FaultMod

Finite Element Software for 3D Physics-Based Fault Models

Michael Barall

Invisible Software, Inc., and USGS

3D Dynamic Rupture Code Validation Workshop
Pomona, California
March 10, 2008

Acknowledgements

- USGS Earthquake Hazards Team.
- Southern California Earthquake Center.
- Pacific Gas & Electric Company / DOE Extreme Ground Motion Project.
- USC Center for High-Performance Computing and Communications.

Plan

1. Introduction to FaultMod.
2. Example – How to set up FaultMod to run TPV8.
3. Technical Details – How FaultMod implements faults.
4. FaultMod architecture for dynamic rupture.
5. Code validation benchmark problems.

FaultMod

- FaultMod is finite-element software, written from scratch for the specific purpose of constructing 3D physics-based fault models.

- Flexible code, designed for a variety of scientific problems:
  - Static mode (coseismic deformation, stress distribution).
  - Quasi-static mode (fault creep, viscoelastic relaxation).
  - Dynamic mode (wave propagation, dynamic rupture).

- "Polymorphic interfaces" make it straightforward to add new capabilities.
  - Cell interface: Supports multiple type of cells (cubes, tetrahedra, prisms, pyramids, and others; linear and quadratic).
  - Material property interface: Supports multiple material types (elastic, viscous, plastic, viscoelastic, and combinations; linear and nonlinear).
  - Fault behaviour interface: Supports multiple fault behavior models (slip weakening, rate-state, kinematic slip).
  - Boundary condition interface: Supports multiple boundary conditions (kinematic, energy-absorbing, free motion, constrained motion).
  - Output driver interface: Supports multiple output formats (time series, contour plot, VTK/ParaView files for 3D visualization, tabular output for testing).

“Tensors All The Way Down”

- The FaultMod finite-element engine is written entirely in terms of tensors:
  - Displacement tensors, force tensors, strain tensors, stress tensors, elasticity tensors, etc.
  - All internal calculations are done in tensor form.

- Mathematically, FaultMod uses the machinery of Riemannian manifolds:
  - Metric tensor (allows use of arbitrary coordinate systems).
  - Physical quantities are represented as covariant and contravariant tensors.
  - Covariant differentiation is performed using Christoffel symbols.
  - Integration performed is using Levi-Civita tensors.

- Advantages of tensors:
  - Provides a rigid mathematical framework for constructing the code.
  - Guarantees that only physically meaningful quantities can appear in the calculation.
  - Guarantees that physical laws are expressed in a proper form.

- FaultMod implements a “tensorized” version of FEM:
  - Replace FEM shape functions with a set of basis vector fields.
  - Derive force-balance equations using tensor mathematics.

Current Status of FaultMod

- Code is portable, modular, and self-contained:
  - Open source, public domain.
  - Written in ISO standard C++ using object-oriented techniques.
  - Not dependent on any external libraries.

- FaultMod versions currently exist for multiple operating systems:
  - Linux / Unix (32-bit and 64-bit).
  - Mac OS X (32-bit and 64-bit).
  - Windows XP / Vista (32-bit and 64-bit, native application).

- Multi-threading via OpenMP is currently running, but the program is only partially converted to multi-threading.

- Multi-processing via MPI is not running yet. Work on an MPI version has started, but considerable work remains to be done.

Example: Running TPV8 on FaultMod

Running FaultMod

- FaultMod can be run in two ways:
  - Interactive: The program gives you a prompt, and you can type in commands.
  - Script-Driven: The program reads a text file and executes the commands therein.
- FaultMod has its own built-in script language, which includes:
  - Script variables and arrays.
  - Integer, floating-point, boolean, and string variables.
  - A library of mathematical functions.
  - Branches, loops, and subroutines.
  - File I/O.
- Advantage: The exact same script files can be used on any system where FaultMod runs.
- Goal: Make it possible to set up and run problems entirely by writing script files, so there is no need to write C++ code for each problem.

The following slides show excerpts from the script file for TPV8, along with an overview of the main steps in setting up and running a problem on FaultMod.

```
# TPV8, 100m cell size

#----- Script variables to define problem parameters

(basefile = "outdir/tpv8_sample")

(CellSize = 100)

(JobTitle = "TPV8, " # CellSize # "m Cell Size, 0.01 sec Time Step, 27 km

(Vp1 = 5716.0)
(Vs1 = 3300.0)
(Density1 = 2700.0)

(mu1 = Vs1*Vs1*density1)
(lambda1 = Vp1*Vp1*Density1 - 2.0*mu1)

(Vp2 = 5716.0)
(Vs2 = 3300.0)
(Density2 = 2700.0)

(mu2 = Vs2*Vs2*density2)
(lambda2 = Vp2*Vp2*Density2 - 2.0*mu2)

(eta_bar_t = 0.1*0.01)
(eta1 = mu1*eta_bar_t)
(zeta1 = (lambda1 + 2.0*mu1/3.0)*eta_bar_t)
(eta2 = mu2*eta_bar_t)
(zeta2 = (lambda2 + 2.0*mu2/3.0)*eta_bar_t)

(TimeStep = 0.01)
(TimeStepCount = 1500)
(TimeStepReport = 2000)

(MeshSize[0] = 42000)
(MeshSize[1] = 60000)
(MeshSize[2] = 54000)

(beta1 = 0.65)
(beta2 = 0.5 * (beta1 + 0.5)**2)

(RatioNormal = 7378.0)
(RatioShear = 7378.0 * 0.55)
(RatioNucleate = 7378.0 * 0.760 * 1.005)
(SWStatic = 0.760)
(SWDynamic = 0.448)
(SWDistance = 0.50)
(SWCohesion = 1.0E6)
```

Step 1. Define script variables to hold the problem parameters:

- Cell size.
- Job name.
- Material properties.
- Viscous damping parameters.
- Time step values.
- Mesh size.
- Algorithmic damping parameters.
- Fault friction parameters.

Use of script variables makes it easy to change problem parameters and rerun the calculation.

```
procedure fault_station (strike, dip) {
    local (filename, fn_strike, fn_dip, eff_strike, eff_dip)

    (eff_strike = idiv(strike, CellSize) * CellSize)
    (eff_dip = idiv(dip, CellSize) * CellSize)

    (filename = basefile # "_fault")

    (filename #= "st")
    (fn_strike = idiv(eff_strike, 100))
    if (fn_strike < 0) {(filename #= "-")}
    if (abs(fn_strike) < 100) {(filename #= "0")}
    if (abs(fn_strike) < 10) {(filename #= "0")}
    (filename #= abs(fn_strike))

    (filename #= "dp")
    (fn_dip = idiv(eff_dip, 100))
    if (fn_dip < 0) {(filename #= "-")}
    if (abs(fn_dip) < 100) {(filename #= "0")}
    if (abs(fn_dip) < 10) {(filename #= "0")}
    (filename #= abs(fn_dip))

    > (filename) {
        call tpv_header ()
        echo ("# location = on-fault, " # fn_strike * 100 # " m along stri
        echo ("# Column #1 = Time (s)")
        echo ("# Column #2 = horizontal slip (m)")
        echo ("# Column #3 = horizontal slip rate (m/s)")
        echo ("# Column #4 = horizontal shear stress (MPa)")
        echo ("# Column #5 = vertical slip (m)")
        echo ("# Column #6 = vertical slip rate (m/s)")
        echo ("# Column #7 = vertical shear stress (MPa)")
        echo ("#")
        echo ("t  h-slip  h-slip-rate  h-shear-stress  v-slip  v-slip-rate
    }

    echo TimeSeries2 @
         (filename) @
         (-eff_dip) @
         (-eff_strike) @
         (0) @
         (10) @
         1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 @
         false false false false @
         true @
         (7) @
         time 1.0 0.0 @
         slip_r 1.0 0.0 @
         slip_rate_r 1.0 0.0 @
         mpa_stress_r 1.0 0.0 @
         slip_d 1.0 0.0 @
         slip_rate_d 1.0 0.0 @
         mpa_stress_d 1.0 0.0
    exit
}
```

Step 2. Write a subroutine to define the file format for on-fault stations:

- Construct file name from station location.
- Create file and write file header.
- Tell FaultMod what values to write on each line of the file.

Similar subroutines define the file formats for off-fault stations and the rupture contour plot.

```
> (basefile # "vmf.txt") {

    echo "FaultMod.Ascii.VisMulti.1.0"

    call fault_station (0, 0)
    call fault_station (4500, 0)
    call fault_station (12000, 0)

    call fault_station (0, 4500)

    call fault_station (0, 7500)
    call fault_station (4500, 7500)
    call fault_station (12000, 7500)

    call fault_station (0, 12000)

    call body_station (3000, 12000, 0)
    call body_station (-3000, 12000, 0)

    call body_station (-3000, 0, 0)
    call body_station (-2000, 0, 0)
    call body_station (-1000, 0, 0)
    call body_station (1000, 0, 0)
    call body_station (2000, 0, 0)
    call body_station (3000, 0, 0)

    call body_station (-1000, 0, 300)
    call body_station (-500, 0, 300)
    call body_station (500, 0, 300)
    call body_station (1000, 0, 300)

    call body_station (-3000, 12000, 12000)
    call body_station (3000, 12000, 12000)

    call contour_plot ()

    call data_table ()

    echo "End"
}
```

Step 3. Specify the desired set of output files:

- One file for each on-fault station.
- One file for each off-fault station.
- One file for the rupture contour plot.

FaultMod can also generate output files for VTK/ParaView, allowing for full 3D visualization of results.

```
#----- Mesh generator

mesh2d {
    reset
    FaultSystem {
        Vertex start (-(15000 + CellSize)) 0
        Vertex end (15000 + CellSize) 0
        Line fault start end
    }
    set {
        BBoxRefine 0
        FaultRefine 1
        FaultQuadratic 5
    }

    basemesh (bigcell) (bigcell) (MeshSize[1]/(CellSize*bigcell))  @
                                 (MeshSize[2]/(CellSize*bigcell))

    refine  1  ((MeshSize[1]/2 - 15000)/CellSize - SmallLayers)  @
               ((MeshSize[2]/(CellSize*2)) - (SmallLayers - 1))  @
               ((MeshSize[1]/2 + 15000)/CellSize + SmallLayers)  @
               ((MeshSize[2]/(CellSize*2)) + (SmallLayers - 1))

    match (CellSize) 0

    draw (basefile # ".gif") 3200 "z1" "z2"

    if (bigcell_v == 1) {
        layer (MeshSize[0]/CellSize) 1 (CellSize)
    } else if (bigcell_v == 2 & bigcell == 2) {
        GDLayer ((CellSize * SmallLayers + 15000)/(CellSize*bigcell_v)) 1
        GDLayer ((MeshSize[0] - (CellSize * SmallLayers + 15000))/(CellSiz
    } else {
        GDLayer (MeshSize[0]/(CellSize*bigcell_v)) 1 (CellSize*bigcell_v)
    }

    export (basefile # "m.txt")
}

reset -n
```

Step 4. Generate the mesh:

- Define the fault system.
- Generate a mesh surrounding the fault system.
- Draw the mesh (so you can check what the mesh generator did).
- Export the mesh to a file.

FaultMod has a built-in mesh generator that can handle curved faults, dipping faults, faults with 3D geometry, surface topography, and to some extent branching faults and multiple faults.

Meshes are constructed with hexahedra, sometimes with prisms adjacent to the fault. Elements can be linear or quadratic.

```
#----- Rheology

grid {
    set quadratic false
}

# Generate rheology query and response files

>(basefile # "gf.txt") {

echo "FaultMod.Ascii.RegionProperty.1.0"

echo 1 Box (-(CellSize*SmallLayers + 15000)) (-(CellSize*SmallLayers + 150
           0 * (CellSize*SmallLayers + 15000) (CellSize*SmallLayers)
echo (Density2) 2 LinElastic (lambda2) (mu2) LinViscosity (zeta2) (eta2)

echo 1 Box (-(CellSize*SmallLayers + 15000)) (-(CellSize*SmallLayers + 150
           (-CellSize*SmallLayers) * (CellSize*SmallLayers + 15000) 0
echo (Density1) 2 LinElastic (lambda1) (mu1) LinViscosity (zeta1) (eta1)

echo 1 Box * * 0 * * *
echo (Density2) 1 LinElastic (lambda2) (mu2)

echo 0
echo (Density1) 1 LinElastic (lambda1) (mu1)

}

rheology {
    queryfile -u (basefile # "m.txt") (basefile # "qf.txt")
    SimpleRheology  -u  (basefile # "gf.txt")  (basefile # "qf.txt")  (bas
}
```

Step 5. Set up rheology:

- Define linear elastic parameters.
- Add viscosity in a layer surrounding the fault, to damp spurious oscillations.
- Store the rheology in a file.

FaultMod supplies elastic, viscous, plastic, and viscoelastic materials, all of which can be selected in the script file. Individual materials can be combined to create more complex materials.

Material properties can be imported from EarthVision, to allow use of 3D geologic models.

```
#----- Boundary conditions

# Generate boundary condition query file

boundary queryfile (basefile # "m.txt") (basefile # "rf.txt") (basefile #

# Generate boundary condition response file for half-space energy-absorbin

boundary generate EnergyAbsorb  ana  (basefile # "bcq.txt")  (basefile # "
```

Step 6. Set up boundary conditions:

– Select energy-absorbing boundary conditions on the bottom and sides of the mesh, free surface on the top.

– Store the boundary conditions in a file.

```
#----- Fault behavior model

> (basefile # "bcx_fm.txt") {

echo boundary generate BoxFaultModel  iiiiiii  (basefile # "bcq.txt")  (b

# Nucleation

for (depth = -13500; depth <= -10500; depth += CellSize) {
echo  -13510  -1510  *  (depth + 10)  1510  *  @
      LinSlipWeak2 0 (SWStatic) (SWDynamic) (SWDistance) 1.0E4 (SWCohesion
     (RatioNucleate*(-depth) + SWCohesion) 0 (RatioNormal*(-depth))  ("@")
}

# Main Fault

for (depth = -15000; depth < 0; depth += CellSize) {
echo  *  *  *  (depth + 10)  *  *  @
      LinSlipWeak2 0 (SWStatic) (SWDynamic) (SWDistance) 1.0E4 (SWCohesion
     (RatioShear*(-depth)) 0 (RatioNormal*(-depth))  ("@")
}

# Surface nodes are placed at an equivalent depth of 1/3 cell

echo  *  *  *  *  *  *  @
      LinSlipWeak2 0 (SWStatic) (SWDynamic) (SWDistance) 1.0E4 (SWCohesion
     (RatioShear*(CellSize/3.0)) 0 (RatioNormal*(CellSize/3.0))

echo
echo exit

}

run (basefile # "bcx_fm.txt")

# Generate boundary condition response file to rupture the desired area

boundary generate BoxRupture  -d  rl  (basefile # "bcq.txt")  (basefile #
  -15010 -15010 *     10  15010 *

reset -n
```

Step 7. Set up fault behavior model:

- Define the nucleation zone.
- Create depth-dependent initial stress.
- Select linear slip weakening with desired parameter values.
- Define the active fault area.
- Store the fault behavior model in a file.

```
#----- Finite element engine

# Load mesh

grid {
    set quadratic false
    set LumpElementMass  1.0
    load (basefile # "m.txt")
}

# Load rheology file

rheology {
    set UseGravity  false
    load (basefile # "rf.txt")
}

# Set up visualization

visualize {
    set MultiFile  (basefile # "vmf.txt")
    set DataType  Multi
}

# Set up stepping parameters

femstep  set  AutoVisualize  true
femstep  set  JobTitle  (JobTitle)

femstep  set  NewmarkBeta1  (beta1)
femstep  set  NewmarkBeta2  (beta2)

# Set up stiffness matrix

femstep  Setup

# Load fault model

femstep  LoadBC  -n  (basefile # "bcr_fm.txt")

# Turn on energy absorbing boundary conditions

femstep  LoadBC  -u  (basefile # "bcr_ea.txt")

# Enable fault rupture in the desired area

femstep  LoadBC  -u  (basefile # "bcr_rup.txt")

# Initial acceleration is zero

femstep  DynamicIC  SetToZero
```

Step 8. Initialize finite element engine:

- Load the mesh.
- Load the rheology.
- Load the output drivers.
- Set up time stepping parameters.
- Set up stiffness matrix.
- Load fault behavior model.
- Load boundary conditions.
- Initialize to zero acceleration.

```
# Loop for time steps

for (TimeStepNumber = 1; TimeStepNumber <= TimeStepCount; ++TimeStepNumber

    # Progress report

    echo  ("Time step number = " # TimeStepNumber)

    # Determine if this is a reporting time step

    (f_reporting = (TimeStepNumber % TimeStepReport == 0))

    # Time step

    femstep  set  StepTitle  @
            ("Time step number = " # TimeStepNumber # ", time = " # (TimeS

    femstep  set  TimeStep  (TimeStep)

    visualize  field  All  (f_reporting)

    femstep  DynamicStep
}

# Finalize visualization

femstep  Finalize

exit
```

Step 9. Perform time steps:

- In a loop, write a progress report and perform a dynamic time step.
- At end of loop, clean up.

Now you can upload the results to the code validation website!

Technical Details –
How FaultMod Implements Faults

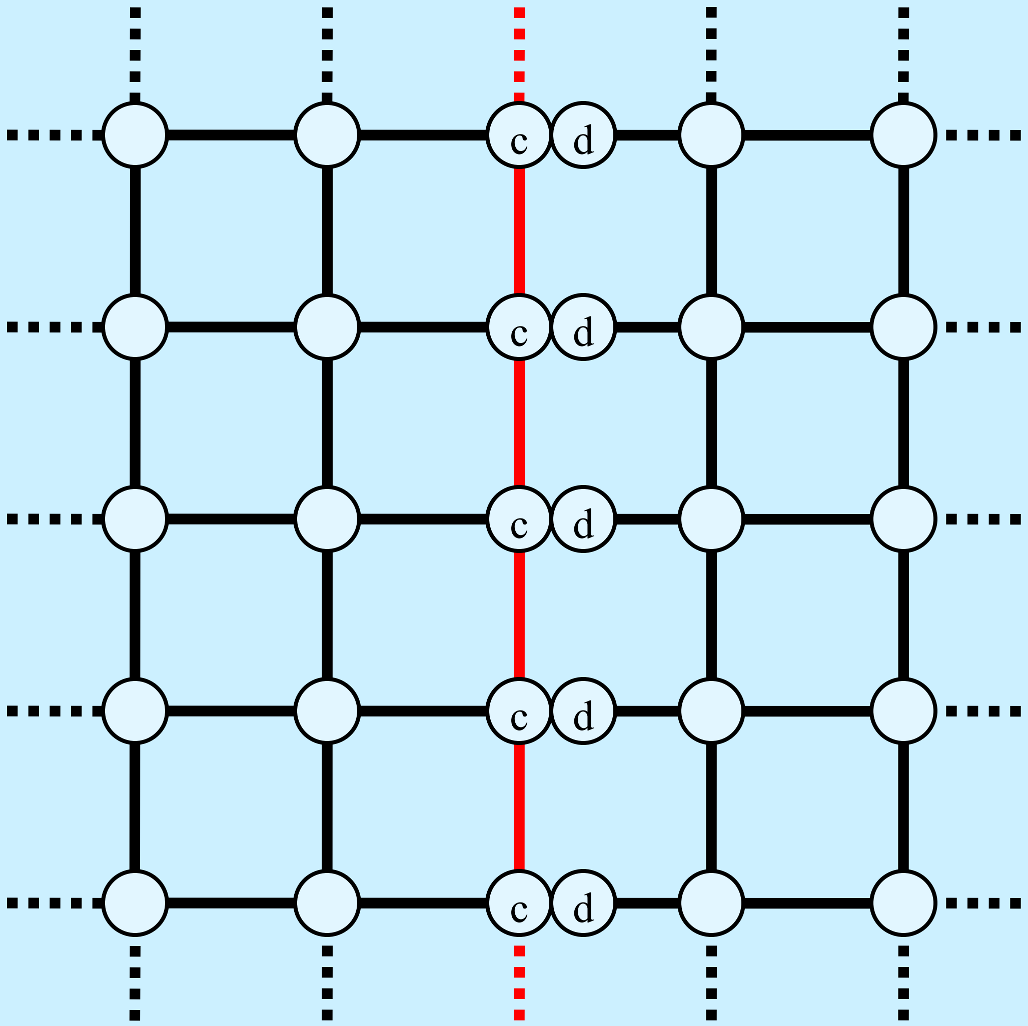
Implementing Faults

- Finite-element “contact problems” are usually solved by adding an auxiliary mechanism, such as Lagrange multipliers or penalty functions.

- Goal: Design an inherent understanding of faults directly into the guts of FaultMod, so no auxiliary mechanism is needed.

- Requirement: Do not introduce any non-physical entities into the calculation.

Common Nodes and Differential Nodes



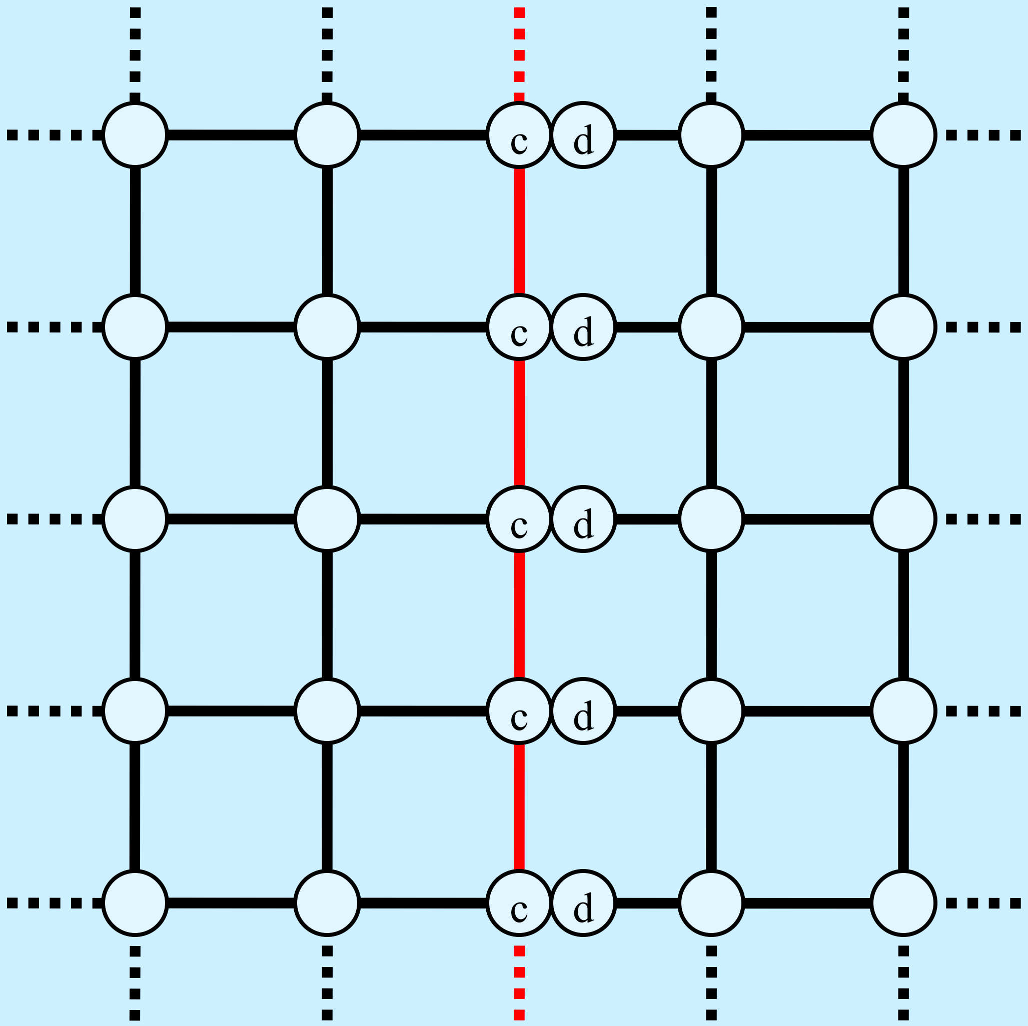
- Fault 🟥
  - Each node location on the fault has two nodes, a common node and a differential node.
- Common Node ⓒ
  - Represents motion common to both sides of the fault.
- Differential Node ⓓ
  - Represents motion of one side of the fault, relative to the other side.
- Displacement on the left side of the fault is:

$$u = u^c$$

- Displacement on the right side of the fault is:

$$u = u^c + u^d$$

Differential Nodes and Fault Variables



With differential nodes, fault variables have simple formulas.

- Fault slip = u^d
- Fault slip rate = v^d
- Traction force on fault = f^d *provided that* $f^c = 0$.
- In dynamic calculations, the traction force is:

$$f^d - (m^d / m^c)\, f^c$$

Where:

u^d = diff. node displacement

v^d = diff. node velocity

f^d = nodal force on diff. node

f^c = nodal force on common node

m^d = mass for diff. node

m^c = mass for common node

Representing Constraints as Projection Operators

Typically we want to constrain the differential node so u^d is tangent to the fault.

Define a 3×3 operator P as:

$$P_{ij} = \delta_{ij} - n_i n_j \qquad \text{where} \qquad n = \text{unit normal to fault}$$

Then P is a projection operator with a two-dimensional range:

$$P^2 = P \qquad \text{(definition of projection operator)}$$

The vector u^d is tangent to the fault if and only if:

$$u^d \in Range(P)$$

or equivalently:

$$(I - P)\, u^d = 0$$

A projection operator P is a general way to specify constraints. For example, you can get pure strike-slip motion by selecting P with a one-dimensional range, and you can lock the node by selecting $P = 0$.

Projection operators can also be applied at boundary nodes, to specify boundary conditions.

Assembling a Global Projection Operator

Assign a projection operator *P* to each node:

- For differential nodes, *P* maintains the fault constraint.
- For boundary nodes, *P* maintains the boundary condition.
- For other nodes, take *P*=*I* to allow free motion.

Assemble all the node projection operators into a global projection operator ***P***, which is a block-diagonal $3N\times 3N$ matrix, where *N* is the number of nodes.

Then all the fault constraints and all the boundary conditions are satisfied if and only if the displacement vector ***u*** satisfies:

$$\boldsymbol{u} - \boldsymbol{u}_0 \in Range(\boldsymbol{P})$$

or equivalently:

$$(\boldsymbol{I} - \boldsymbol{P})(\boldsymbol{u} - \boldsymbol{u}_0) = \mathbf{0}$$

(The additional term $\boldsymbol{u}_0$ permits the specification of kinematic fault slip and kinematic boundary conditions.)

The global projection operator ***P*** is a compact mathematical way to specify all the fault constraints and boundary conditions.

Finite-Element Force Balance Equation

Without constraints, a finite-element code balances the forces at each node by solving the system of equations:

$$\boldsymbol{E}\,\boldsymbol{u} = \boldsymbol{f}$$

where:

$\boldsymbol{E}$ = Stiffness matrix (a positive-definite symmetric $3N\times 3N$ matrix).

$\boldsymbol{u}$ = Displacement vector.

$\boldsymbol{f}$ = Vector of applied nodal forces.

(The above equation is for static problems. The concepts described here work with dynamic problems too, but the equations are more complicated. To keep the presentation simple, we use static equations for the next few slides.)

When there are constraints, the force-balance system of equations becomes:

$$\boldsymbol{P}^T\,\boldsymbol{E}\,\boldsymbol{u} = \boldsymbol{P}^T\,\boldsymbol{f}$$

which must be solved simultaneously with the constraint:

$$\boldsymbol{u} - \boldsymbol{u}_0 \in \mathit{Range}(\boldsymbol{P})$$

Notice that $\boldsymbol{P}^T\,\boldsymbol{E}$ is neither symmetric nor positive-definite, and is not of full rank.

Preconditioned Conjugate Gradient Method

To solve $\boldsymbol{E}\,\boldsymbol{u} = \boldsymbol{f}$:

1. $\boldsymbol{r}_0 = \boldsymbol{f} - \boldsymbol{E}\,\boldsymbol{u}_0$
2. $\boldsymbol{z}_i = \boldsymbol{M}^{-1}\,\boldsymbol{r}_i$
3. $\rho_i = \boldsymbol{r}_i \cdot \boldsymbol{z}_i$
4. $\boldsymbol{p}_i = \boldsymbol{z}_i + (\rho_i \,/\, \rho_{i-1})\,\boldsymbol{p}_{i-1}$
5. $\boldsymbol{q}_i = \boldsymbol{E}\,\boldsymbol{p}_i$
6. $\gamma_i = \boldsymbol{q}_i \cdot \boldsymbol{p}_i$
7. $\alpha_i = \rho_i \,/\, \gamma_i$
8. $\boldsymbol{u}_{i+1} = \boldsymbol{u}_i + \alpha_i\,\boldsymbol{p}_i$
9. $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\,\boldsymbol{q}_i$
10. If not converged, go to step 2.

This method is commonly used, but an auxiliary mechanism is needed if you must enforce constraints.

Constrained Preconditioned Conjugate Gradient Method

To solve $\boldsymbol{P}^T \boldsymbol{E}\, \boldsymbol{u} = \boldsymbol{P}^T \boldsymbol{f}$ subject to constraint $\boldsymbol{u} - \boldsymbol{u}_0 \in Range(\boldsymbol{P})$:

1. $\boldsymbol{r}_0 = \boldsymbol{P}^T\,(\boldsymbol{f} - \boldsymbol{E}\,\boldsymbol{u}_0)$
2. $\boldsymbol{z}_i = \boldsymbol{M}^{-1}\,\boldsymbol{r}_i$
3. $\rho_i = \boldsymbol{r}_i \cdot \boldsymbol{z}_i$
4. $\boldsymbol{p}_i = \boldsymbol{z}_i + (\rho_i / \rho_{i-1})\,\boldsymbol{p}_{i-1}$
5. $\boldsymbol{q}_i = \boldsymbol{P}^T\,\boldsymbol{E}\,\boldsymbol{p}_i$
6. $\gamma_i = \boldsymbol{q}_i \cdot \boldsymbol{p}_i$
7. $\alpha_i = \rho_i / \gamma_i$
8. $\boldsymbol{u}_{i+1} = \boldsymbol{u}_i + \alpha_i\,\boldsymbol{p}_i$
9. $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\,\boldsymbol{q}_i$
10. If not converged, go to step 2.

All that's necessary to maintain fault constraints and boundary conditions is to add some ***P***'s to the conjugate gradient method, and choose a preconditioner ***M*** satisfying:

$$\boldsymbol{r} \in Range(\boldsymbol{P}^T) \;\Rightarrow\; \boldsymbol{M}^{-1}\,\boldsymbol{r} \in Range(\boldsymbol{P})$$

This method is used by FaultMod.

Nonlinear Equations – Constrained Newton's Method

For nonlinear problems, the force-balance system of equations becomes:

$$\boldsymbol{P}^T \boldsymbol{E}(\boldsymbol{u}) = \boldsymbol{P}^T \boldsymbol{f}$$

where $\boldsymbol{E}(\boldsymbol{u})$ is a nonlinear function of $\boldsymbol{u}$. This is to be solved along with the constraint:

$$\boldsymbol{u} - \boldsymbol{u}_0 \in Range(\boldsymbol{P})$$

Introduce an approximation:

$$\boldsymbol{E}(\boldsymbol{u}_0 + \delta\boldsymbol{u}) \approx \boldsymbol{E}(\boldsymbol{u}_0) + \boldsymbol{K}(\boldsymbol{u}_0)\,\delta\boldsymbol{u}$$

where $\boldsymbol{K}(\boldsymbol{u}_0)$ is the tangent stiffness matrix.

Then the force-balance equations become a linear system of equations for $\delta\boldsymbol{u}$:

$$\boldsymbol{P}^T \boldsymbol{K}(\boldsymbol{u}_0)\,\delta\boldsymbol{u} = \boldsymbol{P}^T (\, \boldsymbol{f} - \boldsymbol{E}(\boldsymbol{u}_0)\,)$$

which must be solved simultaneously with the constraint:

$$\delta\boldsymbol{u} \in Range(\boldsymbol{P})$$

The constrained preconditioned conjugate gradient method can solve the above system of equations for $\delta\boldsymbol{u}$, giving an approximate solution $\boldsymbol{u} \approx \boldsymbol{u}_0 + \delta\boldsymbol{u}$.

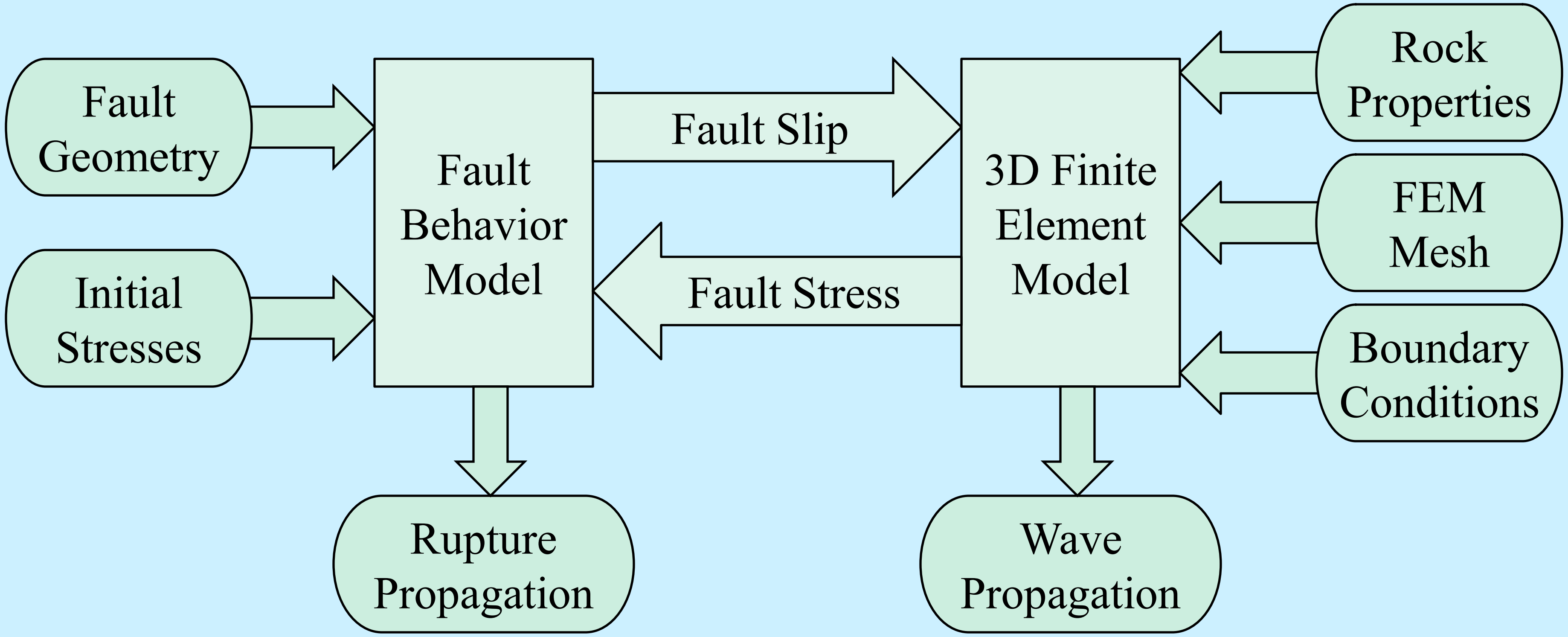
Iterating the above procedure gives a constrained Newton's method, for solving a nonlinear system of equations subject to constraints.

FaultMod Architecture for Dynamic Rupture

Dynamic Rupture Software Architecture

Conceptually, there are two main interacting components:

- 3D Finite Element Model: Performs a dynamic wave-propagation calculation, taking into account rock properties, and calculates stresses acting on the fault.
- Fault Behavior Model: Uses the fault constitutive law to calculate fault slip, slip rate, and acceleration, given fault stresses and geometry.



Dynamic Rupture Computational Cycle

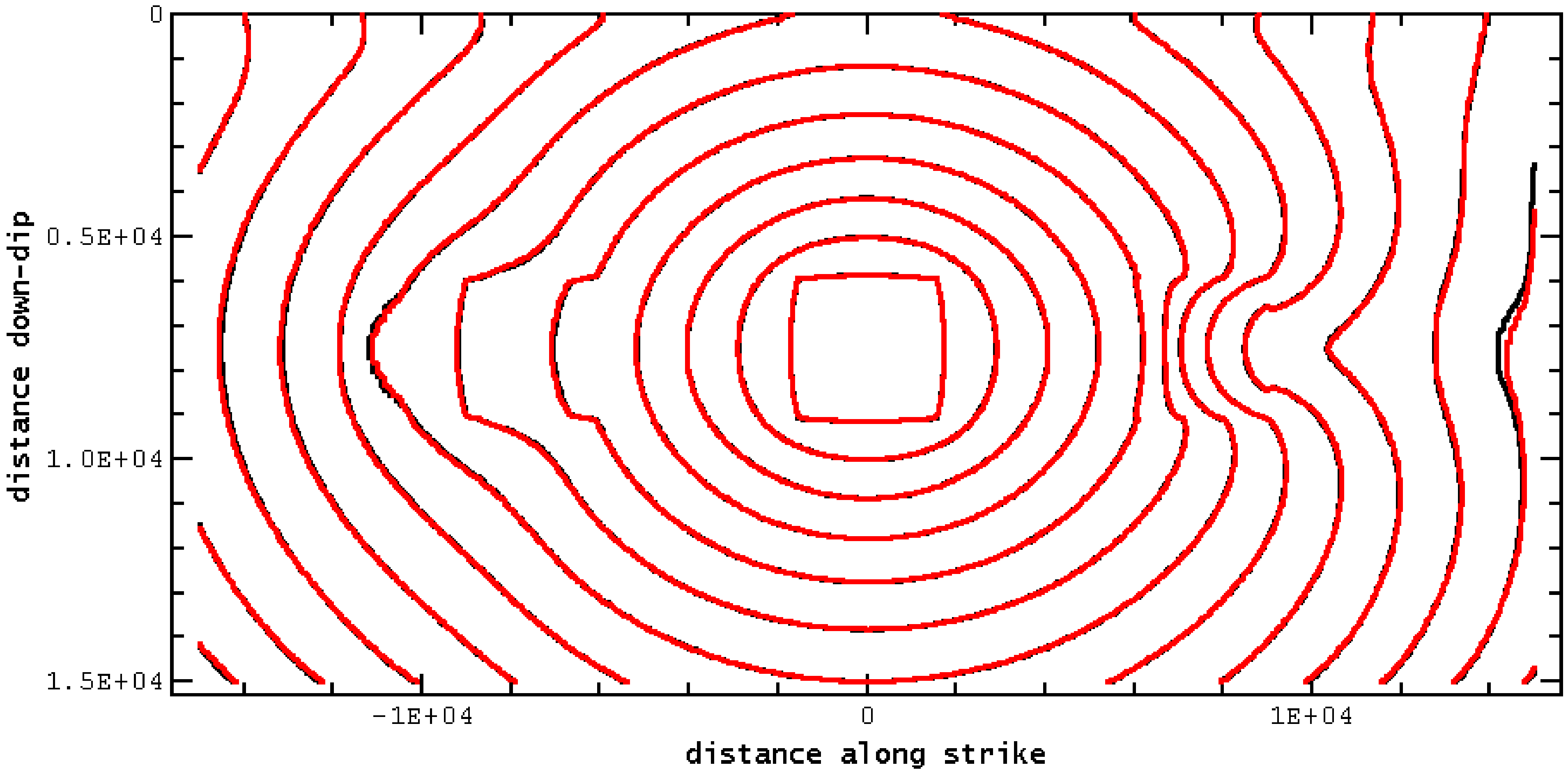
For each time step, FaultMod does one iteration of the following loop:

1. Take a trial step, assuming acceleration is unchanged from the previous time step.
2. Calculate nodal forces at the end of the trial step, and convert them into traction forces on the fault.
3. Fault behavior model – Solve the fault constitutive equation, obtaining fault slip, slip rate, and acceleration.
4. Apply the calculated fault slip, slip rate, and acceleration to the finite-element model as a kinematic boundary condition.
5. Finite element model – Solve the finite-element force balance equations, obtaining displacement, velocity, acceleration, and nodal forces throughout the mesh.
6. Update internal state variables of the fault behavior model.
7. Go to step 1.

This is a generalization of the method described by Joe Andrews, *BSSA*, 1999.

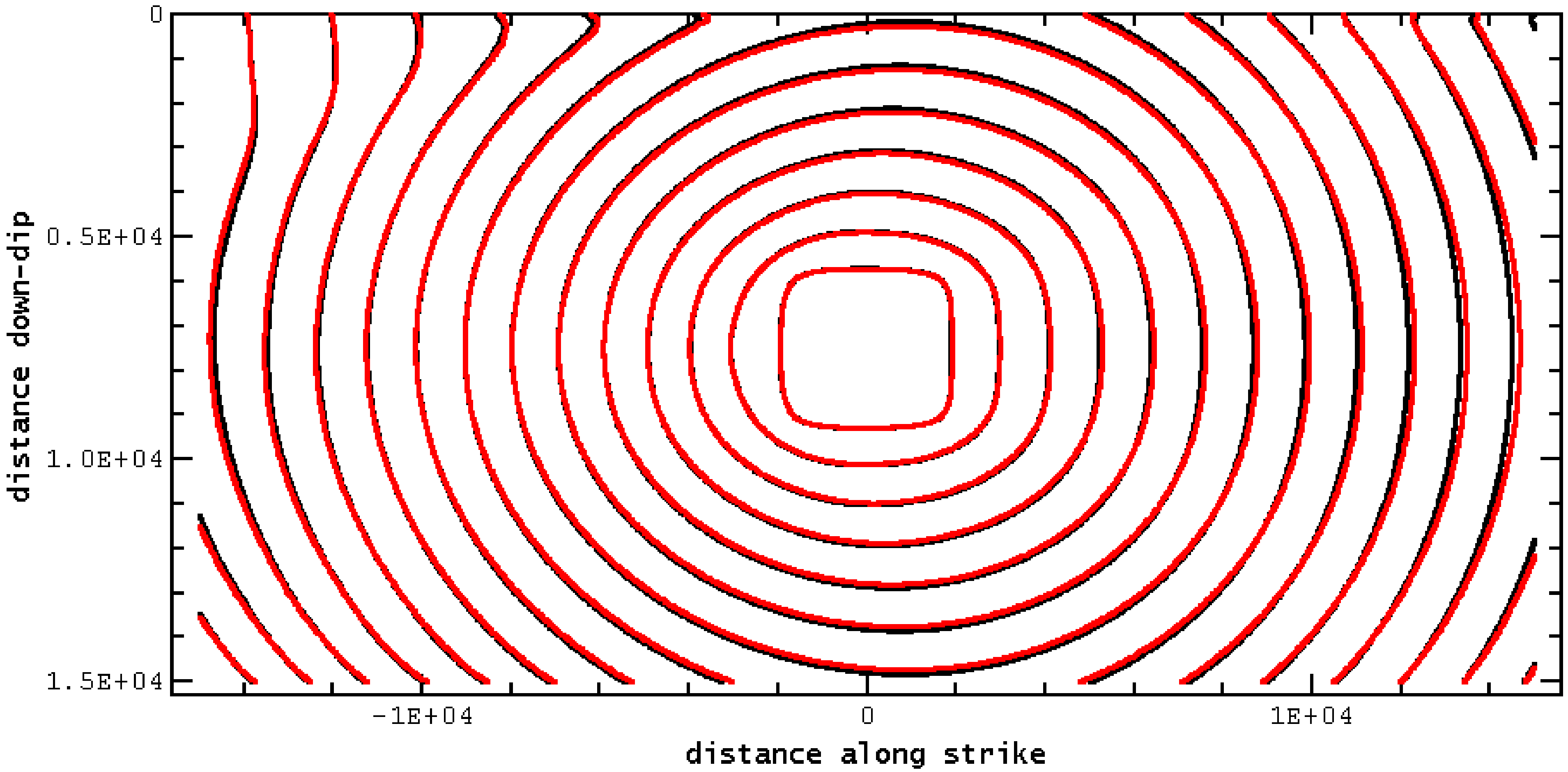
Code Validation Benchmark Problems

TPV5 (100m Node Spacing)



Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

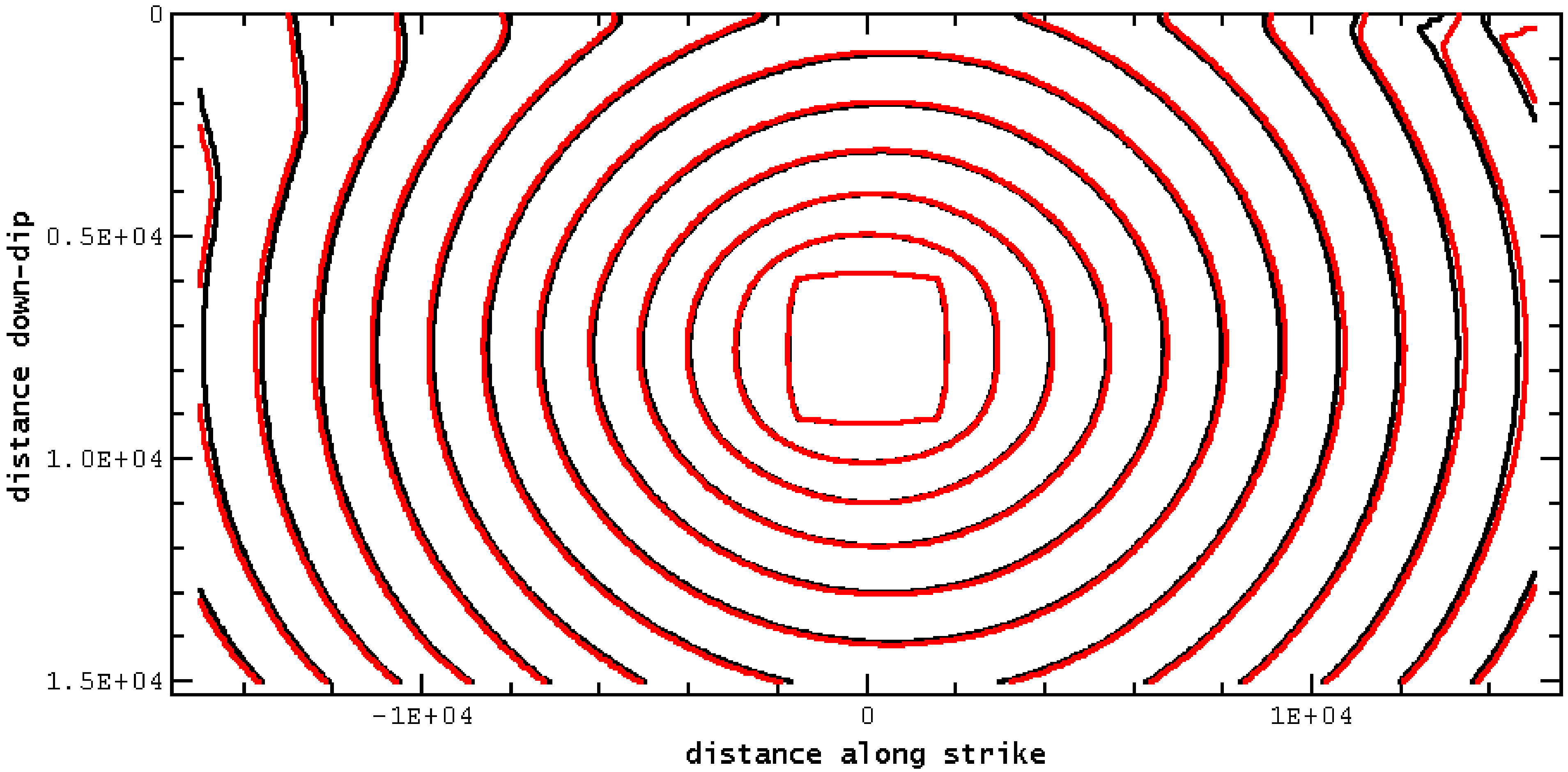
TPV6 (Bi-Material Problem, 50m)



Red = FaultMod Finite-Element (Barall)

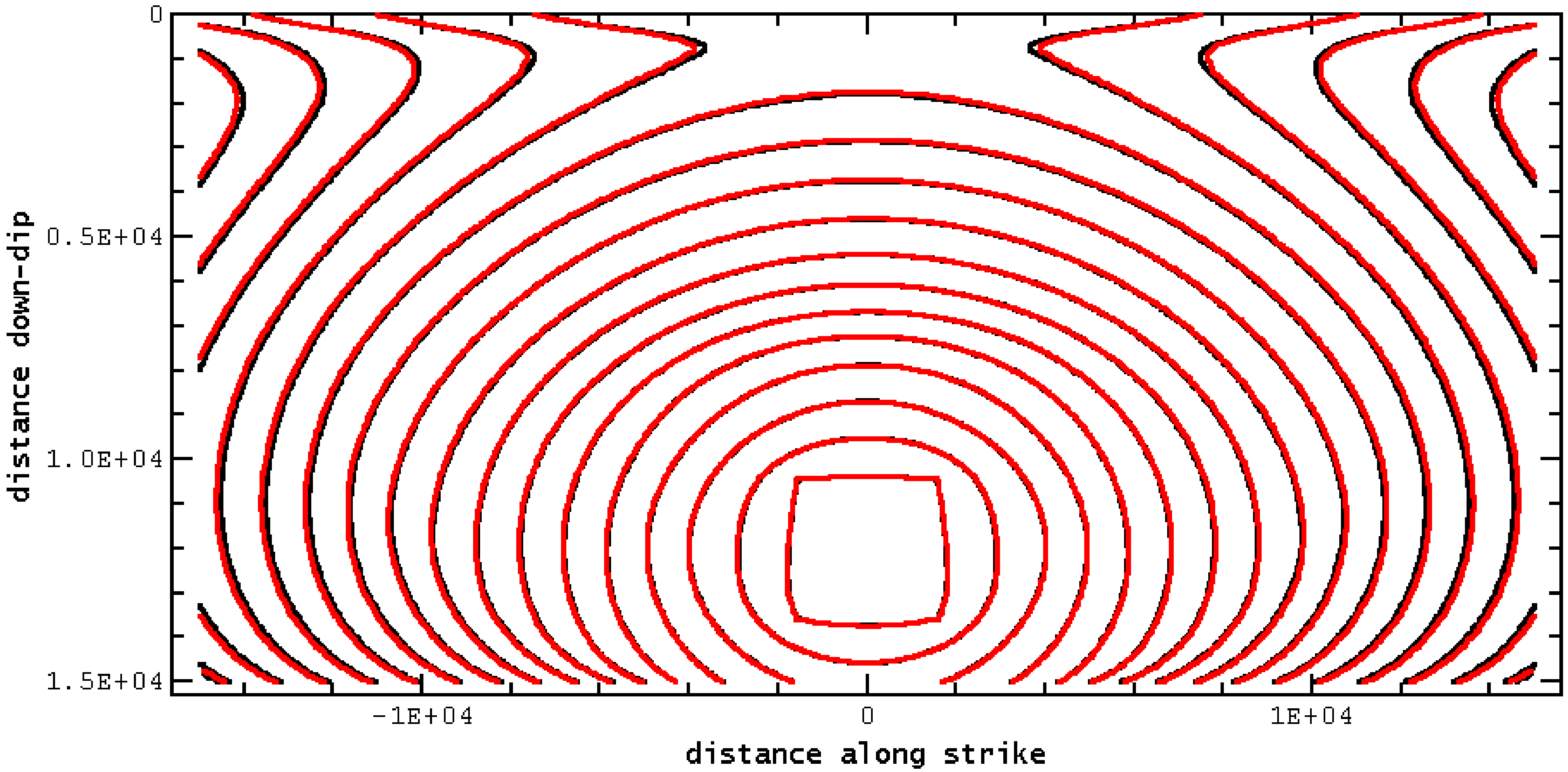
Black = DFM Finite-Difference (Day/Dalguer)

TPV7 (Bi-Material Problem, 100m)



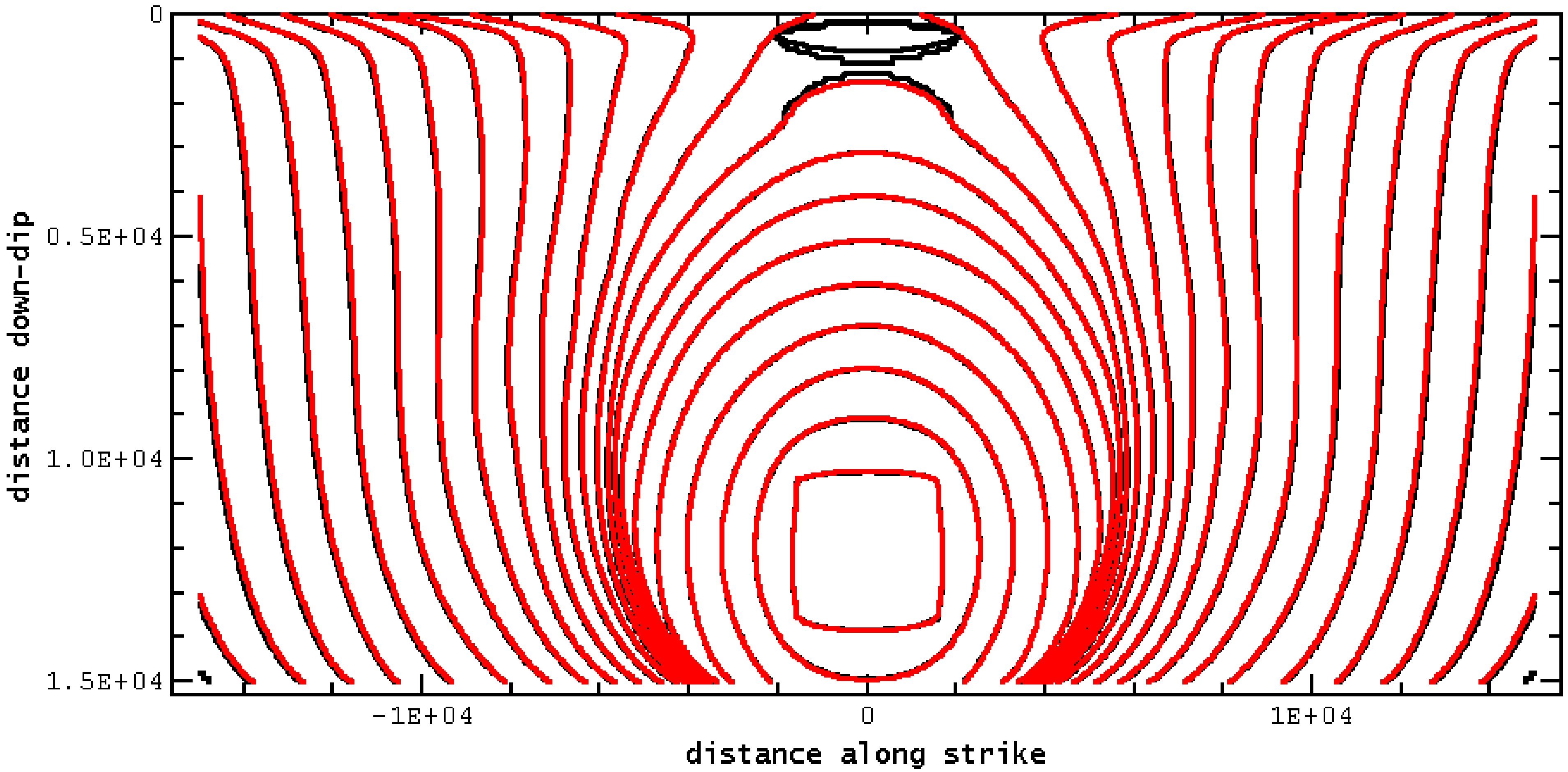
Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

TPV8 (Depth-Dependent Stress, Strike-Slip, 100m)



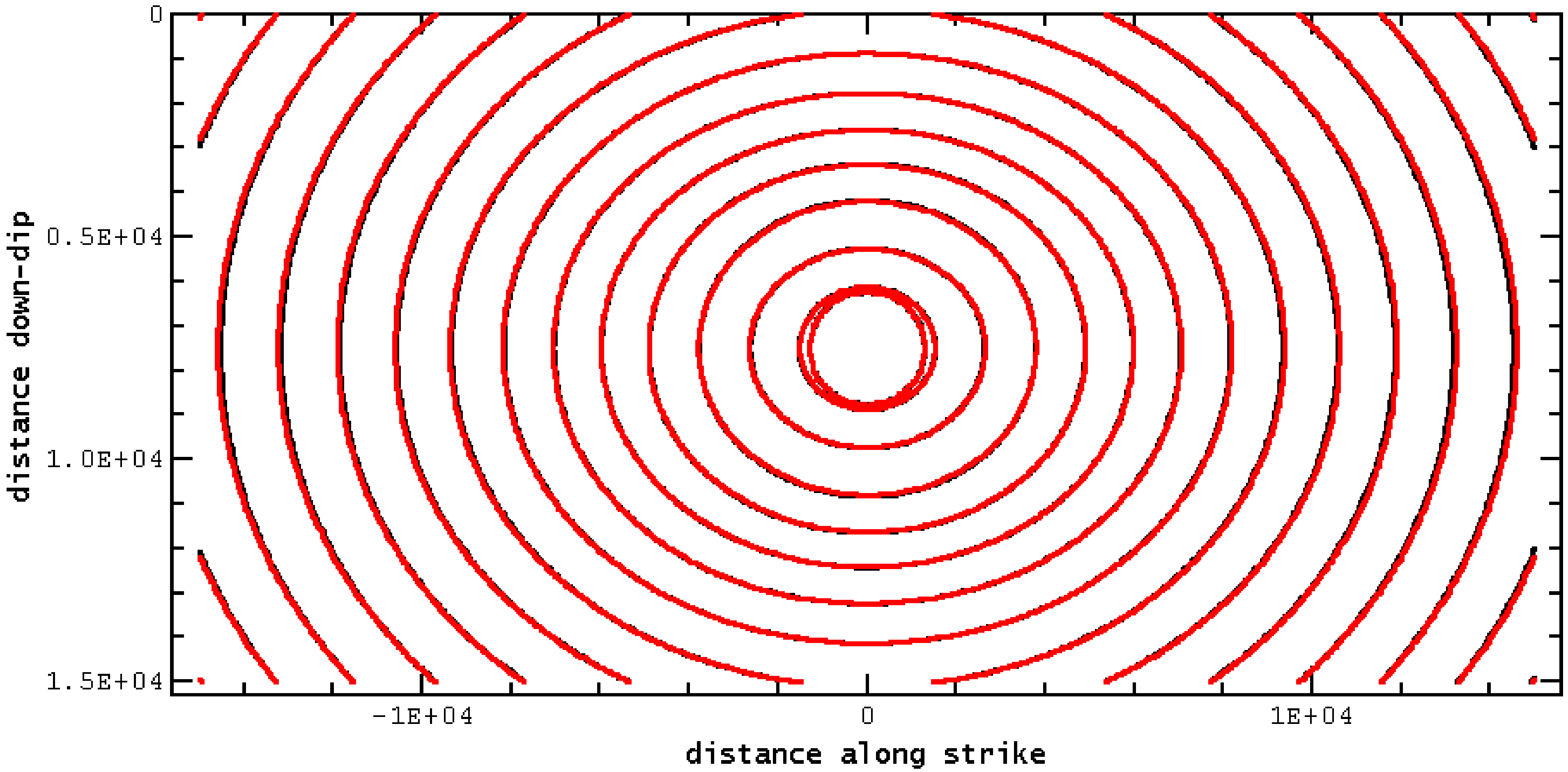
Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

TPV9 (Depth-Dependent Stress, Dip-Slip, 100m)



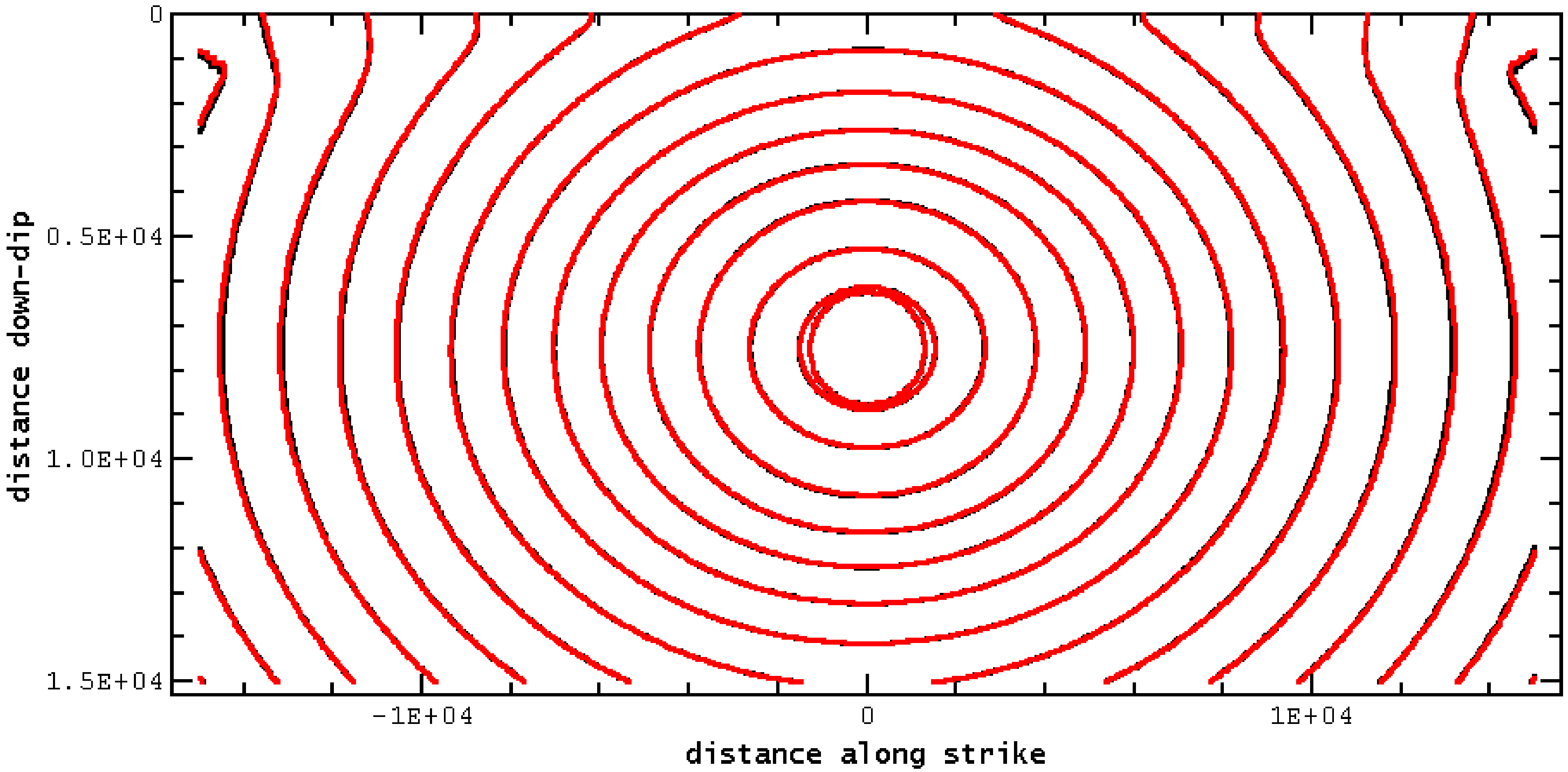
Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

TPV101 (Rate-State Friction, Full-Space, 100m)



Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

TPV102 (Rate-State Friction, Half-Space, 100m)



Red = FaultMod Finite-Element (Barall)
Black = DFM Finite-Difference (Day/Dalguer)

Thank You!