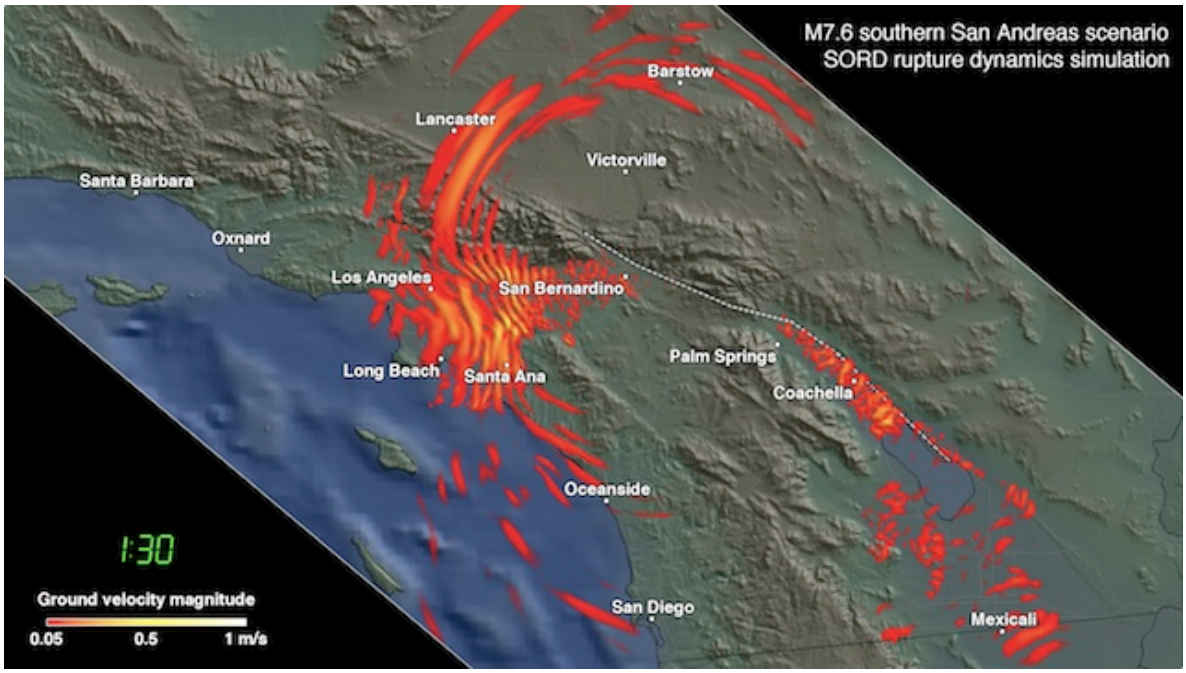
SORD

Support Operator Rupture Dynamics



Contents

Summary

The Support Operator Rupture Dynamics (SORD) code simulates spontaneous rupture within a 3D isotropic viscoelastic solid. Wave motions are computed on a logically rectangular hexahedral mesh, using the generalized finite difference method of support operators. Stiffness and viscous hourglass corrections are employed to suppress suppress zero-energy grid oscillation modes. The fault surface is modeled by coupled double nodes, where the strength of the coupling is determined by a linear slip-weakening friction law. External boundaries may be reflective or absorbing, where absorbing boundaries are handled using the

method of perfectly matched layers (PML). The hexahedral mesh can accommodate non-planar ruptures and surface topography

SORD simulations are configured with Python scripts. Underlying computations are coded in Fortran 95 and parallelized for multi-processor execution using Message Passing Interface (MPI). The code is portable and tested with a variety of Fortran 95 compilers, MPI implementations, and operating systems (Linux, Mac OS X, IBM AIX, SUN Solaris).

Publications

The first two papers give (for wave propagation and spontaneous rupture, respectively) the formulation, numerical algorithm, and verification of the SORD method. The third paper presents an application to simulating earthquakes in southern California.

1. Ely, G. P., S. M. Day, and J.-B. Minster (2008), A support-operator method for visco-elastic wave modeling in 3D heterogeneous media, *Geophys. J. Int.*, *172* (1), 331-344, doi:10.1111/j.1365-246X.2007.03633.x.
2. Ely, G. P., S. M. Day, and J.-B. Minster (2009), A support-operator method for 3D rupture dynamics, *Geophys. J. Int.*, *177* (3), 1140-1150, doi:10.1111/j.1365-246X.2009.04117.x.
3. Ely, G. P., S. M. Day, and J.-B. Minster (2010), Dynamic rupture models for the southern San Andreas fault, *Bull. Seism. Soc. Am.*, *100* (1), 131-150, doi:10.1785/0120090187.

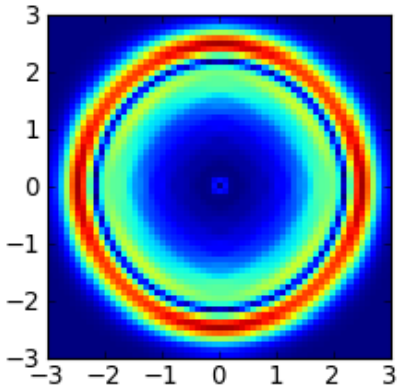
User Guide

Quick test

Run a simple point source explosion test and plot a 2D slice of particle velocity:

```
cd scripts/sord
python sim.py -i
python plot.py
```

Plotting requires Matplotlib, and the result should look like this:



Scripting with Python

For a simple example consider the above test, sim.py:

```
#!/usr/bin/env python
import os                                       # import O/S untilities
import cst                                      # import the Coseis module
prm = cst.sord.parameters()                     # for specifying SORD parameters
s_ = cst.sord.s_                                # for specifying slices
prm.delta = [100.0, 100.0, 100.0, 0.0075]       # step length in (x, y, z, t)
prm.shape = [61, 61, 61, 60]                    # mesh size in (x, y, z, t)
prm.fieldio = [                                 # field variable input and output
    ['=',  'rho', [], 2670.0],                  # material density
    ['=',  'vp',  [], 6000.0],                  # material P-wave velocity
    ['=',  'vs',  [], 3464.0],                  # material S-wave velocity
    ['=',  'gam', [], 0.3],                     # material viscosity
    ['=w', 'v1',  s_[:,:,31,-1], 'vx.bin'], # write X velocity slice output
    ['=w', 'v2',  s_[:,:,31,-1], 'vy.bin'], # write Y velocity slice output
]
prm.ihypo = [31.0, 31.0, 31.0]                    # source location
prm.source = 'potency'                          # source type
prm.source1 = [1e6, 1e6, 1e6]                     # source normal components
prm.source2 = [0.0, 0.0, 0.0]                     # source shear components
prm.pulse = 'integral_brune'                    # source time function
prm.tau = 6 * prm.delta[3]                      # source characteristic time
os.mkdir('run')                                 # create run directory
os.chdir('run')                                 # switch to run directory
cst.sord.run(prm)                               # launch SORD job
```

The first thing to do is import the `cst` module and instantiate a parameter object from `cst.sord.parameters()`. The parameter object contains all of the simulation settings and then passed to `cst.sord.run()`. A complete list of possible SORD parameters and default values are specified in parameters.yaml.

Running jobs

The `cst.sord.run()` function does four tasks: (1) configure job parameters (2) compile the source code if necessary (3) populate the run directory with necessary executable, input, and metadata files, and (4) launch the job interactively or through a batch processing system. By default, step (4) is skipped, giving the user an opportunity to inspect the run directory

prior to launching the job.

Field I/O

Multidimensional field arrays can be accessed for input and output through the `fieldio` list. The fieldnames.yaml file specifies the list of available field variables, which are categorized in four ways: (1) static vs. dynamic, (2) settable vs. output only, (3) node vs. cell registration, and (4) volume vs. fault surface. For example, density `rho` is a static, settable, cell, volume variable. Slip path length `sl` is a dynamic, output, node, fault variable. The `fieldio` list is order dependent with subsequent inputs overwriting previous inputs. So, for example, a field may be assigned to one value for the entire volume, followed by a different value for a sub-region of the volume.

The four-dimensional sub-volume of the array in space and time is specified using Python slicing notation and a helper function `s_` (similar to NumPy index expressions). The notation is extended here to use integers for node indices and integers + 0.5 for cell indices (1.5, 2.5, 3.5, ...). Array indexing starts at 1 for the first node, and 1.5 for the first cell. Negative indices count inward from end of the array, starting at -1 for the last node, and -1.5 for the last cell. Empty brackets `[]` are shorthand for the entire 4D volume. Some examples of slice notation:

```
s_ = cst.sord.s_    # Helper function for specifying slices
s_[10,20,1,:]       # Single node, full time history
s_[10.5,20.5,1.5,:] # Single cell, full time history
s_[2,:,:,::10]      # j=2 node surface, every 10th time step
s_[:,:,:,-1]        # Full 3D volume, last time step
[]                  # Entire 4D volume
```

Each member of the `fieldio` list contains a mode, a field name, and slice indices, followed by mode dependent parameters. The following I/O modes are available, where `'f'` is the field variable name (from the list fieldnames.yaml), and `[]` are the slice indices:

```
['=',   'f', [], val],             # Set f to value
['+',   'f', [], val],             # Add value to f
['=s',  'f', [], val],             # Set f to random numbers in range (0, val)
['=f',  'f', [], val, tfunc, T],   # Set f to time function with period T, scaled
['=r',  'f', [], filename],        # Read from filename into f
['=R',  'f', [], filename],        # Read from filename into f with exrapolation.
['=w',  'f', [], filename],        # Write f to filename
['=wi', 'f', [], filename],        # Write weighted average of f to filename.
```

A letter `'i'` in the mode indicates sub-cell positioning via weighted averaging. In this case the spatial indices are single logical coordinates that may vary continuously over the range. The fractional part of the index determines the weights. For example, an index of 3.2 to the 1D variable f would specify the weighted average: 0.8 * f(3) + 0.2 * f(4).

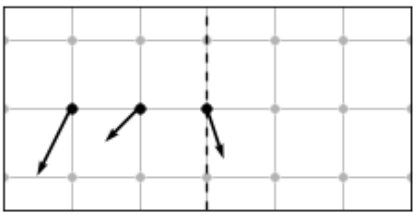
Reading and writing to disk uses flat binary files where j is the fastest changing index, and t is the slowest changing index. Mode 'R' extrapolates any singleton dimensions to fill the entire array. This is useful for reading 1D or 2D models into 3D simulations, obviating the need to store (possibly very large) 3D material and mesh coordinate files.

All input modes may use '+' instead of '=' to add to, rather than replace, preexisting values. For a list of available time functions, see the `time_function` subroutine in util.f90. The routine can be easily modified to add new time functions. Time functions can be offset in time with the `tm0` initial time parameter.

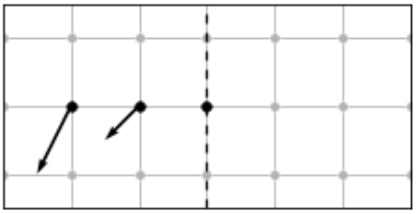
Boundary Conditions

Boundary conditions for the six faces of the model domain are specified by the parameters `bc1` (near-size, x, y, and z faces) and `bc2` (far-side, x, y, and x faces). The symmetry boundary conditions can be used to reduce computations for problems where they are applicable. These are not used for specifying internal slip boundaries. However, for problems with symmetry across a slip surface, the fault may be placed at the boundary and combined with an anti-mirror symmetry condition. The following BC types are supported:

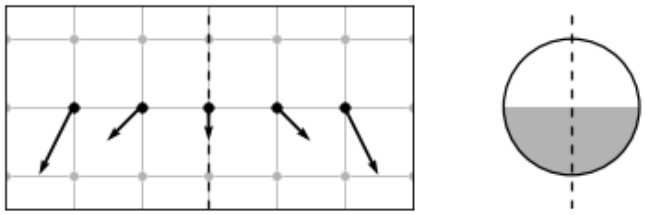
Type 0: Vacuum free-surface. Stress is zero in cells outside the boundary.



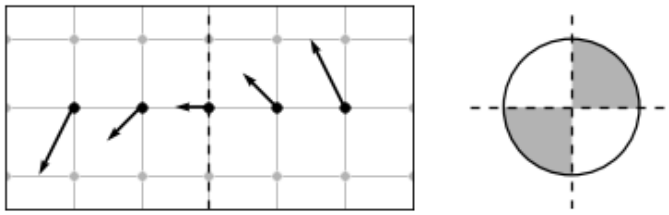
Type 3: Rigid surface. Displacement is zero at the boundary.



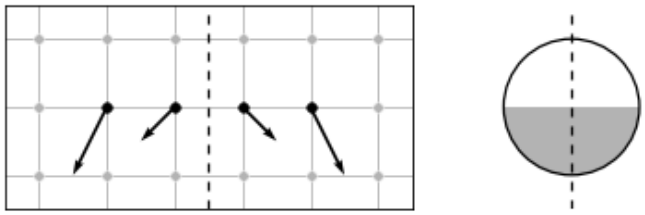
Type 1: Mirror symmetry at the node. Normal displacement is zero at the boundary. Useful for a boundary corresponding to (a) the plane orthogonal to the two nodal planes of a double-couple point source, (b) the plane normal to the mode-III axis of a symmetric rupture, or (c) the zero-width axis of a 2D plane strain problem.



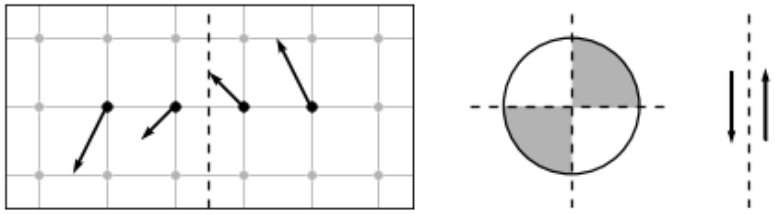
Type -1: Anti-mirror symmetry at the node. Tangential displacement is zero at the boundary. Useful for a boundary corresponding to (a) the nodal planes of a double-couple point source, (b) the plane normal to the mode-II axis of a symmetric rupture, or (c) the zero-width axis of a 2D antiplane strain problem.



Type 2: Mirror symmetry at the cell. Same as type 1, but centered on the cell.



Type -2: Anti-mirror symmetry at the cell. Same as type -1, but centered on the cell. Can additionally be used when the boundary corresponds to the slip surface of a symmetric rupture.



Type 10: Perfectly match layer (PML) absorbing boundary.

Example: a 3D problem with a free surface at Z=0, and PML absorbing boundaries on all other boundary faces:

```
shape = [50, 50, 50, 100]
```

```
bc1 = [10, 10,  0]
bc2 = [10, 10, 10]
```

Example: a 2D antiplane strain problem with PML absorbing boundaries. The number of nodes is 2 for the zero-width axis:

```
shape = [50, 50, 2, 100]
bc1 = [10, 10, -1]
bc2 = [10, 10, -1]
```

Defining the fault rupture surface

Fault rupture always follows a surface of the (possibly non-planar) logical mesh. The orientation of the fault plane is defined by the `faultnormal` parameter. This can be either 1, 2, or 3 corresponding to surfaces normal to the j, k, or l logical mesh directions. Any other value (typically 0) disables rupture altogether. The location of the rupture plane with in the mesh is determined by the `ihypo` parameter, which has a dual purpose of also defining the nucleation point. So, the indices of the collocated fault double nodes are given by `int(ihypo(faultnormal))`, and `int(ihypo(faultnormal)) + 1`. For example, a 3D problem of dimensions 200.0 x 200.0 x 200.0, with a fault plane located at z = 100.0, and double nodes at l = (21, 22), may be set up as such:

```
delta = [5.0, 5.0, 5.0, 0.1]
faultnormal = 3
ihypo = [21, 21, 21.5]
shape  = [41, 41, 42, 100]
bc1 = [0, 0, 0]
bc2 = [0, 0, 0]
```

For problems with symmetry across the rupture surface (where mesh and material properties are mirror images), the symmetry may be exploited for computational savings by using an appropriate boundary condition and solving the elastic equations for only one side of the fault. In this case, the fault double nodes must lie at the model boundary, and the and the cell-centered anti-mirror symmetry condition used. For example, reducing the size of the previous example to put the rupture surface along the far z boundary:

```
shape = [41, 41, 22, 100]
bc2 = [0, 0, -2]
```

Alternatively, put the rupture surface along the near z boundary:

```
ihypo = [21, 21, 1.5]
shape = [41, 41, 22, 100]
bc1 = [0, 0, -2]
bc2 = [0, 0, 0]
```

Further symmetries may present. If our previous problem has slip only in the x direction, then we may also use node-centered mirror symmetry along the in-plane axis, and node-centered anti-mirror symmetry along the anti-plane axis, to reduce computations eight-fold:

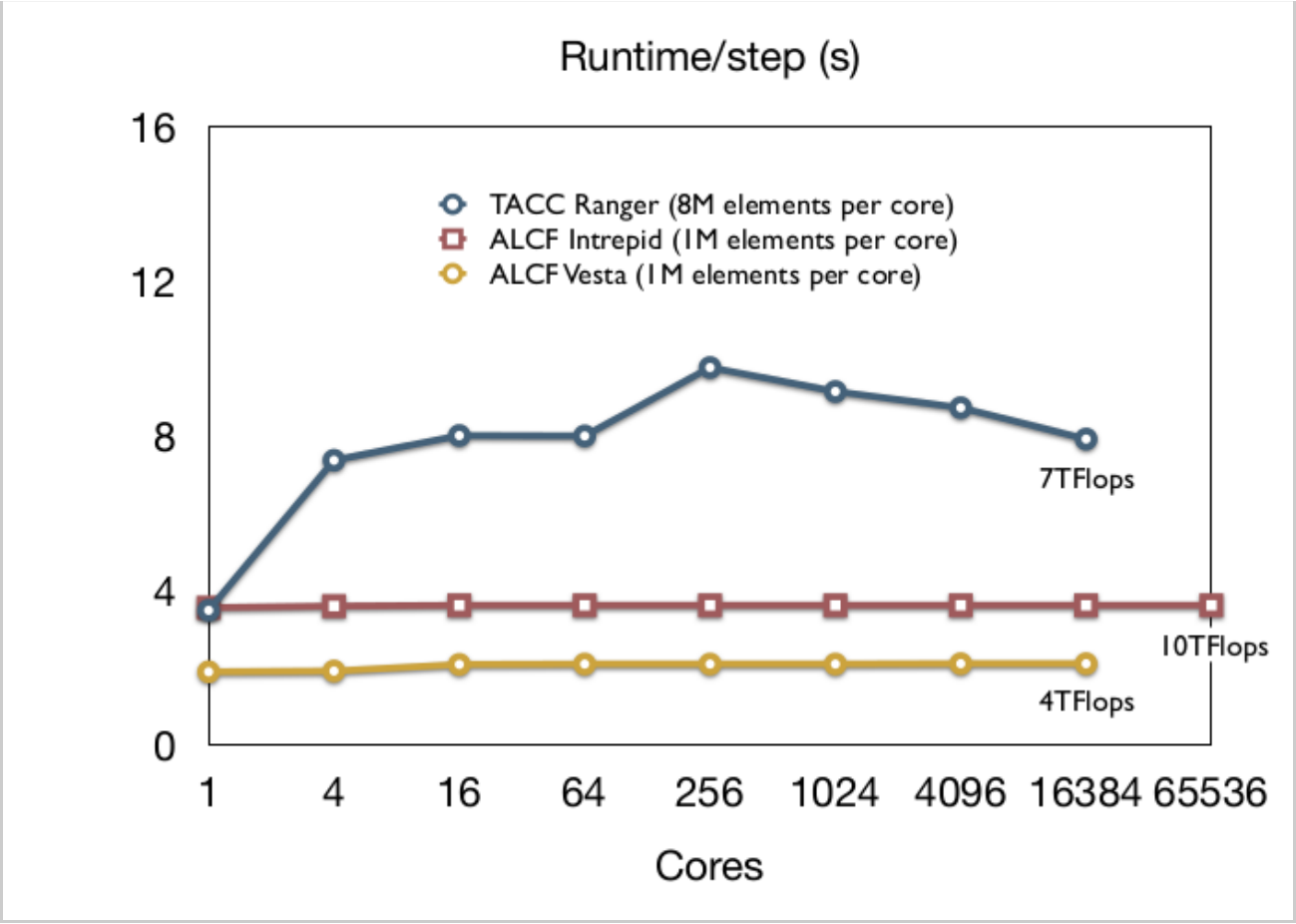
```
ihypo = [21, 21, 21.5]
shape = [21, 21, 22, 100]
bc1 = [0, 0, 0]
bc2 = [-1, 1, -2]
```

Memory Usage and Scaling

For rectilinear meshes, 23 single precision (four-byte) memory variables are required per mesh point. Curvilinear meshes have two options with a trade-off in memory usage vs. floating-point operations. Stored operators require 44 variables per mesh point and give the best performance, while on-the-fly operators require 23 variables per mesh point at the cost of a factor of four increase in floating point operations. As CPU improvement tends to out-pace memory bandwidth improvement, in the future, on-the-fly operators may become faster than stored operators. The operator type is controlled by the `oplevel` parameter, but can generally be left alone, as the default is to automatically detect rectilinear and curvilinear meshes and assign the proper operator type for fastest performance. The allowed values are:

0: Auto pick 2 or 6
1: Mesh with constant spacing dx
2: Rectangular mesh
3: Parallelepiped mesh
4: One-point quadrature
5: Exactly integrated elements
6: Saved operators, nearly as fast as 2, but doubles the memory usage

On current hardware, computation time is on the order of the one second per time step per one million mesh points. SORD scalability has been benchmarked up to 64 thousand processors at ALCF. The following chart is the wall-time per step per core (click for PDF):


Runtime/step (s)
16
12
8
4
0
TACC Ranger (8M elements per core)
ALCF Intrepid (1M elements per core)
ALCF Vesta (1M elements per core)
7TFlops
10TFlops
4TFlops
1
4
16
64
256
1024
4096
16384
65536
Cores