Developers Guide for Diablo

Jason Hicken

September 14, 2009

Contents

1 Introduction

This guide is intended to bring new code developers up-to-speed as quickly as possible. Diablo is not particularly complicated (yet), but there are a few common ideas that, if understood at the beginning, make learning the code much easier. That said, the first thing you may want to do is read the relevant papers.

Finally, this guide is open source, so feel free to add to it — once you understand the material you are adding.

2 Revision Control

We use git for revision control. I recommend the following websites:

main website: `http://git-scm.com/`

user manual: `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`

short tutorial: `http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html`

The manual provides a nice reference, and I suggest reading the first 4 chapters. If you want to understand what is going on "under the hood," I recommend chapter 7.

Many new students are overly cautious when they get a new code. I was no different when I started my masters. I was afraid that I might change something and mess-up the code for everyone else. Fortunately, this kind of experience should be a thing of the past, at least if you use git properly. Therefore, it is important that you appreciate what version control can do, otherwise you will not take full advantage of it. Here are some things to keep in mind

- When you want to try out something new, **do not** copy the code into a new directory. Just create a new branch using git. Then, if it doesn't work out, just return to where you branched off.
- **Commit frequently!** When you are working along on a development branch, save your changes by making a commit. You can always revert to a previous version. Once you get the hang of it, you'll probably commit at least once a day.
- Keep a master branch that contains a working up-to-date copy of the code, and at least one working branch. Merge changes into the master branch when you are happy with them (at least once a month).
- Merge within the group at least once a month. Historically, we have not been good at this, and the result is a mess.

Smooth and frequent merging of the group code remains a work in progress. The plan is to follow the distributed public repository system advocated by git (see "public git repositories" in Chapter 4 of the manual). For example, suppose you find a bug and fix it in your version of the code. You should then push those changes onto your public repository, and then email the rest of the group that they should pull the update from you repository. Well, that's the plan, we'll see how it goes.

2.1 Directory Structure

When you clone the Diablo git repository, you will get several directories. Here, I give a quick rundown of what's in each of directory. You can probably ignore many of these. Note, third party libraries are stored in directories with all caps.

LAPACK: Linear algebra package. This is included for portability, since most of the systems we use have their own built-in versions of this library (I believe).

METIS-4.0: The metis graph partitioning library. Tried using this to load balance the blocks, but it often did a poor job. Presently not used.

SPARSKIT2: Yousef Saad's krylov solver and preconditioner library. Many ideas from SPARSKIT are used in pKISLib, but few if any subroutines from this library are used anymore.

SNOPT: SQP optimizer library. Presently, my favourite nonlinear optimizer.

pKISlib: parallel Krylov Iterative Solvers Library. Our in-house library of Krylov solvers and preconditioners. Unless you are doing research that involves new preconditioners or solvers, you may never need to look at this library.

common: Global constants, structures, and functions needed by the flow solver and the optimizer. Also home to the SBP module that defines some of our finite-difference operators.

movegrid: A Fortran 95 version of Anh Truong's linear elasticity mesh movement code.

diablo: The home of Diablo, our Newton-Krylov flow solver.

doc: Where we keep documentation, like this developers guide.

surface: Aerodynamic surface representation using B-splines. I developed this for my thesis, so it may be of limited usefulness to others.

optimizer: Optimization related routines.

tests: Various test problems to check for consistency between versions.

3 Data Structure Overview

Diablo is a multi-block finite-difference flow solver. In the code, the blocks are represented with the `Block` data type, described in more detail below. The blocks are "joined" together with `Interface` data types, and boundary conditions are defined with the `Boundary` data type. While a `Block` has a 6-sided hexahedral shape, it may have more than one `Boundary` or `Interface` on each side. The data structures are depicted visually in Figure 1

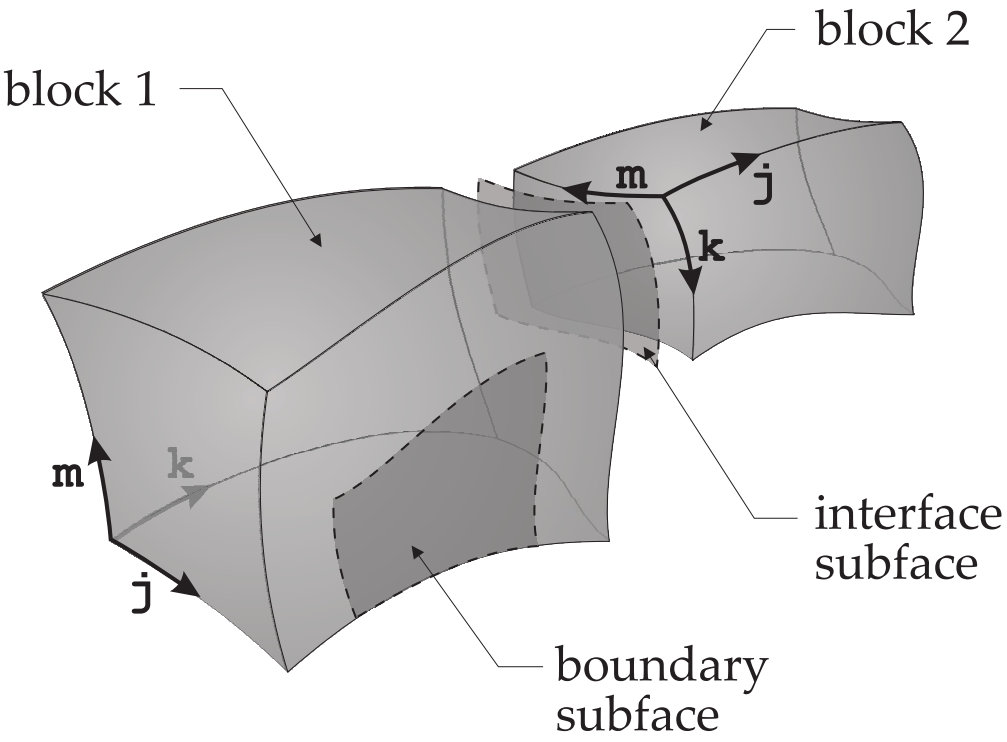

Figure 1: Visual representation of Diablo's data structures.

3.1 Block Structure

The block structures hold geometric and flow solution information for each node. The arrays store this information in (j, k, m) array format; when additional array indices are needed they are appended after m. For example, the $q_2 = \rho u$ momentum component of the $(4, 6, 7)$ node is accessed using `q(4,6,7,2)`. This storage system was adopted for historic and readability reasons. Fortran arrays are stored in column-major order, so this storage scheme is not optimal in many circumstances; however, I have tried different orderings and the CPU time difference is not significant (at least on the Itaniums with the Intel compiler).

The block structures also contain arrays for node indices and stencil numbers which are used to build the Jacobian and preconditioner matrices; more on stencil numbers in section 4.1

Block sides are indexed using the convention shown in table 1. This index convention can be used to find the j-,k-, or m-coordinate direction which is normal to the face. Specifically, if we make the associations $j \rightarrow 1$, $k \rightarrow 2$, and $m \rightarrow 3$, then, for a given side `is`, the normal coordinate direction is given by (`is`+1)/2 (integer arithmetic).

$(\mathbf{j}, \mathbf{k}, \mathbf{m})$	**min/max**	**side index**
j	min	1
j	max	2
k	min	3
k	max	4
m	min	5
m	max	6

Table 1: Diablo's block-side indexing convention.

As you might expect, there is a block structure for each block assigned to a given processor p. In addition, any block adjacent to the blocks on processor p, but belonging to another processor, is represented with a ghost block. A ghost block has a full set of indices in the tangential directions, but a truncated set in the normal index direction; specifically, the ghost blocks are `nhalo` nodes deep. The ghost blocks provide one way to communicate information between processors.

3.2 Interface Structure

A block face is the entire side of a block. A subface is a subset of a particular face and is defined by the block-side and the tangential index ranges. For example, a block with $20 \times 20 \times 20$ nodes might have a subface on side `is` $= 2$ (j max side), and have tangential index ranges of $k = [1, 10]$ and $m = [11, 20]$.

An interface is a particular type of subface that has blocks on either side. An interface indicates how the two blocks are oriented. Hence, the interface must provide several pieces of information:

- which blocks are adjacent;
- the sides (1–6) of the blocks that meet at the interface, and;
- tangential index ranges for both blocks

There are several ways of storing the tangential index ranges; We use ICEMCFD's convention. In this approach, an array of the form `dit(1:2,1:3)` is stored for each block on either side of the interface. The first tangential coordinate direction, i.e. one of $(j, k, m) \rightarrow (1, 2, 3)$, is stored in `dit(1,1)`. The beginning of the index range is given by `dit(1,2)`, and the number of nodes in the first tangential direction is `dit(1,3)`. Note that the coordinate direction, `dit(1,1)`, may be positive or negative; the sign indicates if the tangent indices should be traversed in increasing (positive) or decreasing (negative) order. The second tangential direction information is stored similarly, with the first array index in `dit` replaced with 2.

For example, consider the interface shown Figure 1. For block 1 we might have `dit(1,1)` = 1 and `dit(2,1)` = 3; the first tangential direction is j and the second is m. I say "we might" only because the order and sign of the tangential indices may be different. Similarly, for block 2 we might have `dit(1,1)` = 2 and `dit(2,1)` = 3.

3.3 Boundary Structure

A boundary structure is a one-sided subface and must indicate:

- which block it belongs to;
- the type of boundary (far-field, symmetry, slip, no-slip, etc.);
- the side (1–6) of the block that the boundary lies on, and;
- tangential index ranges for the boundary subface.

Again, a `dit(1:2,1:3)` array is used to indicate the tangential coordinate information.

3.4 Partition Module

The blocks are partitioned among the processors according to some algorithm. This partitioning gives rise to the `Partition_Mod` module. You can think of the `Partition_Mod` as a container for all the data needed by a individual processor.

The block list array, `blk`, is the set of blocks assigned to a processor together with the set of necessary ghost blocks. The local blocks are listed first and the last local block has the index `nBlk`. Including the ghost blocks there are `totBlk` blocks in total.

Similarly, the `iface` and `bcface` arrays are lists of interfaces and boundary faces. The boundary list is a straightforward array with `nbcf` boundary subface structures. The interface list is a bit more complicated, since an interface may be between two local blocks, or between a local block and a ghost block. We will call the former a local interface and the

latter an external interface. The external interfaces are listed first in the `iface` array, and they are grouped according to the neighbouring processor responsible for the corresponding ghost block. The `ifptr` array can be used to access the interfaces corresponding to a particular neighbouring processor. For example, the interfaces for neighbour p are given by `iface(ifptr(p):ifptr(p+1)-1)`. Suppose there are `nBPnbr` neighbouring processors. Then `iface(ifptr(nBPnbr+1))` is the first internal interface and `iface(nface)` is the last internal interface, where `nface` is the total number of interfaces stored locally.

The partition module contains more than lists for the basic structures: this module must coordinate the entire solution process. It is responsible for the highest level subroutines including initialization, outer (Newton) iterations, and residual evaluation. This module also holds the Jacobian and precondition matrix data types of the pKISlib library.

3.5 SBP operator

Our present discretization of choice uses summation-by-parts operators (SBPs) and simultaneous approximation terms (SATs). These are described in detail in many publications. The purpose of this section is to describe a SBP data type that I have proposed, and that some of you may choose to use.

First, a useful observation is that SBP operators are essentially sparse matrices with a special structure. Thus, it makes sense to store them as such. The idea is to adapt the compressed-sparse-row (CSR) storage system. Suppose we have a block with N nodes in a given direction, and we want a 3rd-order accurate SBP operator for d/dx. Define 4 arrays:

as(:) holds the numerical values of the operator, with each row compressed (zeros removed) and concatenated into a 1-dimensional array.

ja(:) holds the position of the matrix entries, one for each value in `as`, relative to the diagonal.

ibeg(:), iend(:) indicate where in `as` and `ja` a particular row is stored.

The arrays `ibeg` and `iend` are always N elements long, since each row in the operator (matrix) needs to know where to find its elements in `as` and `ja`. In contrast, the size of `as` and `ja` depends on the operator.

To illustrate the operator, consider the first four rows of the 3rd-order SBP operator:

$$\begin{bmatrix} -\frac{24}{17} & \frac{59}{34} & -\frac{4}{17} & -\frac{3}{34} & & & & \\ -\frac{1}{2} & 0 & \frac{1}{2} & & & & & \\ \frac{4}{43} & -\frac{59}{86} & 0 & \frac{59}{86} & -\frac{4}{43} & & & \\ \frac{3}{98} & 0 & -\frac{59}{98} & 0 & \frac{32}{49} & -\frac{4}{49} & & \\ & & & & \ddots & \ddots & \ddots & \end{bmatrix}.$$

Consider the fourth row. Now, the first, second, and third rows contain 10 nonzero elements; therefore, the forth row's elements will start being stored in `ibeg(4) = 11`. There are four

nonzero elements in the fourth row, so `iend(4) = 14`. In particular, the values of the fourth row are stored in

$$\texttt{as(11:14) = as(ibeg(4):iend(4))} = \left[\tfrac{3}{98} \quad -\tfrac{59}{98} \quad \tfrac{32}{49} \quad -\tfrac{4}{49}\right].$$

To determine the column positions of these elements, the `ja` array stores the values that need to be added (or subtracted) from the diagonal. In this example, `ja(11:14) = [-3 -1 1 2]`. Essentially, `ja` stores the 1-dimensional stencil.

SBP operators have a number of rows in the interior with identical entries (the interior scheme). The interior scheme is stored once in `as` and `ja`, and `ibeg` and `iend` simply point to the same entries as many times as necessary.

This operator may seem complicated at first, but it simplifies the code significantly. To see the operator in action, look at the subroutine `getEulerFluxesRHS` in the `EulerRHS_Mod.f90` file. The subroutine gives us the discrete derivative of the inviscid fluxes, for any order-of-accuracy (all we need to do is define the SBP operator, and it takes care of the rest).

4 Miscellany

4.1 Stencil Numbers

I define the stencil of a node to be the set of neighbouring nodes used to construct the discrete flow equations. Consider, in 2-D, a second-order accurate approximation to the Laplace equation on a uniform Cartesian grid. The discrete equation at node (j, k) has the stencil $\{(j, k), (j-1, k), (j+1, k), (j, k-1), (j, k+1)\}$. Each node in the full stencil is assigned a binary bit; $0, 1, 2, 4, 8$ in this case. If a node is absent from the stencil, say at a boundary, then the bit corresponding to that neighbouring node is turned off, i.e. set to zero. Hence, adding all the binary bits produces a number which encodes the stencil information.

This may not seem very useful for the second-order accurate Laplace equation, but it becomes very handy in 3 dimensions and higher-order approximations. Why? Once you learn a bit about the compressed-sparse-row and block-compressed-row matrix storage schemes, you may start to appreciate the usefulness of a stencil number. Basically, it tells you (quickly) where to store neighbour information in the compressed, 1-d array for the sparse matrix. The best way to appreciate how it works is to look at the code.

Warning: the default integer in most implementations of the fortran compiler will only be 32 bits. Thus, if you plan on using the stencil integer, your stencil had best not include more than 31 positions!

4.2 Finding Tangential Coordinate Directions

Often, when dealing with the sides of a block, it is necessary to find the coordinate indices corresponding to the normal and tangential directions. The calculation for finding the normal coordinate direction was described earlier; just add 1 to the side index and divide by two. Suppose this produces the coordinate direction index `di`. The two tangential coordinate

directions can be found easily as follows: `it1 = mod(di,1)+1` and `it2 = mod(di+1,1)+1`. Note, that the coordinate system `(di,it1,it2)` obeys a right-hand-rule. This is occasionally very important.

The indices `it1` and `it2` are also used frequently to describe the tangential coordinate directions on an interface or boundary, i.e. `it1=dit(1,1)` and `it2=dit(2,1)`. In these cases, the system `(di,it1,it2)` may *not* obey a right-hand-rule.

4.3 Metric Terms

The transformed Euler equations are given by

$$\partial_t \hat{\mathbf{Q}} + \partial_{\xi_i} \hat{\mathbf{E}}_i = \mathbf{0},$$

where $(\xi_1, \xi_2, \xi_3) = (\xi, \eta, \zeta)$,

$$\hat{\mathbf{Q}} = \frac{1}{J} \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ e \end{pmatrix}, \qquad \text{and} \qquad \hat{\mathbf{E}}_i = \frac{1}{J} \begin{pmatrix} \rho U_i \\ \rho u_1 U_i + p \partial_x \xi_i \\ \rho u_2 U_i + p \partial_y \xi_i \\ \rho u_3 U_i + p \partial_z \xi_i \\ (e+p) U_i \end{pmatrix}.$$

The scalar J denotes the Jacobian of the mapping, and the U_i are the contravariant velocities defined by $U_i = u_j \partial_{x_j} \xi_i$.

The metric terms are of the form

$$\partial_{x_j} \xi_i = \frac{\partial \xi_i}{\partial x_j} = J \left(\frac{\partial x_{j_{t1}}}{\partial \xi_{i_{t1}}} \frac{\partial x_{j_{t2}}}{\partial \xi_{i_{t2}}} - \frac{\partial x_{j_{t2}}}{\partial \xi_{i_{t1}}} \frac{\partial x_{j_{t1}}}{\partial \xi_{i_{t2}}} \right)$$

where $i_{t1} = \mod(i,3)+1$, $i_{t2} = \mod(i+1,3)+1$, $j_{t1} = \mod(j,3)+1$, and $j_{t2} = \mod(j+1,3)+1$.

Notice that a factor of J appears in the metric terms, and a factor J^{-1} appears in the Euler equations. Therefore, in Diablo, I store scaled metric terms of the form

$$\widetilde{\partial_{x_j} \xi_i} = \frac{1}{J} \partial_{x_j} \xi_i$$

Hence, the Euler equations become

$$\partial_t (\frac{1}{J} \mathbf{Q}) + \partial_{\xi_i} \tilde{\mathbf{E}}_i = \mathbf{0},$$

where

$$\tilde{\mathbf{E}}_i = \begin{pmatrix} \rho \tilde{U}_i \\ \rho u_1 \tilde{U}_i + p \widetilde{\partial_x \xi_i} \\ \rho u_2 \tilde{U}_i + p \widetilde{\partial_y \xi_i} \\ \rho u_3 \tilde{U}_i + p \widetilde{\partial_z \xi_i} \\ (e+p) \tilde{U}_i \end{pmatrix}.$$

and $\tilde{U}_i = u_j \widetilde{\partial_{x_j} \xi_i}$.

Now, you might be asking, "why did you do this?" First, it allows me to use $\mathbf{Q}$ instead of $\hat{\mathbf{Q}}$ in the solution vector, and I find this produces much better scaling for the Jacobian-free matrix-vector products, since all the variables are of the same order[1]. Second, sometimes the metric Jacobian is inaccurate, and this inaccuracy corrupts the metric terms and hence the solution; this is avoided with this approach, at least for the Euler equations.

Note, as an aside, when the metrics are scaled this way they become proportional to areas; in fact, they are very closely related to cell face areas in a finite volume scheme.

When implementing viscous terms, this metric scaling approach will have to be reevaluated.

5 Coding Guidelines

Here are some guidelines we would like you to follow when developing code for Diablo, and in general. If you are looking for a handy Fortran 90/95 reference, I recommend Metcalf and Reid's book "Fortran 90/95 Explained, Second Edition" (it is available online through the library).

5.1 General Practices

implicit none: All modules and subroutines must contain `implicit none`. This forces all variables to be explicitly declared, hence there is no ambiguity regarding their type.

Fortran 90/95 syntax: Use Fortran 90/95 syntax for all new code. For example, use `do--end do`, `>=`, `==`, etc. Do not use `goto` statements; use loop control statements such as `cycle` and `exit` instead.

Modules: If you are adding many related subroutines, consider grouping them into a Module. However, try to avoid making modules too big.

Variable Intent: Use `intent(in)` and `intent(out)` in the declaration of subroutine input and output variables, respectively. Use `intent(inout)` if a variable is used as both an input and output. These declarations help catch inadvertent attempts to overwrite input variables. Be careful with `intent(out)` when using structures; the value of a field which has already been set in the structure may be wiped out when passed in as an `out` variable. Use `inout` when you need to keep some of the values in the structure.

Error Handling: Whenever program execution must be stopped due to a failed test, an error statement should be written to the standard output, i.e., write (*,*). This statement must contain

1. which function found the error, and;

[1]Todd Chisholm solved this problem by applying row and column scaling to the matrix

2. what test failed.

#include: Do not use `#include` statements. The variables used by a subroutine should be passed in as arguments. Global variables should be used only as a last resort. If the number of arguments becomes large, consider the need to introduce a structure.

Local Variables: Avoid declaring "large" local variables, e.g., arrays with dimensions on the order of the mesh size. Try to use variables that have already been defined.

Code Lifetime: Do not assume that you are the only one that will use a subroutine that you have coded; keep the code consistent. Moreover, if you find a section of code that is unclear and uncommented, add some comments so the next person will not have to spend as long figuring out the code.

5.2 Coding Style

Comments: Add comments freely, but be concise. Explain what every block of code is doing, but not necessarily every line. Describe the purpose of a subroutine immediately after its interface. Explain what each input variable holds, and what each output variable should hold upon return. Put your initials at sections of the code that you have added to the original subroutine. Issue a warning if the code has not been tested for certain cases. Try to follow the template on page 12.

Naming: Use subroutine and variable names that describe what the subroutine does and the variable holds.

Maximum Column Width: Although Fortran 90/95 lines may extend to 132 characters, *try to restrict line widths to 72 characters* (Fortran 77 standard). The narrower code is more readable and fits on printed pages better (no wrapped text).

Indentation: Use the indentation shown in the template as a guideline. The indentation for the template can be obtained in emacs by adding the lines below to your `.emacs` file (this file is located in your home directory).

```
(setq f90-beginning-ampersand nil       ; no "&" at *start* of continuations
      f90-indented-comment-re "!--"     ; "!--" indented to code
      f90-comment-region "! "           ; string to comment regions
      f90-program-indent 6              ; PROG, MOD, SUB, FUNC
      f90-type-indent 3                 ; TYPE, INTERFACE, BLOCK DATA
      f90-do-indent 3                   ; DO
      f90-if-indent 3                   ; IF, SELECT CASE, WHERE, FORALL
      f90-continuation-indent 5
      f90-break-before-delimiters nil)
```

A Template

```
!
! ======================================================================
!
subroutine example(intvar,dpvar,arrdim,dparr1,dparr2)
!
! ----------------------------------------------------------------------
! Purpose:
!   Explain what the subroutine does, as concisely as possible.
!
! WARNING:
!   Include a warning if there are cases the subroutine has not been
!   tested for, if the input has to conform to a certain format, etc
!
! Inputs:
!   intvar = describe what this variable holds
!   dpvar = describe what this variable holds
!   arrdim = describe what this variable holds
!   dparr1 = describe what this variable holds
!   dparr2 = describe what this variable holds
!
! Outputs:
!   dparr2 = describe what this variable will hold upon return
!
! Author: I.P. Freely, July 2005
!         U.R. Great, December 2006 (what was modified)
! ----------------------------------------------------------------------
! ------------------------
! variable specification

      implicit none

      !-- subroutine arguments
      integer, intent(in) :: intvar, dpvar, arrdim
      double precision, intent(in) :: dparr1(arrdim)
      double precision, intent(out) :: dparr2(arrdim)

      !-- local variables
      integer :: i

! ---------------------
! begin main execution

      !-- perform necessary checks
      if (arrdim <= 0) then
         write(*,*) 'Error in example:: arrdim <= 0!'
         stop'example'
      end if
      if (intvar <= 0) then
         write(*,*) 'Error in example:: intvar <= 0!'
         stop'example'
      end if

      !-- explain what this loop is doing
      do i = 1,intvar

         if (dparr1(i) >= dpvar) then
            dparr2(i) = dparr1(i)
         else
            dparr2(i) = dpvar
         end if

      end do !-- i

#if 0
      !-- preprocessor directives can be used to comment out
```

```
      !-- testing and debugging sections of the code

      !-- URG: output for testing
      write (*,*) 'dpvar', dpvar
      write (*,*) (dparr2(i), i=1,arrdim)
      !-- URG: end of testing
#endif

      return

end subroutine example
```