Growth by Optimization of Work
The User Manual
Version 2015
Oct. 2015

Table of Contents

1. Introduction

Growth by Optimization of Work (GROW) predicts how fractures and faults grow, interact and eventually link through a work optimization approach. This user manual describes 1) the underlying algorithm employed by GROW, 2) the input required by GROW, 3) the output produced and 4) the files and compilers required to run GROW.

We maintain a forum where users of GROW can post questions at: https://www.geo.umass.edu/faculty/cooke/software.html. Please contact us with comments or questions at this address!

When publishing results of GROW models, please reference the following paper, which describes the GROW algorithm and an application to a crustal-scale releasing stepover:

McBeck, J. A., Madden, E., Cooke, M. L., 2016. Growth by Optimization of Work (GROW): A new modeling tool that predicts fault growth through work minimization. Computers and Geosciences.

2. General Algorithm

Tectonic environments evolve to optimize work and fractures grow in the direction that optimizes work (Cooke and Madden, 2014). To model fracture growth through work optimization we use a Boundary Element Method (BEM) tool in which all the fractures, faults and boundaries in a system are discretized into linear displacement discontinuity elements. We call this tool GRowth by Optimization of Work (GROW). To simulate fracture growth with GROW, elements are added radially to the tip of propagating fractures in the direction that optimizes external work divided by newly added fracture area in that increment of growth $W_{ext}/\Delta A$. More specifically, in each iteration of crack growth one element is added to each fault tip that is propagating. If one fault is propagating, the element that optimizes work is added to the tip of the propagating fault in one propagation of crack growth. If two faults are growing, two elements will be added to the corresponding faults after one propagation of crack growth. If two tips of one fault are growing then an element will be added to both tips of that fault. Each added element optimizes the external work of the system divided by the fracture area propagated in that propagation of growth, $W_{ext}/\Delta A$. See the fabulous review Cooke and Madden (2014) for a full explanation of the theoretical basis of work optimization and its application to fault growth.

2.1 Fric2D

Because GROW repeatedly executes Fric2D to calculate W_{ext}, here we briefly describe the functionality of the two-dimensional BEM modeling tool Fric2D (Cooke and Pollard, 1997). In Fric2D dislocation surfaces, such as faults and the boundaries of the model, are discretized into elements that are free to open or slip, but not interpenetrate, in response to tractions or displacements applied to the boundaries (Cooke and Pollard, 1997). Fric2D solves the quasi-static equations of deformation on each element given a set of boundary conditions to determine the displacement and tractions on each boundary and fault

produced by a given stress state (Cooke and Pollard, 1997). Fric2D input files enable the user to describe the boundary conditions and initial fault geometry, and specify how each fault is growing. The user can allow the propagation of the tip of one fault, both tips of a fault and multiple tips of multiple faults. GROW automatically executes Fric2D and reads Fric2D output files to calculate the external work of the system, W_{ext}, in order to simulate the growth of the most mechanically efficient fault network.

2.2. One Fault

If the user specifies that the tip of one fault is propagating, GROW searches for the orientation of the potential element added to the tip of that fault that optimizes external work divided by the new fracture area added in that propagation of growth ($W_{ext}/\Delta A$). GROW uses $W_{ext}/\Delta A$, rather than W_{ext} because systems with more fracture area will generally always be more efficient that systems with less fracture area. Because Fric2D assumes a unit thickness of 1 meter, fracture area added in a propagation of growth is simply calculated by the difference in total fracture area before and after an increment of growth.

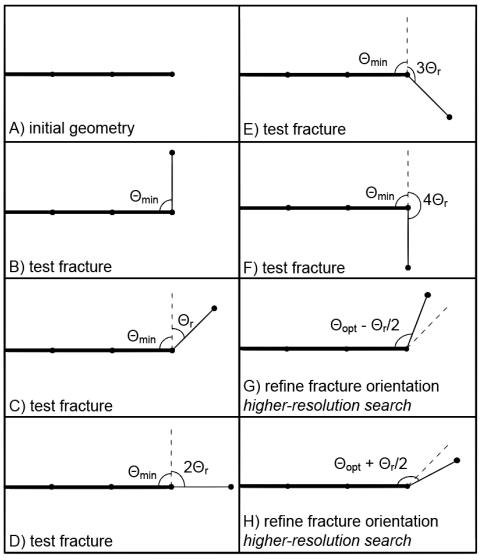
When the user calls GROW they must provide an input file that describes an initial fault geometry and loading conditions, *input.in*, and the value in degrees of the resolution angle, θ_{res}, the minimum angle, θ_{min}, and the maximum angle, $\theta_{max,}$ which describe the parameter space in which GROW searches for the most efficient fault geometry. The input file must end in *.in*, and must exist in the current directory with the GROW executable. For more details about this input file see the "Input" section below.

θ_{min} and θ_{max} specify the range of orientations that GROW searches for the element that optimizes work, and θ_{res} determines the resolution of this search. For example, immediately after the user calls GROW, GROW reads the initial input file, determines what faults are growing and, if one tip of one fault is growing, creates a Fric2D input file that includes the original fault geometry and an element oriented θ_{min} in the clockwise direction from the growing fault tip. GROW then calls the BEM code Fric2D to calculate the tractions and displacements along each element in this new geometry. This execution of Fric2D produces a Fric2D output file from which GROW then determines $W_{ext}/\Delta A$ of the system with the perl script *Wext.pl*. Next GROW creates a Fric2D input file from the initial input file specified by the user with an element added $\theta_{min} + \theta_{res}$ clockwise from the tip of the fault, and similarly calculates $W_{ext}/\Delta A$. Each subsequent orientation tested is θ_{res} more than the previous orientation tested. The last orientation of the new element that GROW tests in this increment of crack growth is strictly less than θ_{max}. After GROW calculates $W_{ext}/\Delta A$ for each orientation, GROW identifies the orientation of the newly added element that optimizes external work, θ_{opt}. After this first broad search GROW creates a new Fric2D input file from the initial input file with an element added $\theta_{opt} - \theta_{res}/2$ clockwise from the tip of the fault, and finds the $W_{ext}/\Delta A$ of this system. GROW also calculates $W_{ext}/\Delta A$ for the element added $\theta_{opt} + \theta_{res}/2$ clockwise from the tip of the fault. We refer to this higher-resolution search as the "tuning" propagation step. We refer to the first initial, lower-resolution search of the efficient orientations as the "first pass" in a propagation step. Figure 1 demonstrates the fault geometries tested in the first propagation of crack growth when one tip of one fault is growing.

After W_{ext} is calculated for each of the aforementioned geometries GROW then identifies the geometry that optimizes $W_{ext}/\Delta A$. If the user specifies displacements as boundary conditions then the most efficient geometry will have the smallest value of $W_{ext}/\Delta A$, and thus minimizes work. If the user specifies tractions as boundary conditions then the most efficient geometry will have the highest $W_{ext}/\Delta A$, and thus maximizes work. Although a system with tractions as boundary conditions will evolve so that $W_{ext}/\Delta A$ increases, we sometimes refer to this general method as "work minimization." The user may use either tractions or displacements as boundary conditions when running GROW, but should not use both methods of loading when imposing non-zero boundary conditions. For example, the user may set a displacement boundary condition where 0 meters of slip is prescribed to one boundary, and a traction boundary condition where 1 MPa of normal stress is prescribed to a different boundary. But the user should not set one boundary to 1 meters of normal displacement and a different boundary to 1 MPa of normal stress. Additionally, displacement boundary conditions often produce more numerically robust results than tractions, and thus is the recommended method of loading.

After GROW identifies the geometry that optimizes work, the Fric2D input file that specifies this geometry is saved as the *efficient input file*, which is named with *.eff* (in our example, *input.eff*). GROW then treats *input.eff* as the new initial input file to which to add potential growth elements, calculate $\Delta W_{ext}/\Delta A$ and find the most efficient geometry in all subsequent propagations of crack growth.

Figure 1: Schematic representation of all fracture geometries tested in first propagation of crack growth when user specifies the minimum angle as θ_{min}, resolution angle as θ_r, and maximum angle as $\theta_{min}+C\theta_r$ (where C=4.5 in this example). A) shows the initial fault geometry, B-F) shows the fault geometries for which $\Delta W_{ext}/\Delta A$ is calculated in the first, lower-resolution search of a propagation of crack growth. G-H) show the fault geometries tested in the *tuning* sequence of this propagation. In this example the orientation of the newly added element that optimizes $\Delta W_{ext}/\Delta A$ in the first broad search of this propagation is $\theta_{min} + \theta_r$. In B-F) the dashed lines show the location of an element oriented at θ_{min} from the fault tip, and in G-H) the dashed line shows the location of the most efficiently oriented potential element identified in the first lower resolution search of this propagation of crack growth.



2.3. Two or More Faults

If the user specifies that two faults are growing, GROW searches for the orientation of the element that optimizes work for one of the two faults by invoking the first pass of the algorithm used to find the most efficient geometry when one fault is growing. After this orientation is found, the element at the specified orientation is added to the initial input file. Then GROW searches for the most efficient element added to the tip of the other fault with an input file that includes the element added to the other fault. GROW sorts the faults by name alphanumerically to determine what fault propagates first, i.e., what fault will have elements added to it first. For example, a fault named *coachella* will propagate before a fault named *san_andreas*.

The tuning propagation step functions slightly differently when more than one fault is propagating. If the most efficient element added to the fault *coachella* is oriented θ_c, and the most efficient element added to *san_andreas* is oriented θ_s, four additional geometries are tested that search the parameter space near the most efficient orientations found in the first broad search. Four possible tuning geometries are listed in Table 1a, which shows the orientation of the element added to each fault in each of the geometries. Table 1b shows an alternative set of tuning geometries, which would be tested for efficiency if one of the tuning geometries is found to be more efficient than the most efficient geometry identified in the first broad search. After $W_{ext}/\Delta A$ is calculated for the four geometries listed in Table 1a or Table 1b, the geometry that optimizes $\Delta W_{ext}/\Delta A$ is identified from the tuning geometries and the most efficient geometry found in the first broad search. At the end of this tuning propagation GROW creates the most *efficient input file* found thus far by adding two elements to the corresponding fault tips in the initial input file. Like the algorithm with only one fault, GROW then uses this *efficient input file* as the next initial input file to which to add elements. Figure 2 illustrates an example propagation sequence when two faults are growing.

If the user specifies that both tips of a fault are growing, GROW handles the propagation of both tips of a fault just as if each tip of two faults were growing. If more than two faults are propagating GROW finds the most efficient geometry with the same first broad search algorithm, where GROW searches for the efficient orientation of each newly added element sequentially. In the tuning propagation step, several more geometries are tested. Table 2 lists six geometries that would be tested if in the first pass of the algorithm GROW determined that the most efficient orientation of the elements added to three hypothetical faults *banning*, *coachella*, and *mill,* are θ_b, θ_c, and θ_m, respectively.

Table 1a: Geometries tested in the tuning sequence of a propagation of crack growth after the first pass of the search algorithm found that the most efficient orientation of an element added to the tip of the *coachella* fault is θ_c, and the most efficient orientation of an element added to *san_andreas* is θ_s. In this example the resolution angle set by the user is θ_r. When $W_{ext}/\Delta A$ was calculated for these geometries, a geometry more efficient than an element added to *coachella* at θ_c and *san_andreas* at θ_c was not identified.

geometry	*coachella*	*san_andreas*
1	θ_c	$\theta_s - \theta_r/2$
2	θ_c	$\theta_s + \theta_r/2$
3	$\theta_c - \theta_r/2$	θ_s
4	$\theta_c + \theta_r/2$	θ_s

Table 1b: An example of how GROW will test slightly different tuning geometries if one of the tuning geometries is found to be more efficient than the most efficient geometry identified in the first broad search. Here, the efficient orientations identified in the first search of *coachella* and *san_andreas* was, respectively, θ_c and θ_s but tuning geometry 2 was determined to produce a greater change in normalized work than the previous most efficient geometry.

geometry	*coachella*	*san_andreas*
1	θ_c	$\theta_s - \theta_r/2$
2	$\boldsymbol{\theta_c}$	$\boldsymbol{\theta_s + \theta_r/2}$
3	$\theta_c - \theta_r/2$	$\theta_s + \theta_r/2$
4	$\theta_c + \theta_r/2$	$\theta_s + \theta_r/2$

Table 2: Geometries tested in tuning sequence of propagation of crack growth after the first pass of the search algorithm found that the most efficient orientation of the elements added to the faults *banning, coachella* and *mill* are θ_b, θ_c and θ_m, respectively. The resolution angle is θ_r. The below sequence of tuning geometries assumes that the most efficient geometry identified in the first broad search is more efficient than tuning geometries tested.

geometry	*banning*	*coachella*	*Mill*
1	θ_b	θ_c	$\theta_m - \theta_r/2$
2	θ_b	θ_c	$\theta_m + \theta_r/2$
3	θ_b	$\theta_c - \theta_r/2$	θ_m
4	θ_b	$\theta_c + \theta_r/2$	θ_m
5	$\theta_b - \theta_r/2$	θ_c	θ_m
6	$\theta_b + \theta_r/2$	θ_c	θ_m

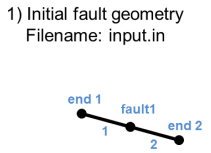
Figure 2: This sequence of sketches shows an example propagation sequence when two faults are growing. This figure also demonstrates the naming convention of files used in the GROW algorithm, which will be explored in more detail in the section "Files Generated". In this example the initial input file provided by the user, shown in 1), and named *input.in* contains a fault *fault1* comprised of 2 elements, and fault *fault2* comprised of 3 elements. *Fault1* is growing from end 2 and *fault2* is growing from end 1.

2) shows that *fault1* propagates before *fault2* in a propagation of crack growth because GROW propagates faults alphanumerically. 2) shows all of the elements tested for *fault1* in propagation 1. For each fault geometry tested, however, only one of the newly added elements (i.e. 6, 7, 8, 9, 10) exist in the file. The table in 2) lists the filename of the Fric2D input file that describes the fault geometry that includes each respective element and does not include the remaining elements from 6-10. In 2) element 7 is bolded because GROW determined that the fault geometry that includes element 7 optimizes $W_{ext}/\Delta A$. Thicker elements indicate elements that were part of the initial fault geometry, or were identified as elements that optimize work. Thinner elements show other element orientations that were tested for efficiency.

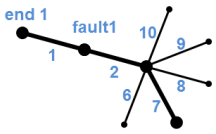
3) shows that after the fault geometry that includes element 7 is identified as the most efficient geometry, GROW searches for the most efficient geometry of elements added to the tip of *fault2*, and finds that element 14 optimizes $W_{ext}/\Delta A$.

4) shows the most efficient geometry found in the first pass of the search algorithm, and the table lists the orientation of the newly added elements. 5-8) show the fault geometries tested in the tuning sequence. Thicker elements indicate elements that exist in the respective tuning fault geometry. Tables in 2-8) lists the orientation of the corresponding element measured clockwise from the tip of the fault.

9) shows that the most efficient geometry identified after the tuning sequence includes element 14 and element 17. If GROW did not execute the tuning sequence the more efficient orientation of element 17 would not have been identified.



2) Propagation 1 of crack growth, growing fault1 end 2

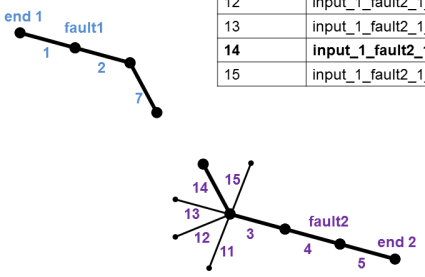


Element #	Filename
6	input_1_fault1_2_90.in
7	**input_1_fault1_2_135.in**
8	input_1_fault1_2_180.in
9	input_1_fault1_2_225.in
10	input_1_fault1_2_270.in

Figure 2 continued

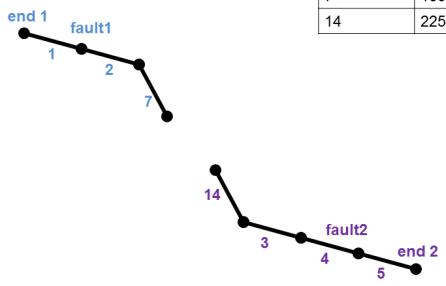
3) Propagation 1 of crack growth, growing fault2 end 1

Element #	Filename
11	input_1_fault2_1_90.in
12	input_1_fault2_1_135.in
13	input_1_fault2_1_180.in
14	**input_1_fault2_1_225.in**
15	input_1_fault2_1_270.in



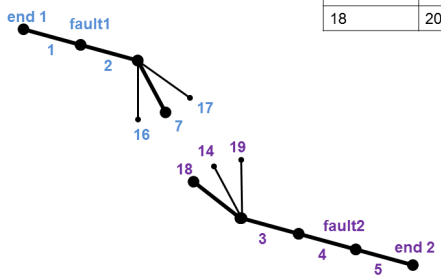
4) Propagation 1 of Crack Growth, before Tuning Sequence
Filename: input_1_fault2_1_225.in

Element #	Orientation
7	135
14	225



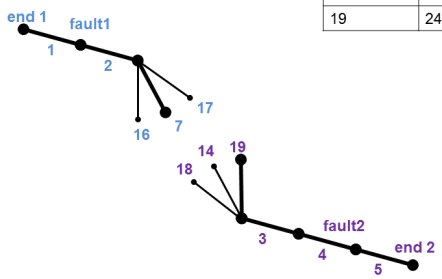
5) Propagation 1 of Crack Growth: Tuning Sequence Geometry
Filename: input_tune_1_fault2_1_202.5.in

Element #	Orientation
7	135
18	202.5



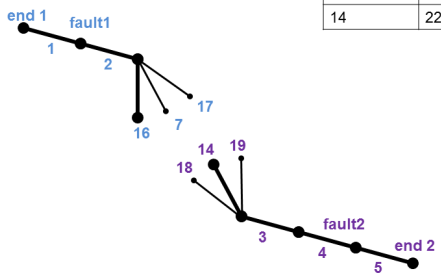
6) Propagation 1 of Crack Growth: Tuning Sequence Geometry
Filename: input_tune_1_fault2_1_247.5.in

Element #	Orientation
7	135
19	247.5



7) Propagation 1 of Crack Growth: Tuning Sequence Geometry
Filename: input_tune_1_fault1_2_202.5.in

Element #	Orientation
16	202.5
14	225



8) Propagation 1 of Crack Growth: Tuning Sequence Geometry
Filename: input_tune_1_fault1_2_247.5.in

Element #	Orientation
17	247.5
14	225

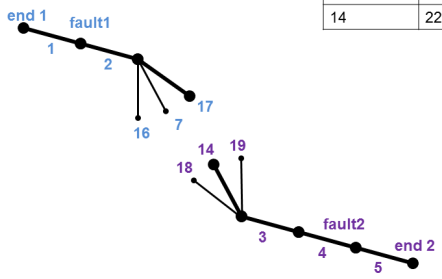
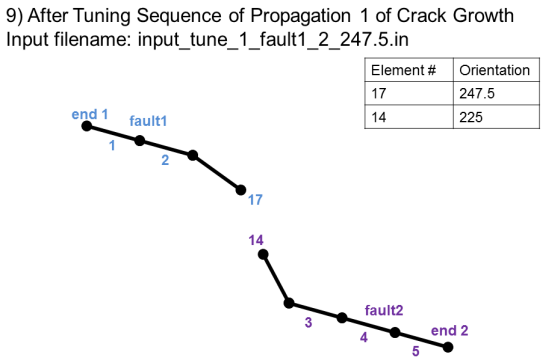


Figure 2 continued



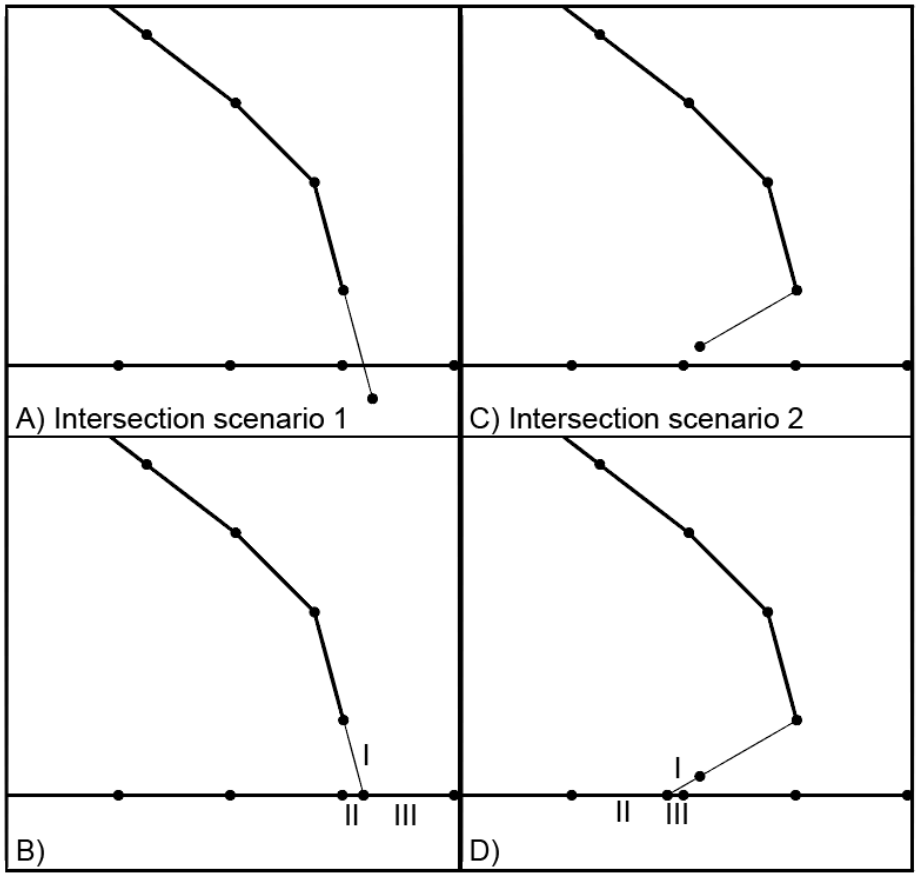
2.4. Intersections

When GROW adds an element to the tip of a fault, this potential element could intersect another element of a fracture or boundary, or fall within the zone neighboring each element where stresses are non-physically high. We consider when a new element lies within this zone or strictly intersects another element as "intersecting" because both geometries will produce non-physical values of W_{ext}. We define this zone as within one element half-length of an element.

GROW handles the intersection of elements differently depending on whether an element of a fault intersects an element of the same fault, or whether the element intersects the element of a different fault or boundary of the model. If GROW creates an input file where the potential element intersects the fault from which it is growing, GROW does not try to correct this intersection, and does not attempt to run Fric2D and calculate W_{ext} for this input file. Instead, GROW will report an error message "Fault intersects itself" to standard output (i.e., the command window or terminal) and continue to test other orientations of the newly added element.

If the element of a fault intersects the element of a different fault or the element of a boundary, then the nodes of the fault or boundary and the propagating fault are slightly adjusted so that the intersection meets at exactly one node. The nodes of the boundary or non-propagating fault are changed so that the maximum possible number of elements maintains the same length as the original element length of the boundary or fault. We strive to keep the element length as consistent as possible because the calculation of W_{ext} is sensitive to element length. After the nodes are adjusted GROW runs Fric2D, calculates W_{ext}, and considers this intersecting geometry when searching for the most efficient geometry. When the most efficient geometry includes a fault that intersects another fault or a boundary, GROW sets a flag in the Fric2D input file that specifies that the intersecting tip is no longer growing. Figure 3 demonstrates how GROW detects and corrects intersections.

Figure 3: Detection of fracture intersection and subsequent modification of element nodes. A) In intersection scenario one, a potential growth element intersects a preexisting fracture. To maintain kinematic compatibility among elements, GROW replaces propagation element tip with smaller element I, and divides an element of lower fracture into two smaller elements (labeled II and III) (B). C) In intersection scenario two, GROW detects fracture intersection because tip of newly added element (thin line) falls within one element half-length of lower fracture. As a result, GROW adds a smaller element (labeled I) to tip of upper, propagating fracture, and divides element of lower fracture into two elements labeled II and III (D).



2.5. Termination

A GROW run will stop when the tips of all the faults are not propagating. The tip of a fault will stop propagating if the element added to the tip of this fault that optimized work in the last propagation of crack growth intersects another fault or boundary. Additionally, GROW signals that a fault stops propagating (by setting a flag in the corresponding input file) if the most efficient element added in the last propagation of crack growth does not fail in shear or in tension. An element fails in tension when the normal stress across its surface, σ_n, which is positive when tensile, exceeds or equals its tensile strength, T:

$$\sigma_n \geq T.$$

Note that T is the tensile strength of the intact material at the fault tip, and not the tensile strength of the fault.

A potential growth element fails in shear following the Coulomb failure criterion, when the magnitude of the shear stress across its surface, τ, exceeds or equals the sum of the inherent shear strength, S_o, and the product of the internal coefficient of friction, μ_0, and normal stress, σ_n across the potential element:

$$|\tau| \geq S_o + \mu_0 \sigma_n.$$

By determining if potential elements fail in tension and/or shear, GROW honors the principles of linear elastic fracture mechanics. And while each potential growth element must meet one of these criteria to be considered a possible growth direction, the propagation direction is determined by the fault geometry that minimizes work.

At the end of a Fric2D output file Fric2D lists the elements that fail in shear or by opening-mode failure for each fault. GROW reads the Fric2D output file to determine if an element added to the propagating fault fails, and if the element does not fail then GROW does not consider this geometry when it searches for the geometry that optimizes work. If none of the elements added to the tip of a fault fail in shear or opening mode, GROW sets a flag in the Fric2D input file that signals that this fault tip is no longer propagating. So if one tip stops propagating, GROW will continue to search for the geometry that optimizes work by adding elements to the tips of any other faults in the system. This functionality is implemented by parsing the most efficient input file after each propagation of crack growth to determine if and what faults are still propagating.

2.6. Long GROW Runs

Under certain loading conditions the tips of faults will continue to propagate for many increments of crack growth. To free memory during such long runs, and effectively reduce execution time, GROW automatically restarts itself after five propagations of crack growth. When GROW restarts the Fric2D input file that specifies the initial geometry is now set as the most efficient geometry identified in the last propagation of crack growth. Additionally, when GROW restarts, GROW passes the external work required by the last efficient

geometry so that GROW will not recalculate the initial W_{ext} of a geometry for which GROW has previously calculated work.

2.7. Growing from a point: Investigating fracture initiation

The previous sections describe the general usage of GROW, in which the user specifies one or more faults in the initial input file that are propagating. The user can also specify a point from which a fault will propagate. In this case, the initial Fric2D input file lists the coordinates of the point and whether or not the fault that grows from this point is propagating from both tips of the fault, or just one tip (see "Input" section for more detail).

When the user invokes this functionality, GROW first searches for the orientation of the element that optimizes work at the point specified by the user. More specifically, if the user sets the resolution angle to θ_{res}, the coordinates of the point to *x*, *y*, and the length of an element of the fault that grows from the point to *h* meters, in the first propagation of crack growth GROW will find the external work of the input file with an element with the end points *x*, *y* and *x*-*h*, *y*. GROW finds the external work for the orientation of an element added from zero to 359° from this line segment in increments of θ_{res} (0, θ_{res}, $2\theta_{res}$, $3\theta_{res}$,…). Like the general GROW algorithm, after GROW finds the most efficient orientation of the element extending radially from the coordinates *x*,*y* (θ_e), GROW enters a tuning sequence where W_{ext} is calculated for the fault geometry with an element oriented at $\theta_e - \theta_r/2$ and $\theta_e + \theta_r/2$**.** Figure 4 shows the orientations of all elements tested in the first propagation of crack growth when the user decides to propagate a fault from a point.

In this usage of GROW, when the user specifies that both tips of the fault are growing GROW subsequently adds elements to both tips of the element that optimized work in the first propagation of crack growth. When the user sets only one tip of the fault growing GROW adds elements to the tip of the first element added that does not have the coordinates originally set by the user. For example, tip 1 of all of the elements tested in the first propagation of crack growth will have the coordinates *x*, *y*, and tip 2 of the most efficient element will have coordinates along the circle with a center of *x*, *y* and a radius of one element length. When one tip of a fault propagates from a point, GROW subsequently adds elements to tip 2 of the previous element. Figure 5 shows how elements are subsequently added to one tip of a fault when the user chooses to propagate a fault from a point, and the propagation sequence when the user specifies that both tips of the fault propagate.

When GROW adds the first element it inserts a header line and line describing other fault properties to the Fric2D input file so that later propagations of crack growth consider this new fault like any other growing fault. After the first element is added the algorithm proceeds as the general algorithm described in previous sections.

Figure 4: Illustration shows the orientations of all elements tested in the first propagation of crack growth when the user grows from a point. In this example the user specified that the flaw begin propagating at the coordinates *x*, *y*; that the length of an element in this flaw is *h*; and that the resolution angle is θ_r. The angle between elements 1-8 radiating from *x*, *y* is θ_r. End 1 of the crack added in the first propagation of crack growth is always at the coordinates specified by the user *x*, *y*, and end 2 of the crack is along the circle with a center of *x*, *y* and a radius of *h*. The bolded element 8 indicates that this fault geometry optimized W_{ext} in the first propagation of crack growth. Element 9 and 10 were tested in the tuning sequence of the first propagation of crack growth. The numbering of elements indicates the order in which the elements were added and W_{ext} calculated.

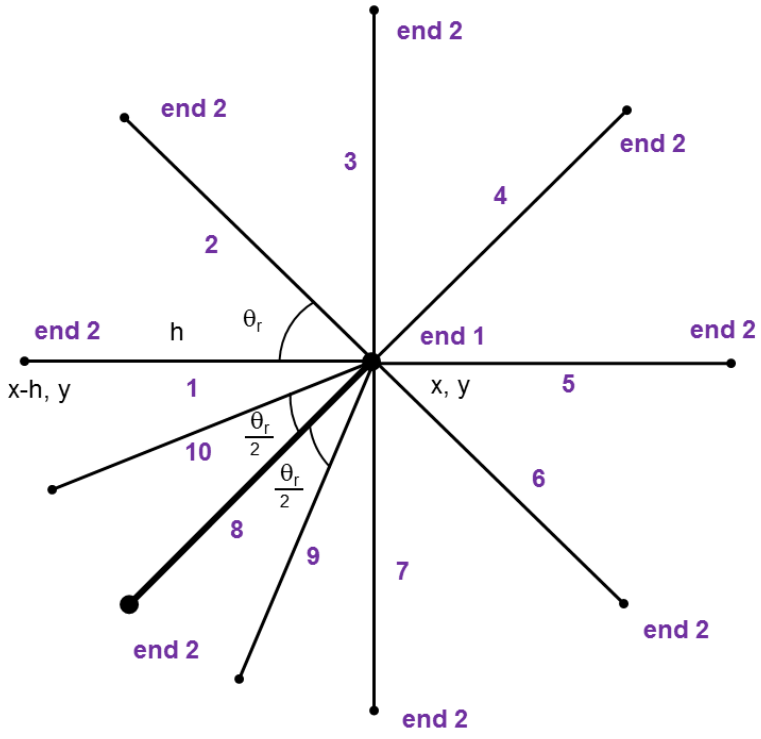
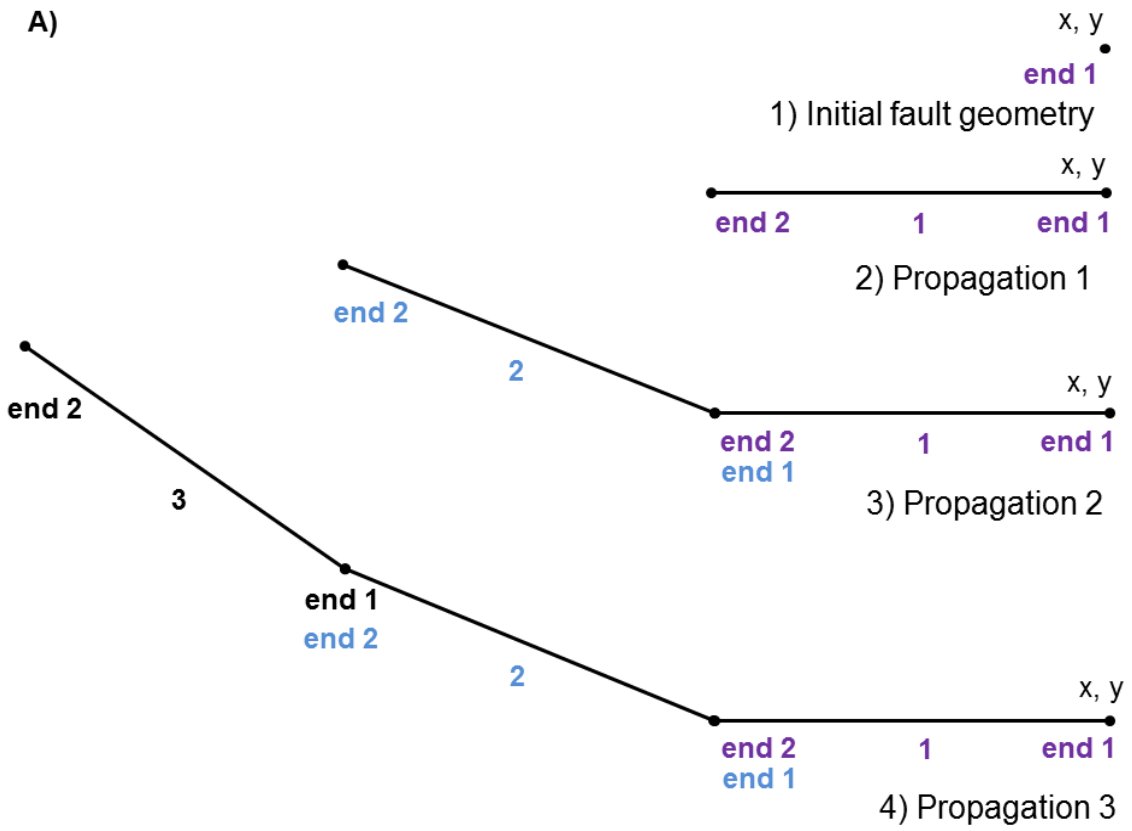
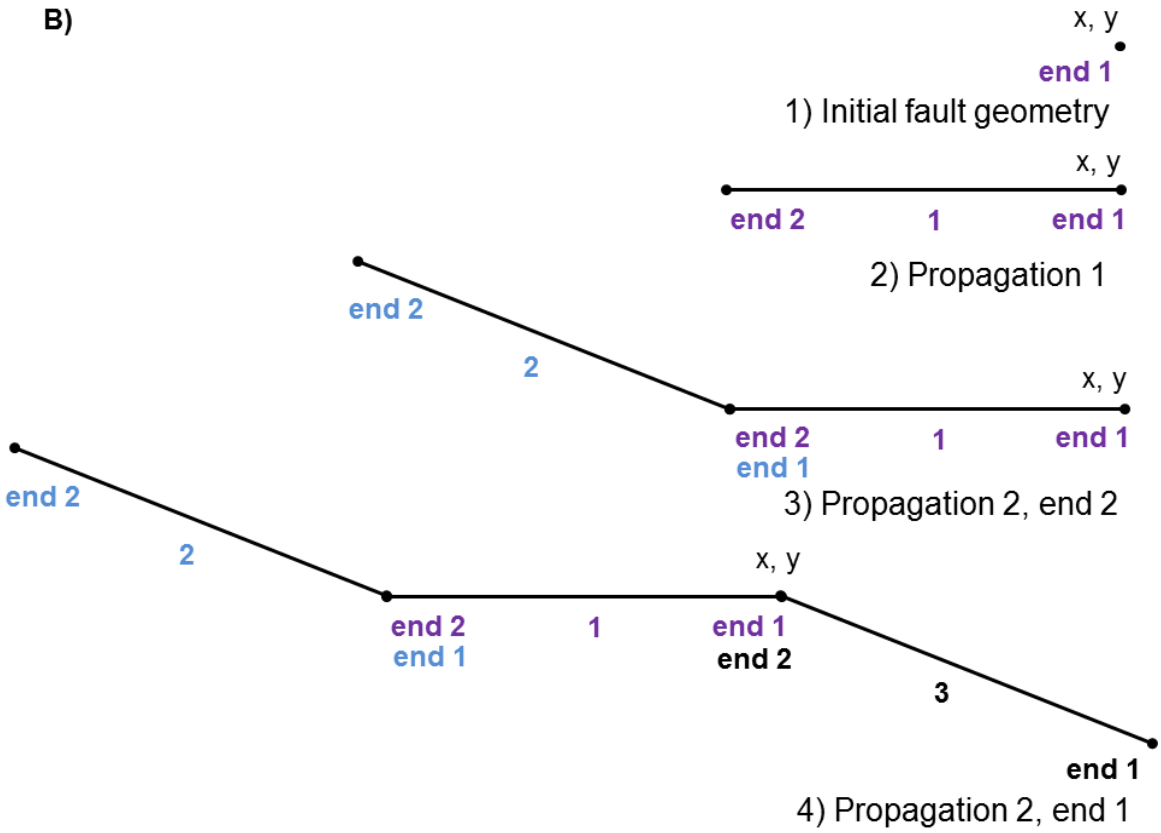


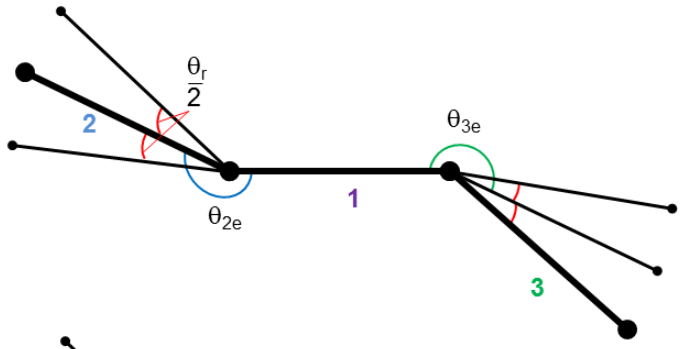
Figure 5: Sketch shows how elements are subsequently added to one tip of a fault when the user decides to grow from a point (A), and the propagation sequence when the user specifies that both tips of the fault propagate (B). With this implementation the user initially specifies the coordinates of the flaw (*x, y*) and the length of the elements that will comprise this fault. A) shows that when the user indicates that only one tip of the fault is propagating a new element will always be added to end 2 of subsequent elements.

B) shows that when the user indicates that both tips of the fault is growing, elements are added to end 1 and end 2 of the crack in each propagation of crack growth. B) 5-8) describe all the fault geometries tested in the tuning sequence of the second propagation of crack growth in this example. Here, the most efficient orientation identified in the first pass of the search algorithm of element 2 and 3 are θ_{2e} and θ_{3e}, respectively, and the resolution angle is θ_r. In 5-8) only bolded elements represent the fault geometry currently being tested, and the table lists the orientation of the elements added in the tuning sequence.



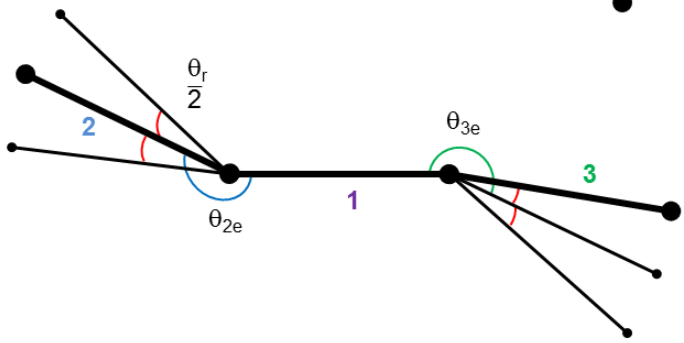
B)





5) Propagation 2
Tuning Sequence

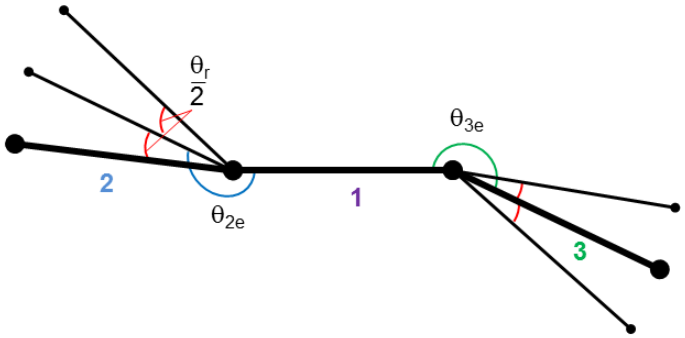
Element	Orientation
2	θ_{2e}
3	$\theta_{3e} + \theta_r/2$



6) Propagation 2
Tuning Sequence

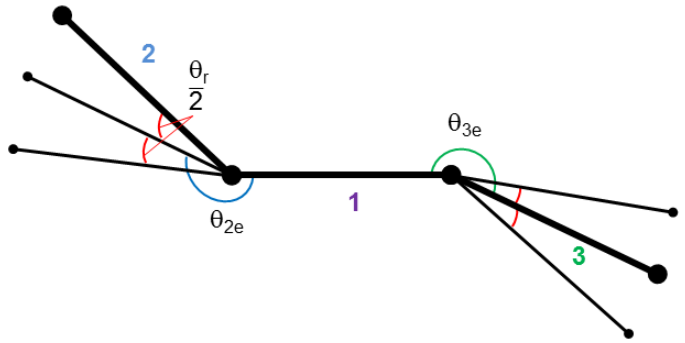
Element	Orientation
2	θ_{2e}
3	$\theta_{3e} - \theta_r/2$

Figure 5B) continued



7) Propagation 2
Tuning Sequence

Element	Orientation
2	$\theta_{2e} - \theta_r/2$
3	θ_{3e}



8) Propagation 2
Tuning Sequence

Element	Orientation
2	$\theta_{2e} + \theta_r/2$
3	θ_{3e}

3. Input

Because GROW repeatedly runs Fric2D, the user must understand how to format Fric2D input files to precisely model the specific tectonic environment in which they are interested. The user must also know the order of input parameters required to run GROW. First we discuss the order of input parameters required to run GROW, and then we describe how to format a Fric2D input file.

3.1. GROW Input

The general GROW usage requires that the user specify an initial fault geometry and loading conditions, which is described in a Fric2D input file. This input file must be named with the postfix *.in*, for example *input.in*. This input file must exist in the same directory as the GROW executable. The user also sets the resolution angle, θ_{res}, the minimum angle, θ_{min}, and the maximum angle, θ_{max} that defines the range of orientations to search. On the command line, GROW is called with the command: perl GROW.pl *input.in* θ_{res} $\theta_{min}\theta_{max}$. Each parameter in the command line is separated by a space.

3.2. Fric2D Input

The specifics of how to format Fric2D input files are fully described at http://www.geo.umass.edu/faculty/cooke/Fric2D/chapter4.html. Here we describe the most important features of a Fric2D input file required to model fault evolution with GROW.

3.2.1. Intact Rock Properties

In the Fric2D input file the user can set the properties of the intact rock in the model, including Possion's ratio, listed as *pr* in the input file, Young's modulus, *e* (MPa), mode-I fracture toughness, *k1c* (MPa $m^{1/2}$), tensile strength, *tensileStrength* (MPa) and density, *density* (kg/m^3). The variables that specify the magnitude of these rock properties are listed under *Rock Properties* or *Gravitational Stresses* in the input file. Figure 6 highlights where these properties are listed in an input file. To suppress the fracture propagation mechanism of Fric2D, and thus ensure that only the work minimization approach implemented in GROW models fracture propagation, the user must set the mode-I fracture toughness to high values (~1e10 MPa $m^{1/2}$). The *gravity* flag determines whether the model should include the forces produced by gravity. If *gravity* is turned on (with a value of 1) then the user must also create a topography file that describes the geometry of the topography, which should list the nodes of the elements of the top boundaries of the model. When the user calls GROW they must include the name of the topography file (that must end in .topo) after the initial input values entered, i.e. after name of initial Fric2D input file, resolution angle value, starting angle value and ending angle value. The *kratio* value controls the ratio of horizontal to vertical normal stresses. This factor only significantly influences results if the force due to topography/gravity is larger than the applied tractions or displacements on the "tectonic" model boundaries.

Figure 6: Example Fric2D input file with bulk rock properties highlighted.

```
*------------------------
*Problem    Variables
*------------------------

*Titles
title1  =   "Simulation of the right-lateral releasing bend"
title2  =   "-4 km horizontal and 4 km vertical spacing in granite"

*Rock   Properties
pr  =   0.17    *(possion's ratio)
e   =   5.00E+01    *(young's   modulus MPa)
k1c =   1.00E+10    *(mode  I   fracture    toughness,  MPa*m^1/2)
tensileStrength =   7   *(tensile   strength    of  sample, MPa)

*Symmetry
ksym    =   1

*Gravitational  stresses
gravity =   0   *(apply gravity:    0=no,   1=yes)
kratio  =   1   *(ratio of  horizontal:vertical gravitation stress)
density =   2550    *(density   of  rock    kg/m^3)
```

3.2.2. Boundaries

In the BEM code Fric2D each boundary is discretized into elements, and the coordinates of the end points of each boundary is listed by the user in the input file. In a Fric2D input file, under the title *Boundary Lines* the user must list each linear segment of the boundary of the model with a separate line (i.e., a line of text followed by a carriage return). Figure 7 shows how the coordinates of the boundary lines must connect to complete a closed shape. The segments of the boundaries must define a complete shape so that one node of the first segment exactly intersects a node of the last segment listed in the input file. When the user defines a solid surrounded by empty space the user must list in the segments in clockwise order. For example, the second boundary line listed in the input file is in the clockwise direction from the first boundary line listed. In this case one end point of the first boundary line (titled *xend* and *yend* for the x- and y-coordinates in the input file header) intersects the end point of the second boundary line (titled *xbeg* and *ybeg*). If the user wants to define a void infinitely surrounded by solid material the boundaries must be listed in counter-clockwise order. The user must set the number of boundary lines with the variable *nblines*, which is listed under the title *Number of Observation Lines and Boundary Lines* in the Fric2D input file before the *Boundary Lines* title.

For each boundary line, the user lists, in order, the number of elements in the boundary (titled *num*), the x and y-coordinate of the one end point of the boundary (xbeg, ybeg), the x and y-coordinate of the other end point of the boundary (*xend*, *yend*), a flag that specifies the type of loading condition on this boundary (*knode*), and four numbers that specify the magnitude and type of loading on the boundary (*static bvs, static bvn, monotonic bvs, monotonic bvn*). Static loading is applied constantly, while monotonic loading is applied in a certain number of loading steps specified by the user (*nsteps*). Each of the aforementioned values must be separated by white space (any number of spaces or tabs). Figure 8 illustrates that Fric2D creates a solid surrounded by void space when the user lists boundary segments in clockwise order, and creates a void surrounded by solid material when the lists boundary lines in counterclockwise order.

The number of elements of a boundary must be a positive integer, and the coordinates of the end points can be any real number (i.e., positive or negative and integer or decimal). The length of each segment of a boundary (as determined by the distance between the coordinates *xbeg*, *ybeg* and *xend*, *yend*) and number of elements (*num*) of that boundary determines the length of elements of that boundary. Models with consistent (as close to equal as possible) element lengths produce the most robust numerical results.

The flag that specifies the loading conditions (*knode*) must be 1, 2, 3, or 4. This flag specifies whether tractions (MPa) or displacements (m) are applied to the elements of a boundary (i.e., the boundary conditions of the model). Table 3 lists the specific boundary condition for each *knode* value. If *knode* is 1, then the user must specify shear traction and normal traction to the boundary; if the *knode* is 2, the user specifies shear displacement and normal displacement; if the *knode* is 3, the user can specify shear displacement and normal traction; and if the *knode* is 4, the user can specify shear traction and normal displacement. However, because a model loaded with displacement boundary conditions will optimize work with the lowest value of $W_{ext}/\Delta A$, and a model loaded with tractions is most efficient when $W_{ext}/\Delta A$ is greatest, the user should not mix the types of loading conditions applied to the boundaries, and only use non-zero values of tractions or

displacements. The user may specify a *knode* that uses both tractions and displacements if the magnitude of tractions is zero when the displacements are not zero, or the displacements are non-zero and the tractions are set as zero. When the displacement of a boundary is set to zero, then that boundary must remain fixed (in the normal or shear sense) as that boundary experiences reaction tractions. When the tractions of a boundary is set to zero, that boundary is free to displace, and can be thought of being on rollers.

The magnitude of *static bvs* and *static bvn* specifies the static loading on the boundary, where *static bvs* represents shear traction (MPa) or shear displacement (m), and *static bvn* represents normal traction (MPa) or normal displacement (m). Similarly, the magnitude of *monotonic bvs* and *monotonic bvn* specifies the loading that is applied in a certain number of steps, *nsteps* in the input file, on the boundary, where *monotonic bvs* represents shear traction (MPa) or shear displacement (m), and *monotonic bvn* represents normal traction (MPa) or normal displacement (m). The static loading remains constant while the monotonic loads are applied. The number of steps in which the monotonic loading is applied is specified under *Option Commands* in the input file with the variable *nsteps*.

The sign of *static bvs, static bvn, monotonic bvs* and *monotonic bvn* (i.e., whether or not the values are positive or negative) depends on the local coordinate system of the elements of the boundary upon which the loads are applied. For example, *bvs* is positive if 1) Fric2D traverses the elements of the boundary clockwise (i.e., the user lists the segments of the boundary in clockwise order) and 2) the direction of the shear displacement or traction is clockwise, or to the right relative to the boundary. If 1) the user lists the segments of the boundary clockwise, and 2) the direction of the shear displacement or traction is counterclockwise, *bvs* is negative. Figure 9 demonstrates the sign convention of the boundary conditions when the boundary segments are listed clockwise. If the user lists the segments of the boundary clockwise and *bvn* is positive, the direction of the normal displacement or traction is towards the outside of the model (i.e., the model boundary will be moved relatively away from the center of the model). If the user lists the segments of the boundary clockwise and *bvn* is negative, the direction of the normal displacement or traction is towards the inside of the model (i.e., the model boundary will be moved relatively toward the center of the model).

Table 3
Boundary condition specified by *knode* value.

knode	Shear (bvs)	Normal (bvn)
1	traction	traction
2	displacement	displacement
3	displacement	traction
4	traction	displacement

Figure 7: Sketch illustrates that the ends of the boundary segments must connect so that the model boundary is a closed polygon. The beginning coordinates of the first boundary line listed must equal the coordinates of the end of the last boundary line listed, and the end coordinates of each boundary line (except the first boundary listed) must equal the coordinates of the beginning of the next boundary line listed. Because these boundary segments are listed in clockwise order, Fric2D interprets this shape as a solid surrounded by void space. The colored numbers show the coordinates of the end points of the boundary lines. The black numbers indicate the number of elements of the corresponding boundary line. The value and sign convention of the loading conditions (i.e., *knode*, *static bvs*, *static bvn*, *monotonic bvs*, *monotonic bvn*) are discussed more in the text and Figure 9.

```
*------------------
*Boundary   Lines
*------------------
*(bvs    =    shear;  bvn =    normal)                STATIC  MONOTONIC
*num     xbeg     ybeg     xend     yend     knode    bvs bvn bvs bvn
*---     ----     ----     ----     ----     -----    --- --- --- ---
29       0        0        0        14500    2        0   0   0       0
21       0        14500    0        25000    2        0   0   0   -4000
100      0        25000    50000    25000    2        0   0   4000    0
29       50000    25000    50000    10500    2        0   0   0   4000
21       50000    10500    50000    0        2        0   0   0       0
100      50000    0        0        0        2        0   0   0       0
```

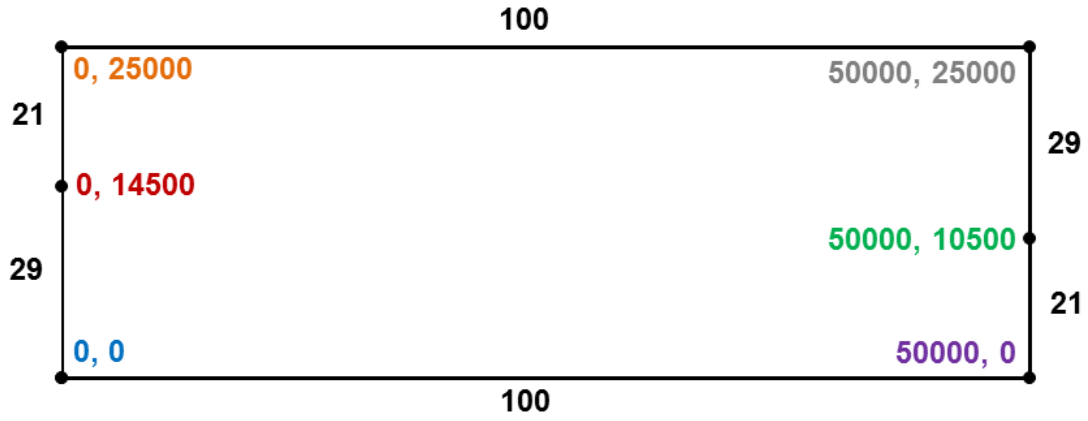


Figure 8: Sketch illustrates that Fric2D creates a solid surrounded by void space when the user lists boundary segments in clockwise order (A), and creates a void surrounded by solid material when the lists boundary lines in counterclockwise order (B). In A) Fric2D traverses the elements of the boundary clockwise; in B) Fric2D traverses the element counterclockwise. The red arrows and numbers indicate the direction that the elements are traversed, and the order in which the boundaries were listed in the input file. The gray region indicates volume that is solid, white region indicates void space.

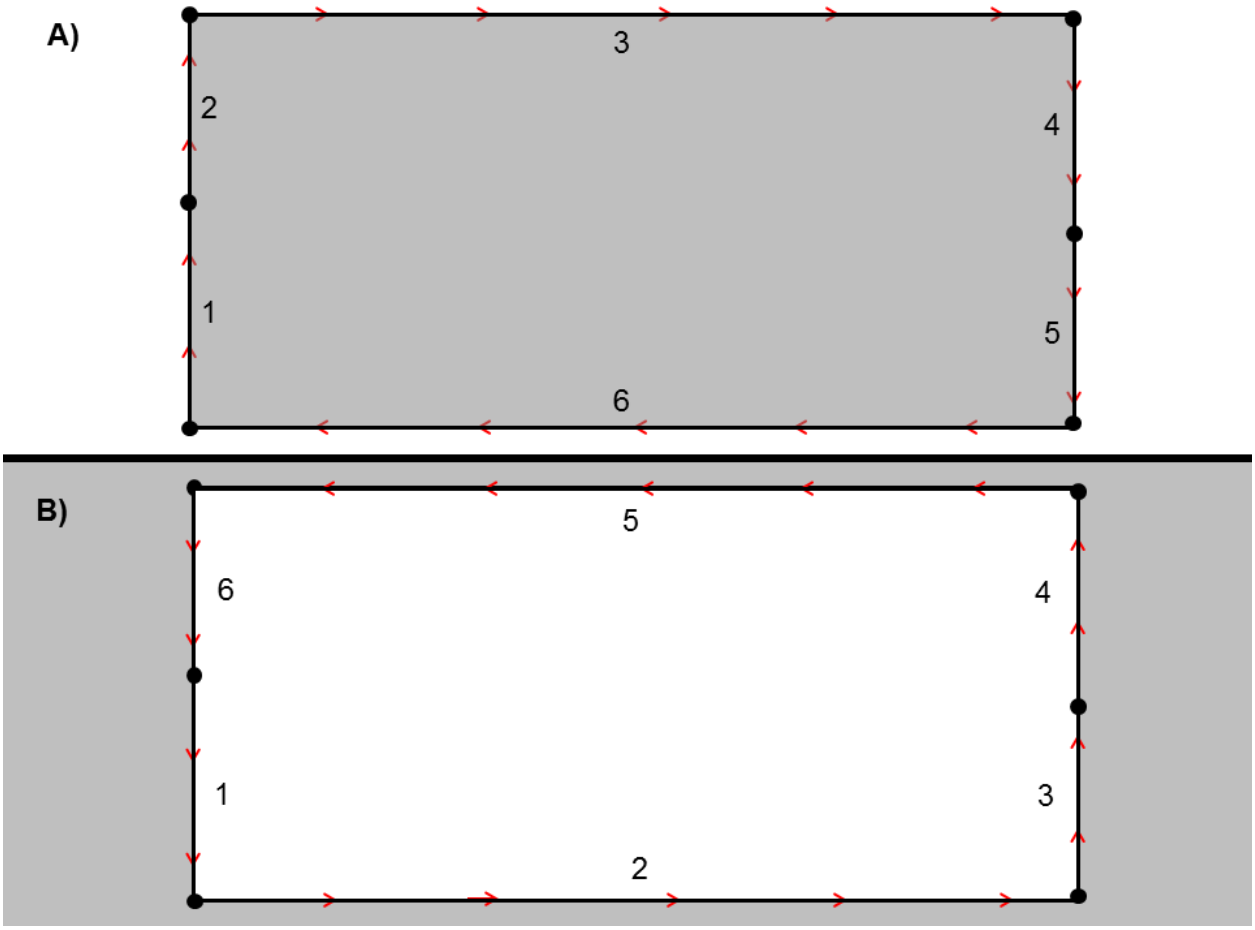
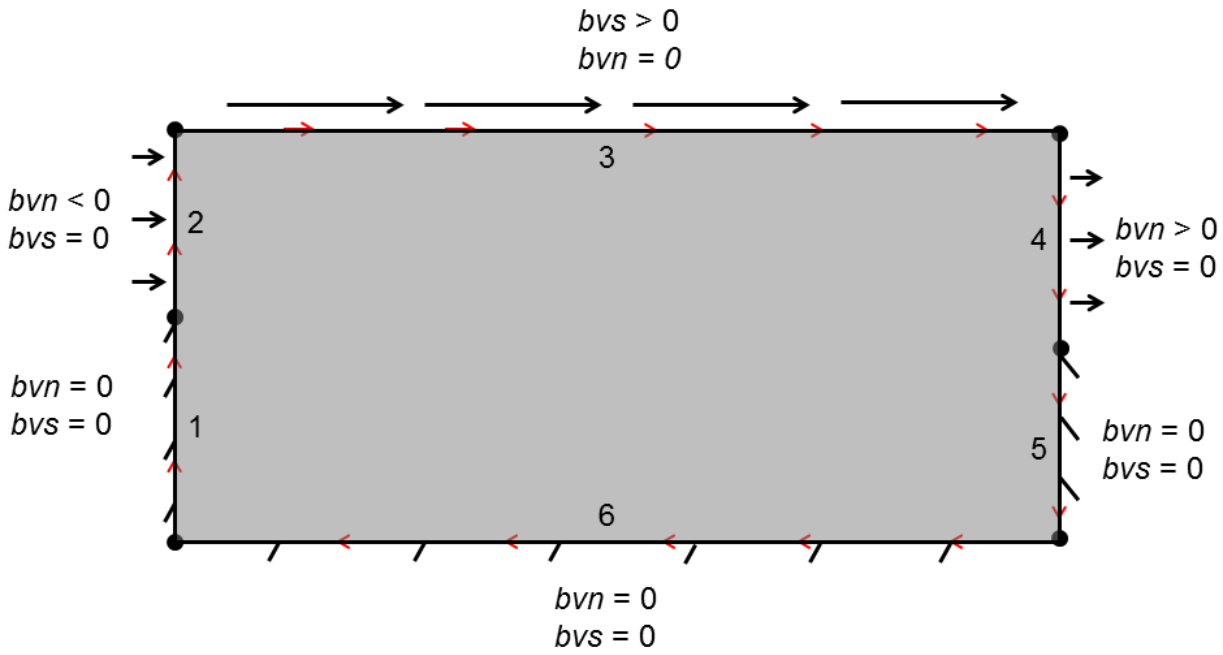


Figure 9: Sketch illustrates sign convention of boundary conditions used in Fric2D model. The red arrows indicate the direction that Fric2D traverses the elements of the boundaries. These arrows run clockwise because the user listed the segments of the boundary in clockwise order, in the order indicated by the numbers on the inside of the model. The end points of each segment are designated by black dots. *Bvn* and *bvs* are zero on boundary segments 1, 5 and 6 because these boundaries are fixed and experience no shear or normal displacement. If the loading conditions were tractions and *bvn* = *bvs* = 0 then the boundaries are said to be on "rollers", because under those conditions they must experience no tractions, and thus no resistance to displacement. On boundary 2, *bvn* < 0 because the boundary is displaced toward the inside of the model and on boundary 4, *bvn* > 0 because the boundary is displaced toward the outside of the model. These conventions hold because we defined the local element coordinate system in the clockwise direction. For boundary 3, *bvs* > 0 because the model must be displaced in the same direction as the elements of the boundary are traversed (clockwise, or to the right on this boundary). Similarly, if we wanted to apply a left lateral displacement to boundary 6 then *bvs* > 0, because the displacement is also in the same direction that Fric2D traverses the elements of the boundary (clockwise, and to the left on this boundary). These boundary conditions correspond to the conditions listed in the input file shown in Figure 7.



3.2.3. Fault Properties

To define a fault in a Fric2D input file the user must enter at least two input lines after the *Fault Conditions* header in the input file. The first input line must begin with the text "fault", after this flag the user lists the name of the fault, which can be any sequence of alphanumeric characters and does not include any white space, a flag that specifies if cracks can propagate from the middle of the fault, labelled *grow_tails?*, a flag that determines if the fault is growing from one tip of a fault, *from_end1?*, and a flag that determines if the fault is growing from the other tip of the fault, *from_end2?*. The value of these flags is set as "yes" or "no" in the input file. When using GROW, the user should set the *grow_tails?* flag to "no." After this line the fault is described by a line that contains, in order, the number of elements in the fault, titled as *num* in the input file, the x-coordinate of one tip of the fault, *xbeg*, the y-coordinate of that tip, *ybeg*, the x-coordinate of the other tip of the fault, *xend*, the y-coordinate of this tip, *yend*, shear stiffness, *stiffS*, normal stiffness, *stiffN*, tensile strength, *ten-str*, initial cohesion, *init-coh*, the sliding cohesion, *slid-coh*, coefficient of static friction, *stat-fric,* coefficient of dynamic friction, *dy-fric,* and the critical slip-weakening distance, *crit-slip*. If the user decides to grow from the *xbeg*, *ybeg* coordinates they must set the *from_end1?* flag to "yes". If the user wants to grow from the *xend*, *yend* fault tip they must set *from_end2?* to "yes". The user can set both of these flags to "yes" and grow from both tips as well.

The number of elements must be a positive integer and the coordinates of the end points specify locations in the model where the unit of distance is meters. The number of elements and length of each fault will determine the element length of that fault. The most robust numerical models have consistent element sizes. The initial coordinates of the faults must either be 2 element lengths from any model boundaries, or the end points of the fault must exactly intersect an element node of the boundary. When the tip of the fault exactly intersects a boundary, the user should not specify that that fault tip is growing. The shear and normal stiffness, tensile strength, and initial and sliding cohesion are defined in MPa. The unit of the critical slip-weakening distance is meters.

The shear stiffness controls the resistance to shear displacement on an element. This influence of shear stiffness can also be modeled by changing the values of friction of the fault. Normal stiffness controls the degree to which elements may interpenetrate: if normal stiffness is high then elements may not interpenetrate, if it is low then the positive and negative sides of the element are allowed to interpenetrate, and the modeled system is effectively more compliant (reduced Young's modulus). Fric2D enables slip-weakening to be modeled with the static and dynamic coefficient of friction and the prescribed slip-weakening distance. When shear slip on an element exceeds this slip-weakening distance the coefficient of friction on the fracture evolves linearly from its static to dynamic value. Thus the dynamic value should be less than the static value if the modeled material is slip-weakening. Figure 10 shows the evolution of 3 input file that describes one fault, and highlights the specific location and formatting of each parameter mentioned above. This example also shows the *Crack* functionality of GROW; see the following section "Crack Properties" to learn more about this implementation. Table 4 synthesizes the various fault properties listed in this input file and lists the title of the property used in the input file.

The user can list multiple faults by separating the lines describing each individual fault with carriage returns (enter keys). The user can define a fault with segments with

differing properties by listing multiple lines after the original input line that begins with the "fault" header. Similar to the requirements of the coordinates of the boundary lines, the end points of the segments of the fault must connect so that the structure is continuous.

Figure 10: Example of fault location and properties listed in 3 Fric2D input files. The initial input file is shown at the top of the figure, and input files produced in subsequent iterations of growth are shown in descending order. Input files show evolution of fracture properties enabled by *Crack properties. Red boxes highlight elements with intact rock properties (*Crack) and blue boxes show elements with fracture properties. Table 4 and Table 5 lists the other properties listed in this input file.

```
*----------------------
*Fault  Conditions
*----------------------
*fault  name    grow_tails? from_end1?  from_end2?
*num    xbeg    ybeg    xend    yend    stiffS      stiffN      ten-str init-coh slid-coh stat-fric dy-fric crit-slip
*----   ----    ----    ----    ----    ------      ------      ---     ---     -----     -------   ------  ---------
fault   left    no  no  yes
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
68  10000   13500   27000   13500   1.00E+10    1.00E+10    0   0   0   0.3     0.3     0.00E+00

fault   right   no  yes no
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
68  23000   11500   40000   11500   1.00E+10    1.00E+10    0   0   0   0.3     0.3     0.00E+00
```

```
*----------------------
*Fault  Conditions
*----------------------
*fault  name    grow_tails? from_end1?  from_end2?
*num    xbeg    ybeg    xend    yend    stiffS      stiffN      ten-str init-coh slid-coh stat-fric dy-fric crit-slip
*----   ----    ----    ----    ----    ------      ------      --- --- -----     -------   ------  ---------
fault   left    no  no  yes
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
68  10000           13500           27000           13500           1.00E+10    1.00E+10    0   0   0   0.3     0.3   0
1   27000           13500           27249.04867452  13521.78893569  1.00E+10    1.00E+10    0   0   0   0.3     0.3   0
1   27249.04867452  13521.78893569  27490.53013109  13586.49369697  1.00E+10    1.00E+10    7   14  0   0.5     0.5   0

fault   right   no  yes no
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
1   22500.00000000  11500.00000000  22750.00000000  11500.00000000  1.00E+10    1.00E+10    7   14  0   0.5     0.5   0
1   22750.00000000  11500.00000000  23000           11500           1.00E+10    1.00E+10    0   0   0   0.3     0.3   0
68  23000           11500           40000           11500           1.00E+10    1.00E+10    0   0   0   0.3     0.3   0
```

```
*----------------------
*Fault  Conditions
*----------------------
*fault  name    grow_tails? from_end1?  from_end2?
*num    xbeg    ybeg    xend    yend    stiffS      stiffN      ten-str init-coh slid-coh stat-fric dy-fric crit-slip
*----   ----    ----    ----    ----    ------      ------      --- --- -----     -------   ------  ---------
fault   left    no  no  yes
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
68  10000           13500           27000           13500           1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
1   27000           13500           27241.48145657  13564.70476128  1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
1   27241.48145657  13564.70476128  27476.40461176  13650.20979712  1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
1   27476.40461176  13650.20979712  27681.19262282  13793.60390621  1.00E+10    1.00E+10    7   14  0   0.5   0.5   0

fault   right   no  yes no
*tag    KS          KN      ten-str shear-str   coh int-fric dy-fric crit-slip
*Crack  1.00E+10    1.00E+10    7   14          0   0.5         0.5     0
1   22266.97949577  11370.91709279  22501.90265096  11456.42212862  1.00E+10    1.00E+10    7   14  0   0.5   0.5   0
1   22501.90265096  11456.42212862  22750.95132548  11478.21106431  1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
1   22750.95132548  11478.21106431  23000           11500           1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
68  23000   1       1500            40000           11500           1.00E+10    1.00E+10    0   0   0   0.3   0.3   0
```

Table 4: Table lists flags and coordinates designated in a Fric2D input file. Table 5 shows additional fault and intact rock properties specified in the example input file shown in Figure 10.

Title	Property
fault	Tag used in input file for parsing
grow_tails?	Flag to indicate if cracks can propagate from interior nodes of elements of a fault. Set as "no" if using GROW.
from_end1?	If fault propagating from end 1 (i.e., the beginning of the fault, xbeg, ybeg)
from_end2?	If fault propagating from end 2 (i.e., the end of the fault, xend, yend)
num	Number of elements in a fault segment
xbeg	x-coordinate of end 1 (beginning) of initial fault (m)
ybeg	y-coordinate of end 1 (beginning) of initial fault (m)
xend	x-coordinate of end 2 (end) of initial fault (m)
yend	y-coordinate of end 2 (end) of initial fault (m)
*Crack	Flag used in input file for parsing to indicate the properties of the newly added elements

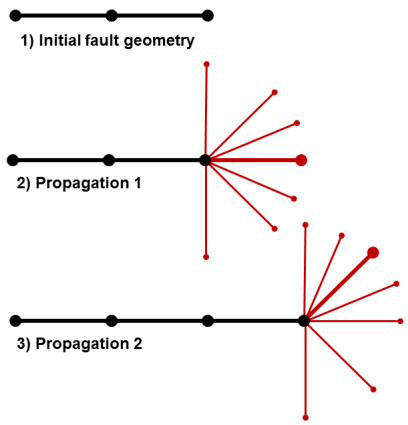
Table 5: Table lists the value of the property listed in the input file shown in Figure 10, and indicates whether the property is a property of the fracture or the potential element/fracture tip, which should reflect properties of intact rock.

Property		Title	Value
Intact rock	Poisson's ratio	pr	0.17
*Crack	Young's modulus	E	50 GPa
	shear stiffness	KS	1e10 MPa
	normal stiffness	KN	1e10 MPa
	tensile strength	ten-str	7 MPa
	inherent shear strength	shear-str	14 MPa
	cohesion	coh	0
	internal coefficient of friction	int-fric	0.5
	dynamic coefficient of friction	dy-fric	0.5
	critical slip-weakening distance	crit_slip	0
Fault	shear stiffness	stiffS	1e10 MPa
fault	normal stiffness	stiffN	1e10 MPa
	tensile strength	ten-str	0 MPa
	initial cohesion	init-coh	0 MPa
	sliding cohesion	slid-coh	0 MPa
	static friction	stat-fric	0.3
	dynamic friction	dy-fric	0.3
	slip-weakening distance	crit-slip	0 m

3.2.4. Crack Properties

If the user wishes to model fault growth where the tip of a growing fault has different characteristics than the remainder of the fault, the user can set the *Crack* properties of a specific fault. To set these properties, the user must insert an additional input line after the input line listing the name of the fault (that begins with "fault") and before the input line listing the fault characteristics, which begins with the number of elements in the fault. This line must begin with the header "*Crack". After this header the user must list the same number of properties they include in the input line that lists the fault properties of the original fault. Figure 10 shows the exact location and formatting required for using *Crack* properties. If the user invokes this functionality each potential element added to the tip of the growing fault will initially have the properties listed in the "*Crack" line. After GROW determines what orientation optimizes work and the new element is added to the model, in the next propagation of crack growth the element with *Crack* properties will then attain the properties of the remainder of the fault. Thus if one tip of one fault is growing, only one element in the model will have *Crack* properties during one propagation of crack growth. Figure 11 demonstrates how the properties along each element of a fault evolve when the user invokes the *Crack* functionality. Table 5 lists of the properties and values of each fault property in the example input file illustrated in Figure 11. This functionality allows the user to simulate the propagation of faults through intact rock because the user can set the *Crack* properties to intact rock values (i.e., inherent shear strength, coefficient of internal friction), and can set the fault properties to values representative of crustal faults (i.e., cohesion, coefficient of static/dynamic friction).

Figure 11: Illustration demonstrates how the user can set different fault properties for potential elements. Black elements are elements of the fault, and red elements are potential elements. The values of the properties of the fault and potential elements are listed in Table 5. 1) shows the initial fault geometry, 2) shows that the elements tested in the first propagation of crack growth in both the first pass and tuning sequence will contain the fault properties specified for newly added elements, and 3) shows that the element that optimized $\Delta W_{ext}/\Delta A$ in the first propagation of crack growth will attain properties of the through-going fault in the next propagation of crack growth. In 2) and 3) the element that optimized work is bolded and colored red.



3.2.6. Flaw Properties

When the user chooses to propagate from a point, they specify a location of a point from which a fault will propagate in the initial Fric2D input file. To define the location and additional properties of this point the user must include a line in the input file after the "*Fault Conditions" header that is separated by the other fault input lines by at least one carriage return. This input line begins with "*Flaw-Intact", and after this header the line should list in order the name of the fault, labelled as *fault* in the input file, the x-coordinate where the point is located, *xcoor,* the y-coordinate of the point, *ycoor,* the length of the elements of this fault (m), *length,* a flag that determines how the fault is growing ("yes" or "no"), *grow_both?*, shear stiffness (MPa), *stiffS*, normal stiffness (MPa), *stiffN,* inherent shear strength (MPa), *strength,* static friction, *friction-s,* dynamic friction, *friction-d,* and slip-weakening distance (m), *L*. The aforementioned fracture properties should represent the intact rock properties of the material because the cracks tested in the first propagation of growth at the coordinates specified simulates fracture initiation, and the failure of intact rock.

The user may also allow the fracture properties to evolve with growth, similar to the *Crack properties mentioned above. To indicate that the properties of the fracture should change with growth, the user must add an input line immediately before the line that begins with "*Flaw-Intact". This line must start with "*Flaw-Fault", and should include fracture properties that simulate weakened slip surfaces (i.e., cohesion, coefficient of static and dynamic friction). Figure 12 shows the exact formatting and location of these input lines.

4. Output

In a GROW run several files are generated and preserved throughout the length of the run, and maintained after GROW terminates. Some files maintain the initial contents written to the given file, and other files are repeatedly overwritten in various propagations of crack growth.

4.1. Files Generated

After GROW reads in the initial Fric2D input file provided by the user, which must exist in the same directory as the GROW executable, GROW copies the contents of this Fric2D input file to several files with differing names. Later in a GROW execution the contents of these files are overwritten to represent other fault geometries, which are expressed in Fric2D input files. Table 6 describes some of the files generated in a GROW run. In addition to the files listed in Table 6, GROW also automatically creates files to identify the most efficient fault geometry in a given propagation of crack growth. For example, if the user calls GROW with the input file *input.in*, a resolution angle of 45°, a minimum angle of 90°, and a maximum angle of 270°; and the Fric2D input file *input.in* contains one fault named *banning* that is growing from end 2, then Table 7 lists all of the Fric2D input files generated in the first propagation of crack growth for which W_{ext} will be calculated, and subsequently used to identify the most efficient orientation of the element added to end 2 of *banning*. In this example the most efficient orientation of the newly added element to *banning* is

oriented 180° from end 2 of the fault. For each of the input files listed in Table 7, a new Fric2D output file is produced after GROW calls Fric2D for the input file. If two faults are growing in *input.in*, and the faults are named *banning* and *coachella*, then the input file *input_1_coachella_1_45.in* contains the fault geometry after the most efficient element oriented radially from the fault *banning* is added to the geometry, and a new element oriented 45° clockwise from end 1 of *coachella* has been added. The fault *banning* grows first—that is, new elements are added to *banning* before *coachella* because faults propagate in alphabetic order in GROW. Figure 2 also lists the specific filename and illustrates the fault geometry for each geometry tested in the first propagation of crack growth.

Figure 12: Images of 3 Fric2D input files that shows evolution of fault properties when user choses to initiate a fracture from a point. Blue boxes highlight fault properties prescribed by the user with the *Flaw-Fault input line, and red boxes show intact rock properties prescribed with the *Flaw-Intact input line. The table below lists the values of the fault properties for the fault *back*.

```
*Tag                                    stiffS  stiffN   cohes   fric-s fric-d   L
*Flaw-Fault                             1e10     1e10      0   0.62    0.58    0.00025
*Tag            fault       xcoor    ycoor       length  grow_both? stiffS stiffN    shear-str   fric-s fric-d  L
*Flaw-Intact    backthrust  0.055424 0.01173379 0.002   yes         1e10   1e10       100      0.96   0.72    0.00025
```

```
*Tag                                    stiffS  stiffN   cohes   fric-s fric-d   L
*Flaw-Fault                             1e10     1e10      0   0.62    0.58    0.00025
*Tag            fault       xcoor    ycoor       length  grow_both? stiffS stiffN    shear-str   fric-s fric-d  L
*Flaw-Intact    backthrust  0.055424 0.01173379 0.002   yes         1e10   1e10       100      0.96   0.72    0.00025
1   0.055424    0.01173379  0.05610804  0.01361318  1e10    1e10    100     0.96   0.72    0.00025
```

```
*Tag                                    stiffS  stiffN   cohes   fric-s fric-d   L
*Flaw-Fault                             1e10     1e10      0   0.62    0.58    0.00025
*Tag            fault       xcoor    ycoor       length  grow_both? stiffS stiffN    shear-str   fric-s fric-d  L
*Flaw-Intact    backthrust  0.055424 0.01173379 0.002   yes         1e10   1e10       100      0.96   0.72    0.00025
fault   backthrust  no  yes yes
*Crack  1e10    1e10    100 0.96    0.72    0.00025
1   0.05473996  0.00985440  0.055424    0.01173379  1e10    1e10    100 0.96    0.72    0.00025
1   0.055424    0.01173379  0.05610804  0.01361318  1e10    1e10    0   0.62    0.58    0.00025
```

Table 6: Table lists files generated in a GROW run.

Filename	Description
input.in	Initial input filename provided by user. Formatted as a Fric2D input file. Must exist in current directory that from which the GROW executable is running. Must end in ".in". The user creates this file.
input.eff	Fric2D input file that represents the most efficient geometry identified in the current propagation of crack growth. GROW overwrites the contents of this file when a more efficient fault geometry is found at the end of a propagation of crack growth.
input.prev	Fric2D input file that represents the most efficient geometry found after the last new, most efficient element was added to a fault. GROW overwrites this file every time a new element is identified as the most efficient orientation of an element added to the tip of one fault. This file is used to generate the fault geometries that are tested for efficiency in the first general search of a propagation of crack growth.
input.prev_seq	Fric2D input file that preserves the most efficient geometry found in the previous propagation of crack growth, before any new elements are added to any of the growing fault tips. Used in tuning search of efficient orientations.
input.raw	GROW output file that records the exact geometry tested in each propagation of crack growth, the W_{ext}, $W_{ext}/\Delta A$ and change in $W_{ext}/\Delta A$ ($\Delta W_{ext}/\Delta A$) produced by each geometry, if a fault stops propagating, intersects another fault or a boundary, and other information related to this GROW execution.
input_cont*N*.in	Fric2D input file that records the most efficient fault geometry found in the last propagation of crack growth before GROW automatically restarts the run. *N* is 1 when GROW terminates and restarts itself after 5 propagations of crack growth. Fric2D generates this file and adopts this naming convention for all subsequent files created in the new GROW run (i.e., input_cont*N*.eff, input_cont*N*.prev, etc.). After 5 more propagations of crack growth, which began with the input file input_cont1.in, if faults are still growing, GROW starts the process again with the input file input_cont2.in.

Table 7: Table describes all the Fric2D input files used to represent the various fault geometries in the first propagation of crack growth when the user calls GROW with the input file *input.in*, a resolution angle of 45°, a minimum angle of 90°, and a maximum angle of 270°; and the Fric2D input file *input.in* contains one fault named *banning* that is growing from end 2.

Filename	Description
input_1_banning_2_45.in	Fric2D input filename that describes the fault geometry of the initial input file input.in, with one element added to end 2 of the fault *banning* that is oriented 45° clockwise from end 2 of this fault. The "1" indicates that the geometry is being tested in the first propagation of crack growth. The text "banning" indicates what fault the new element has been added to. The "2" indicates that the new element has been added to end 2 of the fault *banning*. And "45" represents the orientation of the newly added element.
input_1_banning_2_90.in	Same as above, except the element is oriented 90° clockwise from end 2 of *banning*.
input_1_banning_2_135.in	Same as above, except the element is oriented 135° clockwise from end 2 of *banning*.
input_1_banning_2_180.in	Same as above, except the element is oriented 180° clockwise from end 2 of *banning*.
input_1_banning_2_225.in	Same as above, except the element is oriented 225° clockwise from end 2 of *banning*.
input_1_banning_2_270.in	Same as above, except the element is oriented 270° clockwise from end 2 of *banning*.
input_tune_1_banning_2_157.5.in	First fault geometry tested in the tuning sequence of the first propagation of crack growth if the input file *input_1_banning_2_180.in* describes the most efficient fault geometry found in the first propagation of crack growth. In other words, an element added at 180° clockwise from end 2 of the fault *banning* optimizes $\Delta W_{ext}/\Delta A$ in the first propagation of crack growth, or $\theta_{opt} = 180°$. This fault geometry is identical to the initial input file provided by the user, but also contains one new element added to end 2 of fault banning at 157.5° $(\theta_{opt} - \theta_{res}/2)$.
input_tune_1_banning_2_202.5.in	Second fault geometry tested in the tuning sequence with the same parameters described in the caption above. Here, the fault geometry is identical to the initial input file provided by the user, but also contains one new element added to end 2 of fault banning at 202.5° $(\theta_{opt} + \theta_{res}/2)$.

4.2. Standard Output

GROW also produces print statements in the terminal window so the user can continually monitor the progress of a given GROW execution. These print statements include the fault geometry that is currently being tested for efficiency (i.e., the geometry for which Fric2D is calculating tractions and/or displacements along every element in the model), the external work required by each previously tested geometry, and other pertinent information. Figure 13 shows an example of standard output produced by GROW.

5. System Requirements

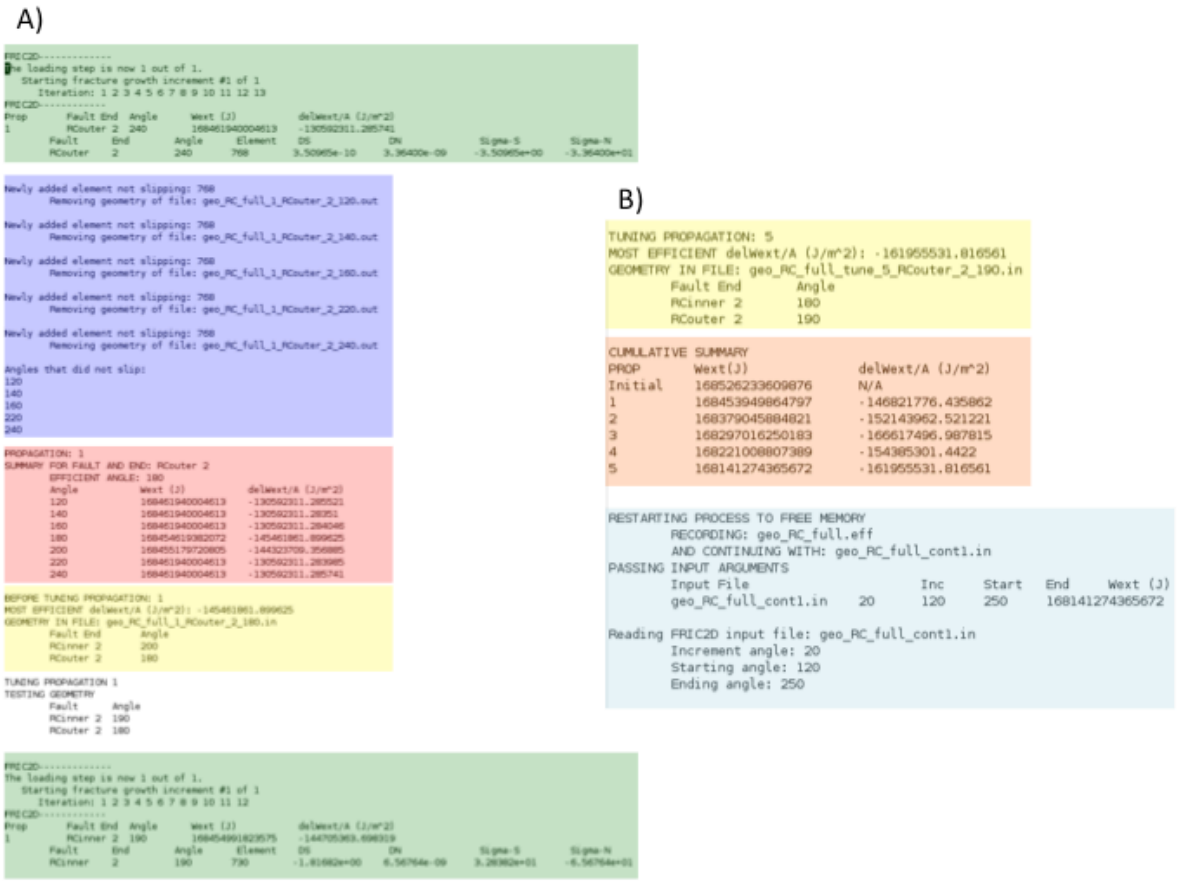
5.1 Files Required

To execute GROW the following files must exist within the same directory: the GROW executable *GROW.pl*, a perl script *Wext.pl*, the Fric2D executable *fric2d*, and the Fric2D input file that represents the initial geometry and boundary conditions of the system. See the *Input* section for more details about this input file and using GROW via the command line.

5.2 Compilers Required

Because GROW is written in Perl and calls Fric2D, which is a C executable, you must have a C and Perl compiler. This website describes more fully how to install and compile Fric2D from source code: http://www.geo.umass.edu/faculty/cooke/Fric2D/chapter1.html#sub4.

Figure 13: Examples of standard output produced by GROW. The green rectangles highlight print statements produced by Fric2D, and the fracture geometry being tested and shear and normal displacements (DS, DN) and shear and normal tractions (Sigma-S, Sigma-N) calculated by Fric2D on the element added at the given orientation from the end of the propagating fault. The red rectangle shows the $\Delta W_{ext}/\Delta A$ calculated for different orientations of an element added at end 2 of the fault *RCouter*. The orange rectangle shows the W_{ext} and $\Delta W_{ext}/\Delta A$ for each increment of growth. In this example $\Delta W_{ext}/\Delta A$ is negative because the example uses displacement boundary conditions, which causes W_{ext} to decrease. The darker blue rectangle shows that some of the elements added to the end of this fault did not fail in tensile or shear failure, and than GROW does not consider these geometries when searching for the geometry that optimizes work. The light blue rectangle highlights the print statements produced when GROW automatically restarts itself after 5 propagations of growth. The yellow boxes highlight print statements that show the most efficient geometry found in this increment of growth. A) shows this geometry before the higher resolution *tuning* sequence is entered, and B) shows this geometry after the final *tuning* sequence.



References

Cooke, M.L. and E. H. Madden (2014), Is the Earth Lazy? A review of work minimization in fault evolution, *Journal of Structural Geology*, 66, 334-346, doi: http://dx.doi.org/10.1016/j.jsg.2014.05.004.

Cooke, M. L., and D.D. Pollard, 1997. Bedding plane slip in initial stages of fault-related folding. Journal of Structural Geology, 19, pp. 567-581.