductivity. The notes below cover the most obscure elements of the notation used in the code but should make sense when read in conjunction with the source code in `src/core/conduct.f90`. It is presented here as these routines are not documented in any of the references.

Firstly the conduction is written in terms of the heat flux vector $\vec{q}$ so that

$$\vec{q} = \left(\vec{k}.\nabla T \right) \vec{n} + \frac{b_{min}^2}{B^2 + b_{min}^2} \kappa \nabla T$$

Then the specific energy density in each cell is updated using super-stepping [13]

It is possible that sufficient heating occurs so that qx exceeds the free streaming heat flux $q_f = v_{the} k_B T$. In this case the Braginskii flux needs to be limited. This is done by calculating the components of the Braginskii (Spitzer-Harn) heat flux, e.g. qsh_x and the free streaming limit q_f and then finding the non-linear limited flux through

$$q_{nl} = \frac{1}{(1/q_s h + 1/q_f)}$$

Experience from measured heat fluxes in laser plasma experiments suggest that actually the heat flux must be limited not to $q_f = v_{the} k_B T$ but to $q_f = F_L v_{the} k_B T$ where F_L is a flux limiter which is usually set to 0.05. The parameter F_L is set in the input `control.f90` file as `flux_limiter`. If flux limiters are unfamiliar you should read papers from the laser plasma community where these are measured and used in codes.

Optically thin radiative losses

Optically thin radiative losses are included by using the Townsend exact intergration method [14]. This scheme works by directly updating the temperature. Thus in *LareXd* the dimensionless specific energy density is first converted to S.I. units. Then the Townsend scheme updates the temperature whch is finally converted back to dimensionless units. For this to work the user must specify the optically thin radiative loss function. Ignoring all other terms the radiative losses are modelled through

$$\rho \frac{d\epsilon}{dt} = -L_r$$

In the example used in the default code this is RTV loss so that $L_r = n_e^2 \chi T^\alpha$ where χ and α are both functions of temperature. For RTV the values of χ and α are given in table 3 for temperature in MK. The user can change the radiative loss to any loss function which is a piece-wise polynomial of temperature. This is documented in the routine `setup_loss_function` in `radiative.f90`

Table 3: RTV radiation coefficients

T_{MK}	$\log_{10}(\chi)$	α
0.02 – 0.0398	-21.85	0
0.0398 – 0.0794	-31.0	2
0.0794 – 0.251	-21.2	0
0.251 – 0.562	-10.4	-2
0.562 – 1.995	-21.94	0
1.995 – 10	-17.73	-2/3

User specified heating function

`conduct.f90` also contains a user specified routine for inputing heat anywhere in the simulation domain. The example given finds the radiative losses as a function of mass density and temperature first in S.I. and then converts to `LareXd` units. This can be replaced with any user specified prescription.

References

[1] T. D. Arber, A. W. Longbottom, C. L. Gerrard, A. M. and Milne. *J. Comput. Phys.* **171**, 151-181 (2001)

[2] `https://wci.llnl.gov/simulation/computer-codes/visit`

[3] D. J. Benson, *Comput. Methods Appl. Mech. Eng.* **99** 148 (1992)

[4] M. L. Wilkins, *J. Comp. Phys.* **36** 281 (1980)

[5] E. J. Caramana et al., *J. Comp. Phys.* **144** 70 (1998)

[6] B. Van Leer, *J. Comp. Phys.*, **32** 101 (1979) & *J. Comp. Phys.*, **135** 229 (1997). The same paper was published twice! It's that good!

[7] C. R. Evans and J. F. Hawley, *Astrophysics J.* **332** 659 (1988)

[8] T. D. Arber, M. Haynes and J. E. Leake, *Astrophysics J.* **666** 541 (2007)

[9] M. Haynes and T. D. Arber, *Astronomy & Astrophysics* **467** 327 (2007)

[10] T. D. Arber and M. Haynes, *Physics of Plasmas* **13** 112105 (2006)

[11] J. Brown, *Solar Physics* **29** 421 (1973)

[12] NRL Plasma Formulary, J. D. Huba (2002)

[13] C. Meyer, D. Balsara and T. Aslam, *MNRAS* **422** 2101 (2012)

[14] R.H.D. Townsend, *ApJS* **181** 391-397 (2009)

[15] T. Gombosi et al. *J. Comp. Phys.* **177** 176-205 (2002)