The PENNANT Mini-App

Charles R. Ferenbaugh
Los Alamos National Laboratory
cferenba@lanl.gov

Version 0.9 – February 2016
LA-CC-12-021
https://github.com/lanl/PENNANT

PENNANT is an unstructured mesh physics mini-app designed for advanced architecture research. It contains mesh data structures and a few physics algorithms adapted from the LANL rad-hydro code FLAG, and gives a sample of the typical memory access patterns of FLAG.

Details on the performance of PENNANT can be found in [5, 6].

1 Building and running the code

1.1 Building

A simple `Makefile` is provided in the top-level directory for building the code. Before using it, you may wish to edit the definitions of CXX and CXXFLAGS to specify your desired C++ compiler and flags, and to choose between optimized/debug and serial/OpenMP/MPI builds. Then a simple "`make`" command will create a `build` subdirectory and build the `pennant` binary in that directory.

PENNANT has been tested under GCC 5.1.0, PGI 15.3, and Intel 15.0.3. Building under other compilers should require only minor changes.

1.2 Running tests

Several test problems are provided in subdirectories under the `test` directory. The command line

```
pennant testname.pnt
```

is used to run a test in serial mode. If running under MPI, this should be preceded by `mpirun` or similar command as appropriate on your system.

The available test problems are listed in Table 1. The smaller problems run quickly and are useful for debugging and regression tests; gold standard files are provided (see next section). The larger tests take longer to run and are suitable for timing tests. Most of these problems have sizes chosen to match a typical FLAG run on a single node of a cluster; the exceptions are the two Leblanc problems specifically labeled as multi-node.

Table 1: Test problems provided with PENNANT.

name	# zones	mesh shape	zone type
Leblanc problems [1]:			
leblanc	900	rectangle	quad
leblancbig	230400	rectangle	quad
Leblanc problems, multi-node versions:			
leblancx4	3.69×10^6	rectangle	quad
leblancx16	5.90×10^7	rectangle	quad
leblancx64	9.44×10^8	rectangle	quad
Noh problems [7]:			
nohsmall	40	radial	triangle/quad
noh	3000	radial	triangle/quad
nohsquare	129600	square	quad
nohpoly	63001	square	mostly hexagons
Sedov problems [8]:			
sedovsmall	81	square	quad
sedov	2025	square	quad
sedovbig	291600	square	quad

1.3 Test inputs and outputs

Each test problem directory contains an input file with the `.pnt` suffix. This is a small text file containing input parameters for the test.

PENNANT can generate output files of two kinds. The `.xy` output file is a text file containing the per-zone values of zone density, energy, and pressure. (It is modeled after a similar file generated by FLAG.) Note that, when running under MPI, the order of the output values in this file may vary when the number of PEs changes. For the smaller tests, two gold standard files are provided for reference. The `.xy.std` file gives the expected results for a serial run (or 1-PE MPI run), while the `.xy.std4` file gives expected results for a 4-PE MPI run. (These files are identical except for the ordering of zones, which is done differently depending on the number of PEs in use.)

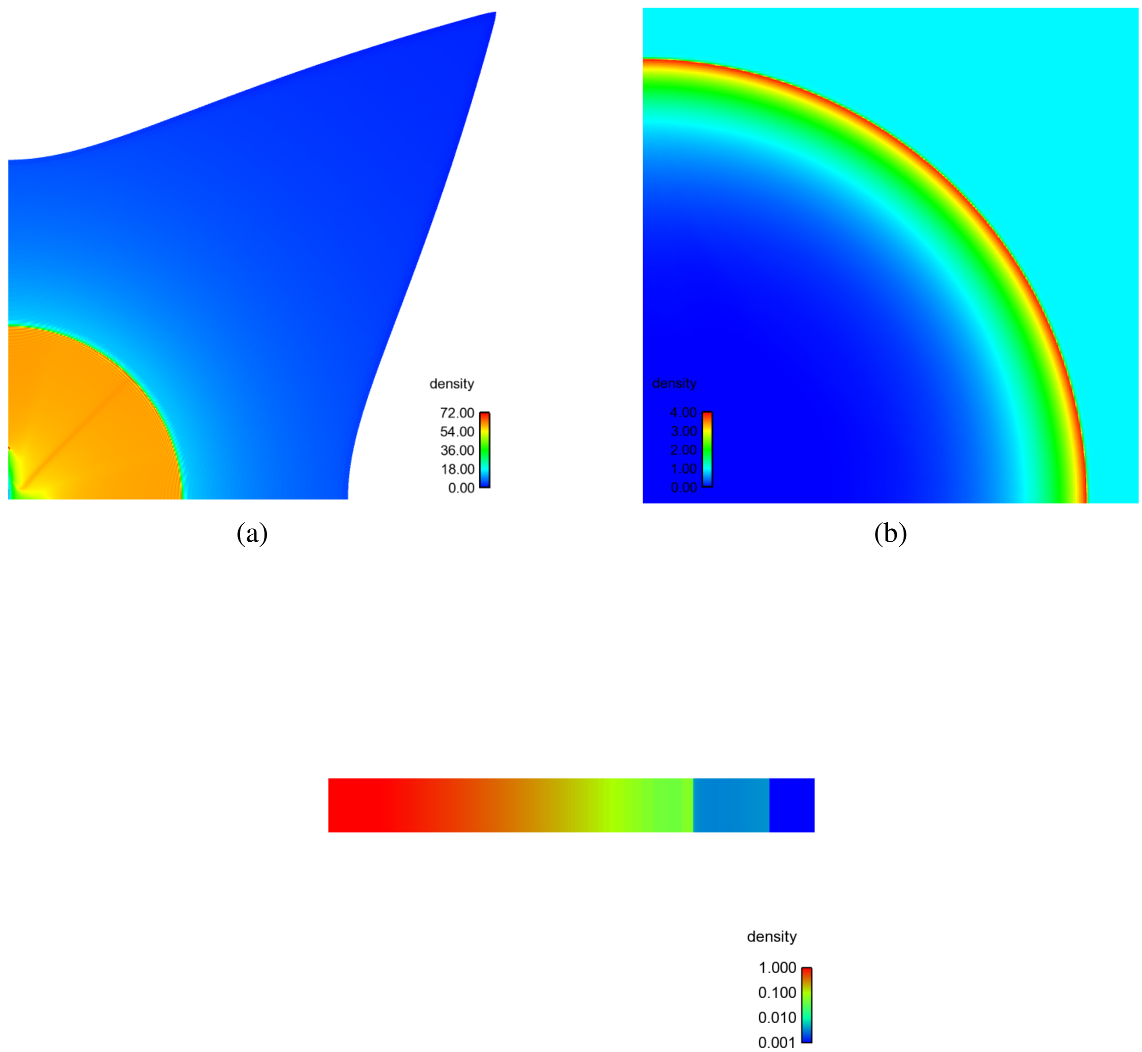
There are also several graphics output files in Ensight Gold format: the main file has suffix `.case`, and it refers to auxiliary files with suffixes `.geo`, `.ze`, `.zp`, and `.zr`. These can be viewed by the proprietary Ensight[1] viewer, or by open-source viewers such as ParaView[2] and VisIt[3]. Sample outputs are shown in Figure 1.

These outputs are off by default, but can be activated using the `writexy` and `writegold` input file flags respectively (see next section). Note that the file writers are not optimized to work well on large numbers of MPI ranks (this will be fixed in a future release).

1.4 Input file parameters

In most cases, there is no need for users to modify input files. However, here are a few parameters that ambitious users might want to know about:

[1] `http://www.ensight.com`
[2] `http://www.paraview.org`
[3] `https://wci.llnl.gov/codes/visit/home.html`



(c)

Figure 1: Final state of (a) *nohsquare*, (b) *sedovbig*, and (c) *leblancbig* problems, colored by zone density.

`writexy` (integer) If nonzero, write `.xy` file at end of run.

`writegold` (integer) If nonzero, write Ensight Gold file at end of run.

`cstop` (integer) Stop run when problem reaches given cycle number.

`tstop` (real) Stop run when problem reaches given simulation time.

`chunksize` (integer) Process mesh elements in chunks of given size; see section 2.3 for more details on chunk processing. If `chunksize` is zero, the entire mesh is treated as a single chunk (this is the default). Typically, for best performance, this value will be chosen so that a chunk can fit in L1 or L2 cache as appropriate; it follows that the optimal value is architecture-dependent.

`meshparams` (list of integers and reals) Parameters for internal mesh generator. These may be modified if additional test cases of varying sizes are desired (e.g., for scaling studies). The format of this line is:
`meshparams` *`nzx [nzy [lenx [leny]]]`*
where the parameters have the following meanings:

`nzx, nzy`	number of zones in x, y directions (no default for nzx; default nzy = nzx)
`lenx, leny`	total length in x, y directions (default for both = 1.0, except when meshtype = pie, in which case default for lenx = 90.0)

For the *pie* mesh type, x and y should be understood as θ and r respectively.

`dtinit` (real) Initial timestep. This shouldn't need to be changed unless the mesh has been changed (see `meshparams` above). As a rule of thumb, if the resolution of the problem is increased by a factor of r in each direction, `dtinit` must decrease by a factor of r.

2 Data structure details

2.1 Mesh data structures

PENNANT is designed to use standard finite-volume meshes similar to those used by many common physics solvers. In particular, PENNANT supports 2-D unstructured meshes composed of arbitrary polygons.

The PENNANT mesh data structures are a subset of those used by FLAG. These are implemented in the `Mesh` class. FLAG supports 1-, 2-, and 3-D meshes with various geometries; for simplicity, PENNANT is restricted to the 2-D, cylindrical geometry case.

The PENNANT terminology for entities within a mesh is shown in Figure 2. The basic mesh elements in 0, 1, and 2 dimensions are called *points, edges,* and *zones* respectively. PENNANT also uses two types of sub-zone entities. Within any given zone, a *side* is a triangle whose vertices are two consecutive boundary points of the zone together with the zone center. A *corner* is a quadrilateral whose vertices are one boundary point of a zone, the midpoints of the two adjoining edges, and the zone center.

For each entity type, the first letter of its name is used as an identifier for variables associated with it. For example, `px` is an array of point coordinates, and `zvol` is an array of zone volumes. There are scalar variables of the form `numX` which give the number of `X`'s in the problem: `nump` is the number of points, and so on.

PENNANT also stores various mapping arrays from sides to other entity types. These are shown in Figure 3. Given a side s, the following mapping arrays are available:

- `mapsz` gives the zone z of which s is a subregion.

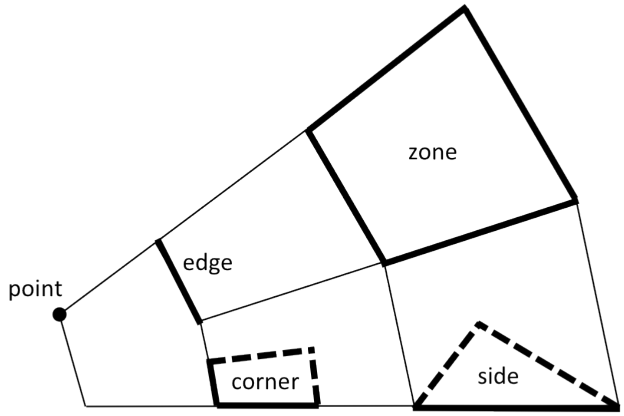


Figure 2: PENNANT terminology for mesh entities.

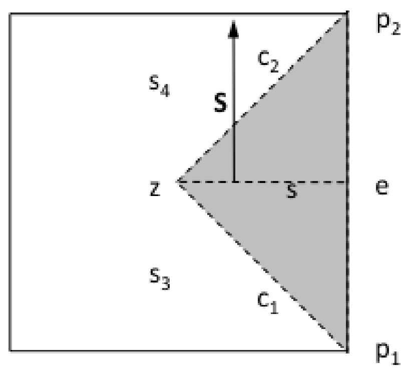
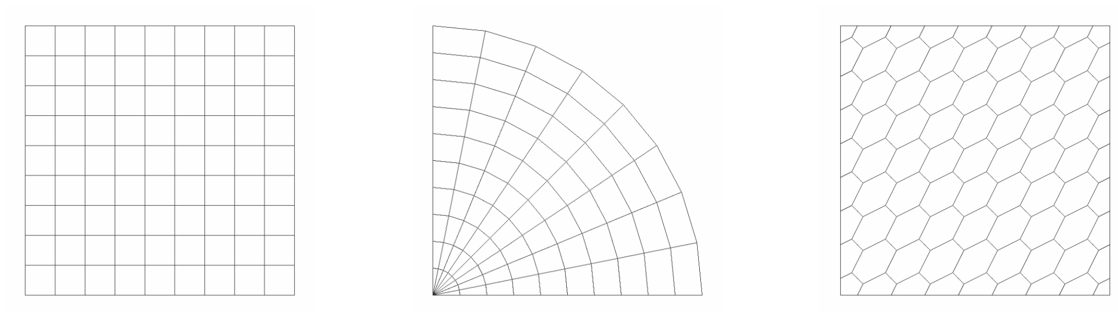


Figure 3: Various side map arrays supported by PENNANT.

- `mapse` gives the edge e on the boundary of s.
- `mapsp1` and `mapsp2` give the two mesh points p_1 and p_2 on the boundary of s. It is assumed that the mesh is oriented according to a right-hand rule, so that the edge from p_1 to p_2 is always in a counter-clockwise direction relative to the zone.
- `mapss3` and `mapss4` give the two sides s_3 and s_4 on either side of s, where s_3 is before s and s_4 is after it in a counter-clockwise traversal of the zone.

Since there is a one-to-one correspondence between sides and corners, side-to-corner arrays are not used. By convention, the corner labeled c_1 in Figure 3 has the same index as s. It follows that c_2 has the same index as s_4, or `mapss4[s]`.

The `Mesh` class also has methods for computing other geometry-related variables, such as edge and zone centers, lengths, volumes, and surface vectors. The surface vector for a side s, shown by the vector **S** in Figure 3, is used by force computations in the hydro algorithms described later.



(a) (b) (c)

Figure 4: Mesh types generated by PENNANT mesh generators: (a) `rect`, (b) `pie`, and (c) `hex`.

2.2 Mesh generators

PENNANT has internal mesh generation code that will generate three different types of meshes, shown in Figure 4.

Note that the physics routines in PENNANT can process 2-D meshes of any geometry; they are not limited to the mesh types shown here. However, the internal mesh generators are set up to also generate domain connectivity information needed by MPI (see section 2.4), which would be difficult to generate for arbitrary meshes.

2.3 Chunk processing

The PENNANT `Mesh` class has been set up to support computation on chunks of the mesh in parallel. This is used to implement OpenMP and CUDA versions of the code and should lend itself to other task-based approaches as well. (MPI parallelism is implemented differently, using geometric domain decomposition, as described in section 2.4). The maximum chunk size is controlled by the input file parameter `chunksize`.

In the current chunking approach, the lists of points and zones are simply divided into chunks of size `chunksize` (except for the final, leftover chunk). The list of sides is handled similarly, except that the size of each individual chunk is rounded down slightly if necessary so that each zone has all of its sides in the same chunk. For each chunk, the *first* and *last* indices of the chunk are stored. (Note that *last* is actually one index beyond the end of the chunk, in a similar manner to STL iterators, so that the sides in a side chunk are those with $s_{first} \leq s < s_{last}$.) The total numbers of each type of chunk (`numXch`, where `X` may be `p`, `s`, or `z`) are also stored.

Then, nearly all of the routines in the main hydro cycle have been modified to take as input first and last indices of the appropriate mesh entity. This allows the hydro processing to be divided into five phases, two on point chunks, two on side chunks, and one on zone chunks. Within each phase, all chunks are independent and can be processed in parallel. See `Hydro::doCycle()` for the complete code flow.

A few of the helper routines, particularly in the `QCS` class, use scratch arrays the size of the chunk currently being processed. The prefix `s0` is used for an array with one entry per side in the current chunk, with prefixes `c0` and `z0` used similarly for corners and zones respectively.

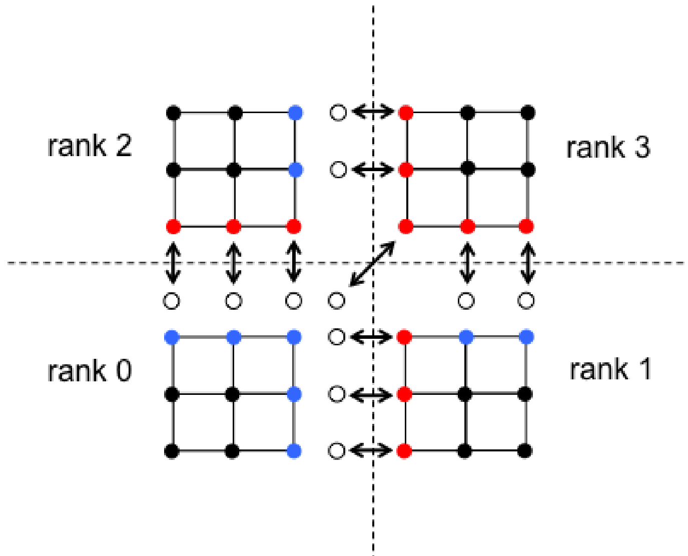


Figure 5: Example MPI decomposition of a mesh. Master points are shown as blue, slaves as red, and proxies as white. MPI communications between slaves and proxies are shown as arrows.

2.4 Domain decomposition

The MPI implementation of PENNANT uses domain decomposition to put a geometrically contiguous subset of the mesh on each processor. It also generates information to allow point data to be communicated across processors, as described below.

PENNANT follows FLAG in assigning each zone, corner and side to a single MPI rank when decomposing the mesh. However, any point on a rank boundary is replicated on each rank which needs it; see Figure 5 for an illustration. For any duplicated point, one of its instances is designated as the *master*, and the others are its *slaves*. (The current PENNANT mesh generators use the convention that the master point is the one on the lowest-numbered MPI rank.) When it is time to sum a quantity from corners to points, the summation is first done for on-processor corners in `Mesh::sumOnProc`. Then the summation is extended across processors in three stages:

1. In `Mesh::parallelGather`, slave point values are assembled into messages, and sent to corresponding *proxy* points on the same rank as their masters (using MPI).
2. In `Mesh::parallelSum`, master points sum their own values and all proxy values, and store sum at master and all proxies (on-processor only, no MPI is used).
3. In `Mesh::parallelScatter`, the updated proxy point values are assembled into messages and sent back to their corresponding slave points (using MPI).

Table 2: Basic data flow for FLAG/PENNANT hydrodynamics.

step	main inputs	main outputs
Predictor step:		
1. Update mesh	point velocity, position	point position (half-advanced)
1a. Update mesh geometry	point position	**side and zone volume; zone density; side surface vector**
2. Compute point masses	zone density, volume; side mass fraction	**point mass**
3. Update thermodynamic state	zone density, specific energy, work rate	zone pressure, sound speed
4. Compute forces	side surface vector; zone pressure	side and **point force**
4a. Apply boundary conditions	point force, velocity	point force, velocity (constrained)
5. Compute accleration	point force	point acceleration
Corrector step:		
6. Update mesh	point acceleration, velocity, position	point velocity, position (fully advanced)
6a. Update mesh geometry	point position	**side and zone volume**
7. Compute work	point position, velocity; side force	**zone work, work rate, total energy**
8. Update zone state variables	zone volume, mass, total energy	zone density, specific energy

3 Physics details

3.1 Basic hydro algorithms

PENNANT provides a subset of the compatible Lagrangian staggered grid hydrodynamics (SGH) algorithms implemented in FLAG and described in [3]. These are implemented in the `Hydro` class. An outline of the main steps is given in Table 2.

The PENNANT hydro algorithm is a *Lagrangian* method, meaning that the computational mesh moves with the material as the problem state advances. This implies that the mass and material type within each zone are constant throughout the problem, but the zone's position and shape will change over time.

It is also a *staggered-grid* method, meaning that mesh positions and related variables (velocity, acceleration, etc.) are stored on points, while most state variables (density, energy, pressure, etc.) are stored on zones. Therefore, the calculation must frequently use values of zone-based variables to compute point-based results, or vice-versa. In Table 2, such results are shown in **bold**. (Note that this is true for five of the 11 steps shown.)

To facilitate these calculations, many of the calculation loops are done over sides, and some intermediate variables are stored on sides. This works since each side can be easily correlated to its corresponding zone and points using the mapping arrays in the `Mesh` object. A few routines use corners in a similar manner.

PENNANT hydro uses a *predictor-corrector* time integration method. Each cycle can be broken into two steps, shown in the table. The cycle begins with all problem state defined for the beginning of the timestep. In the predictor step, some variables are advanced to the middle of the timestep, in order to compute half-advanced point acceleration values. In the corrector step, the new accelerations are then used to advance all

Table 3: Examples of PENNANT variables and their dependence on timestep.

	Part of time step		
quantity	begin	middle	end
point coordinate	px0	pxp	px
point velocity	pu0		pu
point acceleration		pap	
point force		pf	
point mass		pmaswt	
zone center coordinate		zxp	zx
zone mass			zm
zone volume	zvol0	zvolp	zvol
zone density		zrp	zr
zone specific energy			ze
side volume		svolp	svol
side surface vector		ssurfp	

variables to the end of the timestep.

To implement the predictor-corrector scheme, it is necessary to store multiple values of some of the problem variables. This is done using the following notation convention:

- suffix 0 = the beginning of the timestep ("cycle n")
- suffix p = half-way through the timestep ("cycle n + 1/2")
- no suffix = completion of the timestep ("cycle n + 1")

Some examples are shown in Table 3. Note that some entries in the table are blank, since not all quantities are needed at all times.

3.2 Energy check

The hydro algorithms in PENNANT are intended to conserve total energy as a simulation progresses. However, in the 2D cylindrical geometry used by PENNANT, the conservation is not always exact. An energy check diagnostic is printed at the beginning and end of each run to verify conservation. For the `leblanc` problems, energy is typically conserved to within the accuracy of the printout (seven decimal places). For other problems, there can be a relative error of about 2×10^{-4} (for the `noh` problems) or 2×10^{-2} (for the `sedov` problems). These errors are due to limitations of the numerical algorithms in the cylindrical case, and should not change significantly between different platforms or implementations of PENNANT.

3.3 Material model

PENNANT provides finite-volume, arbitrary-polygon cells with a gas material model, implemented in the `PolyGas` class. This class includes code to compute a simple gamma-law gas equation of state, and to compute the resulting pressure-based forces.

3.4 Subzonal pressures

PENNANT provides the Temporary Triangular Subzoning (TTS) algorithm described in [4, 9]. This is implemented in the `TTS` class. This prevents certain kinds of distortions of zones, such as "hourglassing," by estimating a pressure for each side, and adding a force to each side based on the difference between the zone and side pressures.

Note to FLAG users: The FLAG implementation of TTS contains, in addition to the subzonal pressure treatment, an artificial viscosity algorithm based on the subzonal pressures; the artificial viscosity is not part of the standard TTS description in the references. Only the subzonal pressure part of TTS is implemented in PENNANT.

3.5 Artificial viscosity

PENNANT provides the tensor artificial viscosity algorithm of Campbell and Shashkov, described in [2]. This is implemented in the `QCS` class. (The symbol q is traditionally used to denote artificial viscosity, hence the `Q` prefix on the class name.) Artificial viscosity is a fictitious term commonly introduced into fluid flow equations to correctly handle shock regions with large discontinuities in the problem state variables.

Acknowledgements

Thanks to Mikhail Shashkov and the ASCR "Mimetic Methods for PDEs" project, and the ASC Hydrodynamics project, for providing support for this work. Thanks to Pat McCormick and the ASC Programming Models project for providing some additional support for development of the MPI version. Thanks to Sriram Swaminarayan and the ASC Code Strategies for Emerging Platforms project for supporting PENNANT modifications for the APEX NERSC-9/Crossroads procurement.

Thanks also to the Lagrangian Applications Project members who have contributed to the FLAG code; parts of the PENNANT code and documentation are adapted from their work. And finally, thanks to the participants in the Intel EPOCH workshop in the summer of 2012; several optimization ideas from that workshop have been incorporated into subsequent PENNANT versions.

References

[1] D.J. Benson. Momentum advection on a staggered mesh. *J. Comput. Phys*, 100:143–162, 1992.

[2] J. Campbell and M. Shashkov. A tensor artificial viscosity using a mimetic finite difference algorithm. *J. Comput. Phys*, 172:739–765, 2001.

[3] E.J. Caramana, D.E. Burton, M. Shashkov, and P.P. Whalen. The construction of compatible hydrodynamics algorithms utilizing conservation of total energy. *J. Comput. Phys.*, 146:227–262, 1998.

[4] E.J. Caramana and M.J. Shashkov. Elimination of artificial grid distortion and hourglass-type motions by means of Lagarangian subzonal masses and pressures. *J. Comput. Phys.*, 142:521–561, 1998.

[5] C. Ferenbaugh. PENNANT: An unstructured mesh mini-app for advanced architecture research. *Concurrency Computat.: Pract. Exper.*, 27:4555–4572, 2015.

[6] C. Ferenbaugh. Performance evaluation of unstructured mesh physics on advanced architectures. *Proceedings of IEEE Cluster 2015*, Chicago, IL, 721-728, 2015.

[7] W.F. Noh. Errors for calculations of strong shocks using an artificial viscosity and an artificial heat flux. *J. Comput. Phys.*, 72:78, 1987.

[8] L.I. Sedov. Similarity and Dimensional Methods in Mechanics. Academic Press, New York, 1959.

[9] K.B. Wallick. Temporary triangular subzoning (TTS), in REZONE: A method for automatic rezoning in two-dimensional lagrangian hydrodynamics problems. Technical Report LA-10829-MS, Los Alamos National Laboratory, Los Alamos, NM 1987.

A Version Log

0.9 February 2016
Added leblancx64 problem. Added energy check diagnostic for verifying large problems.

0.8 November 2015
Added multi-node test problems. Added information for APEX benchmark testing.

0.7 February 2015
Further optimizations for MPI+OpenMP.

0.6 February 2014
First MPI version. MPI capability is working and mostly optimized; MPI+OpenMP is working but needs optimization. Replaced GMV mesh reader with internal mesh generators. Added QCS velocity difference routine to reflect a recent bugfix in FLAG. Increased size of big test problems.

0.5 May 2013
Further optimizations.

0.4 January 2013
First open-source release. Fixed a bug in QCS and added some optimizations. Added Sedov and Leblanc test problems, and some new input keywords to support them.

0.3 July 2012
Added OpenMP pragmas and point chunk processing. Modified physics state arrays to be flat arrays instead of STL vectors.

0.2 June 2012
Added side chunk processing. Miscellaneous minor cleanup.

0.1 March 2012
Initial release, internal LANL only.

B Copyright and License Information