MDSBI: Multi-Dimensional Spectral Boundary Integral Code, Version 4.1.7

Eric M. Dunham
edunham@fas.harvard.edu

20 September 2008

1 Introduction

MDSBI is a Fortran 95 boundary integral equation code for solving elastodynamic problems, specifically those dealing with shear ruptures or frictional instabilities. The code handles both two- and three-dimensional problems, as well as the response of a single spatial Fourier mode, and the case of a spring-block system. The code handles the case of identical materials on either side of the fault, as well as the more general bimaterial problem, including its poroelastic correction. Various friction laws have been implemented, including slip-weakening, rate-and-state (with flash heating), and thermal pressurization.

The elastodynamic response is calculated by a convolution over space and time of the slip or slip velocity history with appropriate kernels. The spatial convolution is performed in the Fourier domain using FFTs. The code was developed based on ideas from several papers, in particular, P. H. Geubelle and J. R. Rice (A spectral method for three-dimensional elastodynamic fracture problems, Journal of the Mechanics and Physics of Solids, 43, 1791–1824, 1995), P. H. Geubelle (A numerical method for elastic and viscoelastic dynamic fracture problems in homogeneous and bimaterial systems, Computational Mechanics, 20, 20–25, 1997), M. S. Breitenfeld and P. H. Geubelle (Numerical analysis of dynamic debonding under 2D in-plane and 3D loading, International Journal of Fracture, 93, 13–38, 1998), and N. Lapusta, J. R. Rice, Y. Ben-Zion, and G. Zheng (Elastodynamic analysis for slow tectonic loading with spontaneous rupture episodes on faults with rate-

and state-dependent friction, Journal of Geophysical Research, 105, 23,765–23,789, 2000). These papers are an excellent place to start if you wish to read more about the method.

For details on the poroelastic fault zone model, see Dunham, E. M. and J. R. Rice (2008), Earthquake slip between dissimilar poroelastic materials, Journal of Geophysical Research, 113, B09304, doi:10.1029/2007JB005405. A description of the numerical method used to solve the coupled flash heating and thermal pressurization equations is in preparation by H. Noda, E. M. Dunham, and J. R. Rice (2008). You can find these papers on my website.

The current version of MDSBI requires an MPI library (though you can still run it on a single processor, of course). Memory and computational work are distributed across the processors and the only interprocessor communication occurs during the spatial FFT. I have included several options for performing the FFT. Some use the distributed memory FFT in FFTW-2.1.5; others gather all information to the master process, perform the FFT there, and scatter the information back to the other processes. Efficiency depends on the problem size and number of processors you are using. Data output of the distributed memory arrays is also done in parallel using MPI I/O routines.

2 Installation

1. Download the compressed and archived source code `mdsbi-v4.1.7.tgz` from
   `http://www.people.fas.harvard.edu/~edunham/codes.html`
   to a chosen installation directory (taken to be `INSTALL` in this text).

2. Unzip and extract:
   `tar xvzf mdsbi-v4.1.7.tgz`
   or:
   `gunzip mdsbi-v4.1.7.tgz` followed by `tar xvf mdsbi-v4.1.7.tar`
   This will create a directory `INSTALL/mdsbi/` that contains some files and subdirectories. The subdirectories are

   (a) `analysis`: routines to analyze the output data

   (b) `data`: where the output data will be written (you may wish to create a symbolic link of the same name if you want to store the data elsewhere)

(c) `kernel`: convolution kernel tables

(d) `problems`: files configuring the problems to be solved

(e) `src`: source code

3. Download the kernel tables `kernels.tgz` from
`http://www.people.fas.harvard.edu/~edunham/codes.html`
to the kernel directory `INSTALL/mdsbi/kernel`.

4. Unzip and extract:
`tar xvzf kernels.tgz`
or:
`gunzip kernels.tgz` followed by `tar xvf kernels.tar`
This should produce two files containing the tabulated values of the convolution kernel; make sure they are in the `kernel` directory.

5. The code uses two libraries that are not included in this package. They are LAPACK, a package of linear algebra routines—see `www.netlib.org/lapack/` and FFTW, the Fastest Fourier Transform in the West—see `www.fftw.org`. The FFTW routines can perform FFTs on arrays of all sizes, not just powers of two.

   To set up these libraries, you need to make a few changes in the `INSTALL/mdsbi/src/` directory:

   (a) Install the LAPACK library. This library is often pre-installed on Linux distributions, and is included in recent OS/X distributions (in the vecLib framework), so you might want to check if you already have it before installing. One option is to download it from `www.netlib.org/lapack/`. You can actually get by with only a subset of the complete LAPACK library known as ATLAS (also available on netlib).

   (b) Edit `Makefile` to link to the appropriate LAPACK library (a few examples are given in the makefile).

   (c) Set up the FFT routines. All FFT calls will be contained in `fft_routines.f90`. I've provided three implementations of this: two that use FFTW-2.1.5 and one that uses FFTW-3.x. FFTW 3.x routines (in file `fft_routines_fftw3.f90`) are for a single processor only, so I gather data from all processes to the master,

perform a serial FFT on the master, and then distribute the data out to all processes. In my experience, this is the most efficient method unless you use more than about 32 processors. Also, in the current version of the code, I have only implemented 2D transforms (for 3D elasticity problems) using FFTW-3.x. The FFTW-2.1.5 library has distributed memory parallel transforms; I include two implementations of this (but only for 1D transforms for 2D elasticity problems). The 1D transform is only a complex-to-complex transform, but we have real data. In one implementation (in file `fft_routines_fftw2.f90`), I pack a real vector of length N into a complex vector of length $N/2$, perform the distributed memory parallel FFT on this complex vector, and then unpack the resulting complex vector to yield only the non-negative wavenumbers. However, this requires additional interprocessor communication. The other option (in file `fft_routines_fftw2c.f90`) is to pack the real vector of length N into a complex vector of length N (setting the complex part equal to zero). After taking a distributed memory parallel transform, the code works with both positive and negative wavenumbers. Despite the fact that this means sacrificing a factor of 2 in terms of memory and computational time, it eliminates the extra interprocessor communication step. Which routine is fastest depends on your system so experiment with all of them. If you want to use another library, then you can use any of these files as a template. If you choose FFTW, then download and install the library from `www.fftw.org`, making sure that Fortran libraries are created. To use FFTW 3.x:

i. Make a symbolic link (or rename) the file:
`ln -s fft_routines_fftw3.f90 fft_routines.f90`

ii. Edit `Makefile` to link to the FFTW3 library: `-lfftw3`.

and to use FFTW 2.1.5:

i. Make a symbolic link (or rename) the file:
`ln -s fft_routines_fftw2.f90 fft_routines.f90` or
`ln -s fft_routines_fftw2c.f90 fft_routines.f90`

ii. Edit `Makefile` to link to the FFTW2 libraries: `-lrfftw -lfftw`.

6. Set your compiler options in `Makefile`. I have a few options in the makefile; basically uncomment your compiler or add it and do likewise

with the flags. The program (well, at least previous versions—I haven't checked for this version) successfully compiles and runs with the Intel compiler `ifort`, the Sun compiler `f90`, IBM'S compiler `xlf95`, and the G95 compiler `g95` (available at `www.g95.org`). Please let me know if you experience any errors with yours.

One item of note concerns the use of non-Fortran 95 intrinsic subroutines.I have made use of only one such subroutine: `flush`. The `flush` subroutine forces the program to automatically write all queued output to a specific file. This option is nice when you want to look at the contents of the output files (or the `status` file, discussed later, that tells you which time step you are on or any error messages), before the program is finished running. For certain compilers or optimization levels, this data is not output immediately when the program is directed to write output. The `flush` subroutine causes this to occur. If you experience problems with this non-intrinsic, simply comment out the appropriate line in the source code. The specific subroutine is `flush_io` in `io.f90` in the `INSTALL/mdsbi/src/` directory.

7. Choose the precision with which you would like the calculations to be performed. The default installation of the FFTW library includes only double precision routines. For single precision routines, you will need to change the FFTW installation options. Within the program, edit the file `INSTALL/mdsbi/src/constants.f90`. Specifically set `pr = real4` for single precision or `pr = real8` for double precision. The default is double precision. You will also have to modify the appropriate routines in `INSTALL/mdsbi/src/fft_routines.f90`. Sometimes the solution degrades when using single precision; other times it seems fine. Use it at your own risk!

8. Compile by typing `make` in the source directory. This generates the executable `mdsbi` (or something similar, depending on what you've chosen in the makefile) in the main directory `INSTALL/mdsbi/`.

3 Configuration

Specific problems are found in the problems directory: `INSTALL/mdsbi/problems/`. All options are configured in a file named `problem_name.in`, where `problem_name`

is the name of the problem (e.g., `slipweak.in`). Input is primarily accomplished using Fortran namelists. Several input files are included that illustrate different problems (which are described under Included Problems). These files are not extensively commented (except for tpv3.in, which has a number of comments), and the best place to find out more about the options is within the source code itself, which is extensively commented. If you wish to create your own problem, copy one of the existing problem files, giving it a different name (e.g., `new_problem.in`), and modify it as needed.

4 Running

The program has the ability to solve several problems in succession (with each defined in a separate input file in the `INSTALL/mdsbi/problems/` directory). The code reads the file `INSTALL/mdsbi/input.in` to obtain the name of the problem list file, the name of the status file, and the name of the error file.

The problem list file contains a list of all the problems to be solved. By default, this is `INSTALL/mdsbi/problems/list.in`. However, you may use a different list located elsewhere. This is useful when you wish to run several instances of the program simultaneously. Your problem list file must have the extension `.in`.

Most messages from the program (such as the current problem name, current time step, etc.) are written to the status file, which will appear in the main directory. The default name is `status`, though you can change this as well in the file `INSTALL/mdsbi/input.in`.

Error and warning messages from the program are written to the error file, which will appear in the main directory. The default name is `error`, though you can change this as well in the file `INSTALL/mdsbi/input.in`.

1. Edit the problem list file, either `INSTALL/mdsbi/problems/list.in` or your chosen file, to include the names of all of the problems to be solved. Specify this list and the name of the status and error files to which messages will be written in `INSTALL/mdsbi/input.in`.

2. Switch to the main directory: `cd INSTALL/mdsbi/`. Run the executable by typing: `mpirun -np 2 mdsbi` (or the equivalent for your MPI library). This will read the file `input.in` to determine which problem list and status file to use. Data files are written within the

`INSTALL/mdsbi/data/` directory, which you can link to your favorite storage location.

3. Check the status of the running program by examining the status file (e.g., type `cat status`).

4. Check for error and warning messages by examining the error file (e.g., type `cat error`).

5 Analysis of Data

The data is output to the directory `INSTALL/mdsbi/data/`. There are two ways to output the fields, each of which is configured in the `problem_name.in` configuration file:

1. `mesh`: Output the fields on a mesh. The spatial mesh can be 2D, 1D (a line), or 0D (a point). Each field component is saved as a binary data file containing the field values on the mesh at specified time steps.

2. `front`: Output the time at which the rupture front (defined as when the value of a particular field exceeds a specified value) passes each point.

A Matlab script, `analyze.m`, is provided as an example in the analysis directory `INSTALL/mdsbi/analysis/`. Cut and paste the commands into the Matlab window (run Matlab in that same directory) and several figures will be created. The comments in the file explain what you are seeing.

6 Included Problems

Several problems are included as examples:

1. `slipweak`: Two-dimensional rupture with slip-weakening friction law for a fault between identical materials.

2. `rs_ageing`: Two-dimensional rupture with rate-and-state friction, ageing law.

3. `rs_slip`: Two-dimensional rupture with rate-and-state friction, slip law.

4. `rs_flash`: Two-dimensional rupture with rate-and-state friction, slip law with flash heating.

5. `poro1a` and related: Two-dimensional rupture with regularized slip-weakening friction between dissimilar poroelastic materials. These are the six cases from Dunham, E. M. and J. R. Rice (2008), Earthquake slip between dissimilar poroelastic materials, Journal of Geophysical Research, 113, B09304, doi:10.1029/2007JB005405.

6. `thermal_crack.in` and `thermal_pulse.in`: Two-dimensional crack-like and pulse-like ruptures with thermal pressurization and flash heating (in slip law framework). These are the input files for the BlueGene/L runs shown in H. Noda, E. M. Dunham, and J. R. Rice (in preparation, 2008). *These require 100+ GB of memory.*

7. `thermal_crack-small.in` and `thermal_pulse-small.in`: Small versions of the BlueGene/L runs (lower resolution and shorter propagation distance and calculation time) that demonstrate crack-like and pulse-like rupture modes.

8. `tpv3`: Three-dimensional rupture with slip-weakening friction. Based on SCEC TPV3, but additional heterogeneity is added.

9. `tpv101`: Three-dimensional rupture with rate-and-state friction, ageing law. This is SCEC TPV101. In contrast to other problems, heterogeneities in initial conditions and friction law parameters are read in from data files (which permit arbitrary heterogeneity, not just the simple types that are covered by the asperity list input feature in the input files. To set up the parameters for this problem, you must run `input_tpv101.m` in the `INSTALL/mdsbi/problems` directory.

7 Miscellaneous Notes

7.1 Field Names and Sign Conventions

Let me comment on the names of the fields, which appear both in the code itself and in the `.in` problem files and the Matlab analysis scripts. I use similar names for the identical materials version and the bimaterial version, which might lead to confusion. For both the identical materials version and

the bimaterial version, `Ux` and `Uy` refer to slip in the x and y directions, `Vx` and `Vy` refer to slip velocity, `sx` and `sy` refer to stress, `sx0` and `sy0` refer to loads. For the identical materials version, `fx` and `fy` refer to stress transfer functionals. For the bimaterial version, the response of the two half-spaces is computed separately before boundary conditions are applied that couple them. So there are fields for each side, the upper (or "plus") side denoted by a "p", and the lower (or "minus") side denoted by an "m". Hence, the displacements are `uxm`, `uxp`, etc., the particle velocities are `vxm`, `vxp`, etc., and the stress transfer functionals are `fxm`, `fxp`, etc. The stresses are continuous so they remain the same as in the identical materials case.

The code keeps track of the stress components of traction, instead of the tractions direction (which have different signs on the different sides of the fault). Slip is right lateral. Compression is negative and tension is positive for stresses (and opposite for pore pressure).

7.2 Units

The program performs no conversion of units, so you are free to use any self-consistent system (e.g., SI units). In several of the examples, units relevant to large-scale earthquakes are used. These are not SI units; instead I use a mixture of units for different fields that keep fields of order unity. (The code performs no rescaling of fields to prevent round-off error. This is your responsibility.) In these examples, stresses are in MPa, shear modulus is in GPA; the extra factor of 10^3 is canceled by measuring displacements in m, but distances in km.

8 Program Details

The program is organized into a set of modules, typically one per file, containing the main data types and associated subroutines and functions. In alphabetical order, the modules are

1. `asperity`: Linked-list routines for adding heterogeneity to fields
2. `constants`: Constants and precision definitions
3. `convolution`: Convolution variables

4. `convolution_routines_bm`: Routines to perform convolution, bimaterial version
5. `convolution_routines_im`: Routines to perform convolution, identical materials version
6. `fault`: Fields on the fault
7. `fft_routines_fftw2`: FFT routines using FFTW2.x
8. `fft_routines_fftw3`: FFT routines using FFTW3.x
9. `fields`: Routines related to fields
10. `fourier`: Fourier-domain fields
11. `friction`: Friction laws
12. `friction_routines`: Routines for friction laws
13. `friction_solver`: Solver for friction laws
14. `front`: Output timing of rupture front
15. `history`: History of Fourier coefficients of slip or slip velocity
16. `init`: Routines to initialize problems
17. `integration`: Routines for integrating fields in time
18. `io`: Input/output routines
19. `kernel`: Convolution kernel
20. `linalg`: Linear algebra routines
21. `load`: Time-dependent loads
22. `main`: Main program routines
23. `mesh`: Output history of fields on a mesh
24. `model`: Basic model parameters
25. `mpi_routines`: MPI routines

26. `problem`: Main problem data type
27. `problem_routines`: Routines to solve a single problem
28. `rates`: Routines to set rates (time-derivatives of fields)
29. `ratestate`: Rate-and-state friction
30. `rk`: Runge-Kutta coefficients
31. `slipweak`: Slip-weakening friction
32. `substep`: Routines for substepping
33. `thermpres`: Thermal pressurization
34. `timestep`: Routines for time steps
35. `utilities`: Assorted useful routines

Each of these modules comes in its own file, `module_name.f90`. The code has more features that I haven't discussed in this guide, and other modules. You are, of course, welcome to use these, but (as with everything else) I urge you to test them extensively before trusting the results.