DGM

1.2.10

Generated by Doxygen 1.8.10

Sat Jul 25 2015 11:07:15

Contents

Chapter 1

Introduction and Overview

1.1 Overview

This is the Discretization ToolKit (DTK) that includes a module for discontinuous Galerkin (DGM) and the beginnings of a module for finite-difference (FDM). In the future, other discretizations such as finite volume, finite element and spectral element are envisioned.

Starting with DTK, one can readily build modeling and simulation applications and an example is the Reo appliation that is built on DTK::DGM and supports hybrid, one-, two- and three-dimensional discretizations for: Adv_Diff, Burgers, Euler, LinEuler_quasi, Navier_Stokes,Euler3d, and Navier_Stokes3d equations. Note that not all PDEs can be solved at all spatial dimensions so see the particular module to learn more.

Reo/DGM is designed using an object-oriented approach to PDE simulation and optimization/inversion. The class heirarchy begins at the highest level with the Problem class which defines a simple forward problem on a space-time Domain, Ω. One can also derive off of Problem to generate optimization, control, or error estimation Problems and examples of each are provided.

The next level in the heirarchy is the Domain class that holds a space-time domain that logically contains a prescribed set of physics. The physics must be an equivalent first-order PDE suitable for discontinuous Galerkin methods of the form

$$\boldsymbol{U}_{,t} + \boldsymbol{F}_{i,i}(\boldsymbol{U}) - \boldsymbol{F}^v_{i,i}(\boldsymbol{U}) = \boldsymbol{S}$$

where $\boldsymbol{U}$ is the solution vector, which in DGM notation is called a vField (short for vector Field). The convective flux in the i^{th} direction is $\boldsymbol{F}_i(\boldsymbol{U})$ and the diffusive (or viscous) flux in the i^{th} direction is $\boldsymbol{F}^v_i(\boldsymbol{U})$. Note that derivatives are denoted by subscripts following a comma with summation over repeated indices and t denotes time.

The Domain is an (partially) abstract class that one derives off of in order to implement a particular system of PDEs. For example, Euler derives off of Domain and implements the 2d compressible Euler equations of fluid dynamics. Likewise, Navier_Stokes derives off of Euler and adds the Navier-Stokes diffusion approximation for molecular viscosity to the Euler equations.

Each Domain, holds at least two vector_Field objects, $\boldsymbol{U}$ and $\tilde{\boldsymbol{U}}$ where $\boldsymbol{U}$ is the primary solution variable and $\tilde{\boldsymbol{U}}$ is a temporary solution vector, typically used to hold the solution value at the previous time step. Each vector_Field object is actually a lightweight container of Field objects. Each Field contains a list of Elements that define the spatial representation of that Field component. In general, the Element representations for each component can be different, although in practise the Element topology is usually the same for each component but the polynomial order of the representation may differ. Element itself is also a base class that one derives from to implement particular elements such as Line, Quad, Tri, and Hex.

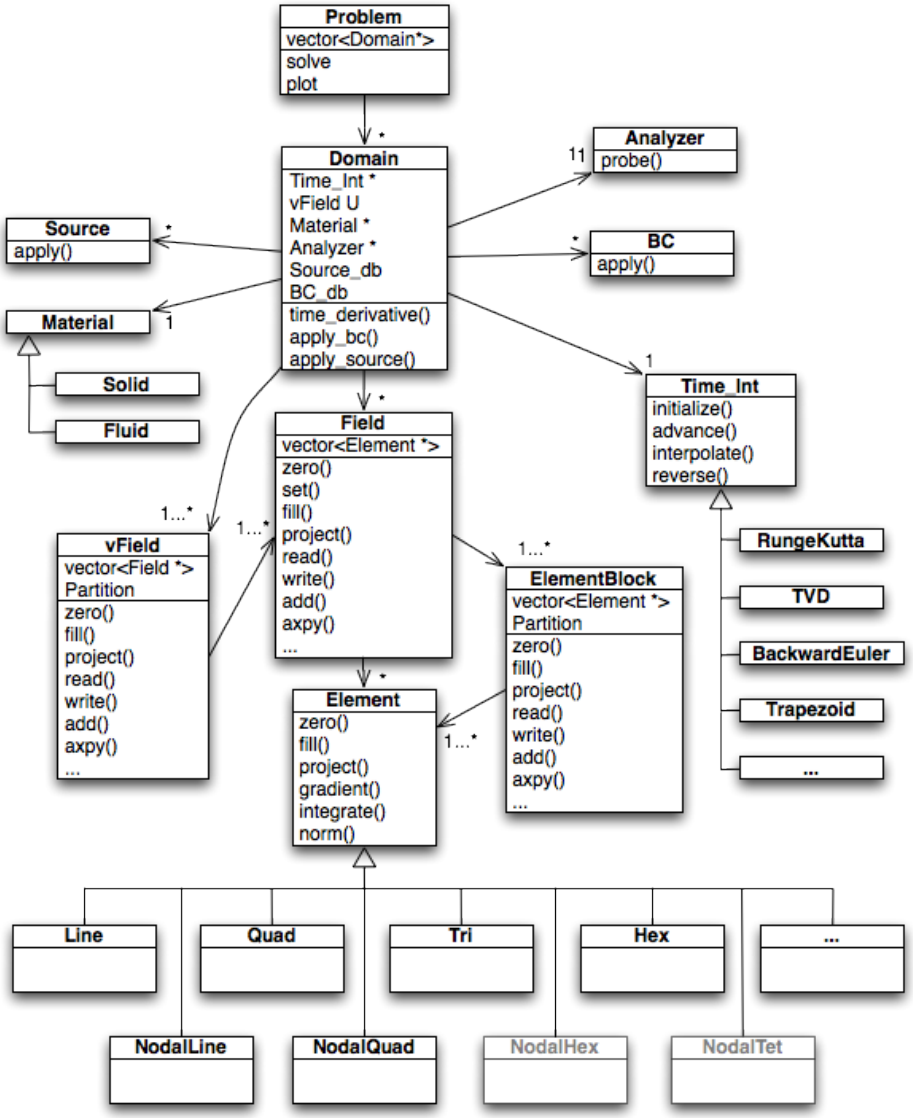

Figure 1.1: DGM overall UML

Domain objects also hold an instance of the DGM time advancement class called Time_Int. Time_Int is a base class that one derives from to implement a variety of time advancement methods including Runge_Kutta, TVD, Multistep and implicit methods such as Backward_Euler and Trapezoidal. Time_Int also, when compiled with Trilinos, support Trilinos enabled DGM::Backward_Euler and DGM::Trapezoidal. Since the Time_Int class defines the interface for all DGM time integrators, it is very easy to add a new time integration method since it is entirely self-contained by providing the Time_Int interface.

The Domain also holds both a Material object that defines the material properties for the particular PDE and an Response object that can query the solution in a number of flexible ways. The Material base class makes it very easy to develop new material types in either liquid, gas, solid or combinations. Finally, the Domain holds a database of both boundary conditions and source terms that allows for a very flexible mechanism to define and enforce new boundary conditions (BC) and source terms (Source). Likewise, all the code associated with a given BC or Source is easily localized making it much easier to add, revise, and maintain boundary conditions and source terms, which, in the author's experience is one of the major challenges and weaknesses of many existing simulation toolkits.

1.2 Optimization and Inversion

The true power of DGM is that it is designed from the ground up to support adjoint-based optimization. This is done in such a way that the discrete problem setup mimics the continuous formulation and setup for optimization problems. In the following discussion, we introduce some of the key classes within the optimization formulation.

The State interface defines a base class that a Domain may derive off of to provide the general functionality needed for the forward problem. Analogously, Domains must also implement the Adjoint interface. The Objective class provides the concept of an objective or merit function that is typically minimized. The Objective class is implemented in a very general way as a database of Objective::Entry that each hold an observation (Obs) object. The Obs base class provides the basic functionality of an observation term within an objective function including the ability to compute its cost, norm, terminal_cost etc.

Similarly, there is the Control class that is a database of Control::Entry that each hold and manage a Ctrl object. Any quantity (parameter, source, boundary condition) that one wishes to control as part of an optimization problem must derive off of Ctrl and provide its basic funtionality.

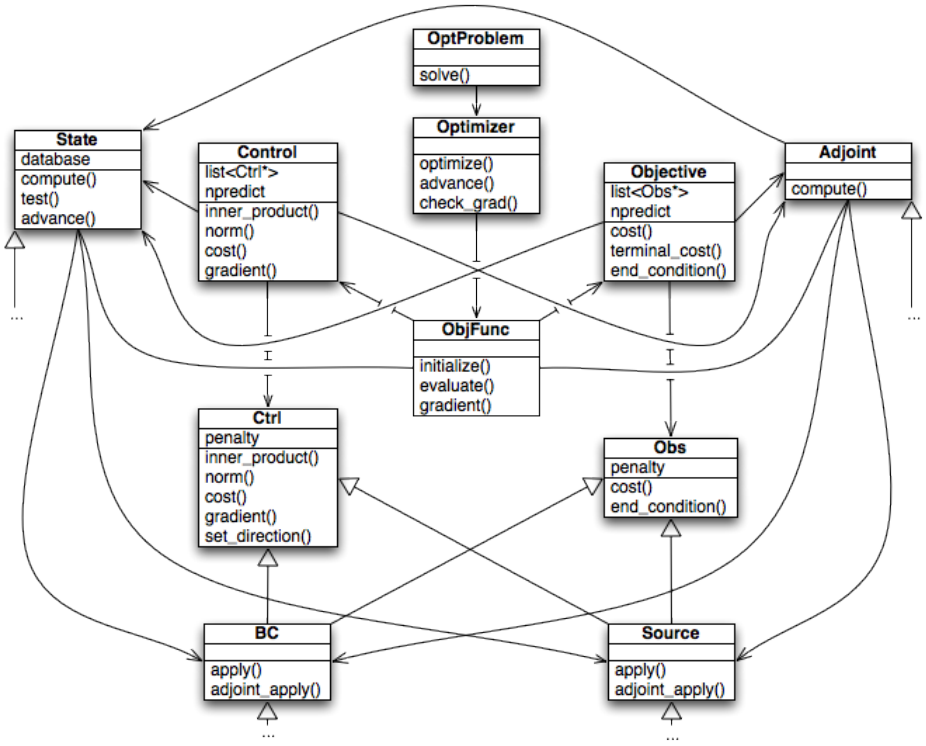

Figure 1.2: DTK Optimization UML

Given the objective oriented design, it is relatively easy to introduce both new Obs and Ctrl objects and this was one of the major thrusts of the overall DGM code design.

In order to form the complete objective function, that involves both observations and controls, we use the ObjFunc class which literally takes both a Control database and an Objective database object and assembles them into a unified objective function that can evaluate itself. In particular, the ObjFunc has knowledge of the underlying Time_Int object so that time integrals in the objective function are computed consistently.

The final ingredient in the DGM optimization framework is the Optimizer class which provides a simple, yet general interface to large-scale optimization algorithms and error estimation. In order to implement a new Optimizer in DGM,

you simply derive off of the base Optimizer class and provide concrete implementations for several methods. DGM supports BlackBox optimization which provides a general DAKOTA-style interface. DGM also has a native nonlinearCG Optimizer as well as an adjoint-based error-estimation (ErrorEst) capability. Finally, there is a limited memory BFGS Optimizer, however, this remains a work in progress and should be used with caution.

1.3 Directory Stucture

The DGM source tree is organized as:

- `src` - All source and headers are located here
- `util` - Some useful utilities (dgm2tec, dgm2p3d, dgm_clean, dgm_test)
- `regression` - Beginnings of a regression test harness with scripts
- `runs` - A large number of sample runs (try 1d, 2d, cyl/cyl)
- `examples` - Examples demonstrating advanced capabilities
- `docs` - A collection of documents (some incomplete) describing DGM
- `math` - Some useful Mathematica notebooks
- `matlab` - Some useful Matlab scripts
- `test` - Some stubs of tests and trial code

1.4 Executables: Drivers, Utilites and Scripts

DGM is setup to be used primarily as a library. As such, there are a number of DGM drivers that each exercise different DGM capabilities. The following is a partial list of DGM drivers and utilities. Each of these builds into an executable with the extenstion `cpp` replaced by `exe`.

1.4.1 Drivers

- `dgm.cpp` - forward solver
- `dgm_opt.cpp` - optimization solver using native optimizers
- `dgm_dak.cpp` - DAKOTA direct interface
- `dgm_me.cpp` - Trilinos::ModelEvaluator direct interface
- `dgm_err.cpp` - error estimation solver
- `dgm_ctl.cpp` - feedback control solver
- `dgm_tdd.cpp` - time-domain-decomposition solver (deprecated)

1.4.2 Utilities

- `dgm_mesh.cpp` - mesh generator (1, 2 and 3-d rectangular meshes)
- `dgm_n2e.cpp` - pre-processor (convert nodal to element connectivity)
- `dgm_post.cpp` - post processor (generates Tecplot or Plot3d files)
- `dgm_loc.cpp` - locate the elements on which nodes live
- `dgm_cpost.cpp` - optimization and control post processor
- `dgm_stats.cpp` - statistics post-processor for turbulence simulation

1.4.3 Scripts

The `dgm/util` directory has a number of scripts that can simplify the use of the DGM utilities for common use cases. These scripts set common commandline options and use the unix shell to loop over multiple files

- `dgm2tec` - converts `rst` files to Tecplot `dat` files
- `dgm2p3d` - converts `rst` files to Plot3d `xyz` and `q` files
- `dgm2aux` - converts `aux` files to Tecplot `dat` files
- `dgm_clean` - cleans output files from a DGM run directory
- `dgm_test` - executes multiple DGM `tst`
- `ctl2tec` - converts `ctl` files to Tecplot `dat` files
- `ctl2aux` - extracts the gradient from the ctl and converts to Tecplot
- `exo2ien` - converts ExodusII mesh to IEN mesh (ready for `dgm_n2e.exe`)
- `fv2tec` - converts a finite-volume `rst` to Tecplot `dat` file

1.5 Copyright and License

DGM is released as open source under the following `license`.

1.6 Primary Contact

S. Scott Collis, Computational Science and Mathematics Sandia National Laboratories, Albuquerque, NM 87175 (505) 284-1123, `sscoll@sandia.gov`

1.7 Contributors

Noel Belcourt, Guoquan Chen, Scott Collis, Axel Gerstenberger, Kaveh Ghayour, Eric Lee, Curt Ober, James Overfelt, Srinivas Ramakrishnan, Guglielmo Scovazzi, Bart van Bloemen Waanders, and Joseph Young.

Chapter 2

DGM Users Guide

This is the DGM Users Guide which provides detailed instructions on how to build, setup, run and analyze the results from DGM. The following quick links will take you directly to the section of your interest.

2.1 Building DGM

The DGM build system and many of the run scripts associated with DGM require that you set the `DGM_DIST` environmental directory to point to the distribution directory of your DGM source tree. On my system I use

```
> setenv DGM_DIST $HOME/dgm
```

where we have assumed that DGM was installed in your home directory and that you are using [t]csh.

The following build instructions give the basics for both serial and MPI-parallel enabled versions of DGM. More detailed installation instruction can be found at `$DGM_DIST/Install`.txt and the reader is encouraged to consult that document.

2.1.1 Serial DGM Build

The simplest procedure for building DGM is as follows. First, make sure that you have the required third party libraries available in `$HOME/local`. Then, do the following

```
> cd $DGM_DIST
> mkdir GCC
> cd GCC
> ln -s ../src/Makefile .
> make -j2 dgm
```

where the `-j2` tells make to do a parallel build on 2 processors. You should set this to be close to the number of cores you have on your system. This should (at least under Linux and Darwin) leads to a GCC (Gnu Compiler Collection) build of DGM. You can make other targets including many of the drivers and utilities listed at Executables: Drivers, Utilites and Scripts. An easy way to see what make targets are supported is to simply type `make` which will return an output similar to

```
DGM Makefiles:

Run make with one of the following options:
a) dgm.....dgm optimized solver
b) pdgm....dgm parallel optimized solver
```

```
 c) mdgm....dgm maximally optimized solver
 d) pmdgm...dgm parallel maximally optimized solver
 e) ddgm....dgm debug solver
 f) mesh....simple mesh generator
 g) n2e.....pre processor
 h) post....post processor
 i) stats...statistics post processor
 j) clean...remove all objects
 k) docs....make Doxygen documentation
 l) opt.....DGM with optimal control
 m) popt....parallel DGM with optimal control
 m) dopt....debug DGM with optimal control
 n) tdd.....time-domain decomposition
 o) cpost...post processor for control output
 p) diff....norm of the difference of two rst files
 q) all.....make all dgm serial codes
 r) pall....make all dgm parallel codes

 ARCH = Darwin-i386, OSVER = 8.11.1, BISON = 1.28
```

To quickly build all major DGM targets, simply use

```
make -j8 all
```

2.1.2 Parallel DGM Build

DGM also supports distributed-memory parallel execution using MPI communication. To build an MPI-enabled version of DGM do the following

```
> cd $DGM_DIST
> mkdir GCCp
> cd GCCp
> ln -s ../src/Makefile .
> ln -s $HOME/local/openmpi mpi
> make -j2 pall
```

where we have assumed that you are using a version of OpenMPI that is available in your `$HOME/local` directory. Of course, you can use any functional version of MPI that uses a consistent set of compilers that you used for the third-party libraries in `$HOME/local`. Note that we have used the `pall` target to automatically build all MPI-enabled targets.

2.1.3 Shared Libraries

A common issue when building and running DGM is that some third-party libraries may default to using shared (or dynamic) libraries which are resolved at runtime. For example, under Linux you might need to set

```
> setenv LD_LIBRARY_PATH $HOME/local/lib:$HOME/local/gsl/lib
```

in you `.cshrc` file. In Darwin (Mac OS-X) you would set

```
> setenv DYLD_LIBRARY_PATH $HOME/local/lib:$HOME/local/gsl/lib
```

2.2 DG Formulation

This section provides a brief background on discontinuous Galerkin methods.

2.3 Generating Meshes

Two of the required files are the Mesh File and the Connectivity File, and they can be generated by the rectangular mesh generator, `dgm_mesh.cpp`, or by Converting Exodus Meshes.

2.3.1 Rectangular Mesh Generation

To generate a rectangular mesh, the utility `dgm_mesh.exe` can be used, and help can be obtained by

```
> dgm_mesh.exe -help
Simple DGM mesh generator
================================================
Usage:          dgm_mesh.exe [Options]
================================================
Options:         Description
================================================
-nsd <int>       Number of dimensions
-nx <int>        Elements in x
-ny <int>        Elements in y
-nz <int>        Elements in z
-Lx <double>     Length in x
-Ly <double>     Length in y
-Lz <double>     Length in z
-x0 <double>     x offset
-y0 <double>     y offset
-z0 <double>     z offset
-Cx <double>     Stretch in x
-Cy <double>     Stretch in y
-Cz <double>     Stretch in z
-Dx <int>        Stretch dir x [-1:2]
-Dy <int>        Stretch dir y [-1:2]
-Dz <int>        Stretch dir z [-1:2]
-theta <double> Rotation about y-axis
-phi <double>   Rotation about z-axis
-r <string>     Root filename
```

The following will generate a 2D quadrilateral mesh with 850x175 elements over the domain $(x_{\text{min}}, x_{\text{max}}) = (0, 17000)$ and $(y_{\text{min}}, y_{\text{max}}) = (0, 3500)$.

```
> dgm_mesh.exe -nsd 2 -nx 850 -ny 175 -x0 0 -y0 0 -Lx 17000 -Ly 3500 -r root
```

Three output files are produced from `dgm_mesh.exe`: *root.con* (the Connectivity File); *root.dat* (a Tecplot ASCI↩ I mesh file); and *root.msh* (the Mesh File) where `root` is the root filename specified using the `-r root` parameter. If no root name is supplied, `dgm_mesh.exe` will use the name `new`.

Note that DGM uses the native coordinate system shown here

2d problem in DGM are *always* in x, y while 1d problems are always in x. Users must account for this when using `dgm.mesh.exe`.

2.3.2 Converting Exodus Meshes

Often one may wish to use an unstructured mesh produced by a mesh generation package (e.g., Cubit). One format is the Exodus database, (i.e., `root.gen` or `root.exo`), which contain information on the node locations, nodal connectivity, element types, sidesets, nodesets, attributes, ... This information can be converted to

1. Connectivity File (`root.con`, ASCII, and/or `root.cn`, binary)
2. Mesh File (`root.msh`, ASCII, and/or `root.grd`, binary)

3. Boundary Condition File (`root.bc`)
4. Element Order File (`root.ord`)
5. Curved Side File (`root.crv`)

using `exo2ien`, `dgm_n2e.exe` and dgm_test_support.py scripts. There are several steps to create these files.

2.3.2.1 Using exo2ien

Starting with an Exodus mesh (e.g., `root.gen` or `root.exo`), one can convert it to ASCII versions of the mesh information, using `exo2ien`.

```
> exo2ien root.gen
```

`exo2ien` creates several files

1. an attribute file, `att.asc`
2. an element connectivity file, `ien.asc`
3. a sideset file, `side.asc`
4. a nodal coordinate file, `xyz.asc`.

Attribute File: `att.asc`

The `att.asc` file contains the attributes specified within Cubit. An attribute is simply an integer or real defining some quantity on a block of elements. Generally it is assumed that all element blocks have the same number of attributes, and the order of the attributes is important for utilities in `dgm_test_support.py` to work correctly.

The current convention is the first attribute is used for specifying the information about the elements in the element block: curve flag, polynomial order, and the quadrature order. These values are encoded into a single integer using the following formula

```
attribute 1 = 10000*c + 100*p + q
```

where `c` is the curve flag (0 -> an affine element and 1 -> a non-affine element), `p` is the polynomial order, and `q` is the quadrature order. Using the `dgm_test_support.py` (`mkcrv_cubit` and `mkord_cubit`), these values can be used to create the curve file (`root.crv`) to handle skewed elements, and the order file (`root.ord`) to use variable polynomial and quadrature order. If `p` or `q` are set to zero the default value specified in `root.inp` will be used.

One convention currently used is the attributes are ordered as follows:

Attribute	Value
1	Encoded element information (see above)
2	Material density
3	Primary wave speed
4	Secondary wave speed

Element Connectivity File: `ien.asc`

The `ien.asc` file is the traditional finite-element connectivity listing, where for each element the nodal IDs are listed (a.k.a. nodal connectivity). `ien.asc` is used by dgm_n2e.exe to create the edge connectivity files (Connectivity File; `root.con` and/or root.cn) needed by DGM.

Sideset File: `side.asc`

The side.asc file contains information about the sidesets specified within Cubit, and is used to set boundary conditions on the mesh boundaries using dgm_n2e.exe and optionally mkbc_cubit (Boundary Condition File).

Nodal Coordinate File: xyz.asc

The xyz.asc file has the (x, y, z) coordinate locations for the nodes in the Exodus mesh. xyz.asc is used by dgm_n2e.exe to create the mesh file (Mesh File; root.msh and/or root.grd) needed by DGM.

2.3.2.2 Using dgm_n2e.exe

dgm_n2e.cpp uses ien.asc, side.asc and xyz.asc generated by exo2ien to produce the Mesh File, Connectivity File and Boundary Condition File. Again dgm_n2e.exe -help will provide basic man page information. Executing the following command

```
> dgm_n2e.exe -r root TFE
```

produces four files:

1. a dual-graph for use with a graph partitioner such as Zoltan or Metis/ParMetis, *ien.asc.dgraph*
2. the Mesh File, root.msh (ASCII) and/or root.grd (binary)
3. the Connectivity File, root.con (ASCII) and/or root.cn (binary)
4. the Boundary Condition File, root.bc

Note that the last argument to dgm_n2e.exe indicates the input file format where TFE is the traditional finite-element format generated by exo2ien. Other formats are available including UCD for GridGen mesh format and NEU for Gambit mesh format.

The Mesh File contains the nodal locations for each element, and can be read in two formats: root.msh (ASCII) and root.grd (binary).

The Connectivity File describes the edge connectivity and similarly has two formats: root.con (ASCII) and root.cn (binary).

2.3.2.3 Using mkbc_cubit, mkord_cubit, and mkcrv_cubit

The Boundary Condition File generated by dgm_n2e.exe introduces placeholder variables (i.e., bc1, bc2, ...) for each sideset. One needs to replace these with the appropriate boundary conditions for the given physics (e.g., absorbing boundary condition, Z; Dirichlet boundary condition, D; flux boundary condition, F; prescribed boundary condition, S; wall boundary condition, W; or user defined Boundary Condition Type File). Using scripts from dgm_test_↩ support.py (mkbc_cubit and mkbc_cubit_dict), the above replacement of bc1, bc2, ... with the appropriate boundary conditions can be automated. **Note:** These boundary condition codes are examples only. The actual values depend on the physical domain being used.

To complete the conversion of Exodus meshes, the optional Element Order File and Curved Side File can be created from the first attribute in att.asc. Using mkord_cubit in dgm_test_support.py, the Element Order File, root.↩ ord, can be made where the polynomial and quadrature order for every element within each element block will be set according to the value of its first attribute (see above for details).

Lastly, with most unstructured meshes, quadrilateral elements are generally be skewed, that is they will not have an affine mapping. For efficiency, DGM assumes that every Quad element is affine unless otherwise indicated. For a general, unstructured quadrilateral mesh, each element will need to be defined as a general straight-sided element using the *root.crv* file. If the Curved Side File is not present, the solution will not be incorrect and will likely lead to unstable simulations. Using mkcrv_cubit in dgm_test_support.py, the Curved Side File, the first attribute can be used to create the curve file, root.crv. Please refer to Curved Side File for details on the root.crv format.

2.4 Running DGM

- DGM Input Specification
- Examples

2.5 Analyzing DGM Output

- DGM Output Specification

A bibliography for the Reference Manual is provided in:

- DGM Bibliography

Chapter 3

DGM Developers Guide

This is the DGM Developers Guide which provides information on various aspects of developing in DGM and developer's description of design choices. The following are quick links to some documented topics.

3.1 32 and 64 bit

The use of 32 and 64 bit integer types is set by the types Size and Ordinal. The definitions are

1. Size: For integral types that index quantities across the whole mesh/machine, which may require integer values that are larger than a 32-bit representation.
2. Ordinal: For integral types that index quantities local to an MPI rank, and may or may not require 64-bit.

These types should used instead of "int" and "unsigned" directly. The nice thing about it is that with this approach all combinations should (must) work. Our standard builds still use

Size = int
Ordinal = int

but we also build nightly with

Size = uint64
Ordinal = uint64

Even with that, there are still a number of places in the code where we have$*$ to use a boost::numeric_cast$<$int$>$ since:

1. MPI is hardwired to use int for many things
2. A good chunk of Trilinos (old stack) is stuck with int
3. STL containers always use size_t but this type varies by platform/OS

When we really run big problems, a likely setup for us will be

Size = uint64_t
Ordinal = size_t

since size_t is designed to be large enough to address things locally whereas we almost certainly need a full 64bits for huge models globally.

Chapter 4

DGM Input Specification

4.1 Input File Summary

The DGM input files for a particular run case are indentified by each having a unique root file name. If we look in the `dgm/runs` directory you will see many examples of input files each identified by a unique root file name, e.g. `1d` or `vortex` or `bump`. Given a root name for the input files, there are both required and optional files that define the input parameters for a DGM run. The following table lists the extensions for each of these files, provides a brief description and indicates whether the file is required or optional. If a file is optional and that the file exists, it will be read by DGM and that feature/capability will be activated. Note that the example runs available in `dgm/runs` is a great place to explore DGM capabilties and the input files and syntax required.

Extension	Full Name	Purpose	Status
`inp`	`Input file`	Sets overall problem parameters	**required**
`msh`	`Mesh file`	Specifies element geomtry	**required**
`con`	`Connectivity`	Shows how element sides are connected	**required**
`ic`	`Initial condition`	Specifies the initial solution values	**required**
`bc`	`Boundary conditions`	Prescribes the boundary conditions on element sides	**required**
`bct`	`Boundary condition types`	Defines new boundary condition types	optional
`ord`	`Order file`	List of elements with custom polynomial order and quadrature	optional
`src`	`Source file`	Defines the source terms	optional
`anl`	`Analysis file`	Defines the Response analysis which extracts solution values	optional

ref	Referece solution	Defines a reference solution in rst format for testing	optional
crv	Curved side control	Defines curved side types and which elements sides they live on	optional
imp	Implicit solver control	Specifies parameters to implicit solvers	optional

4.2 DGM Input File

The primary input file for DGM is denoted by root.inp and is used to set many of the overall input parameters that control the execution of DGM. The file uses a very simple but robust parser. Input parameters are specified by name with the syntax name = value. The parser ignores white space and it also ignores all information on a line that follows a "#" sign. An example inp file for the dgm/runs/1d case is:

```
# DGM test case
#
# One-dimensional advection diffusion
#
# Author:  S. Scott Collis
#
eqntype  = 0       # advection diffusion
inttype  = 2       # 3-step multistep method
p        = 10      # polynomial order
Nt       = 1000    # number of time steps
Ntout    = 1000    # restart output interval
dt       = 0.004   # time step
cx       = 1.0     # convection velocity in x-direction
vis      = 0.01    # viscosity coefficient
```

Notes:

1. Enumerations currently must be assigned a numerical value
2. If a parameter is defined more than once, the last value is used
3. There is only limited checking and validation of parameters
4. If a parameter is defined that is not used it will be ignored with no warning
5. Each parameter has a default value, the inp file overrides the default
6. Some parameters may be set as commandline arguments. These override the values in the inp file
7. To see what parameters can be set via the commandline, type dgm.exe -help 1d where 1d should be replaced by your root filename.

Example -help output:

```
> cd $DGM_DIST/runs
> dgm.exe -help 1d
================================================================
Discontinuous Galerkin Solver
================================================================
Usage:    dgm.exe [Options] run_name
----------------------------------------------------------------
Options:  Description
----------------------------------------------------------------
-help     Detailed help
----------------------------------------------------------------
```

```
Problem Class Options
-------------------------------------------------------------
    None currently supported
-------------------------------------------------------------
Domain Class Options
-------------------------------------------------------------
-p        Polynomial order
-q        Quadrature order
-inttype  Time integration
-dt       Time step
-CFLmax   Max allowable CFL
-dtmax    Max allowable time step (0=ignore)
-Nt       Number of time steps
-Ntout    Restart file output interval
-ntout    Minor information output interval
-Ntprb    Response probe output interval
-tf       Final time
-vis      Diffusion coefficient
-stab     Diffusion stabilization parameter
-bstab    Boundary stabilization parameter
-noIO     Turn off Domain IO
-------------------------------------------------------------
Adv_Diff Class Options
-------------------------------------------------------------
-cx         Wave speed in x-direction
-cy         Wave speed in y-direction
-cz         Wave speed in z-direction
-fv         Finite volume reconstuction (0=none)
-cfield     File name for convective field
-------------------------------------------------------------
```

4.3 Mesh File

The required `msh` file is used to specify the number of elements, the number of space dimensions, the element topology and the element nodal coordinates. A sample of the file format is given here:

```
** Mesh Data **
2 1 Ne Nsd
Element 0 Line
 2.5000000e-16  2.0000000e+00
Element 1 Line
 2.0000000e+00  4.0000000e+00
```

Notes:

1. In general, lines surounded by "$**$" are **required** comment lines.
2. `Ne` is the number of elements
3. `Nsd` is the number of space dimensions
4. Comments denoted by "#" are *not* currently supported in this file
5. Elements do *not* have to be listed in numerical order
6. The keyword "Element" is currently required followed by its id and type
7. Each Element then reads the required information to define its geometry
8. Line elements are assumed to always be in x and their geometry is fully specified by the x-coordinates of the end points.

9. For a multi-dimensional problem, one first lists the x-coordinates, then the y-coordinates, and in three-dimensions then the z-coordinates.

10. Keywords are not case-sensitive

11. Here is a two-dimensional example from `$DGM_DIST/runs/2d.msh`

```
** MESH DATA **
4 2 NE NSD
ELEMENT 0  QUAD
 0.0000000e+00  2.0000000e+00  2.0000000e+00  0.0000000e+00
 0.0000000e+00  0.0000000e+00  2.0000000e+00  2.0000000e+00
ELEMENT 1  QUAD
 2.0000000e+00  4.0000000e+00  4.0000000e+00  2.0000000e+00
 0.0000000e+00  0.0000000e+00  2.0000000e+00  2.0000000e+00
ELEMENT 2  QUAD
 0.0000000e+00  2.0000000e+00  2.0000000e+00  0.0000000e+00
 2.0000000e+00  2.0000000e+00  4.0000000e+00  4.0000000e+00
ELEMENT 3  QUAD
 2.0000000e+00  4.0000000e+00  4.0000000e+00  2.0000000e+00
 2.0000000e+00  2.0000000e+00  4.0000000e+00  4.0000000e+00
```

4.4 Connectivity File

The required `con` file describes how element sides are connected. Note that this is different from standard finite element method connectivity which normally shows how element nodes are connected. An example `con` file from `$DGM_DI↩ST/runs/1d.con` is

```
** Connectivity **
2 ne
E       0       0       1       1
E       0       1       1       0
E       1       0       0       1
E       1       1       0       0
```

Notes:

1. The first line is a required comment line and is ignored

2. The second line gives the total number of elements, `ne`

3. The remaining lines are of the form "E" followed by the element id, side id, linked element id, linked side id

4. Not that it is okay (in fact it is the default of `dgm_mesh.exe`) that sides be doubly connected in order to specify periodicity

5. If a side is *not* connected to another side, then it *must* have a boundary condition specified on that side (see Boundary Condition File).

4.5 Initial Condition File

The required `ic` file describes how the initial solution variables are initialized. There are a variety of ways that initial conditions can be specified, in fact, each physics Domain can and likely does add some of their own. However, there are several default methods for setting initial conditions that are described here. The following is a sample `ic` file from `$DGM_DIST/runs/1d.ic`.

```
** Initial Conditions **
Given
u(x) := 1.0 - cos(1.0 * PI * x);
```

Notes:

1. The first line is a required comment line and is ignored
2. The second line declares the *type* of initial condition
3. default types include `Given` and `Restart`
4. The `Given` initial condition is a simple but powerful way of specifying the solution variables in terms of expressions of the spatial coordinates. On the line following the keyword "Given" DGM reads expressions (one per line) for each of the dependent variables. **Order matters**. The dependent variables are read in exactly the same order that they are allocated in DGM. The expression parser looks for ":=" and then takes the expression as all characters to the right of this symbol. Therefore, you can think of characters to the left of ":=" as a description/comment for each expression. Typically we use the symbol for the dependent variable. A ";" is required at the end of the expression
5. The `Restart` initial condition reads the solution variables from a DGM `rst` (restart) file. This provides a easy method to continue a simulation from a prior run. The following examples show how one could continue the `1d` run starting from the solution in `1d.rst`.

```
** Initial Conditions **
Restart
1d.rst
```

4.6 Boundary Condition File

A typical boundary condition file is shown below. It is similar in form to the `con` file. After the required comment line, the number of boundary condition sides `nbc` is given followed by a listing of boundary conditions in the form

```
BCname ElementID SideID [Value]
```

The `BCname` is physics Domain dependent which each Domain assigning particular names to its supported boundary conditions. That said, "D" is often used for Dirichlet and "F" for flux boundary condition. "Z" is commonly a zero state boundary condition and the "S" is used for a prescribed state (generalized Dirichlet) boundary condition. Note that new boundary conditions can be defined at runtime using the `bct` file.

```
** Boundary condition data **
0 nbc
D       0       0       0.0
F       1       1       0.1
```

4.7 Boundary Condition Type File

The `bct` file is used ot declare new boundary condition types at runtime that can then be used in the `bc` file. The idea is that DGM provides the based types of boundary conditions that are appropriate for each supported physics and then the use can instantiate specific versions of these boundary conditions in the `bct` file along with appropriate `BCname` tags that can be used to associated boundary conditions with specific element sides in the `bc` file.

```
State S_M=0.2
{
  1.0 1.0 0.0 45.14285714285715
}
State S_M=0.3
{
  1.0
  1.0
  0.0
  20.34126984126985
}
Isothermal_Wall W_M=0.2
{
  1.0 45.14285714285715
}
Isothermal_Wall W_M=0.3
{
  1.0 20.34126984126984
}
Isoflux_Wall AW
{
  0.0
}
```

4.8 Element Order File

The `ord` file defines the element polynomial order and quadrature order for each element individually. After the required comment line, there is simply a list of `ElementID`, `Porder`, and `Qorder` for each element. Only a subset of elements need be listed (or none at all) and they do not have to be in numerical order. Note that `Qorder` may be zero which uses the internally computed quadrature for the selected value of `Porder`.

```
** Order data **
0 10  0
1  5 12
```

4.9 Source-term File

Defines the body source terms. The supported types are physics dependent although there are three default types ConstSource, FileSource, FuncSource. An example of a FuncSource input specification is

```
Function f { f := sin(x); }
```

Which defines a function `f(x)` that is a sine wave in `x`.

4.10 Analysis File

The `anl` file describes the points where one whats to extract time series traces of the solution using a DGM::Response. An example analysis file is:

```
*** Analysis input ***
2 1
NodeID    NodeX    Nelemts  ElementID
0          1.0        1         0
2          3.0        1         1
```

Notes:

1. First and third lines are required comment lines (they are ignored)
2. In the second line the number of analyzer nodes is 2 and the number of space dimensions is one (i.e. 1d problem).
3. The `NodeX` values are the x-coordinate of the analyzer point
4. `Nelemts` is the number of elements that this point touches. While this can be greater than one in general, DGM requires that this only be one currently.
5. The `ElementID` is the global element number that the point lives in (see the .msh file). For structured meshes this can often be figured out by hand. For more complex meshes there is a utility called `dgm_loc.exe` that can compute the element ids for a list of points.

4.11 Reference Solution File

The reference solution file is used to define either analytic (or high-fidelity computed) solution that can be used as a reference for computing either errors (e.g. for convergence studies) or as a reference solution for evaluating that the code has not changed. In this latter capacity, one might use the `dgm_diff.exe` to compare a recently computed solution with this reference to make sure that the solution has not changed do to code updates. This is commonly done in the `root.tst` scripts that are availabed in `$DGM_DIST/runs`.

4.12 Curved Side File

Important: DGM by default assumes that all elements use an affine mapping from physical space to computational space. This is done to increase efficiency since such a mapping results in a constant determinate of the Jacobian of the mapping over the element. For simplicial elements such as Tri and Tet, the mapping is affine as long as there is not a curved element side. For Quad and Hex the situation is a bit more complex.

In practice, however, one often does *not* have affine elements. The primary case of this with higher-order methods is the presense of a curved side on an otherwise straight-sided element. Another case, which arrises in non-simplicial elements (such as Quad and Hex) is if the sides of the element are not parallel.

A classic example case for both non-affine and curved element sides is given by the `runs/cyl` test case which solves Mach=0.3 inviscid flow over a 2d circular cylinder. The `cyl.crv` file looks like

```
** Curved Sides **
2 Number of curve type(s)

Circle
 arc 0.0 0.0 -0.5

Straight
 skew

144 Number of curved side(s)
0 3  arc
6 3  arc
...
138 3  arc

1 0 skew
2 0 skew
...
143 0 skew
```

where ... denotes lines that have been removed to shorten the file for display in this manual. This curve file declares two types of curved elements, one of type `Circle` which is given the name `arc`, and one of type `Straight` that is given

the name `skew`. The `arc` curved element type is applied to denote the sides of the circular cylinder which is centered at (x,y)=(0,0) and with a radius of 0.5 (the minus sign denotes that the mesh is outside the cylinder. The second curve type, `skew`, is applied to all unstructred, non-affine, quadralateral elements which, for this mesh, are elements 0 through 143 excepting the elments that touch the cylinder surface which are of type `arc`.

4.13 Implicit Solver File

Work in progress...

Chapter 5

DGM Output Specification

5.1 Output Files

The DGM input files for a particular run case are indentified by each having a unique root file name. If we look in the `dgm/runs` directory you will see many examples of input files each identified by a unique root file name, e.g. `1d` or `vortex` or `bump`. Given a root name for the input files, there are both required and optional files that define the input parameters for a DGM run. The following table lists the extensions for each of these files, provides a brief description and indicates whether the file is required or optional. If a file is optional and that the file exists, it will be read by DGM and that feature/capability will be activated. Note that the example runs available in `dgm/runs` is a great place to explore DGM capabilties and the input files and syntac required.

Extension	Name	Purpose	Status
`rst`	`Restart file`	Complete checkpoint/restart file	allways
`dat`	`Tecplot file`	Tecplot files in ASCII format	optional, never in parallel or 3d
`grf`	`Graph file`	A Side connectivity graph for mesh partitioning	always in serial execution
`wgf`	`Weighted Graph file`	Element weights for use in mesh partitioning	always in serial execution
`prb`	`Probe file`	Solution at a probe point as a function of time	Only when specified in an `anl` input file
`rak`	`Rake file`	Snapshot of the solution at each defined probe point	Only when specified in an `anl` input file

5.2 Restart File

The DGM `rst` file has a 10-line ASCII header that can be displayed with the unix `head` command

```
> head 1d.rst
dgm.exe                                 Session
Sun Oct 18 05:52:54 2009                Created
1d.rst                                  Name
2           1           11              Nel, Nsd, Lmax
1000                                    Step
4                                       Time
0.004                                   Time step
0.01                                    Viscosity
```

```
u                                       Field names
Normalized                              Format
```

The remainder of the file is binary data that fully specifies the DGM solution for the conditions described in the header. The files are written in such a way that one can arbitrarily restart DGM from any `rst` file and continue the simulation as it the original execution had never stopped. Furthermore, one can arbitrarily change the polynomial-order (either globally with `p` in the `root.inp` file or locally with a `root.ord` file).

5.3 Tecplot File

Since DGM uses an unstructed, variable polynomial-order discretization, plotting the solution in a consistent manner is an important capability. To this end, we use one of several available third-party tools. A particularly useful tool is `Tecplot` which allows us to visualize the full DG solution on each element in a way that is truthful to the actual DG representation.

When run in serial on 1d or 2d problems, DGM automatically generates Tecplot `dat` files each time it would normally output a `rst` file. However, when run in parallel or for all 3d problems, DGM only outputs `rst` files and one uses a post-processing utility to generate visualization files

5.3.1 Converting to Tecplot format

To generate a Tecplot compatible file from a DGM `rst` file one uses the `dgm_post.exe` utility. Since this is a very common operation and one usually wants to visualize multiple files, there is a simple helper script in `$DGM_DIS↩T/util/dgm2tec` that simplifies the process.

To convert a single `rst` file to a Teplot `dat` file use the command

```
> dgm_post -r root.100.rst root
```

or equivalently

```
> dgm2tec root.100.rst
```

To convert all `rst` files in a given directory, one would use

```
> dgm2tec *.rst
```

Notes:

1. All DGM input files for this run must be in the same directory as the `rst` files for these postprocessing utilities to work
2. The output file is the same name as the input `rst` file but with `rst` replaced by `dat` and these `dat` files are ASCII Tecplot data files. Since the files are ASCII, they can get quite large, making Tecplot mainly useful for 1d and 2d problems but not as useful for 3d setups. We currently use Paraview for 3d visualization (see below).
3. Currently, `dgm_post.exe` can only execute in serial which means that for large problems, again more common in 3d, there may not be enough memory on your system to run `dgm_post.exe`.

5.3.2 Converting to Plot3d format

Instructions for generating Plot3d format files for use with Paraview go here...

5.4 Graph Files

Currently upon serial execution, `dgm.exe` generates a graph file and associated element weights that represent the inter-element connection graph weighted by an estimate of the computational expense for each element. The `grf` file can be input to a tool such as Sandia's Zoltan or Metis/Parmetis to generated a partition input file that DGM will use for subsequent parallel execution.

For example, consider the `$DGM_DIST/runs/cyl` case and run in serial for zero timestep using the command

```
> dgm.exe -Nt 0 cyl
```

This generates the `cyl.grf` file that can be partitioned using Metis

```
> cp cyl.grf cyl
> kmetis cyl 16
```

Which will generate the partition file `cyl.part.16` which is the correct naming convention to be read by DGM for parallel execution on 16 MPI ranks.

If one is using local p-refinement or if there are other features of the problem setup that lead to load-inbalance, then one should used a weighted graph partition. DGM provides the `wgf` file which lists relative weights for each element based on an internal estimate of the computational expense for each element. There is a utility in `$DGM_DIS↩ T/util/weight-graph` that takes `root.wgt` and `root.grf` and outputs a weighted graph file

```
> $DGM_DIST/util/weight-graph cyl.grf cyl
```

will generate a weighted graph file called `cyl` that can be directly partitioned with `kmetis` as described above.

Notes:

1. You will need to run `kmetis` for each core number that you want to run on
2. If you don't supply a `root.part.#` file for the core number that you are using, DGM will use a very simple (and likely non-optimal) internal partition based upon an equipartition of elements on each node based on simple element ids.
3. In the future, we plan to combine Zoltan and/or ParMetis with DGM so that partitions are computed inline.

5.5 Probe File

Describe the DGM probe file format here...

5.6 Rake File

Describe the DGM rake file format here...

Chapter 6

DGM Runs and Examples

6.1 Types of Runs

The `$DGM_DIST/runs` directory holds a number of sample runs that exercise a range of DGM capabilities and provide an intial test harness. That said, the runs do not provide complete coverage of the extensive capabilities within DGM. To execute all the runs use

```
> cd $DGM_DIST/runs
> ../util/dgm_test *.tst | tee results
```

where we have used the `dgm_test` script to execute all `tst` scripts and to summarize the results. After all runs are completed, you should see a message the message

```
******************************
****   ALL TESTS PASSED   *****
******************************
```

If you do not see this message, then one or more tests have failed and you should consult the `results` file to see which test failed.

The tests use the `dgm_diff.exe` utility which does a difference between two DGM `rst` files. This utility computes a variety of different norms of the difference and evaluates the differences between the computed solutions and reference solutions using both absolute and relative tolerances. These tolerances have been set to ensure that tests pass on standard Linux and Darwin systems. However, it is possible that a failure is just do to slightly larger round-off error on a particular platform so that may be the first place to look

If instead, there is a significant difference in the computed and reference solutions, then there is something wrong with your DGM build or you have found a bug in DGM that should be reported.

The following table gives a brief summary of the DGM runs:

```
Name       Physics         Description
-------------------------------------------------------------------------------
1d         Adv_Diff        Simple 1d advection diffusion on a Line mesh
2d         Adv_Diff        Simple 2d advection diffusion on a Quad mesh
3d         Adv_Diff        Simple 3d advection diffusion on a Hex mesh
2dtri      Adv_Diff        Same as 2d but using a triangular mesh
2dtri2     Adv_Diff        Same as 2dtri but with different triangularization
1dlayer    Adv_Diff        Advection diffusion with a boundary layer
1dlayer2   Adv_Diff        Varient of 1dlayer
bump       Euler           Inviscid flow over a bump
burg1d     Burgers         1d Burgers with a shock
burger1d   Burgers         1d Burgers with a shock
c2d        Adv_Diff        Same as 2d but using curved Quad elements
ctri       Adv_Diff        Save at 2dtri but with curved Tri elements
```

```
cvquad     Navier_Stokes   Vortex on curved Quads
cvtri      Navier_Stokes   Vortex on curved Tris
shock      Burgers         1d shock capturing test
tri        Adv_Diff        Advection diffusion on a Tri mesh
vhex       NavierStokes3d  Acoustic wave
vortex     Navier_Stokes   Single vortex with local p-refinement
vquad      Navier_Stokes   Single vortex on a Quad mesh
vtri       Navier_Stokes   Single vortex on a Tri mesh
vtri2      Navier_Stokes   Single vortex on a different Tri mesh
```

6.2 Examples

Currently there are no specific examples but these may following in the future as part of this users guide.

6.3 Regression

The `$DGM_DIST/regression` directory contains scripts and utilities to perform simple regression tests of DGM using the runs described above. Scripts are provided for both Hudson and a simple Bash-based cron regression setup.

The primary DGM regression testing capability is based on the Hudson continuous integration server. The Hudson utilities are located in `$DGM_DIST/regression/hudson`. As part of this set of tools, there is a Python script called `runtests.py` which provides a very flexible way of conducting DGM tests. For example, if in the $DGM_DIST/runs directory, you could do

```
% ../regression/hudson/runtests.py
Wed Jan  5 07:23:27 2011
Test Results from Directory: /Users/sscoll/dgm/runs
Total number of test(s): 47
------------------------------------------------------------------------
  pass         0.74s  1d.tst
  pass         0.60s  1dlayer.tst
  pass         0.60s  1dlayer2.tst
  pass         0.89s  2d.tst
  pass         0.64s  2dtri.tst
  pass         0.61s  2dtri2.tst
  pass         1.95s  2dwave.tst
  pass        19.68s  3d.tst
  pass         4.88s  bump.tst
  pass         0.38s  burg1d.tst
  pass         0.08s  burger1d.tst
  pass         0.59s  c2d.tst
  pass         0.30s  cbshock.tst
  pass         1.67s  ctri.tst
  pass         5.40s  cvquad.tst
  pass         9.09s  cvtri.tst
  pass         0.51s  oned.tst
  pass         0.39s  shock.tst
  pass         1.24s  tri.tst
  pass        51.94s  vhex.tst
  pass         4.62s  vortex.tst
  pass         3.03s  vquad.tst
  pass         7.31s  vtri.tst
  pass         3.13s  vtri2.tst
  pass        93.01s  chan/chan.tst
  pass        40.44s  cyl/cyl.tst
  skipped      0.02s  poisson/poisson.tst
------------------------------------------------------------------------
 Pass: 26    Fail: 0    Skipped: 21    Total: 47

Total Runtime:     253.97s
```

Note that each .tst file includes information that tells the Hudson testing system whether it is `active` as well as additional parameters that should be used during testing. In the example given above, the `poisson` test was skipped since it required a Trilinos-enabled build.

Chapter 7

DGM Bibliography

- S.S. Collis *The DGM Reference Manual*, Sandia National Laboratories, Albuquerque, NM.
- George Em Karniadakis and Spencer J. Sherwin *Spectral/hp elment methods for CFD*, Oxford University Press, 1999.
- E.F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics*, 2nd Edition, Springer, 1999.
- Timothy Warburton, *Spectral/hp Methods on Polymorphic Multi-Domains: Algorithms and Applications*, Ph.D. Thesis, Brown University, 1999.

Chapter 8

DGM Release Notes

8.1 DGM v1.2

8.1.1 Updates in DGM v1.2.10

1. Initial design and implementation of DTK general API
2. Templated update to Polylib
3. Support for 64bit ordinal types
4. New Json input file format
5. Json headers on most binary files
6. New templated DTK::TimeIntegrate class
7. Support for implicit time-integration using Trilinos
8. Online partitioning using Zoltan
9. Initial support for hanging node (non-conforming) elements
10. Nodal Tetrahedral elements
11. Support for Modal, Nodal, and Spetral elements

8.1.2 Features

1. New binary mesh reader that supports MPI-IO. This addresses the first "known issue" in v1.1
2. Now handles C++ exceptions correctly with nice error messages
3. New Python interface for basic DGM::Program and DGM::OptProblem. Includes wrapping of DGM::Control so that you have full access to a basic optimization interface from within Python.
4. Fixed memory errors in InCore/Griewank (issue 2 in v1.1)
5. Testing environment updated to use Jenkins
6. Significant improvements in underlying documentation
7. Improved build scripts with the "-y" option to build shared libraries needed for the Python interface.

8.2 DGM v1.1

8.2.1 Features

1. New Response that allows for sensors anywhere in the spatial domain
2. New InCore storage that greatly reduces computational costs when things can fit in memory
3. New Griewank class that manages incore and recomputation. This remains a "beta" feature in that we are seeing memory problems on some platforms.
4. New non-blocking communication that greatly improves scalability and performance
5. New python-based testing infrastructure that is compatible with Hudson
6. More use of shared-pointers to improve memory management
7. Improved documentation and installation instructions
8. New build scripts that greatly simplify building and managing multiple types of DGM builds
9. Support for Peopt and trust-region methods (beta)
10. Fixed memory leak in Control::Entry and Objective::Entry

8.2.2 Known issues:

1. There is still a serial bottleneck in reading a mesh in parallel. There is a new mesh format and reader underdevelopment that will scale in parallel but this is not yet ready.
2. Griewank has a memory error on some systems.