Imogen User's Manual

Erik Keever

September 23, 2015

1 Setup And Prerequisites

This section will guide you through the Imogen acquisition and setup process.

The setup process consists in making sure that you have the proper libraries available and informing Imogen of where they are through the Make.config file in the top level directory. The template Make.config.default should be copied to Make.config and then setup per instructions below.

1.1 Download

Imogen lives online at `https://github.com/imogenproject/gpuImogen`. To download it into directory `./imogen` from the public repo,

```
$ git clone https://github.com/imogenproject/gpuImogen.git ./imogen
```

This will fetch the latest commit and is readonly.

1.2 Prerequisites

1.2.1 Filesystem

Invoking Imogen does not explicitly require a shared filesystem, but the same runfile must be present at the same directory for all ranks.

To save data however, Imogen does require a shared filesystem.

FIXME: A non-shared FS is acceptable in principle but one rank per host must create output directories. As it stands, Imogen will spinloop forever waiting for global visibility.

1.2.2 Matlab

Imogen runs inside the Matlab runtime & therefore obviously a working installation of Matlab is required. Imogen does not require any of the extended toolkits.

If

```
$ matlab -nodesktop
```

fails to bring up the usual Matlab command line, something is wrong.

1.2.3 MPI

Whether MPI-parallel processing is used or not, the MPI libraries and headers must be available. Imogen uses the MPI API to check that it is running serially or not.

Setting up an MPI installation larger than one machine is less than entirely trivial, but on any institutional cluster you should not have to since MPI is pretty much the universal standard in parallel distributed-memory programming.

As an extremely basic sanity check, if

```
$ mpirun -np 3 date
```

fails to print the current time 3 times something is definitely broken.

1.2.4 Parallel Gateway

Imogen uses the Parallel Gateway (PGW) library as a go-between with MPI.

Make.config's *PGW_DIR* variable must be set to the install directory containing PGW.

1.2.5 CUDA

Imogen's GPU routines are CUDA. You need both a working runtime, as well as the development headers.

From the SDK code samples, you should be able to build `Samples/1_Utilities/deviceQuery` and have it print out various pleasing information about installed CUDA-capable cards.

Imogen will not be able to run on devices of compute capability under 2.0 for 3 reasons:

- Imogen launches 3D grids for 3D simulations
- Arch 1.x suffers from a crippling lack of shared memory
- Arch 1.x does not support peer memory access or cudaMemcpyPeerToPeer

1.3 Configuration

Imogen comes with a file `Make.config.default` file in the root.

Actual configuration must be in `Make.config`, so copy it there and set variables appropriately. The library directories:

- `MATLAB_DIR` must be the directory containing `bin/matlab`. Matlab likes to make symlinks but the output of `$ ls -als $(which matlab)` should be useful.
- `CUDA_DIR` must be the directory containing `include/cuda.h`. The output of `$ which nvcc | sed -e 's@/bin/nvcc@@'` should be correct.
- `MPI_DIR` must be the directory containing `bin/mpirun`. The output of `$ which mpirun | sed -e 's@/bin/mpirun@@'` should be correct.
- `PGW_DIR`

The compilation settings:

- `CUDA_LDIR` - either 'lib' on 32bit or 'lib64' on 64bit machines
- `NVARCH` - `compute_20` for Fermi, `compute_30` for Kepler
- `NVCODE` - `sm_20` for Fermi, `sm_30` for Kepler

NVARCH and NVCODE are further explained by the nvcc manual page.

1.3.1 Cluster environments / MODULES system

Cluster environments will rarely have a single version of any of these libraries available.

For Imogen to work, it is essential that `the same compiler builds MPI, PGW, and provides the host backend for 'mex' and 'nvcc'` AND THAT `the same MPI, Matlab and PGW libraries are used at compile-time and runtime`

Otherwise, library mismatch errors or bizzare runtime malfunctions are extremely likely to occur.

1.4 Compiliation

Once all the above variables are set, run `make` in both the `mpi/` and `gpuclass/` directories.

The `mpi` directory must build in serial. The `gpuclass/` is happy with -j.

1.5 Cluster access

Imogen invokes itself in parallel with the './imogen cluster ...' format.

Before attempting to fire off Imogen on a cluster, you will with almost 100% certainty need to evaluate Imogen's cluster start process at the bottom of `run/imogen`.

At the very least, your cluster will probably have different package names than come in the file. If you are not using Torque there will be further complication.

1.6 Testing And Troubleshooting

1.6.1 Self test 1: tortureTest.m

If everything appears to have built successfully, it's time to send Imogen through the self-test wringer.

The first job is to run the CUDA stress-tester, the `tortureTest.m` function in the `gpuclass/` directory. It takes two arguments. The first enumerates the GPUs to be used for multi-device tests. The optional second, if 'y', runs the free radiation unit test without asking. Typical invocation: `tortureTest([0 1],'y')`

This will test almost all of the compiled functions used by Imogen for fluid dynamic simulations. If you run with multiple devices and run the radiation test, it should finish by printing a happy allcaps `UNIT TESTS PASSED!` message.

If you test only one one device or do not test radiation, it will yell that some tests weren't run.

If anything goes wrong, it will yell `UNIT TESTS FAILED!`.

1.6.2 Self test 2: run_fullTestSuite.m

The full test suite program loops through a bunch of custom simulations living under `run/fullTestSuite/ts*.m`. It will test by scaling the resolution and checking that the convergence order is, in fact, approximately two.

1.6.3 MPI Symbol Resolution Problems

If Imogen fails to start and Matlab prints a bitter complaint about being unable to resolve an MPI symbol, the problem is that Matlab defaults to lazy symbol resolution when loading dynamic libraries and apparently something about the MPI library trips on this. The solution is to force resolution at load time:

- LD_PRELOAD="$MPIDIR/lib/libmpi.so"

Putting this in your .bashrc (or as sysadmin, in the system's /etc/profile.d) will make it go away permanently.

2 Invocation

Imogen is run through the script run/imogen

There are three separate general types of run, all of which end up doing "matlab -r":

- serial - One process uses ones GPU; 'imogen' calls Matlab.
- parallel - N processes select N GPUs; 'imogen' calls mpirun calls Matlab
- cluster - N processes on M nodes; 'imogen' writes a script for qsub that calls Matlab

The 'serial' type run is for a simulation small enough to run on a single node. The 'parallel' type directly uses mpirun. It's only particularly useful for testing that parallelism is correct in a test setting. The 'cluster' run will probably be needed to launch large jobs that require a proper cluster.

The invocation syntaxes are available through `run/imogen -help`

3 Physics solved by the Imogen simulation engine

3.1 CFD

At its core Imogen solves the Euler equations, written here in conservative form:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} &= -\frac{\partial}{\partial x_i}\rho v_i \\
\frac{\partial(\rho v_i)}{\partial t} &= -\frac{\partial}{\partial x_j}(T_{ij} + \mathbb{I}P) \\
\frac{\partial E}{\partial t} &= -\frac{\partial}{\partial x_i}v_i(E + P)
\end{aligned}$$

with mass density ρ, fluid bulk velocity v_i, momentum stress tensor $T_{ij} = \rho v_i v_j$, identity tensor $\mathbb{I}$, scalar thermal pressure P, and total energy density $E = \frac{1}{2}\rho v^2 + \epsilon$.

These are closed by the equation of state $P(\epsilon)$. Imogen implements the adiabatic ideal gas equation of state, $P = (\gamma - 1)\epsilon$ for $\gamma > 1$ (physical gas values are $1 < \gamma \leq 5/3$).

With the assumption of a gamma law, the energy flux is written into the code as $v_i \times (\frac{1}{2}\rho v^2 + \frac{\gamma}{\gamma-1}P)$ with P calculated before fluxing.

3.1.1 Operator splitting

Imogen uses a dimensionally split flux solving scheme. Assuming a splittable Hamiltonian exists in the form

$$\mathbf{H} = \mathbf{E} + \mathbf{F} + \mathbf{G}$$

where in our case, E, F, and G represent the fluid fluxes in 3 non-degenerate directions (normally taken as 3 orthogonal directions in a curvilinear coordinate system: Here, fluxes in the x, y, and z directions), an approximate scheme of the form

$$e^{\epsilon\mathbf{H}} = e^{\epsilon\mathbf{E}} e^{\epsilon\mathbf{F}} e^{\epsilon\mathbf{G}} + \mathbb{O}(\epsilon^2)$$

always exists, and the reader may easily verify that the local error associated with doing this with two operators equals their commutator times ϵ^2.

Strang [reference!] showed that by performing such a first order step, then applying the same operators in exactly reversed order, will cancel all the first-order errors. This Strang splitting provides an easy way of constructing second spatial order solvers.

Denoting the operators that solve the inviscid flux equations in the three directions as $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$, there are 6 ways to order them which are the elements of the permutation group of $\{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$. In a two-dimensional simulation, there are only two orderings ($XYYX$ and $YXXY$).

Imogen cycles through them iteration to iteration, the idea being that this will average away (or at least partly so) any systematic errors associated with a given ordering.

There are additional operators associated with MHD (magnetic fluxing operators), non-inertial frames, radiation, and scalar potental fields (fixed potentials, point objects or self-gravitation).

The MHD algorithm has major problems with low-beta plasma and seems to be subtly broken at present.

3.2 MHD

Imogen can evolve magnetic fields in the limit of Ideal MHD where the Euler equations become

$$\begin{aligned}
\frac{\partial \rho}{\partial t} &= -\frac{\partial}{\partial x_i}\rho v_i \\
\frac{\partial (\rho v_i)}{\partial t} &= -\frac{\partial}{\partial x_j}(T_{ij} + \mathbb{I}(P + B^2/2) \\
\frac{\partial E}{\partial t} &= -\frac{\partial}{\partial x_i} v_i(E + P) - B_i(v_j B_j) \\
\frac{\partial B_i}{\partial t} &= \frac{\partial}{\partial x_i}(v_i B_j - B_i v_j)
\end{aligned}$$

now with the stress tensor $T_{ij} = \rho v_i v_j - B_i B_j$ and under the requirement $\nabla \mathbf{B} = 0$.

There is no computational requirement that the **B** field be initialized this way, however Imogen solves no advection equation for **B**-charge and any divergence introduced will simply stay there.

3.3 Frame rotation

In many cases a simulation is desired of an object that exhibits coherent rotation at high speeds.

Imogen has the Eulerian explicit CFL constraint is set by

$$dt < dx(|v_i| + c_s)$$

and so if v_i can be significantly reduced by setting the grid to co-rotate with a flow, the timestep can be significantly increased for advection-dominated flows.

The non-inertial sourcing operator introduces its own errors and it is up to the user to determine the point at which the change is worth it.

Imogen checks for a rotating frame at simulation initialization and performs the transformation to the noninertial frame for you - simulations should be initialized in the lab inertial frame.

3.4 Radiation

Imogen can solve optically thin radiation with power law cooling,

(power law lambda)

Programming in different cooling laws is extremely simple as a test by making additional entries to the experiment/Radiation.m file, templatted after the thin power law radiation one. Of course, don't expect them to be as fast as the GPU accelerated routines.

While the power law cooling is reliable enough in and of itself, it has exhibited a remarkable ability to derange Imogen's fluid solver routines. None of them has exhibited the ability to handle an extremely slowly-moving radiating shock front (i.e. the motion associated with the finite accuracy of the simulation) without some measure of oscillatory density error at the adiabatic jump.

(Post pics of evil radiative shocks here)

3.5 Gravity

3.5.1 Point gravity

Imogen allows the addition of massive points to the fluid simulation which obey Newton's laws. Both point-point and point-fluid gravitational force are computed.

These points, "compact objects" as Imogen calls them, are used as standins for stars or planets which cannot be resolved at reasonably available levels of resolution.

Far away from the object, they behave as point masses. Within a distance defined as its 'radius', matter is considered as having been accreted. Each step, the mass and linear and angular momentum in cells whose center is within r of a compact mass are added to the compact object and replaced with grid's vaccuum values instead.

NOTE: This is not the desired behavior in some cases - e.g. a planet would require a surface pressure, not a surface vaccuum, in order to be embedded in the disk.

Point does not feel fluid gravity presently.

3.5.2 Self gravity

This is basically dead for now, especially in parallel.

The serial solver could easily enough be resurrected but there's no way I'm going to have time to setup a parallel Poisson solver, let alone shovel it onto the GPU.

4 Physics solvers in the initializers

Many fluid and physics subproblems come up when writing initial condition generators for Imogen. Some of these have convenient self-contained solvers available within Imogen.

The available units include:

- `J = HDJumpSolver(Mach,` γ`,` θ`)`: Returns a structure J describing the only physical shock solution to a hydrodynamic shock of strength $Mach$ in a fluid of adiabatic index γ and preshock shock-normal flow inclination θ (in degrees).

- `J = MHDJumpSolver(Mach, Alfven Mach,` γ`,` θ`)`: Solves the MHD jump equations for the slowest shock permitted.

- `F = fim(X, @f(X))`: Short for Fourier Integral Method, calculates the integral of anonymous function f on the presumed regularly spaced points X by calculating a polynomial that renders f 3 times continuous at the endpoints, integrating the polynomial directly and the residual by Fourier transform.

- `The ICM`: `icm.m` is currently very much a prototype and not ready for normal use. It is short for Integral Constraints Method.

- `t = pade(c_n, a, b, x)`: Given a polynomial $\sum_n c_n x^n$, calculates the (a,b) Pade approximant P(a)/Q(b) of the polynomial and evaluates it at x, for any $a + b \leq 6$.

- RadiatingFlowSolver class: Solves the equations of a power-law-radiating hydrodynamic or magnetohydrodynamic flow. It initializes itself using the 4th (MHD) or 6th (HD) order power series solution of the flow and then integrates using a 6th order implicit LMM, restarting with reduced step size when it encounters sharp flow features (knees).
- Linear wave eigenvectors: The function eigenvectorEntropy, eigenvectorAlfven, eigenvectorSonic, eigenevectorMA(rho, csq, v, b, k, wavetype) return the linear wave eigenvectors associated with the flows with the given uniform density, thermal acoustic soundspeed, velocity, magnetic field and wavevector. Alfven and sonic waves have wavetype +1 for a wave which advances towards **k** and -1 for against **k**. The MA wavetype can have either sign, with magnitude 2 for a fast MS wave and magnitude 1 for a slow MS wave. If a nonzero B is passed to a sonic wave, it will invoke the fast MS solver (as the fast wave is the one connected with the nonmagnetized sonic wave).
- The bow shock problem, in two-dimensional testing of a source with outflow, has acquired a routine analyticalOutflow() which solves the exact adiabatic expansion of a $\gamma = 5/3$ gas away from a cylinder with the inner boundary condition of $v_{radial} = v_0$ at $r = r_0$.

5 Programming Experiments

5.1 Common parameters of all experiments

All experiment initializers should inherit from the `Initializer` class and share its not inconsiderable breadth of parameters. Among the more important names are

- `cfl` coefficient by which the CFL-limit timestep is scaled. **WARNING IMOGEN DOES NOT SANITY CHECK THIS**
- `gamma` - The adiabatic index of the fluid
- `grid` - The [nx ny nz] grid resolution of the *global* domain. This should normally be set when creating the initializer and not messed with again.
- `iterMax` - the maximum number of timesteps to take
- `useInSituAnalysis` - If nonzero, Imogen uses the given in-situ analysis tool
- `inSituHandle` - @AnalyzerClass which must meet the requirements of an in-situ analyzer (see REF)
- `stepsPerInSitu` - Number of timesteps between calls to the analyzer
- `frameRotateOmega` - If nonzero, Imogen places the simulation into a rotating frame and applies appropriate source terms. **Simulation is still initialized in lab frame**
- `frameRotateCenter` - If frame is rotating, this is the center (**in cells**) about which the frame rotates on the (x, y) plane.

5.2 How to setup frame rotation

To turn on the rotating frame during start, simply set a nonzero value for the `frameRotateOmega` parameter, and assign a 2-element array $[x_0, y_0]$ to `frameRotateCenter` and Imogen will automatically transform the fluid momentum field into the noninertial frame at start.

Caution: x_0 and y_0 need to be in cell coordinates... yes it's dumb.

The rotation rate can be altered during simulation time using `alterFrameRotation(run, mass,`
`ener, mom, newOmega)` in the user's in-situ analysis function. This has no physical effect (outside truncation error) and will not e.g. generate the Euler term, it simply alters the rate at which the grid traverses the fluid in the lab frame. Note also that the alterFrameRotation function is not GPU accelerated and probably shouldn't called willy nilly.

One *could* directly alter `run.frameRotateOmega` during the run, and this will generate physical consequences. However the sourcing function assumes the rotation rate is constant and therefore will *not* account for the Euler term caused by varying rotation rate.

5.3 How to setup radiation

5.4 How to setup a potential field

5.5 How to set up a compact object

5.6 How to setup in-situ analysis

Recognizing that it is not practical to save the entire simulation output for post-hoc analysis in most cases, Imogen allows the construction of in-situ analyzers. These are classes that must have two public members:

- `.FrameAnalyzer(mass, ener, mom, mag, run)`
- `.finish(run)`

They must be set at initialization time with these three parameters in the initializer:

- `useInSituAnalysis` - Set to nonzero to enable
- `inSituHandle` - @AnalyzerClass which must meet the requirements of an in-situ analyzer (see above)
- `stepsPerInSitu` - Number of timesteps between calls to the .FrameAnalyzer() function

The `.finish` member is called when the simulation is exiting and presumably should take the data that has been gathered and safely flush it to disk since Imogen is about to return.

6 Experiments

6.1 Centrifuge Test

The centrifuge test checks the ability of the code to evolve an analytically time-independent hydrodynamic flow and maintain its original time-independent values, and also provides a valuable test of the rotating frame's implementation.

6.1.1 Analysis

The centrifuge test balances centripetal acceleration against a pressure gradient,

$$\rho r \omega(r)^2 = dP/dr$$

Under the ideal gas law,

$$P = K\rho T$$

and assuming that we are dealing with a thermodynamically simple gas or mixture whose internal degrees of freedom are pleasantly unchanging,

$$\rho r \omega(r)^2 = KT d\rho/dr + K\rho dT/dr$$

Two of ω, ρ and T have to be specified, then our differential equation plus boundary conditions defines the third.

We chose to define an arbitrary ω, except for the requirement that velocity behave itself at the origin (ω diverging slower than $1/r$) and with the awareness that ω will only be evaluated on a compact region normalized to $[0, 1]$, outside of which it is set to zero (in other words, fluid at rest at infinity).

A pressure-supported structure can't be isobaric but isothermal, isochoric and adiabatic equilibria can all be defined for any sane initial conditions. The ODE is solved by separation of variables; Because it occurs often, the quantity

$$\int_{r0}^{r} r\omega(r)^2 dr \equiv \Phi(r, r0)$$

to save space.

Isothermal With a fixed temperature (represented in the code as a fixed isothermal soundspeed), the ODE

$$\Phi(r, r_0) = KT \int_{\rho_0}^{\rho} d\rho/\rho$$

is separated and has solution

$$\rho(r) = \rho_0 e^{\Phi(r,r_0)/KT}$$

With ρ_0 specified at the outer edge radius r_0 and an isothermal soundspeed KT, a physical solution exists for sane inputs.

Isochoric At a fixed volume (here taken as fixed density), the ODE

$$\Phi(r, r_0) = K \int_{T_0}^{T} dT$$

is separated and has solution

$$T(r) = (a^2 + \Phi(r, r0))/K$$

which gives pressure

$$P = \rho K T = \rho_0 (a^2 + \Phi(r, r0))$$

With the initial temperature set by proxy by isothermal soundspeed a at the center and density fixed, the temperature ramps up as required and a solution exists for sane inputs.

Adiabatic Under adiabatic conditions we use the relation $P = K\rho^\gamma$ and so

$$\frac{\Phi(r, r_0)}{K\gamma} = \int_{\rho_0}^{\rho} \rho^{\gamma-2} d\rho$$

with solution

$$\rho(r) = \left[\rho_0^{\gamma-1} + \frac{(\gamma-1)\Phi(r, r_0)}{K\gamma}\right]^{1/(\gamma-1)}$$

Defining ρ_0 at r_0 and given K, a solution exists for all sane inputs; Imogen defines it at the outer edge.

6.1.2 Initial Conditions

The physical input parameters to the centifuge test are:

- `rho_0` specifies the density at either the center (Isochoric) or edge (isothermal or adiabatic) of the simulation region
- `P_0` specifies the pressure at the same points. The isothermal c_s and adiabatic k are defined through the pressure.
- `omegaCurve` is an anonymous function of an array r which gives $\omega(r)$. This is assumed to obey the constraints given. r is normalized between 0 (center) and 1 (edge).
- `eqnOfState` must be symbolically set to one of the `CentrifugeInitializer class' symbolic values EOS_ISOTHERMAL, EOS_ISOCHORIC or EOS_ADIABATIC.`

Two additional numeric parameters are

- `edgeFraction` - Sets how far the simulation domain extends past the edge of the rotating region, as a multiple of the radius of the centrifuge: $2(1 + e) = Nx$
- `frameRotateOmega` - Sets the rate at which the simulation frame is rotated. This can improve the timestep considerably if rotation is in one direction and advection speed is high. In fluid velocities, positive ω about $\hat{z}$ is such that particles on $+\hat{x}$ have momentum $+\hat{y}$. The fluid frame is opposite, such that positive `frameRotateOmega` will create the appearance of points on $\hat{x}$ swinging towards $-\hat{y}$.

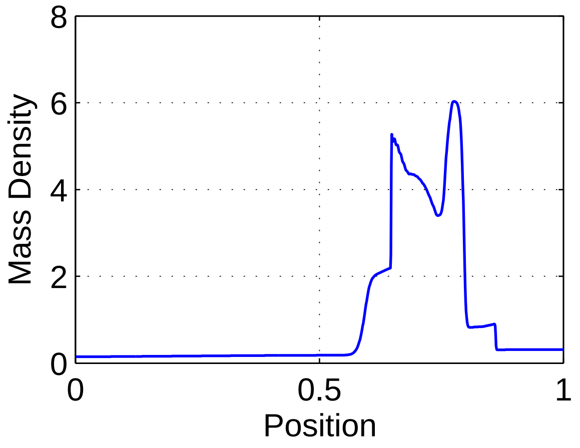

Figure 1: Double Blast at t = 0.038

6.2 Double Blast Test

Two interacting blast waves on a 1-dimensional grid. This test is very difficult to solve on an Eulerian grid due to the strong shocks and multiple interactions with rarefactions and contact discontinuities, and is useful in measuring a solver's ability to handle these types of events.

6.2.1 Analysis

The two blast waves propogate toward one another and collide, producing more contact discontinuities. Could calculate the exact time they collide. Solve sod problem.

6.2.2 Initial Conditions

Boundary conditions are mirrored around the X axis and periodic everywhere else.

The physical input parameters to the double blast test are:

- `Pl - Sets the pressure of the blastwave originating in the left-most tenth of the grid`
- `Pr - Similarly sets the pressure of the right-most blastwave`
- `Pa - Sets the ambient pressure`

with the initial conditions set by each of the three zones being defined as an adiabatic gas with $\gamma = 1.4$ everywhere. We set `Pl` = `1000`, `Pr` = `100`, and `Pa` = `0.01`. Initial momenta are zero everywhere.

6.3 Einfeldt Strong Rarefaction Test

This test measures the solver's vulnerability to very strong rarefactions that can, in some cases, produce negative mass densities or pressures. The input parameters can be tweaked

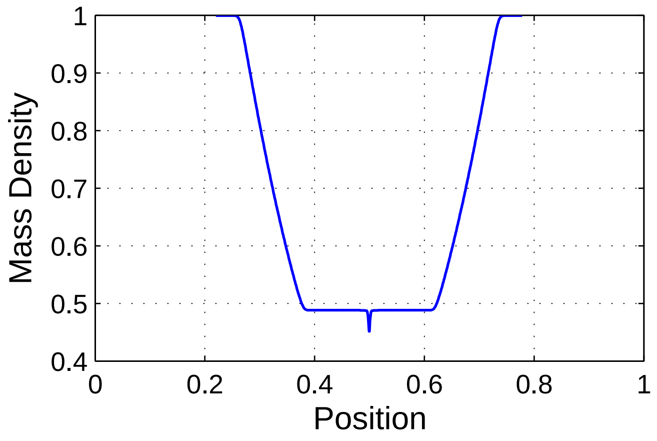

Figure 2: Einfeldt test at t = 0.1

slowly to determine exactly how strong the rarefaction can become before producing values that are NAN.

6.3.1 Analysis

6.3.2 Initial Conditions

Boundary conditions are constant around the X axis and periodic everywhere else.

Einfeldt's Strong Rarefaction test consists of a 1-dimensional grid divided exactly in half. The properties of each half are defined separately, using four physical parameters. When run, the test produces two strong rarefaction waves moving away from the midpoint of the grid.

The physical input parameters to the Einfeldt Strong Rarefaction test are:

- `rho - Defines the mass density of the region`
- `m - Defines the X-momentum of the region (parallel to the grid)`
- `n - Defines the Y-momentum of the region (perpendicular to the grid)`
- `e - Defines the energy density of the region`

The four parameters are saved separately as rhol and rhor, ml and mr, and so on.

6.4 Richtmyer-Meshkov Instability Test

RMI occurs whenever a plane shockwave becomes incident upon a non-uniform density interface. The plane shock refracts through the interface, imparting differential vorticity that drives a jet from the denser fluid into the lighter fluid.

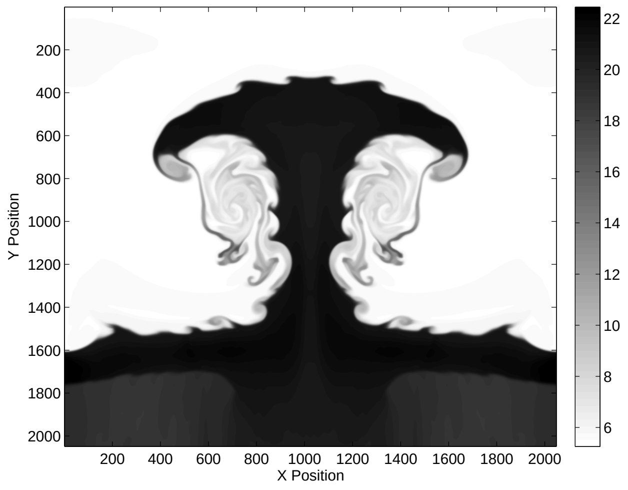

Figure 3: Richtmyer-Meshkov instability at t = 6504

6.4.1 Analysis

For this run, we used a square grid 2048 zones across, for 20,000 iterations.

6.4.2 Initial Conditions

Boundary conditions are periodic around the X axis, constant around the Y axis, and mirrored around the Z axis.

The physical input parameters to the Richtmyer-Meshkov Instability test are:

- `mach - Defines the mach speed of the incoming shock wave`
- `rhotop - Defines the density of the lighter fluid`
- `rhobottom - Defines the density of the heavier fluid`

To create the non-uniform interface, we create a cosine wave with an amplitude 1/20 of the height of the grid and with a wavelength 2x the length of the grid. We then place the center of the wave 4 total wave amplitudes below the center of the grid, and set the mass density of the entire region below the wave equal to rhobottom. The rest of the grid is previously set to have density equal to rhotop, and is unchanged by the wave. We then create a shocked region from the top of the grid to a height just 20 zones above the peak of the wave. This shock is composed of a momentum and pressure equal to a blast wave with mach set by the input 'mach'.

Note that the entire region must then be given a momentum in the direction opposing the shock to maintain position on the grid. This opposing momentum is calculated by the same function as is used to calculate the initial shock.

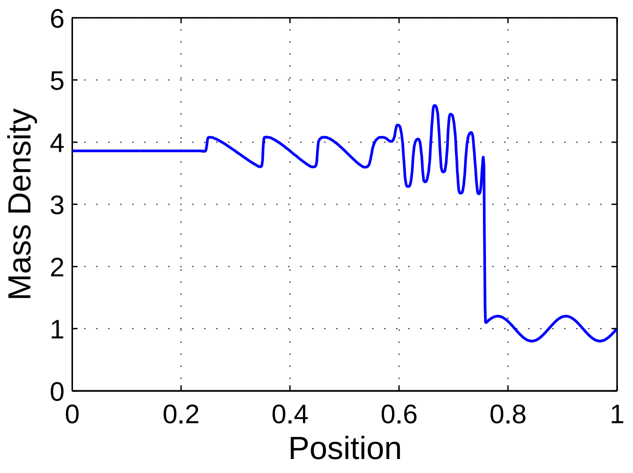

Figure 4: Shu-Osher Shocktube at t = 0.178

6.5 Shu-Osher Tube Test

Shu, C and Osher, S., "Efficient Implementation of Essentially Non-Oscillatory Shock-Capturing Schemes, II", J. Computational Physics, 83, 32-78 (1989). The test is Example 8.

The grid is divided in half, with the left containing a shock and the right containing a sinusoidal structure in mass density. As the shock becomes incident upon the sinusoid, the structure of the sinusoid propogates out with the shockwave, translating and evolving but not acquiring any additional structure.

6.5.1 Analysis

6.5.2 Initial Conditions

Boundary conditions are constant around the X axis and periodic everywhere else.

The physical input parameters to the Shu-Osher test are:

- `lambda - Defines the wave number of the sinusoidal mass density structure`
- `mach - Defines the speed of the shockwave toward the sinusoid`
- `waveAmplitude - Defines the amplitude of the sinusoid`