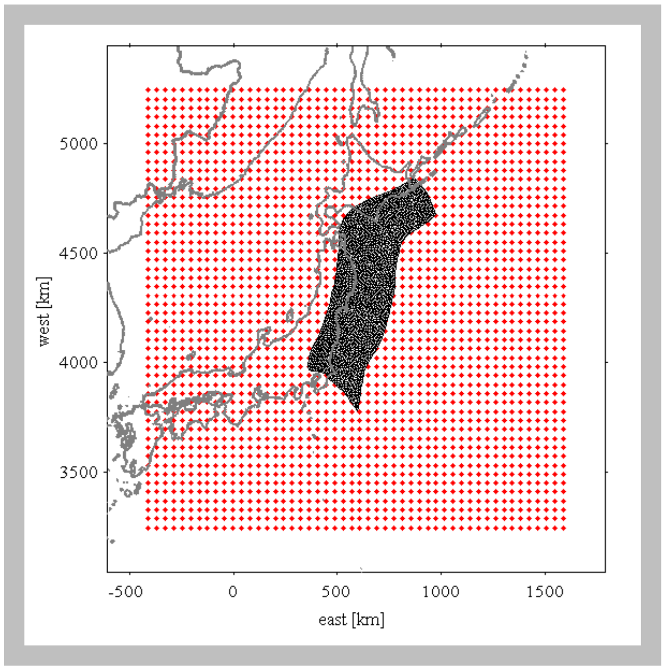
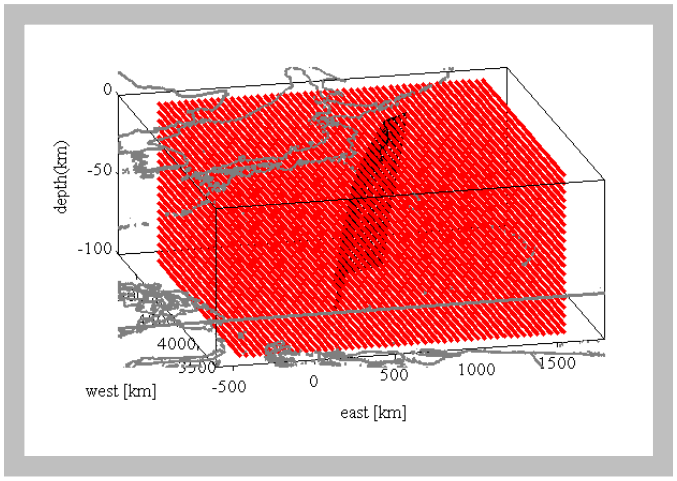
Eileen Evans
Phoebe DeVries





These two figures show the input fault mesh and the observation points used in our largest, final data inputs. On the left is our fault mesh and observation points in map view, and on the right in three dimensions. Vertical exaggeration is 10 x.

Data

In order to solve for the matrix G in our codes, we need to have a triangular mesh representation of the Japan Trench, the large subduction zone on which the Tohoki-Oki earthquake occurred. Our research group has created smoothly interpolated meshes of triangular elements to represent the geometry of the subduction zone based on data from past earthquakes. We also needed to specify the locations of our observation points. Because our goal is to calculate stress changes throughout the crust after the Tohoki-Oki Earthquake, we create a regularly spaced 3D grid of points in the earth's crust around the Japan Trench. In order to calculate the stress changes due to the Tohoki-Oki earthquake, we also needed a slip distribution on the subduction zone. We use a slip distribution created in our research group.

The meshes are stored in two text files; one contains the triangle vertex locations (v.txt), and the other contains the indices to the triangles (c.txt). Both of these text files are generated in Matlab with codes that were written by our research group. The observation point locations are stored in another text file, Obs.txt. Again these text files was generated in Matlab. Note that the observation locations are points in the crust at which we calculate stresses, not GPS observation locations.

The final product of our parallelized code is the Jacobian matrix G. With this Jacobian G, we can solve for the stress changes due to the Tohoku-Oki earthquake in a large 3-dimensional region around Japan and visualize the stress field.

Methods and Design

Each entry in the final Jacobian G depends only on one observation location and one triangle of the TDE. To calculate a single Green's function for a single TDE and observation point, we require two sets angular dislocation calculations for each triangle edge. In addition, we are interested in all 6 independent components of the 3-dimensional stress tensor, so every triangle and observation location pair contributes 6 rows to the Jacobian. This is shown in the schematic figure below: for every triangle and station pair, a total of 2 sets x 6 angular dislocations calculations x 3 triangle edges = 36 total calculations are required. These calculations are computationally intensive, with hundreds to thousands of arithmetic operations involved in each one.

To parallelize the Green's function calculations -- the computation-heavy part of this matrix assembly -- we defined three steps of our project. In each step, we parallelize more of the computation. Because we are interested in problems with far more stations than triangular dislocation elements (in Japan and southern California there are thousands of GPS stations), in all of these steps we parallelize across the stations, and loop over the triangles. Further parallelization, if desired, would involve parallelization across both stations and triangles.

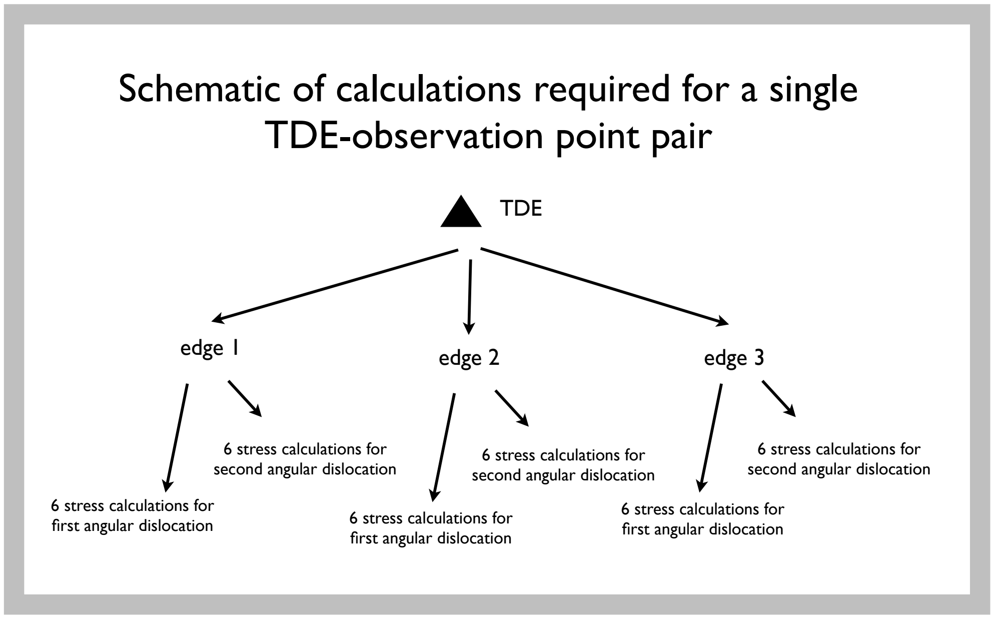

Schematic of calculations required for a single TDE-observation point pair
TDE
edge 1
edge 2
edge 3
6 stress calculations for second angular dislocation
6 stress calculations for first angular dislocation
6 stress calculations for second angular dislocation
6 stress calculations for first angular dislocation
6 stress calculations for second angular dislocation
6 stress calculations for first angular dislocation

FIGURE 1. A schematic diagram of the calculations needed for one station and one triangle pairing.

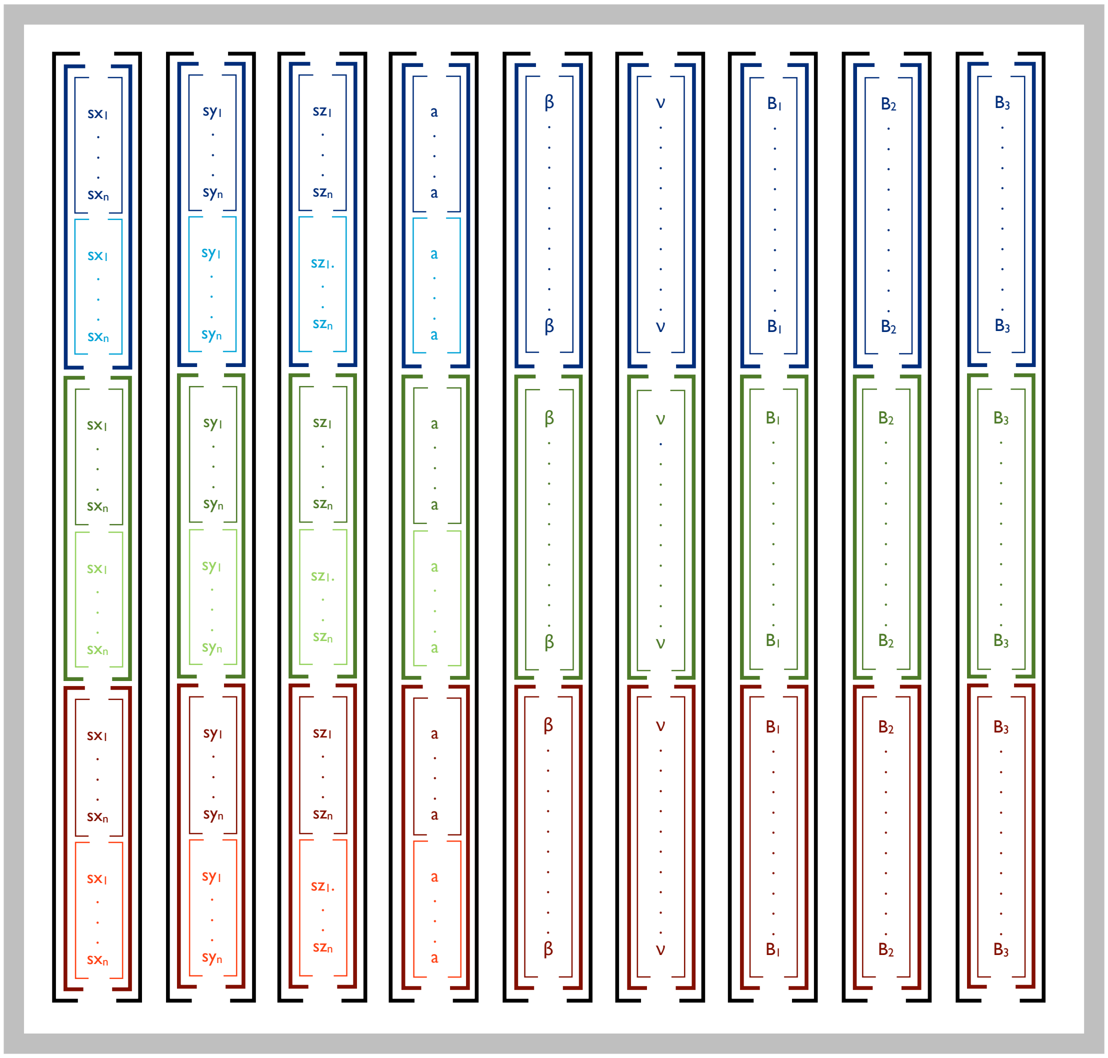


FIGURE 2. Matrix assembly for STEP 3 to parallelize the calculation for each triangle over all stations.

STEP 0. Write the serial code (`JapanStrains_serial.py`).

Our serial code is based on Matlab code by Brendan Meade (Meade, 2007). In this code, the function **`advs`** calculates one set of six angular dislocations for all stations. This function is therefore called twice for every triangle edge, and six times for every triangle. We optimized the serial original code from Meade (2007), by pre-computing some of the repeated expressions.

Inputs into the **`advs`** function are vectors sx, sy, and sz relating a triangle edge to each station location (these are different for each of the two calculations per edge); alpha describing the angle of an angular dislocation (different for each of the two calculations per edge); beta describing the orientation of the triangle edge (different for each edge); and B1, B2, and B3 describing the direction of slip on the triangle (different for each edge).

STEP 1. Parallelize the calculations of the 6 angular dislocations across all stations (`JapanStrains_1.py`).

In this first step, we parallelized the function **`advs`** by including it as a PyCUDA kernel in the file **`advsparallel_1.py`**. We parallelize the calculations of the six angular dislocations across all observation points, but we still have to call this CUDA kernel twice for every triangle edge, and six times for every triangle. However, we get significant speedups in this step (see the Verification and Performance page), because we typically have so many more so observations points than triangles.

STEP 2. Parallelize the calculations of the six angular dislocations for each edge of each triangle across all stations (`JapanStrains_2.py`).

In the second step of parallelization, we wrote a PyCUDA kernel in the file **`advsparallel_2.py`** that calculates both sets of six angular dislocations for all stations in parallel. That is, now, all the calculations for each edge of each triangle are done in parallel. This CUDA kernel is now called only three times for every triangle.

This parallelization step required restructuring of the inputs to the `advs` function. In order to have coalesced memory reads, we made every input a vector with one entry for every observation location. Therefore inputs alpha, beta, B1, B2, and B3 are now repeated to make a vector of the appropriate size.

STEP 3. Parallelize the calculation for each triangle. (`JapanStrains_3.py`).

In the third and final step of parallelization, we wrote a PyCUDA kernel in the file `advsparallel_3.py` that calculates both sets of six angular dislocations for all three edges of a triangle in parallel. That is, now, all the calculations for each triangle are done in parallel.

In this step the inputs to the `advs` function from STEP 2 are stacked to create a single vector for each input for each edge (Figure 2).