FDMAP User Guide
Version 1.0

January 10, 2013

FDMAP was written by Eric M. Dunham (edunham@stanford.edu) and Jeremy E. Kozdon (jekozdon@nps.edu), based on an initial prototype by David Belanger, with additional contributions from Lin Cong. The numerical method, developed by the authors in collaboration with Jan Nordstrom, is based on summation-by-parts finite differences with boundary and interface conditions enforced weakly using the simultaneous approximation term method. Details are given in papers by the authors, particularly

Kozdon, J. E., E. M. Dunham, and J. Nordstrom (2012), Simulation of dynamic earthquake ruptures in complex geometries using high-order finite difference methods, Journal of Scientific Computing, doi:10.1007/s10915-012-9624-5.

Kozdon, J. E., E. M. Dunham, and J. Nordstrom (2011), Interaction of waves with frictional interfaces using summation-by-parts difference operators: Weak enforcement of nonlinear boundary conditions, Journal of Scientific Computing, 50(2), 341-367, doi:10.1007/s10915-011-9485-3.

Dunham, E. M., D. Belanger, L. Cong, and J. E. Kozdon (2011), Earthquake ruptures with strongly rate-weakening friction and off-fault plasticity, 1: Planar faults, Bulletin of the Seismological Society of America, 101(5), 2296-2307, doi:10.1785/0120100075.

Development was initiated with support from the Southern California Earthquake Center, and continued with support by NSF grants EAR-0910574 and EAR-1114073 to Eric Dunham, as well NSF Fellowship for Transformative Computational Science using CyberInfrastructure OCI-1122734 to Jeremy Kozdon.

1 Requirements

To compile and run FDMAP, you will need a Fortran compiler, an MPI library, and LAPACK.

2 Installation

Modify `Makefile` with information about your compiler and libraries. Type `make` to build (with optimization enabled by default). Or use

`make BUILD=debug` ⇒ compile with debugging flags
`make BUILD=production` ⇒ compile with optimization flags.

The executable is `fdmap`.

Decide where you will save data. A simple option is to create a `data` directory off of the main directory. You will probably find it useful to create subdirectories to store sets of simulations, as this is not done automatically, and a generic data directory can quickly fill up with files.

Decide where you will store input files. One option is to create a `problems` directory off of the main directory.

To run, pass the input file name, including path if necessary, as the sole argument to the executable. For example,

`mpirun -np 4 fdmap problems/example4blocks.in`.

Of course, if you run on a cluster, there may be a more complicated job submission process.

3 Input Files

Look at the input file `example4blocks.in`, which sets up the problem of rupture along a branchign nonplanar fault governed by rate-and-state friction.

Most parameters are specified in Fortran namelists, for example,

```
&problem_list
 name = 'data/ex4b', ninfo = 10,
 nt = 3001, CFL = 0.3d0, refine = 1d0 /
```

The namelist is called `problem_list` and there are multiple variables (`name`, `ninfo`, etc.) that are contained between the start of the namelist and the end, which is designated as `/`. Here, `name` is the problem name, including path (absolute or relative) to the data directory. This serves as a prefix for

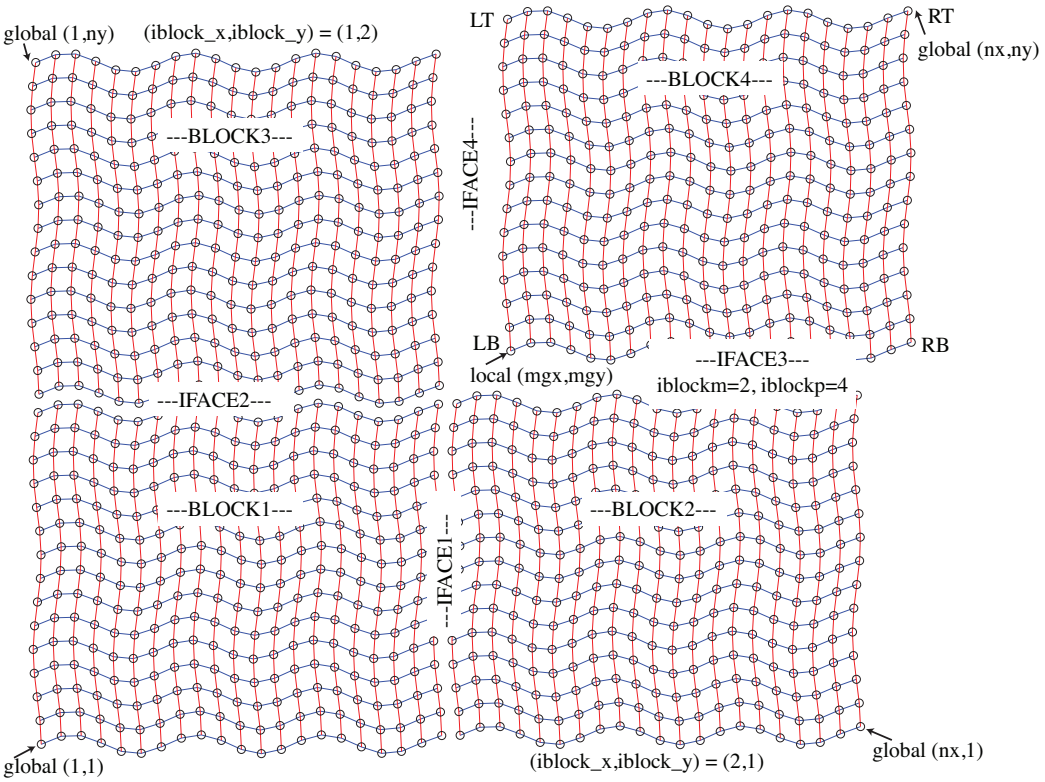

Figure 1: Domain containing 4 blocks.

all data files written by the code. `ninfo=10` prints information about the simulation to stdout every 10 time steps. `nt` is the number of time steps and `CFL` is used, together with other information, to set the time step. `refine` is a global grid refinement parameter; both the number of grid points in space and the number of time steps are increased by this factor.

The "domain" consists of multiple four-sided "blocks" (Fig. 1). Due to details of the Fortran and MPI implementation, these blocks must be arranged in a Cartesian grid. The entire domain has `nx` by `ny` grid points, prior to refinement. This is divided into `nblocks` blocks, which are laid out in an `nblocks_x` by `nblocks_y` grid. Blocks can have exterior boundaries, where boundary conditions are given, or they can be coupled to other blocks across "interfaces."

Parameters for each block are specified in namelists, following a line giving the block index, for example, `!---BLOCK1---`. First look at `grid_list`. Each

block has Cartesian indices in the two directions, stored as `iblock_x` and `iblock_y`. These reflect an underlying Cartesian communicator used in MPI calls. The physical coordinates of each block are given as a pair of numbers, `(x,y)`, corresponding to each block corner, with `LT` being `L`eft `T`op, `RB` being `R`ight `B`ottom, etc. By default, these corners are connected with straight lines. The number of grid points within each block is given as `nx` and `ny`. These are local block sizes, so will always be less than or equal to the global `nx` and `ny`. The indices of the bottom left point (`LB`) in the block, in the global grid, are `mgx` and `mgy`. Here, `mg` stands for "minus global" (internally, `pg` is used for "plus global").

Boundary conditions for external sides are in `boundaries_list`. As before, `L` is for the left side, `R` is for the right side, etc. Options include `absorbing` (to reduce wave reflections from artificial exterior boundaries), `free` for traction-free (really, zero traction change) boundaries, and `rigid` for zero particle velocities. When a block "boundary" is really an interface, set the boundary condition to `none`.

Initial stresses are specified in `fields_list` and material properties in `material_list`. These namelists can be included after each block, if properties are different for the different blocks, or included after all blocks, when properties are common to all blocks.

The interfaces are specified next. As with blocks, a set of namelists follows `!---IFACE1---` for the first interface, for example. The `interface_list` contains the two block indices, `iblockm` and `iblockp`, for the blocks on the minus and plus sides, respectively. Make sure the order matches that used in the Cartesian block communicator. The `direction` is roughly the unit normal to the interface, without sign. The global interface indices lie between `mg` and `pg`. The type of coupling is specified though `coupling`. Examples include `locked` for perfectly "welded" contact and `friction` for frictional contact.

Some types of coupling require additional namelists, like `friction_list`. Here, certain parameters in the namelist are derived types, and the order in which parameters are specified matters. See `friction.f90` for details. A good way to search for namelists is with `grep friction_list *.f90`, for example.

Simulation data output is specified in the output list, starting with `!---begin:output_list---`. This is not a Fortran namelist, but just a set of output items, each containing several lines of variables. For example,

```
I2
iface2
x y
0d0 0d0 1
```

will output the coordinates `(x,y)` of all points on interface 1. These will be stored as files containing `I2_x` and `I2_y`. The first line, `I2`, serves as a prefix and can be anything (use it to keep track of output items). `iface2` identifies the interface. `x` and `y` are the fields to be output. The last line `0d0 0d0 1` contains `tmin`, `tmax`, and `stride_t`. Data is saved between times `tmin` and `tmax`, every `stride_t` time steps. In this case, since the coordinates do not change as a function of time, we just save at the initial time step.

The next output item contains more fields on the fault, `D V S N Psi`, which are slip (`D`), slip velocity (`V`), shear stress (`S`), normal stress (`N`), and state variable (`Psi`). These are output between times `0` and `1d10`. The latter is a huge number, which makes the output time window span the full simulation. These fields are written every 10 time steps. Next are the body fields, both coordinates and fields; the "body" includes all blocks.

4 Complex Geometries

FDMAP handles complex geometries using coordinate transforms. It automatically performs interior mesh generation, given a set of points and unit normals describing any nonplanar block boundaries. These boundaries must be describable as smooth curves. Fault kinks and other boundary/interface slope discontinuities should be placed at one of the corners of a block. The nonplanar geometry is specified in a binary data file called a curve file. An example MATLAB script for generating such a file is provided (`example4blocks.m`, which uses `write_profile.m`).

5 Troubleshooting

By far the trickiest part is getting the input file, especially the multiblock grid, set up correctly. The code does not check inputs very carefully for consistency, so it is up to the user to get this right.

6 Analysis

Put `init.m` and `loadfast.m` in your MATLAB path (e.g., `~/matlab/`). Start MATLAB. Load an overall simulation data structure with

```
A = init('ex4b','~/fdmap/data');
```

where `ex4b` is the problem name and `~/fdmap/data` is the path to the data files. `A` is a data structure that contains lots of information about the simulation. Among its components are `A.I2_x`, `A.I2_V`, etc. These are also data structures and contain information about one of the fields that was output. Load in the coordinates for interface 2 with

```
x2 = loadfast(A,'I2_x'); y2 = loadfast(A,'I2_y');
```

This loads two vectors, `x2` and `y2`, that contain the coordinates of interface 2. Plot with

```
plot(x2,y2).
```

Next load the slip velocity on that interface, which was a frictional fault:

```
V2 = loadfast(A,'I2_V');
```

This returns a data structure containing the output time vector `V2.t` and the actual slip velocity array `V2.V`. The time step index is always the last one. Make a space-time plot of slip velocity:

```
pcolor(x2,V2.t,V2.V'),shading flat,colorbar
```

You can see that that rupture nucleates at the origin and propagates bilaterally. When it hits the left boundary, there is a reflected rupture. This is artificial and you should be cautious to put domain boundaries well beyond the edges of faults, unlike in this example problem. Next, load in the corresponding data for interface 3, the second frictional fault that is connected with this first one. You can combine the coordinate arrays and slip velocity arrays:

```
x3 = loadfast(A,'I3_x'); y3 = loadfast(A,'I3_y');
V3 = loadfast(A,'I3_V');
X = [x2; x3]; V = [V2.V; V3.V]; t = V2.t;
pcolor(X,t,V'),shading flat,colorbar
```

This now shows the entire fault, which is continuous across the block interface at $x = 15$. No numerical artifacts are evident across this internal interface.

You can make similar plots of fields in the medium:

```
x = loadfast(A,'B_x'); y = loadfast(A,'B_y');
vx = loadfast(A,'B_vx'); vy = loadfast(A,'B_vy');
for n=1:vy.nt
    pcolor(x,y,vy.vy(:,:,n))
```

```
    shading flat,axis image
    colorbar,caxis([-2 2])
    drawnow
end
```

7 Stanford-specific information

Check out a working copy of the code using the School's SVN server:

`svn checkout https://sesvcs.stanford.edu/var/svn/repos/fdmap`

Job submission on the CEES cluster is done use PBS. See `launch.sh` for an example job submission script. Submit the job with

`qsub launch.sh`

and monitor the cluster with various commands like `showq`, `qstat`, or

`qstat -u edunham`, where `edunham` is your user name.