User's Guide to Diablo

(Dated: February 10, 2015)

I. GETTING STARTED

In order to use Diablo, it is necessary to install a Fortran 90 compiler and FFTW, if they are not already installed. The Fortran 90 compiler is required since some features used in Diablo such as zero-sized arrays are not supported by Fortran 77. Fortunately, high quality Fortran 90 compilers are freely available for various platforms. Free compilers that have been successfully used with Diablo are g95 (www.g95.org), gfortran (http://gcc.gnu.org/wiki/GFortran#download), ifort (The Intel Fortran compiler which is free for non-commerical use at: www.intel.com/cd/software/products/asmo-na/eng/282038.htm). FFTW can be downloaded from www.fftw.org. Diablo has been tested using FFTW version 2.1.5. Unfortunately the newest versions of FFTW (3.x) use a different calling sequence than versions 2.x, so the older version must be used. In order to update Diablo to use the newer FFTW, the wrapper subroutines in "fft.f" need to be changed to reflect the new calling conventions.

A. Optional packages

In order to run Diablo on multiple processors with distributed memory, it is necessary to install MPI libraries on each computer. Diablo has been tested using LAM/MPI version 1.5, OpenMPI 1.6.1 and MPICH 3.0.4. Since only standard MPI subroutines are used, it should work on any standard MPI distribution. In order to run on a cluster, you will need to install MPI, FFTW, and Diablo on all computers. It will also be necessary to allow the computers to communicate without requiring the user's password. One way to do this, is to add a list of all hosts that will be running in parallel to the "authorized_keys" file in the ".ssh" directory inside the user's home directory. Make sure this is working by trying to ssh between computers and verify that the login prompt is bypassed.

Diablo includes subroutines for writing data in the NetCDF format. NetCDF is a self-describing binary format that can be used by a variety of data processing and visualization software. In order to use NetCDF, the library files must be downloaded and installed on the local system. The NetCDF package is freely available from http://www.unidata.ucar.edu/software/netcdf. Diablo has been tested using NetCDF version 3.. A list of software that is capable of reading NetCDF files can be found at `http://www.unidata.ucar.edu/software/netcdf/software.html`.

Diablo also includes subroutines for writing data in HDF5 format, which is a very flexible standard data format designed to handle large amount of data. In order to use these subroutines, the HDF5 library must be downloaded and installed in the local machine. We strongly recommend the use the HDF5 library for IO operations, especially when running in parallel with MPI. The HDF5 subroutines allow a large degree of flexibility and to use the same grid files and flow fields produced by other simulations, even when using different number of processes. **At the present stage, only the channel version of diablo permit the use of HDF5 subroutines.**

In order to enable additional library in Diablo, the MAKEFILE should be updated accordingly. Once new libraries are installed, the LINKDIR and INCLUDEDIR variables in the Diablo Makefile should be set to the locations where the relevant library and include files can be found. When more library are to be used, all the relevant path should be accessible. However, it is worth mentioning that nowadays MPI and HDF5 library generally comes with wrappers (`mpif90` for MPI, `h5fc` for serial HDF5 and `h5pfc` for parallel HDF5). The wrapper should be substituted to the compiler and will make sure that the right files are included in the compilation. In the case that wrapper are used, it is enough to follow the compiling rules outlines in section III.

Once FFTW and a Fortran 90 compiler are installed, we are ready to perform our first simulation of Diablo. The latest version of diablo can be cloned on the local machine using Mercurial (see the tutorial on Mercurial for further informations) using

```
ed441: anyFolder $  hg clone https://edeusebio85@bitbucket.org/edeusebio85/diablo
```

This should create a new working version of diablo. The diablo release contains the following subfolders

- *bin* contains some useful bash scripts to handle simulations with diablo;
- *for* contains the source code;
- *guides* contains userguides on various matters (among which this quick start);
- *input_files* contains examples of input files;
- *post_process* contains subroutines/packages to be used for post-processing using different tools used in scientifici computing (e.g. Matlab, Python and Vapor);
- *pre_process* contains a number of Matlab scripts to be used to generate the grids;
- *tests* allows to carry out some tests to verify that diablo compiles correctly and gives the right outputs.

II. CODE STRUCTURE

The main code directory contains the Fortran source code for Diablo, header files and the Makefile. The basic source code consists of the following:

- diablo.f - The main program for Diablo. This contains the main time-stepping loop.
- diablo_io.f - Input/Output subroutines for Diablo.
- periodic.f - Subroutines for time-stepping of the triply-periodic case (zero wall-bounded directions).
- channel.f - Subroutines for time-stepping of the channel geometry (one wall-bounded direction).
- duct.f - Subroutines for time-stepping of the duct geometry (two wall-bounded directions).
- cavity.f - Subroutines for time-stepping of the closed cavity geometry (three wall-bounded directions).
- courant.f - A subroutine for setting the time-step based on the CFL condition for a specified Courant number.

- fft.f - Wrappers for the FFTW library which are used for the periodic, channel, and duct cases.
- mpi.f - (Optional) Subroutines for running Diablo with distributed memory parallelization
- netcdf.f - (Optional) Subroutines used to write output files in the self-describing binary NetCDF format.
- hdf5s.f - (Optional) Subroutines used in *serial* runs to write output files in HDF5 format.
- hdf5.f - (Optional) Subroutines used in *parallel* runs to write output files in HDF5 format.

The direcory “dummy_code” contains empty versions of the subroutines in mpi.f, netcdf.f and others which allows Diablo to be compiled and used without MPI or NetCDF enabled or installed.

III. MAKEFILE

In the main code directory, a Makefile is provided to compile Diablo and create and executable. At the top of the Makefile are various compile-time user options. These include:

- COMPILER - The name of the compiler to use. Note while the source code is written in Fortran 77 fixed-form, a Fortran 90 compiler is required due to the use of several Fortran 90 features including zero sized arrays. Free compilers that have been successfully used with Diablo are g95 (www.g95.org), gfortran (http://gcc.gnu.org/wiki/GFortran#download), ifort (The Intel Fortran compiler which is free for non-commerical use at: www.intel.com/cd/software/products/asmo-na/eng/282038.htm).
- USEROPTS - Any number of user specified compiler options. Some common options are ”-g” for debugging and ”-OLEVEL” where LEVEL is an integer specifying the amount of compiler optimization. Larger LEVEL numbers will lead to a higher level of optimization at the expense of longer compile times.
- LINKDIR - Location where FFTW (and other libraries) has been installed. The default location on linux systems is often /usr/local/lib.
- INCLUDEDIR - Locations where other include files are located. The default location on linux systems is often /usr/local/include.
- PARALLEL - An option to compile the code with distributed memory parallelism using MPI. PARALLEL = TRUE to use MPI, PARALLEL = FALSE to compile in serial mode. This will also specify the `mpif90` wrapper as the compiler to be used. Make sure that MPI compatible Fortran compiler (such as mpif90) can be found in the enviromental variable PATH.
- NETCDF - An option to use the NetCDF file format for output. This makes it easy to use the Diablo outupt files in a wide range of NetCDF compatible post-processing programs. Note that the NetCDF library needs to be installed on the local system if this option is enabled by NETCDF = TRUE.
- HDF5 - An option to use the HDF5 file format for output. Note that the HDF5 library needs to be installed on the local system if this option is enabled by NETCDF = TRUE. For serial cases, e.g. PARALLEL=FALSE, this will set COMPILER to `h5fc`. For parallel compilation, this will set COMPILER to `h5pfc`.

- LES - If this option is enabled with LES = TRUE, then the subroutines necessary for a large-eddy simulation will be compiled. Since the LES model adds many additional storage arrays, this should be disabled (by for direct numerical simulations (DNS) when the LES subroutines are not needed.

Diablo can be compiled simply by typing "make". When running "make" the Makefile will determine which files have been modified since the last time the Makefile was run, and only the modified files (and their dependencies) will be re-compiled. Sometimes it is desirable to re-compile all of the source code. To do this, first run "make clean", followed by "make". When values of the above variables differ from the default value, this can be specified in the command line by typing,

```
ed441: for $ make PARALLEL=TRUE
```

for compiling in parallel, and

```
ed441: for $ make PARALLEL=TRUE HDF5=TRUE
```

for compiling in parallel with HDF5 libraries.

IV. GRID

Diablo uses a staggered grid in each direction that is treated with finite differences. This avoids the even/odd gridpoint de-coupling that can occur when using a co-located grid. Specifically, the wall-normal velocity in each finite difference direction is staggered with respect to other variables. Since Diablo uses cartesian grids, each three dimensional grid can be represented by a combination of three one-dimensional arrays of gridpoint locations. Diablo uses the 'base grid', designated by GX, GY, and GZ for all variables in the periodic flow case. This grid is also used for the wall-normal velocity in each direction treated with finite-differences. Other quantities are defined on the 'fractional grid', designated by GXF, GYF, and GZF. The fractional grid is defined to be exactly halfway between neighboring base grid points. Note that when grid-stretching is used, the converse is not true, the base grid points are not necessarily halfway between neighboring fractional grid points. A schematic of the staggered grid with wall locations is shown in Figure 1.

FIG. 1: Grid layout of Diablo in the wall-bounded directions treated with finite differences. The wall-normal velocity is stored at G points (open circles), all other variables are stored at G_F points (closed circles). Note that G here stands for G_X, G_Y, and/or, G_Z, depending on which directions are wall-bounded.

When a given direction is treated with finite-differences, the grid spacing can be made non-uniform. For example, when considering wall-bounded flows, it is generally desirable to stretch the grid to increase the resolution near the wall in order to resolve the large gradients that then to occur in this region. In order to allow for nonuniform grids, Diablo requires a file containing the user-defined grid locations for each direction that uses finite-differences. Grid may be defined either with (a) ASCII files or (b) HDF5 files if the library are installed. In (a), the grid files should be named “xgrid.txt”, “ygrid.txt”, and “zgrid.txt” and placed in the case directory, as needed depending on which directions will be treated with finite differences (set through the NUM_PER_DIR parameter). In order to make it easier to create a grid, a script called “create_grid.m” has been created and can be found inside the `pre_process` directory. Either Matlab or Octave should be able to run this script, and the script can be modified to include user-defined stretching functions. In (b), the grids are contained in a HDF5 file called “grid.h5” (see the HDF5 guide for further information). This file can be generated by a script in the `pre_process` directory, called “create_grid_h5.m” in a very similar manner as for “create_grid.m”. We recommend to use the (b) method when running simulation in parallel. When running in parallel, the total number of points N_y should be chosed such that

$$N_y = N_p \cdot \left(N_y^p - 1\right) + 1, \tag{1}$$

where N_y^p is the number of points per process (equal among the processes) and N_p is the number of processes. For instance, $N_y = 129$ is a suitable combination for $(N_y^p = 17, N_y^p = 8)$, $(N_y^p = 33, N_y^p = 4)$ and $(N_y^p = 65, N_y^p = 2)$.

V. INPUT PARAMETER FILES

When Diablo is executed, two input files containing user-specified parameters will be read in. All simulations, regardless of how many periodic or wall-bounded directions are used, will read the file “input.dat” from inside the specified case directory. This file contains the following paramters:

- FLAVOR - A string used to distinguish the modes of operation for Diablo. For example, if you want to add a specialized forcing term to Diablo, you could set FLAVOR to a flag of your choice, say FLAVOR=”Body Force”. Then inside the source code, appropriate hooks can be added with If statements to check the value of FLAVOR.
- VERSION - The version number of the Makefile. Used to ensure that the input file is compatible with the source code.
- USE_MPI - Option to use MPI
- USE_LES - If TRUE, then solve the LES filtered equations with the addition of a LES model. Note that in order to use the LES, the flag LES should also be set to TRUE inside the Makefile.
- NU - The viscosity in code units. If the code has been nondimensionalized, then this constant becomes 1/Re where Re is the Reynolds number using the selected velocity and length scale.
- LX, LY, LZ - The domain size.
- NUM_PER_DIR - The number of periodic directions. This determines which timestepping algorithm will be used:
  3=periodic flow, 2=channel flow, 1=duct flow, 0=cavity flow
- CREATE_NEW_FLOW - If TRUE, then initialize the velocity field using the CREATE_FLOW subroutines, if FALSE, then initialize the velocity field from a restart file called diablo.saved

- N_TIME_STEPS - Number of integration time steps. Each timestep contains three Runge-Kutta substeps.
- TIME_LIMIT - Maximum wall-clock time in seconds
- DELTA_T - The size of the timestep if constant time-steps are used
- RESET_TIME - If TRUE, then the simulation time will start at zero, if FALSE, then the simulation will start using the time contained in the initial velocity field (if CREATE_NEW_FLOW=FALSE)
- VARIABLE_DT - If VARIABLE_DT=TRUE then the CFL number will be used to specify the timestep every UPDATE_DT timesteps
- CFL - The constant CFL number that will be used to determine the size of the timesteps if VARIABLE_DT=TRUE
- UPDATE_DT - How often (in timesteps) to update the CFL number. Since the procedure for determining the size of the time-step based on the CFL number is expensive, it is often more efficient to do this periodically instead of at every time step.
- VERBOSITY - How much information should be print to the screen at runtime. High levels of VERBOSITY (integer value 1-5) will result in more information.
- SAVE_FLOW_INT - How often (in timesteps) to save the entire 3d velocity field to a restart file.
- SAVE_STATS_INT - How often to write mean statistics to a file
- MOVIE - If TRUE, then write out 2d slices through the velocity field every SAVE_STATS_INT timesteps in order to create movie files. Note that these files can become very large, so make sure that enough disk space is available when using this option.
- CREATE_FLOW_TH - If TRUE, then initialize the scalar field using CREATE_TH subroutines. If FALSE, then initialize the scalar field by reading diablo_th.start
- FILTER_TH - If TRUE, then apply a low-pass filter to the scalar field. Since sharp fronts often form in the scalar field which may result in numerical errors, applying a low-pass filter can help improve the quality of the simulation.
- FILTER_INT - If FITLER_TH=TRUE, then FITLER_INT sets how often (in timesteps) to apply the filter to the scalar field.
- RI_TAU - Quantified the influence of the scalar on the velocity field through the buoyancy term. In the code, the buoyancy term is RI_TAU*THETA where THETA is the local scalar value. For example, if the density is used as the scalar in dimensional terms, then RI_TAU should be $= -g/\rho_0$. If the equations have been nondimensionalized, then RI_TAU should be also be made nondimensional using the appropriate scales.
- PR - The Prandtl number for the scalar (Defined as the kinematic viscosity divided by the scalar diffusivity ν/κ).

- BACKGROUND_TH - This parameter is useful for considering a density stratification in a periodic direction. If this parameter is TRUE then perturbations to a linear background field are computed. This has been implemented in periodic.f to allow simulations of homogeneous stratified turbulence.

VI. CHANNEL GEOMETRY

When NUM_PER_DIR=2 the the "input.dat" parameter file, the x and z directions will be made periodic and derivatives in these directions will be treated with a pseudo-spectral method, while derivatives in the y direction will be evaluated with second-order, central finite differences. The time-stepping subroutines are contained in the file "channel.f". The grid will be staggered in the y direction with the y-velocity, U2 defined on the GY grid, and all other variables defined on the fractional GYF grid. The required input files for a simulation using the channel flow geometry are "ygrid.txt" which contains the location of the staggered grid points, "grid_def" which contains the grid size and the number of scalars to consider, "input.dat" which was described in Section V, and "input_chan.dat" which contains parameters that are native to the channel flow geometry. The file "input_chan.dat" contains the following parameters:

- VERSION - The version number of the Makefile. Used to ensure that the input file is compatible with the source code.

- TIME_AD_METH - The time-stepping method used. Two options are available for the channel geometry, when TIME_AD_METH=1 all terms in the momentum equations involving wall-normal derivatives are treated implicitly using Crank-Nicolson and all other terms are explicitly time-stepped using a third-order Runge-Kutta method. When TIME_AD_METH=2 all viscous terms are time-stepped using Crank-Nicolson and other terms are treated with Runge-Kutta.

- LES_MODEL_TYPE - This parameter gives the type of subgrid-scale model to use when performing a large-eddy simulation. When LES_MODEL_TYPE=1, a constant Smagorinsky model is used (a wall-damping function can be turned on or off inside les.f). When LES_MODEL_TYPE=2, a dynamic Smagorinsky model is used. The Smagorinsky constant is computed using the dynamic procedure, but this model is much more computationally expensive than the constant Smagorinksy. When LES_MODEL_TYPE=3, a scale-similar part is added to the dynamic model for a "dynamic mixed model". This is the most expensive of the three options, but has the advantage of providing a mechanism for energy backscatter from the subgrid scales to the resolved scales.

- IC_TYPE - If CREATE_NEW_FLOW=TRUE, this flag determines which set of initial conditions to use in the subroutine CREATE_FLOW_CHAN.

- KICK - If CREATE_NEW_FLOW=TRUE, then we often want to add a random perturbation on top of the initial condition. This parameter sets the amplitude of the perturbation.

- I_RO_TAU - Sets the wall-normal rotational number (inverse of the Rossby number).

- F_TYPE - Sets the type of body force applied: F_TYPE=1 force with a constant pressure gradient PX0, F_TYPE=0 use a pressure gradient that keeps the bulk velocity constant UBULK0, F_TYPE=2 force with an oscillatory pressure gradient, amplidude AMP_OMEGA0, frequency OMEAG0, and F_TYPE¿2 will not add a body force.

- U,V,W,TH_BC_LOWER - Set the type of boundary condition to be applied at the lower wall to the x-velocity, U, the y-velocity, V, the z-velocity, W, and the scalars TH (if any are used). If this parameter = 0, then Dirichlet boundary conditions will be applied with the value at the wall specified by _BC_LOWER_C1. If this parameter = 1, then Neumann-type boundary conditions will be applied by specifying the wall-normal gradient in _BC_LOWER_C1. The other constants here, _BC_LOWER_C2, etc. are not presently used, but are provided in case user-specified boundary conditions need more input data.
- U,V,W,TH_BC_UPPER - Same as for _BC_LOWER, but specifies the boundary conditions at the upper wall.

VII. CREATING A NEW SIMULATION

In order to run a serial case (for a channel simulation), do the following steps:

1. Create a new directory called `ser`
2. Copy the `grid_def`, `input.dat` and `input_chan.dat` to `ser`
3. Set the number of points in x, y and z in `grid_def`
4. Go to the `for` folder of diablo
5. Copy the `grid_def` file in `for` folder
6. Compile the code with `make HDF5=TRUE` (hoping with no errors!)
7. Copy the executable `diablo` to the run directory `ser`
8. Generate the grid by running the Matlab script `create_grid_h5.m` in the `pre_process` folder
9. Copy the output file `grid.h5` to the run folder
10. Set the appropriate input parameters in `input.dat` and `input_chan.dat`
11. If the simulation start from a flow field, copy the flow field `flow.h5` to the folder and rename it as `start.h5`
12. Run the simulation by issuing `./diablo`

Some of the binaries in the `bin` folder can be helpful to automize the process. For instance `compDia.sh` is a shell-script which automatically does from 4-6 and `genGridHDF5.sh` does 8-9. To see how these work, first set the enviromental variable `PATH_DIABLO`

```
ed441: ser $ export PATH_DIABLO=path/to/diablo/folder
```

where `path/to/diablo/folder` is the absolute path of the diablo folder. Then make the bash scripts in `bin` visible by issuing

```
ed441: ser $ export PATH=$PATH_DIABLO:$PATH
```

At this point, you can use the shell script, simply by issuing for the `ser` directory

```
ed441: ser $ compDia.sh -Options=’’HDF5=TRUE’’ -Clean=yes
```

and generate the `grid.h5` file with

```
ed441: ser $ genGridHDF5.sh grid_def
```

taking the number of points N_y in `grid_def`.

In order to run a parallel case, the some workflow outlined above applies, although with some modifications. The `grid_def` used to compile the code should have the number of points per prossess N_y^p instead of N_y. It is good procedure to store the total resolution in a file called `grid_def.all` and generate the appropriate `grid_def` file by issuing

```
ed441: mpi $ genGridFiles.sh Np
```

where `Np` is the number of processes to be used. Diablo can then be compiled by using the bash script

```
ed441: mpi $ compDia.sh -Options=’’HDF5=TRUE PARALLEL=TRUE’’ -Clean=yes
```

and run with

```
ed441: mpi $ mpirun -np Np ./diablo
```

Runs in parallel will produce only one flow field but multiple statistic files. These files can be merged into a single file by using the python package which is described below.

A. Post-processing with Python

In order to post-process the data with Python, the Python package which permit to read in HDF5 files must be installed (`h5py`, `http://www.h5py.org`). Before starting ipython, make sure that the relevant path are included in the enviromental variable `PYTHONPATH`, e.g.

```
ed441: ser $ export PYTHONPATH=$PYTHONPATH:$PATH_DIABLO/post_process/python/
```

```
ed441: ser $ export PYTHONPATH=$PYTHONPATH:$PATH_DIABLO/post_process/python/ext
```

Start ipython from the run folder. Import the `diablo_pack` module by

```
[1] import diablo_pack as d
```

You can then create a run object with

```
[2] r=d.run.RunDiablo('grid_def.all')
```

The `r` object will contain a bunch of attributes and methods, characteristic of the run. If a parallel run has outputted seperate statistic files, these can be merged by using

```
[3] r.join_stats()
```

and then the statistics can be read in the object with

```
[4] r.read_stat()
```

This will create a `StatsDiablo` object file in `r.st`, which contains all the statistics. For instance if mean profiles (averaged along the simulation) are to be plotted, use

```
[5] plot(r.st.yf,np.mean(r.st.Var['ume'],0))
```

(this assumes that you have imported the module `numpy` as `np`). Finally, if a flow field needs to be read, just create a `FlowDiablo` object by issuing

```
[6] r.read_flow('end.h5')
```

which store the object in the dictionary `r.flow` pointed by its name, *e.g.*

```
[7] myFlow=r.flow['end.h5']
```

.

For further information about the objects and methods of the Python package, please refer to its manual.

VIII. TRIPLY PERIODIC GEOMETRY

When NUM_PER_DIR=3 in “input.dat”, periodic boundary conditions will be applied in all three directions and all spatial derivatives will be treated with a pseudo-spectral method. Since all directions are treated in the same manner, and since all variables are co-located on the same grid, the algorithm for this case is the most straightforward of the four options in Diablo, and this case may be a good place to start to learn the methods used in Diablo. The user-defined options specific to this case are specified in a file called “input_per.dat” located inside the case directory. The contents of this file are:

- VERSION - A flag to set the version number of the input files. This is used to make sure that the source code and input files are compatible.
- TIME_AD_METH - Integer flag to specify which time-stepping method to use. At this time, only one method is active in periodic.f, so this parameter should always be set to 1.
- LES_MODEL_TYPE - This flag will be used to specify which LES model should be used. However, at this time the LES model has not been written for the triply periodic geometry.
- IC_TYPE, KICK - Parameters used to set the initial conditions for the velocity field. If CREATE_NEW_FLOW = TRUE, then IC_TYPE=0 will initialize the velocity with a Taylor-Green vortex, IC_TYPE=1 will use an ideal vortex with an axis aligned with the y-axis. Users can create new initial conditions in the CREATE_FLOW_PER subroutine inside periodic.f
- BACKGROUND_GRAD(N) - This parameter should be set for each scalar that is used in the simulation. If BACKGROUND_GRAD(N)=TRUE, then scalar number N will be solved for perturbations about a linear background profile, with a gradient in the y-direction. This is designed to allow simulations of stratified turbulence in a triply periodic geometry where density perturbations are assumed to be periodic in all three directions.

IX. FREQUENTLY ASKED QUESTIONS

Q: Why are indices for the three-dimensional arrays ordered i, k, j?
A: This is done in order to speed-up the FFTs as much as possible. With the exception of the cavity geometry, it is expected that the FFTs will be the most expensive part of the Diablo algorithm. Therefore, we can get a significant speedup by ordering the arrays to keep the data for the Fourier transforms as close together as possible in memory. Fortran stores data in memory starting with the first index. That is data with the index (i,k,j) will be stored next to the data for (i+1,k,j). Since the duct flow geometry uses FFTs in the x direction, the i index is first, and since the channel flow geometry performs FFTs in the x and z directions, it makes sense to index over k next.

Q: Can I compile and run Diablo under Windows?
A: The short answer is yes, but we don't recommend it. High performance computing is normally done in C or Fortran on a machine running a unix-like operating system (such as linux, mac os X, ibm aix, etc). If you don't have access to a dedicated unix (or linux) machine, probably the best option is to make your computer running Windows dual-boot either by creating a linux partition on your current hard drive, or by installing linux on a separate hard drive. However, some students in MAE 223 at UCSD were able to get Diablo running under Windows using "cygwin", although it took significant work to get this going. Here is a description of how to do this, courtesy of Karsten Breddermann.

1. Erase any already installed versions of g95
2. Download cygwin from http://www.cygwin.com/ Just the blue link under the logo
3. Install the "Devel" packages using cygwin

4. Download g95 from g95.org for cygwin (http://ftp.g95.org/g95-x86-cygwin.tgz)

5. Download fftw 2.1.5 from www.fftw.org.

6. Decompress the FFTW install directory. Using a terminal in cygwin, go to this directory. In the Makefile, change the lines F77 = g77 to F77 = g95 and ac_ct_F77 = g77 to ac_ct_F77 = g95. In the FFTW install directory, run the configure script by "./configure", followed by "make" and "make install". After the installation is finished, you can check it by running "make check". Note after "make install" the directory that it has installed the FFTW library, this will probably end in /usr/local/lib.

7. Download Diablo from http://fccr.ucsd.edu and hope for the best!

Q: Why do I get an error like: "In function init_fft_: undefined reference to fftwnd_ftt_"?
A: For several possible reasons. The first thing to do is to make sure that the FFTW library is installed in the same location as the LIBDIR specified in the Makefile. Look for the files: "librfftw.a" and "librfftw.la" in this directory, and make sure that you have installed FFTW version 2.*. If this are ok and you still get the error, there may be a problem with appended underscores. The following is from the website http://g95.sourceforge.net/docs.html, although it applies to compilers other than g95 too: "While g95 produces stand-alone executables, it is occasionally desirable to interface with other programs, usually C. The first difficulty that multi-language program will face is the names of the public symbols. G95 follows the f2c convention of adding an underscore to public names, or two underscores if the name contains an underscore. The -fno-second-underscore and -fno-underscoring can be useful to force g95 to produce names compatible with your C compiler." Try adding these options to the USEROPS in the Makefile. If this still doesn't work, then you may have to go into fft.f and add one or more trailing underscores to the subroutines names in the FFTW library. In fft.f, add an underscore to all call statements to subroutines containing "FFTWND". Try compiling the Makefile again, and if you get a similar error, try adding a second underscore. I have even come across a situation where I needed three underscores (it was on a shared machine where multiple users had installed different version of FFTW).