DISCRETE FRACTURE MODELING FOR FRACTURED RESERVOIRS USING VORONOI GRID BLOCKS

A Thesis

by

MATTHEW EDWARD GROSS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2006

Major Subject: Petroleum Engineering

DISCRETE FRACTURE MODELING FOR FRACTURED RESERVOIRS USING VORONOI GRID BLOCKS

A Thesis

by

MATTHEW EDWARD GROSS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee, David Schechter
Committee Members, Walter Ayers
John Keyser
Head of Department, Steve Holditch

May 2006

Major Subject: Petroleum Engineering

ABSTRACT

Discrete Fracture Modeling for Fractured Reservoirs Using Voronoi Grid Blocks.

(May 2006)

Matthew Edward Gross, B.S., Texas A&M University

Chair of Advisory Committee: Dr. David Schechter

Fractured reservoirs are commonly simulated using the Dual Porosity model, but for many major fields, the model does not match field results. For these cases, it is necessary to perform a more complex simulation including either individual fractures or pseudofracture groups modeled in their own grid blocks.

Discrete Fracture Modeling (DFN) is still a relatively new field, and most research on it up to this point has been done with Delaunay tessellations. This research investigates an alternative approach using Voronoi diagrams, yet applying the same DFN principles outlined in previous works.

Through the careful positioning of node points, a grid of Voronoi polygons can be produced so that block boundaries fall along the fractures, allowing us to use the DFN simulation methods as proposed in the literature. Using Voronoi diagrams allows us to use far fewer polygons than the Delaunay approach, and also allows us to perfectly align flow so as to eliminate grid alignment errors that plagued previous static systems. The nature of the Voronoi polygon further allows us to simplify permeability calculations due to orthogonality and, by extension, is more accurate than the commonly used corner-point formulation for non-square grid blocks.

DEDICATION

To:

Robert and Maria Gross

Samuel and Richard Gross

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

INTRODUCTION

1. The History of Modeling Fractured Reservoirs

Ever since the first numerical simulations for fluid flow modeling were developed, there has been the issue of how to handle the myriad fractures that lace naturally fractured reservoirs and spread out in stress-oriented directions from hydraulically fractured wells. Early formulations of square grids adjusted the permeability and porosity of the matrix blocks in an attempt to compensate for additional flow, or simply ignored the issue completely.

The most commonly cited and widely adopted first approach to the problem of simulation fracture networks was proposed by Warren and Root, and is used today in modified form as the Dual Porosity model.[1] Essentially, an additional analytical flow term is added to each block to represent the flow from the matrix into the fracture, and the block contains a mix of matrix and fracture permeability. Figure 1 presents a visualization of what is commonly known as the sugar-cube model:

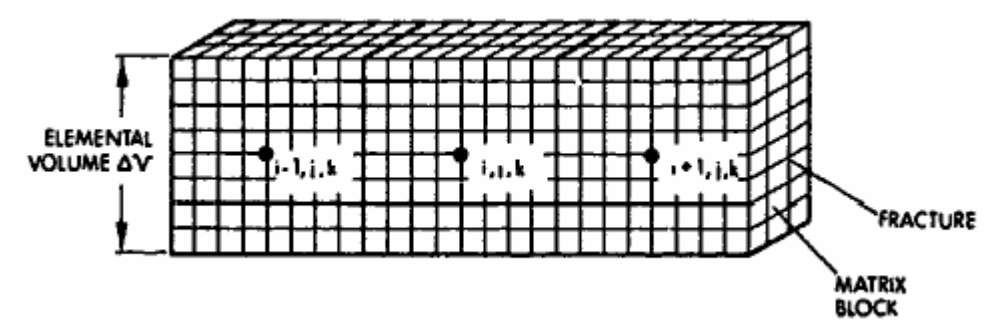


Fig 1 – "Sugar Cube" analytical model for Dual Porosity[1]

This thesis follows the style of *SPE Journal.*

This approach was a great improvement over earlier methods. However, inter-fracture flow was not modeled, and the description worked best on models with a very uniform set of fractures that did not induce flow anisotropy. This model is still used today as a way of including the effects of a fracture network without modeling the individual fractures, which may be too computationally intensive, even when sufficient data to visualize them is present.

Later adaptations improved the basic model,[2] expanding the technique as far as triple porosity models, which attempted to add an additional term for further realism. The Dual Porosity would persist as the general model for fractured reservoirs up to the present day, in spite of the numerous attempts to supplant it.

However, in part due to the general applicability to most reservoirs, and in part due to the limits of computational power, advances in terms of simulating individual fractures were limited. Some early attempts were made at simulating very small, limited cases by discretely representing the fractures,[3] but work remained rather sparse until the early 1980's.

2. The Discrete Fracture Network Approach

Discrete Fracture Networks (DFN) used in attempts to directly model the fractures, either as a virtual grid-block or some other separate entity. No longer is the fracture an additional factor influencing our matrix grid blocks, but a separate grid block with its own properties. This allows us to conduct simulations without blending matrix and fracture properties as was done in previous models. Additionally, it is now possible to directly model inter-fracture flow.

Work on DFN to model fluid flow in porous media dates back to at least the early 80's, when authors such as Noorishad, Mehran, and Baca applied the technique. Further refinements for use in petroleum engineering were made by Karimi-Fard and Firoozabadi in their 2001 paper.[4] In that paper, the authors advocate using a Finite Element approach to avoid problems with simulating the very small grid blocks that would be used to represent the fractures. To build his DFN grid, Karimi-Fard used a Delaunay tessellation to align the block boundaries on the fractures.

Aziz[5] would attempt to make a general framework for applying the DFN approach more widely. One of the more important aspects of the Aziz paper was expanding the use of DFN to finite difference simulators, which Karimi-Fard and Firoozabadi had dismissed, saying that an accurate representation "…may not be possible using a finite difference approach."

The DFN approach Aziz outlined called for the representation of fractures as their own discrete grid blocks, which he illustrated in Figure 2. However, while these grid blocks had volume as a property, in the grid itself, they were represented only as the intersection between two matrix blocks. As such, the boundaries of those blocks must be aligned along the fracture, and the fracture will be added during the solving phase, with flow occurring between neighboring fractures and the matrix blocks whose sides lie on the fracture itself.

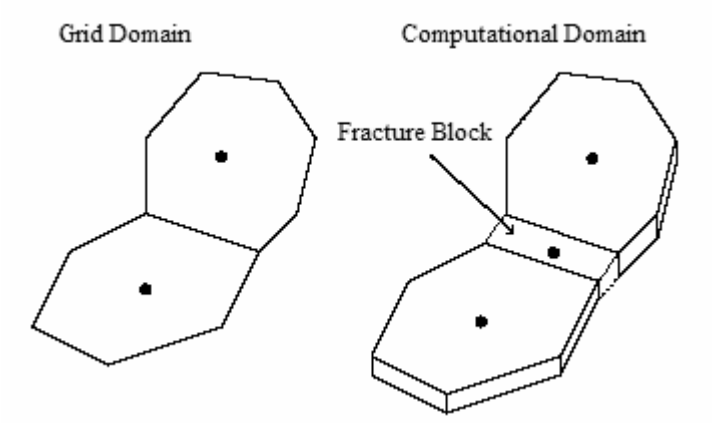


Fig 2 – Aziz's formulation of the DFN domains; fractures are grid boundaries, yet have volume for computational purposes.[5]

Aziz also compared the DFN 2D simulator to traditional square-grid models which used different blocks for the fracture and matrix. While these were completely artificial case studies, they did demonstrate that DFN simulators provided results that replicated those of very densely gridded traditional simulators. The Eighth SPE Comparative Solution Project had earlier examined the use of dynamic grids (Without a DFN component) and found that they can "allow a significant computer time saving during a reservoir simulation."[6]

While verification of DFN simulators against theoretical models is a useful first step, it doesn't link their improved accuracy to tangible field results. For a true experimental calibration of the method, a relatively recent addition to the repertoire of fluid flow visualization can be used. The CT scanner allows us to view the flow inside a core, distinguishing between fluids either through their density or the addition of a doping substance. A mounted core could provide the saturation front needed to assure that the flow modeled in the DFN simulator is an accurate portrayal of what is occurring in the reservoir itself.

Once DFN simulators have been calibrated to lab results, the next plausible approach is to apply the method to fields that have a history anisotropic fracture flow, such as the Spraberry field in West Texas. The Spraberry field was discovered in 1949,

and is thought to have originally contained 10 billion BBLs of oil, of which, only 10% has been recovered, in part due to the difficulties of water flooding a field where flow is primarily controlled by the fracture network.[7] Such fields are poorly served by the current methods for fracture characterization, yet contain large possible payoffs, making them ideal candidates for DFN analysis.

Fracture data will have to be extracted either from seismic results, approximated from outcropping data, or FMI logs, to cover only a few of the possible sources for fracture data. Techniques such as pressure transient testing, tracer tests, and compositional gradients, while not providing exact fracture data, can also be used to discover fractures and causes of flow anisotropy. While fractures are represented discretely, the modeled fracture does not have to be a single fracture, but can instead be a pseudofracture, with properties derived from actual field results, as a form of history matching.

While the DFN approach is currently limited to experimental and two dimensional simulators, as computational power expands and is complemented by increased data on the fractures in the reservoir a full 3D approach should be usable. For current purposes, however, petroleum engineering and the geosciences in general benefit from the layering of sedimentary rocks, allowing for the use of two dimensional models to represent beds. Vertical permeability is often a tenth of horizontal permeability, making the treatment of layers as individual flow units possible in many reservoirs.

This approach should increasingly supplant a basic Dual Porosity model in many areas where fractures dominate such as Type I fractured reservoirs, where fractures provide both essential porosity and essential permeability, as per Nelson's classification.

Extending any kind of polygon into the third dimension can be difficult, and extending the DFN technique to do so is also a challenge. Work on Voronoi diagrams in the third dimension is still ongoing in the world of computational mathematics, although there are some techniques for building Voronoi polygons that extend to the third dimension as well as the first two.

However, given that our blocks often represent one layer and the aforementioned bedded nature of sedimentary rocks, a straight-down extension as demonstrated in Figure 3, or a curvilinear approach can be added to the 2D Voronoi diagram to handle additional layers.

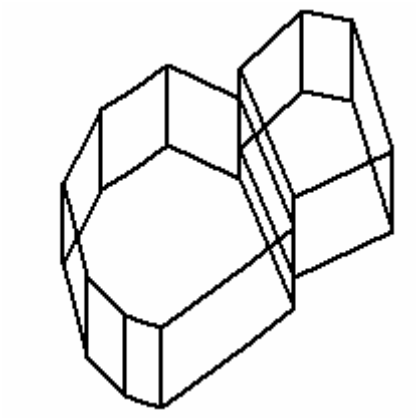


Fig 3 – Example 3D Voronoi blocks, extended directly downwards

It should be noted that the three dimensional aspect of this adaptation will necessarily make the flow between blocks not completely orthogonal, in the case of varying height between grid blocks, which is not uncommon in a reservoir setting.

3. The History of Voronoi Diagrams

Simplistic Voronoi-like diagrams were employed as early as 1644 by Descartes; however, the first comprehensive formulations were developed by Peter Dirichlet and Georges Voronoi. Voronoi proposed the general case of the Voronoi diagram in 1907.

These diagrams are also known as Dirichlet domains, and consist of a polygon with all edges equidistant between the center point and neighboring points. Alternatively, to quote Wolfram Research, a Voronoi diagram is: "The partitioning of a plane with *n* points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other."[8]

Voronoi polygons were already being applied as early as 1909 in the earth sciences to estimate ore reserves. However, well into the 20th century, Voronoi diagrams were being rediscovered (and renamed) by various scientists in a variety of fields, culminating in perhaps the last know rediscovery in 1987 by Icke for use in astronomy. Voronoi diagrams have been used in just about every imaginable scientific field, including use to estimate the growth area of trees to areas of language dialects. Well into the 1960's, the application of Voronoi diagrams was limited, due to the difficulties of drawing them by hand, but advances in computational geometry and microcomputers have put them into the spotlight.

As geometry goes, the actual techniques for building Voronoi diagrams in a non-graphical sense are quite new. While several methods exist for drawing a Voronoi diagram by hand, they are of little use for simulation purposes. Work on boundary Voronoi diagrams started in 1987 with a paper by Wang and Schubert defining the concept,[9] and work continued on the subject as recently as 1995.[10] Thus, Voronoi diagrams are a subject many people in both the petroleum industry and the broader world are unfamiliar with.

Much of the recent work with Voronoi diagrams has been centered around Japan, including what is now the 2nd Annual Symposium on Voronoi Diagrams which was held in 2005 in Seoul, Korea. This area of research is quite active, and already several different methods for constructing two dimensional Voronoi diagrams exist, with research continuing into the third dimension.

Techniques vary from older methods such as Flipping which resulted in a worst case time of n^2 to relatively newer methods including Gift-Wrapping and Divide-and-Conquer which see $O(n)$ run times and superior worst case scenarios.[11] Modern computers provide the computational power necessary to create these grids on demand, with recent advances[12] allowing for the use of graphics co-processors to aid the building process, which should allow for building denser grids, as well as real-time rebuilding of the grids for user convenience.

4. Why the Voronoi Diagram Was Selected

Given the ease and years of use of traditional square-block grids, some explanation is needed for the application and advantages of unstructured grids. While the order of a structure grid is sufficient to determine its neighbors, and unstructured grid must track its location as well as the neighbors to which it is adjacent. As such, the use of dynamic grids is necessarily more complex than a square grid model.

One of the earliest reasons for going with an unstructured grid was the issue of grid alignment. This problem occurs due to the way that flow is simulated, as flow takes place only perpendicular to grid sides. This results in poor modeling of flow when the primary direction does not correspond well with any given side, as the fluid is actually moving farther through grid boundaries than it is physically moving in the Reservoir, as shown in Figure 4:

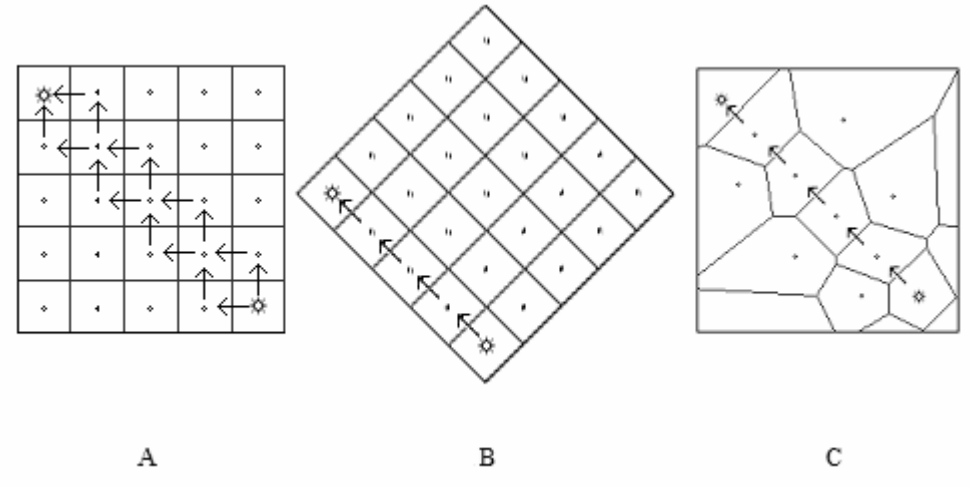


Fig 4 — (A) An improperly aligned square grid, (B) A grid rotated to align, and (C) A Voronoi grid which requires no rotation

Grid rotation was the first and most obvious solution to grid alignment problems, however it is limited to patterns where the grid can be rotated to fit all flow paths. In the case of field anisotropy, it may not be possible to simply rotate a grid to assure flow alignment.

Early solutions following the use of alignment were curvilinear, so called "Nine Point"[13] versions of square grids which allow for flow through the corners of square grid blocks as well as through the sides. The nine-point approach merely allowed for flow through the corners of a square grid, calculating the properties of that flow path from an average of the flow paths of the true side. This complicated flow formulations, and also was only a "split-the-difference" solution to alignment problems, as it merely reduced the angle to which it was possible for flow to be misaligned.

The curvilinear approach is a modification of the square grid that slants the grid as a whole, while maintaining the constant number of sides and orthogonality, a cross-section of which is displayed in Figure 5. Much like grid rotation, this solution assumes a relatively simple reservoir, and is inapplicable to more complicated grid geometry. However, it did provide an early solution to simple grid alignment problems and also had a formulation to allow for automatic grid generation.[14]

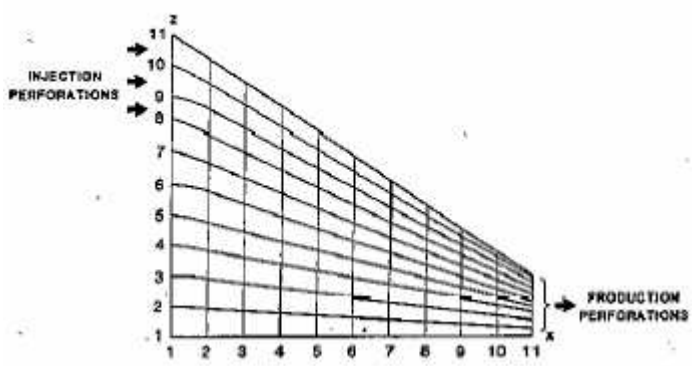


Fig 5 – An example of a curvilinear grid in 2D[15]

The use of polygonal patterns in which the number of sides are fixed, continues up to the modern day[16] in an attempt to address this issue. Although this technique is more elegant than the kind of mathematical fix embodied in the nine-point solution, it suffers from the same drawback of essentially attempting to minimize the flow error by increasing the shear number of possible flow paths. Truly unstructured gridding allows the simulator to align the grid faces precisely with the direction of flow, reducing the computational complexity of the work as a whole.

It is worth noting that dynamic grids do not inherently fix the problem of grid alignment. Commonly used fixed-side-number methods such as the Delaunay tessellation do allow for perfect alignment for flow between two wells, however, for complex flow geometries, it becomes impossible to align the grid correctly, as there are insufficient sides to allow all of them to perpendicular to the required flow directions.

The advantages of dynamic gridding also become increasingly apparent when we have to deal with large areas that are poorly described, or features that we wish to model in place. Frequently, this includes either a fracture network or faulting. When we wish to grid up a field with irregular features, we are somewhat limited by the basic square formulation. The basic formulation can be reduced in size so that the feature in question can be modeled, but doing generally produces so many grid blocks that the resulting grid is too computationally demanding to be practical.

Unstructured gridding techniques such as the Delaunay tessellation and corner-point gridding allow us to align the grid boundaries along the contours that we desire. However, the Voronoi diagram enjoys several advantages over either technique.

Being dual to the Delaunay tessellation in the graphical sense, the Voronoi diagram can describe any shape that the Delaunay tessellation does, however, it can do so using less grid blocks (although of a varying number of faces.) The advantage of the reduced number of blocks is somewhat mitigated by the variable face number, which requires the use of sparse matrix solving equations rather than the more traditional and more efficient banded matrix solvers.

Flow for the Voronoi diagram is guaranteed to be orthogonal, thus allowing us to use the simpler orthogonal flow equations from our square grid rather than the corner-point formulation with suffers accuracy due to the non-orthogonal approximation. This is particularly evidenced in the formulation of the transmissibility equations, which gain both accuracy and simplicity compared to the nine-point formulation, or a similar approximation for the Delaunay triangle. So long as were able to construct a grid consisting only of true Voronoi polygons, we gain the simplicity of the square grid

calculations with the multifaceted nature of center-point geometry or other, more generic polygon solutions.

The trade-off is limited grid flexibility. As we must build our diagram so that all polygons are Voronoi polygons and aligned with the boundaries desired, we are occasionally left with smaller grid blocks than more flexible corner-point geometry would allow for. This is only to a limited extent, as the corner-point grid's accuracy is impacted by the degree of distortion of the gravity centers of the grid block.[17]

Having outlined the strengths of the Voronoi diagram, it is only fitting that the most striking disadvantage of the system should be detailed. That is, with a variable number of grid faces, one ends up with a sparse matrix rather than a banded matrix, requiring that the matrix solver is a more general (and computationally intensive) sparse matrix solver.

These difficulties can be addressed, however, given the proper techniques. Once the Voronoi diagram is generated, additional points can be added to reduce Voronoi diagrams to a reasonable number of sides, at the expense of increasing block numbers.

Alternatively, there exist several algorithms, such as the Gibbs-Poole-Stockmeyer and Cuthill-McKee which can reduce sparse matricies, allowing us to go from $O(n^3)$ which is the general case to $O(n^2)$ for gaussian elimination. It should be noted that while performance of these algorithms is generally linear, worst case performance can be as bad as not using the algorithm at all.

While the idea of using the Voronoi diagram is that its lower block count and orthogonality of flow surfaces can reduce computational time, this is counteracted by the use of a sparse matrix solver. Using Voronoi diagrams may or may not increase the computational time of the simulation, the advantages are primarily the improved flow alignment and improved accuracy through orthogonal flow (over approximations such as center-point,) both of which lead to increased accuracy when modeling field performance.

PROBLEM STATEMENT

Our goal is to create a gridded structure consisting of Voronoi polygons, with the boundaries aligned on our fractures. For simulation purposes, all polygons must be valid Voronoi polygons (due to the assumptions made for orthogonality of flow) and their boundaries must be on the fractures, so that when they are simulated, their positions are accurately modeled. Thus, we will be able to build an accurate simulation model of a field where fractures dominate the reservoir and produce significant flow anisotropy. Higher accuracy in simulation of these fields will allow for optimized production techniques and ultimately greater recovery of the oil in place.

The generation of proper Voronoi polygons from points has been solved since the early 80's in computational geometry. For the purposes of using only points to align our boundaries, there are three central problems that need to be addressed:

The first is how the basic step of how points dictate polygonal edges in the Voronoi diagram. This is a deduction from the nature of the Voronoi and forms the basis for our other approaches. The second deals with the nature of intersections, and how to assure that points from a nearby intersection don't distort it in such a way that it does not align correctly. The third and final issue is a verification process to assure that no points distort edges on top of fractures, and to add points to assure that the boundaries stay aligned.

While much of the work is devoted to explaining how to build a Voronoi diagram that can be used with the DFN approach, additional details are included as to exactly why a Voronoi diagram should be used and how it can be applied to solve other issues that arrive in the course of simulation.

METHODOLOGY

1. Building the Voronoi Diagram from Points

Our first concern is the generation of a Voronoi Diagram from any set of points, within a polygonal boundary. So long as we possess a method of doing so, we can produce any grid required so long as the grid points are positioned appropriately to form the structures we desire. To do so, the visibility shortest-path Voronoi diagram generation method is utilized.

The first step to building the Voronoi diagram is essentially to build a constrained Delaunay tessellation, which is defined a triangulation where the circumcircle of each triangle does not contain in its interior any other vertex which is visible from the vertices of the triangle.[18] This concept is demonstrated graphically in Figure 6:

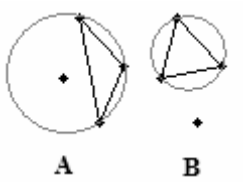


Fig 6 – (A) An invalid Delaunay triangle contains extra points within the circumcircle, (B) A valid Delaunay triangle constructed for those same points

Essentially, any point will be connected to the nearest available points to form a triangle. This will give us a series of shortest distances to each point, which will in turn be used to produce the Voronoi polygon.

There are numerous approaches to building the polygon itself, the earliest proposed by El Gindy and Avis in 1981.[19] From there, the visibility graph is built, which can then been solved for the shortest path using any number of methods, such as Dijkstra's.[20]

From a geometric standpoint, the process used is as follows: We take a system which has been gridded up using a Delaunay tessellation, and examine each vertex in turn, as each will form the center point of a Voronoi polygon. Figure 7 shows our example series of Delaunay triangles:

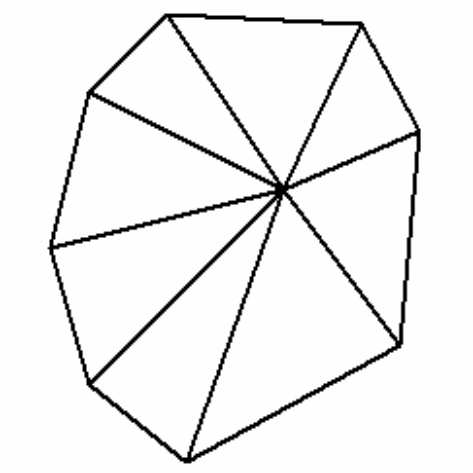


Fig 7 – A single vertex with accompanying Delaunay triangles

Each line segment going into the vertex is bisected and its midpoint, and a perpendicular line drawn through it, extended outside the polygon. Figure 8 shows the above example, with the bisecting lines added:

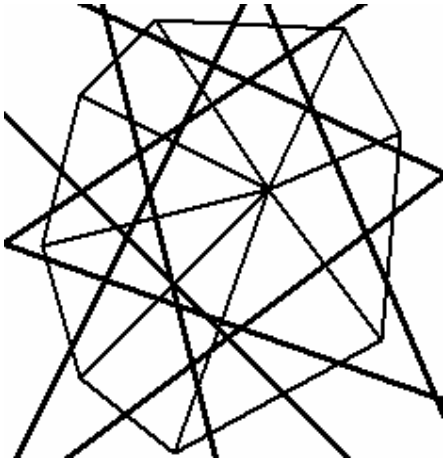


Fig 8 – Vertex and triangles with perpendicular lines added

We detect the intersections and truncate the lines at the intersections, which gives us the polygon in Figure 9:

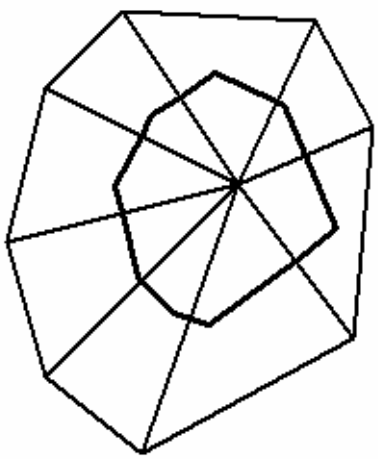


Fig 9 – A finished Voronoi polygon after line truncation

This procedure is repeated for all vertices in our grid, and then extended all external intersections to our boundaries to form the boundary polygons shown in Figure 10.

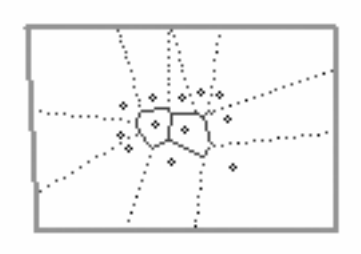


Fig 10 – A bounded Voronoi Diagram, with extended lines dotted

Our technique will not modify existing methods of producing Voronoi polygons themselves, but will focus solely on positioning points so as to align the fracture boundaries on polygon borders, so that any standard technique for generation can be used.

2. Aligning Boundaries on Fractures

With the technique to construct Voronoi polygons from any set of points in a 2D space available, it is possible to proceed with the task of placing the points in such a manner that they form boundaries along our fractures. Three basic issues require addressing: how to grid individual fracture lines, how to handle intersections, and how to verify neighboring points do not disturb boundaries.

The approach outlined in this paper is two dimensional. Although it is possible to extend the model itself downwards in a straight or curvilinear approach to allow for 3D modeling, the formulation inherently considers fractures to be two dimensional. For most of our applications, this is a reasonable approximation, and generally our grid blocks are the entire layer of interest. However, this simplification is worth noting, as fractures into adjacent water-bearing layers can be a major influence on the production.

So while fractures must be constrained in the z direction to the layer of their origin, nothing prevents this approach from being compensated with advanced fracture modeling techniques of an analytical nature. This can include the use of analytical techniques beyond the simple Parallel Plate model, or simply the inclusion of a "roughness" or tortuosity factor to match experimental results. For the vast majority of naturally fractured reservoirs with anisotropic flow, and most correctly contained hydraulically fractured reservoirs, the model outlined here will be applicable.

Now that we have constrained ourselves to two dimensions, it is possible to further abstract our fractures as a series of straight lines. While any complex curve may be reduced into a series of straight lines, it will be important for the purposes of gridding that these lines be as straight as possible. While fractures are rarely truly straight, for the purposes of modeling they may be abstracted as such generally due to the scale involved, as well as the interest in using as few polygon sides as possible for each block.

Once we have a system of straight lines, borders can be aligned on them by simply placing points equidistant on each side of the line, as our algorithm to build the grid (and the Voronoi diagram, itself, by definition) places borders in the exact middle of

two neighboring points. This can be done rather simply by using the equation of the fracture line and taking a line perpendicular to it, then going a set distance down that line on either side and placing a point, as demonstrated in Figure 11.

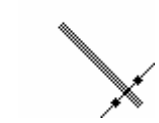


Fig 11 – Adding bracketing points to a fracture to align the Voronoi boundaries

Generally, an even spacing of points could be used to grid up fractures, or the spacing could be tuned to use a certain number of blocks per fracture line. Alternatively, as is detailed towards the later part of the methodology, only the end points can be bracketed, and the exact number of points needed to properly represent that fracture segment calculated with the algorithm used to assure that no neighboring points interfere with the fracture boundary, and added dynamically.

There is, however, one more concern, regarding the simulation dynamics. Since we are treating the fracture as a flow block, the easiest approach is to have an entire side dedicated to the fracture. If endpoints are simply placed perpendicular to the end of the fracture, this will not happen:

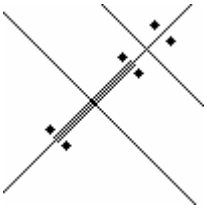


Fig 12 – A point configuration where grid blocks have only part of a fracture on the resulting sides

The extra points to the side of Figure 12 merely represent other non-fracture related points in our grid; the final grid block could end up containing almost no fracture

on its side if the other neighboring point is sufficiently far away. Rather than attempt to come up with a formula for compensating for part block-to-block transfer and part block-to-fracture transfer, it is easier to simply adjust the gridding to assure that all blocks have sides consisting of either all fracture or no fracture.

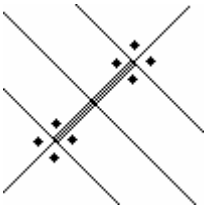


Fig 13 – The fracture from Figure 12, with end point placement modification

In Figure 13, we have placed four grid points at each end of the fracture. Both sets are equidistant from the actual fracture end point, which results in the block boundary being aligned perpendicular to the endpoint. The obvious drawback to this technique is the use of more grid blocks, which will have to be weighed in light of how well the actual fracture end points are known, as well as the number of fractures in the simulation as a whole.

To use the validation technique to assure the optimal amount of grid points, it may also be necessary to place the end points very close together, as any point placed between them would reintroduce the problem they were placed to solve.

3. Handling Fracture Intersections

The aforementioned technique will work for a reservoir consisting of non-intersecting fractures, so long as the fractures are either finely gridded or relatively widely spaced. However, this is not useful for the vast majority of fractured reservoirs, so it is necessary to extend our technique to intersecting fractures.

Using our previous technique, two crossing fractures would, unless by coincidence their spacing was perfectly aligned, distort the area of their intersection (Figure 14):

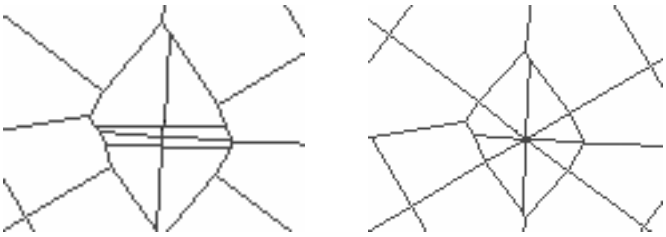


Fig 14 – An unadjusted intersection, and one where points have been adjusted

To avoid this, the area immediately around the fracture needs to be cleared out, and a pair of points set around each line segment involved in the intersection, an equal distance away from the intersection itself.

To facilitate this, all existing fractures should be reduced to a set of line segments, split at the intersections. In this form, rather than analytically comparing lines to determine if they intersect, the end points can simply be compared.

Once all line segments involved have been identified, the distance cleared out should be maximized to prevent the formation of very small grid blocks. The distance will be necessarily restrained to less than one half of the distance to the nearest intersection to prevent overlap and interference.

With our intersection cleared of points, we travel a fixed distance down each line segment from the intersection, and place a pair of points on each side, to form a polygon boundary on the fracture, as shown in Figure 15.

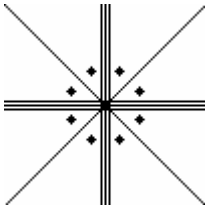


Fig 15 – A 90^o intersection with intersection points present

This solution will work for all intersections, regardless of the number of intersecting points or size of the angles between the intersecting lines, however, it is a non-ideal solution, in that we are creating two grid blocks between each set of line segments when we should be producing only one. Furthermore, for low angle intersections, this will require a truly huge number of points to preserve the boundary angles. Therefore, an optimized solution is suggested to produce a sparser grid in the event of these types of intersections.

4. Low Angle Intersection Optimization

To produce a single block with sides along each of the two lines segments, we need to place a single point between each set of segments in such a way that it is equidistant to neighboring points in our intersection on a perpendicular line.

We can place all the points by proceeding a set distance down each line segment, and drawing a perpendicular line through each of them, as demonstrated in Figure 16. These perpendicular lines will intersect, and the intersection point will be where we place our first point. Although it is possible to get an intersection with angles larger than 90^{o}, the distance at which we must place the points makes it undesirable for use.

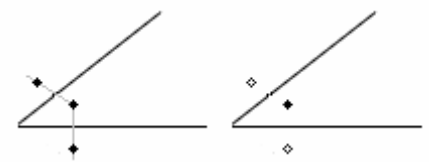


Fig 16 – Perpendicular lines are used to create an intersection point for properly aligning gridding for an acute angle

Once the point has been placed, the distance from the intersection point to each line segment should be measured, and another point placed an equal distance down the perpendicular line, so that the grid borders will align perfectly with the line segments. This approach can be continued around the intersection, by recycling the first line to our newly placed point (which is now the middle point of the line segments were are examining) and taking a line with the perpendicular slope of the neighboring line segment through the intersection point, generating a new point with each angle, until we come full circle around the intersection, as demonstrated in Figure 17:

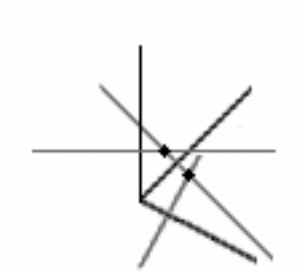


Fig 17 – The technique expanded to grid around the intersection, with an intersection point for each angle

This process will have closely mimicked the procedure outlined earlier in which a Voronoi polygon was built from a Delaunay tessellation, with the intersection as the middle vertex of the neighboring triangles. However, there are important differences. In the procedure, rather than the midpoint of the line segments, an arbitrary (and generally small) distance was chosen as the starting point. Additionally, the intersecting line must be placed so as to go through the later points rather than forming them (Figure 18):

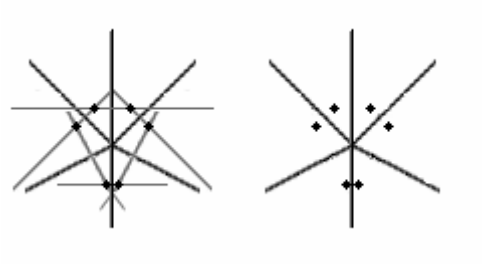


Fig 18 – A finished intersection using this technique, with and without the lines

This technique only works for angles that are relatively similar. Figure 19 demonstrates a case in dissimilar angles cause the technique to fail:

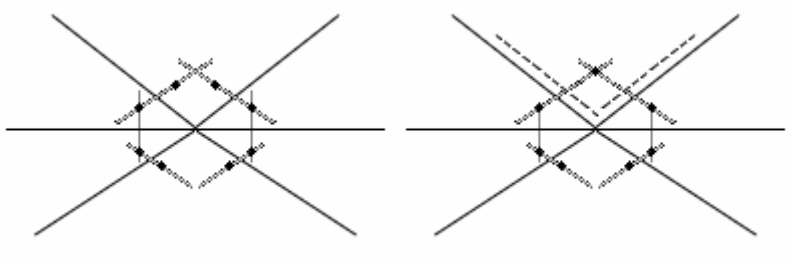


Fig 19 – An intersection with obtuse angles constructed using the low angle technique, dotted lines show where the Voronoi boundaries would fall if this technique was attempted with an obtuse angle

In that case, it will be necessary to simply place a point on either side of the obtuse angle, in essence "skipping" that section and using an extra grid block to properly align the sides. An example of that is shown in Figure 20, with the added points used to compensate for the obtuse angle shown with a hollow center.

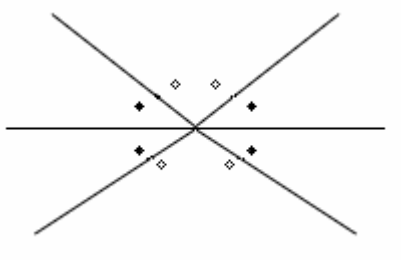


Fig 20 – Handling obtuse intersections, with obtuse angles "skipped"

Furthermore, we are only guaranteed a properly gridded intersection using this method if the intersection is symmetrical. This isn't as critical a restriction as one might think, as fracture intersections are often symmetrical, having formed perpendicular to stresses which are relatively low angles apart. However, it means that this technique cannot be applied to branching intersections or asymmetrical fracture sources. Thus, most of our intersections still must use the earlier case, although this technique will find particular use in addressing low angle intersections or kinks in existing fractures.

Calculating the distance to clear from the center of the intersection, the worst-case functioning scenario will be $\sqrt{2}$ times the distance down the line you use. The does mean this technique uses more space compared to the two-dot bracketing method, which can be an issue when there are multiple fracture intersections in very close proximity.

While the two-dot bracketing method does provide a far more useful general solution, simulation time will be almost entirely a function of the node points, and thus extra diligence in constructing the grid mesh is desirable, as that operation need only be performed a small number of times (or only once, if no modification is desired) compared to the possibility of hundreds of runs of a finished simulation. Finally, smaller blocks can lead to convergence problems, providing a final justification for the increased complexity of adding a secondary intersection method.

5. Evaluating Proximity Issues

Now that intersections and basic gridding have been handled, one more issue requires addressing, namely how to make sure that the points placed so far are not distorting the boundaries that the simulation will need. When two gridded fractures are close to each other, it is possible that the points on one distort the boundary on the other as shown in Figure 21:

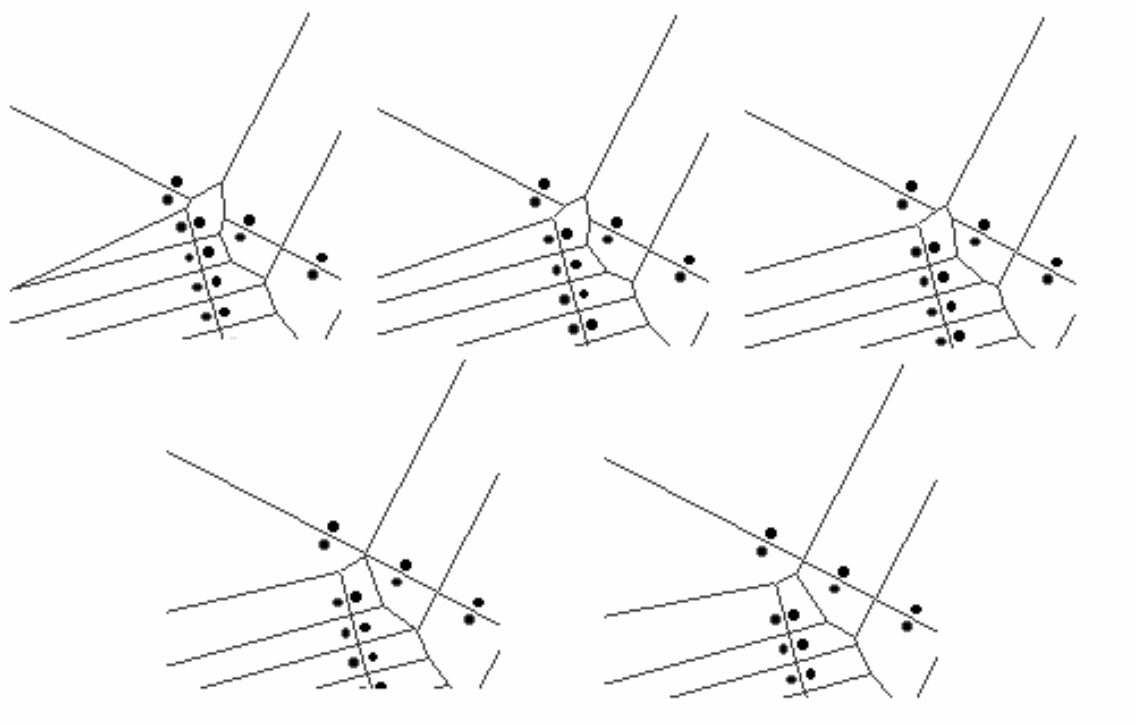


Fig 21 – Interference of a nearby point on Voronoi boundaries as a function of proximity, from upper left to bottom right

While many of these problems occur in intersection situations, extending the radius of the intersection points to encompass the problem area is often impractical due to fracture spacing or the need for high resolution in the area in question. Increasing the number of points around each fracture can also be prohibitive, if done across the board. Thus, we need a method of adding grid points only in areas where it is necessary to preserve the gridding structure.

At this point, the implementation of the data structure tracking our points becomes important. Arrays are generally used due to a fast *O(1)* look-up time for any particular value. However, insertion can be problematic as it generally requires that either our array be unsorted (in which case insertion is *O(1)*) or that it be resorted, which is *O(n log n)* in the case of an optimal sorting algorithm. For the purposes of constructing the point network, it is suggested that a linked list be used, which allows for insertion at any point, which is finally cast to an array when we are finished with our grid construction.

Maintaining a "sorted" list is important, so as to know which points are adjacent to each other and assigned to be bracketing points on which fracture line, in order to verify all points are compliant. This may be done either by adding metadata to the point, identifying the line the point is assigned to as well as its ordinal position in placement, or simply adding the points to the storage structure in a known order (although this later case requires an ordered structure at all times.)

As this process should be performed after we've done all other modifications to our grid, we will have points placed to manage intersections. These points also need to be considered, and the two-points-per-line-segment technique is easily incorporated into our scheme as simply another set of bracketing points ending the line segment. The single-point-per-angle scheme may be also be included in a similar manner.

Our process for grid verification is thus: all fractures have already been reduced to straight line segments, which either do not intersect other segments or terminate at fracture intersections. On each side of this fracture are a set of points running down it to align the grid. In the case of line segments that end in fractures, the final points will consist of two of the fracture bracketing points.

For each fracture segment in our grid, all points must be verified against all other fractures. The basic comparison is shown in Figure 22, and will be between the points on the nearest side of each line segment in question, and each point on our segment will be compared to each two adjacent points on the other, running down the length of the segment:

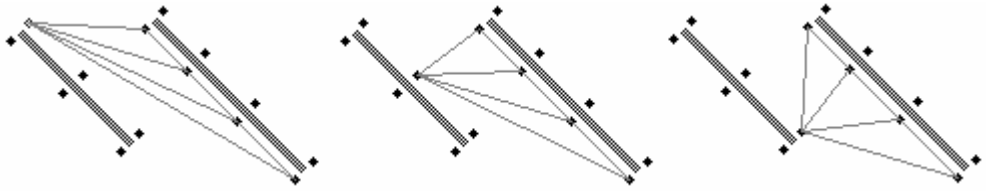


Fig 22 – One set of comparisons between two line segments, as each point in one side compares to all neighboring pairs on the other

While this may be sufficient for most cases, when the line segments in question are nearly perpendicular, the end points of each side of line segment may need to be included, regardless of which side is chosen for the comparison (Figure 23.) Comparing bracketed points on one line segment to those on the other, and vice versa is generally sufficient to determine which sides need examining, although a slope check can help verify which end points do or do not need to be included.

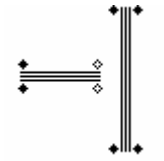


Fig 23 – An example of a case where both endpoints must be evaluated

The distance we must first gauge is that between the closest point on the line to our point of interest. This can be found by running a line with the perpendicular slope of our line through our point of interest until it intersects the line. The intersection point will be our closest point (Figure 24.)

Fig 24 – Cases where the intersection point will be the closest point, and cases where the end point will be the closest point

There are geometric equations available to determine the intersection point of two lines, including whether or not they intersect at all. Paul Borke, from Swinburne University Center details the following equations[21]:

$$u_a = \frac{(x4 - x3)(y1 - y3) - (y4 - y3)(x1 - x3)}{(y4 - y3)(x2 - x1) - (x4 - x3)(y2 - y1)}$$
$$u_b = \frac{(x2 - x1)(y1 - y3) - (y2 - y1)(x1 - x3)}{(y4 - y3)(x2 - x1) - (x4 - x3)(y2 - y1)} \quad \text{……..……(1)}$$

Either of the above coefficients can be plugged into the following equations to supply the X and Y coordinate of the intersection point, as follows:

$x = x1 + u_a (x2-x1)$……..….(2)

$y = y1 + u_a (y2 - y1)$……..…...(3)

If either of the two coefficients are less than zero or greater than one, then there is no intersection point. In that case, one of our endpoints will be the point on the line closest to our point of interest. When that occurs, the point of interest will not exhibit a distortion of the fracture boundary, and we will not have to examine that case for the purpose of verifying our grid.

Once we have the point of intersection, the distance to both the bracketing points on the same line and the point of interest can be evaluated (Figure 25.) If either of the distances to the bracketing points is farther than the distance to the point of interest (Figure 26,) an additional set of bracketing points will need to be placed.

Fig 25 – The distances from each point to the intersection point (hollow) are shown as dashed lines, which the distance from the point of interest is a solid line

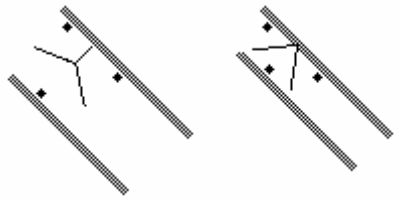


Fig 26 – A point triangle that doesn't require correction and a second set that does

Provided that we used the line between the bracketing points rather than the true line for our examination, our extra pair can consist of our intersection point, and another placed by taking a the perpendicular slope of our line, and proceeding from our intersection point twice the bracketing distance, so as to place a point balanced on the other side of our fracture (Figure 27):

Fig 27 – Placement of a new point set to correct a fracture boundary

However, placing the new point at the intersection will only solve our problems if the slopes of the two line segments are very close, as the new point is constructed

perpendicular to the first line, not necessarily the second (Figure 28.) Especially with low angles, this can cause a great number of added points.

Fig 28 – Different line slopes result in a point which is not perpendicular to the intersection

What has occurred in the figure above is that the point placed is placed using the slope of the second line (which is necessary, to assure that the boundaries are aligned on our fracture line.) Thus, we have formed a new triangle for evaluation where it's possible it still fails validation. If this is the case, our algorithm would try to go back and place a point on top of the one it just placed.

A more general solution is to simply add another point equidistant from the two points (Figure 29.) While this may result in a non-ideal solution for line segments that are parallel or near parallel, it allows us to evaluate all line segments without worrying about special cases, and avoid the error outlined previously.

Fig 29 – Placement of a new point equidistant to our two trial points

We must insert the new point into our list of points so as to maintain the order of the examination, as the points in order on the line segment may be used later in the process for evaluating the validity of other nearby line segment's bracketing points.

Either the point is dynamically inserted if we are using a linked list or other dynamic data structure, or the point is inserted in the end of our array and the array resorted to assure proper ordering. Fortunately, as we already have a mostly ordered array, the sorting process can take place in *O(n)* time.

Assigning points to their line for the purposes of use in this method is simple for the two-point-per-segment intersection method, as each line segment will get two points, one on each side. The greater difficulty comes from using the single-point method, which requires a point to share two line segments, for the purpose of comparison. An example of situation in which the line must be considered is shown below (Figure 30):

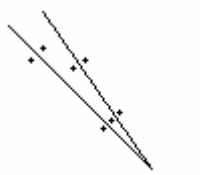


Fig 30 – An example of a case requiring mid-point examination due to misaligned bracketing points in close proximity

Rather than bracketing fractures with points at a set interval, this technique can be used to add the optimum number of points. To do this, each line segment is bracketed with only 4 points, one set at the each end point (or as mentioned earlier, 8 points to guarantee all sides are 100% fracture.) Then intersections are dealt with, and the verification procedure is run to place only those points which are needed.

If points have been be added to lines that have already been evaluated, then were must either back the entire procedure up, or the procedure should be run iteratively until no further points need to be added. It should be noted that only the points that were added in last run need be checked against all others for verification purposes, hence run time will decrease as the process iterates.

The choice between the fixed interval and the automatic generation is generally one of resolution. If a desired spacing is already known, or the grid is full or expected to be filled with many non-fracture points, a fixed spacing technique supplemented with the evaluation technique may be appropriate. From a strict standpoint of minimizing run-

time, only the end point should be bracketed and all other points added via the method described, so as to guarantee the bare minimum number of points placed and therefore the minimum number of grid blocks.

The operation described requires that roughly half of the points be compared to half of the points, each, giving the operation an average time of $O(n^2)$. This is not influenced by the choice between an array or a linked list implementation, so long as both are handled well, assuring that a pointers are kept to current locations in the list to prevent walking through it excessively, or in the case of arrays, that they are sorted with an algorithm that gets $O(n)$ performance for a mostly sorted array. This is, of course, assuming that the number of points added is relatively small compared to the total number of points, an assumption that may not be true in the case of using this technique to place points optimally, rather than a fix-distance bracketing solution.

User-added points are somewhat more complicated than in traditional models, as we must verify that they don't move the Voronoi boundaries off of the fractures. As such, following point insertion we must run the evaluation technique, comparing that added point to the nearest side of all fractures to insure that boundaries are valid, and adding points as necessary. The user would also need to be constrained so that they could not place points closer to the fractures than the bracketing distance, which would be impossible for even the aforementioned evaluation technique to fix, although in practice a relatively small bracketing distance could make such an event unlikely.

The user has a definite, legitimate need to place points, either to increase resolution in an area, or for the purposes of grid alignment and flow. However, in part due to the nature of the automated gridding, such techniques will be somewhat limited by the approach we have chosen.

6. Using Real World Data

Once we have all the tools required to build the grid, we need to import actual field data in the simulator. Traditional maps for features such as porosity, permeability and other reservoir properties can be imported in the usual manner, with properties averaged across grid areas. This averaging process is a little more complex for Voronoi polygons than simpler block forms, however, the formula does not add a great deal of computational complexity to the task.[22]

The real notable difference in the importation task comes with the inclusion of the fracture set. For the purposes of demonstration, a sample outcrop study will be used (Figure 31):

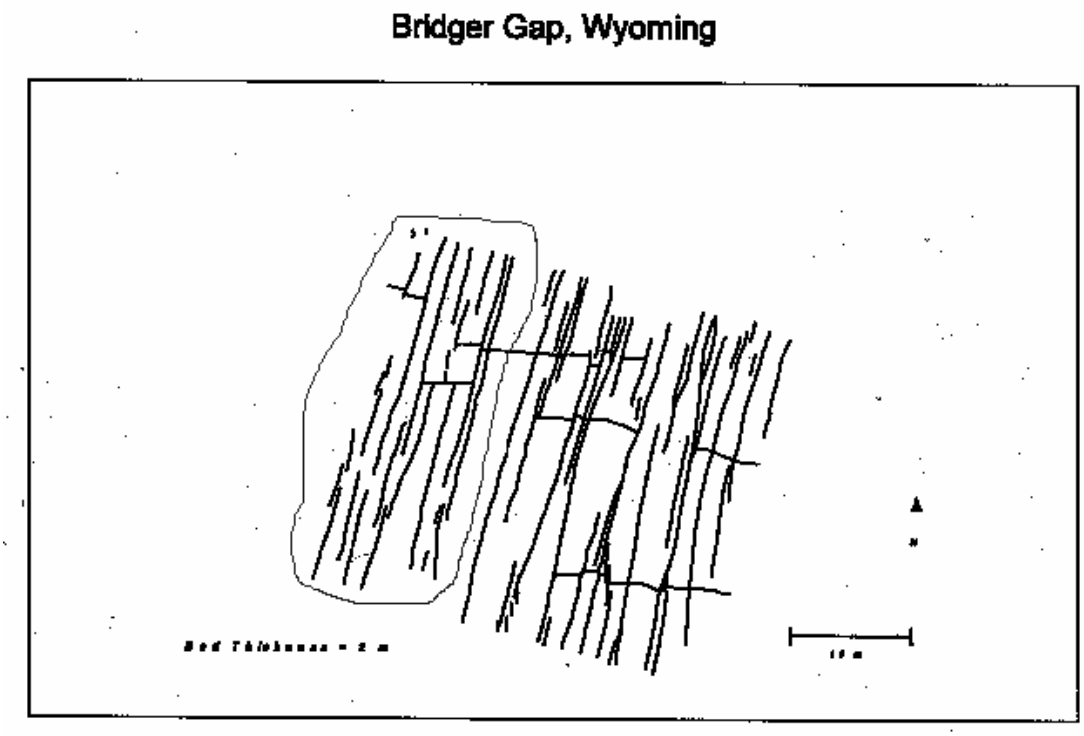


Fig 31 – An outcrop study, with the portion to be gridded circled [23]

For clarity, and due to the resolution limitations of printed medium, only the left section of the study was selected. The fracture or pseudofracture data to be analyzed must be imported in line form. This can involve either a vector data format, such as

SVG, a standard proposed by the World Wide Web Consortium,[24] in which case only interpretation of the format is needed, or it can involve more traditional raster graphics formats such a the commonly used JPEG and GIF standards. In the case of the later, it is necessary to use line detection methods to find the end points for each line; however, such techniques are beyond the scope of this paper. Commercial applications are available which will perform this conversion, allowing the matter to be left to external programs, although this does add an extra step for the user to import the data. For the purposes of expediency, this example was vectorized by hand.

The transformation of our above data sample to a vector form usable in the simulator is presented here (Figure 32):

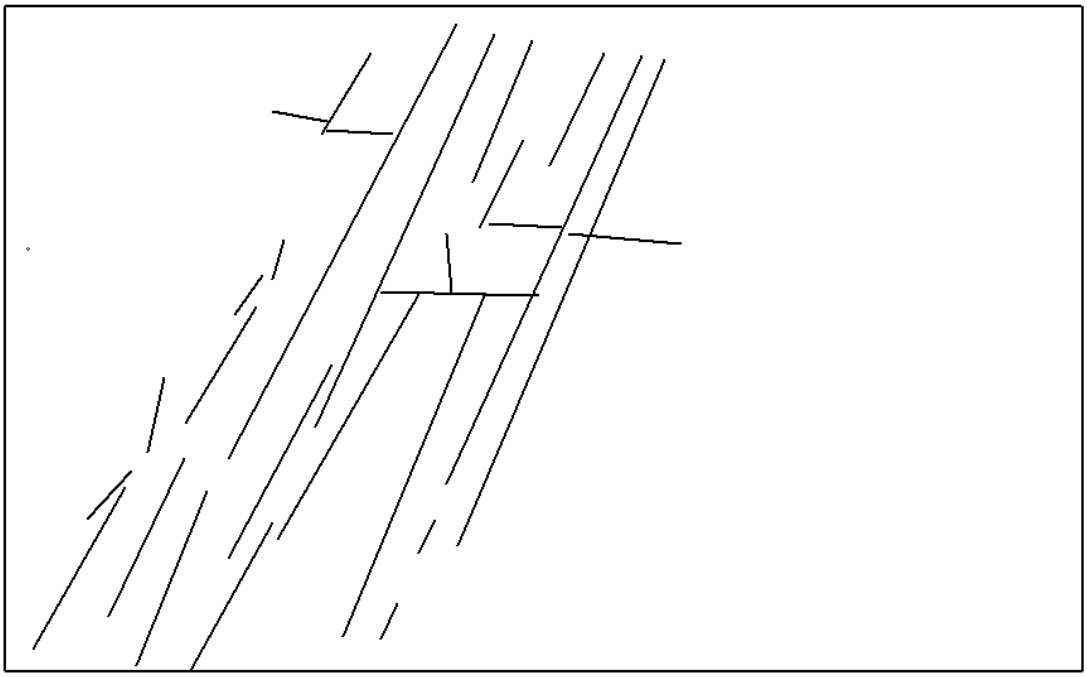


Fig 32 – The fractures in vector form, as imported into the simulator

In a real simulation run, the outlying areas away from the fractures would be gridded so as to prevent huge blocks. A single point was placed on the left hand side to demonstrate this, but otherwise, the only points placed are to preserve fracture boundaries. All the aforementioned techniques for aligning the Voronoi grid are used

here, with the exception of the low angle intersection technique, which doesn't see application due to the nature of the fracture pattern, which sees mainly aligned fractures intersected by perpendicular fractures, but no low angle intersections. This results in the finished grid seen below (Figure 33):

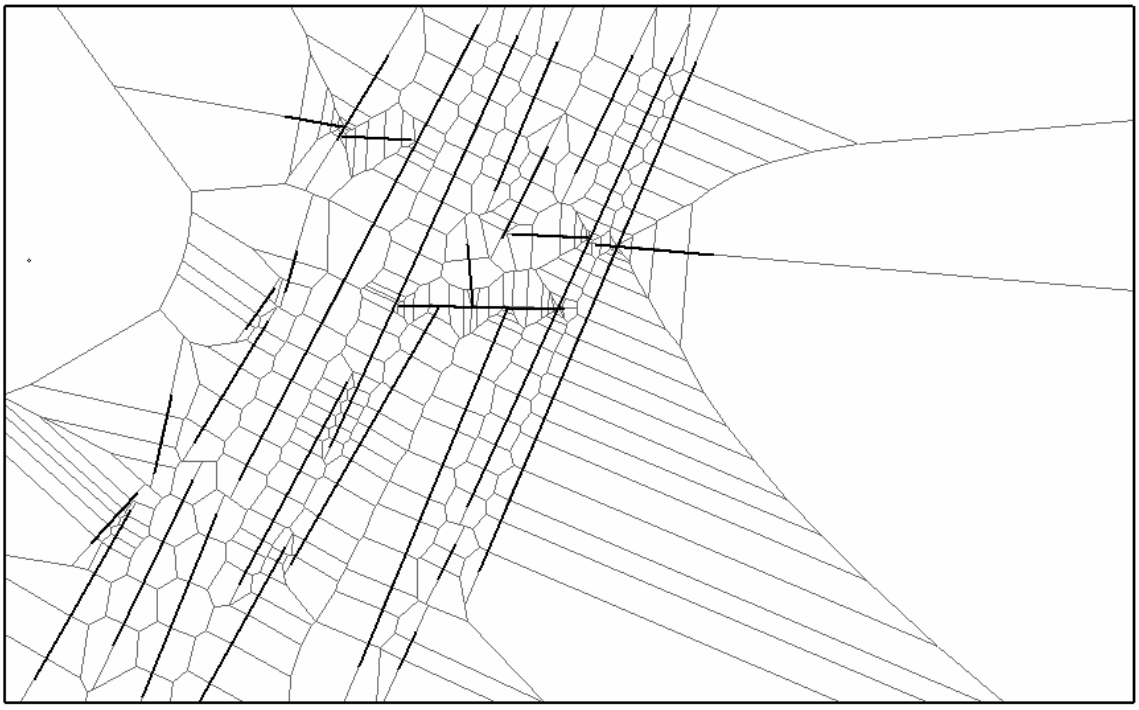


Fig 33 – The gridded version of the outcrop study

The gridding around the fractures is auto-generated from the points and fractures supplied. This leaves the actual grid itself as unalterable directly by the user, although it can be rebuilt after the addition of user-placed points. This is a requirement to maintain true Voronoi polygons so the assumptions made by our flow equations continue to hold.

In the areas beyond the fractures any traditional block pattern could be used, and would be desirable to lower the number of blocks with large numbers of sides, and thus reduce matrix size. This is also easily automated, allowing for the generation of a complete grid with only a suggested block size, and any user-desired touch-ups around areas of interest.

SUMMARY

The techniques discussed in the Methodology section outlines what is required to build a properly-aligned Voronoi diagram so that DFN techniques can be used. An outline of the full procedure to be utilized, from start to finish, is as follows.

Once a fracture set has been imported or created as a set of lines, it is necessary to identify intersections and break the lines into segments. This will facilitate the handling of intersections as well as the use of points to delineate each fracture. This is necessary because many of the techniques mentioned earlier assume we are dealing with line segments that intersect only at the end points.

To break each line into a set of line segments, Equations 1 through 3 can detect an intersection between any two lines. A pair of nested loops can identify all possible intersections, in the following form:

```
For LoopVariableA = 1 to (NumberOfLines - 1)
        For LoopVariableB = LoopVariableA to NumberofLines
                Call Findintersection()
        Next LoopVariableB
Next LoopVariableA
```

This yields an $O(n^2)$ time for the operation, which is a one-time cost. Each line should be split at each intersection, dividing into two line segments, each with one set of the previous lines end points and sharing the common intersection point as their new endpoints. Intersections that are in fact the end points of the line segments in question should be ignored for the purposes of splitting, as they indicate a successful splitting that has already taken place, or a situation in which splitting was not necessary to begin with.

To avoid the issue of splitting a line incorrectly, all the lines should be discretized into line segments to begin with, and only the current line segment used for comparisons, rather than the original line. This implies the *NumberofLines* variable used

above actually represents the number of current line segments, not the number of lines we begin with. This complicates the formulation slightly, as *NumberOfLines* will not be constant for the duration of the loops above. Intersections should be cached at this stage for use in the intersection handling routines.

At this point, every line segment should be assigned its bracketing points, which can either be only two at each end if the earlier method to use minimal points is be employed, or a set distance or number of points per line can be used. The routine will consist of going a set distance down the line for each set of points, and using the technique mentioned earlier to place one on each side.

Once the bracketing points have been placed, it will be time to handle intersections. The distance at which to set intersection points should be determined, generally with a set value reflective of the desired gridding resolution for the project as whole. However, intersections in close proximity will require precautions to assure the added points for each don't overlap or interfere. Thus, each intersection should be checked against each other one using a simple distance comparison, with a loop structure similar to the one outlined for identifying line segments, to assure all possible cases are compared. If two of the intersections are closer than the desired value, a value of slightly less than half the distance between them should be used for each.

Reducing the lines to line segments that only intersect at end points allows us to quickly determine the exact number of line segments (as well as which line segments) involved in an intersection by comparison of the end points. Thus, a quick linear trip through the list of line segments will yield the number and identities of all line segments involved and required for the use of intersection techniques.

The intersection technique used should default to the first technique (two points per line segment) discussed, with a check for the applicability of the second technique which uses fewer grid blocks. The two basic tests for this consist of verifying that at least two line segments involved in the intersection possess the same slope, and that all angles involved are less than 90 degrees. Additionally, it may be desirable, in the case

of only two line segments forming an acute angle, to use the simple version of the second technique, which would require a simple line segment check and an angle check.

Before any points are placed to form the intersection, all points within the radius where we are to place our points need to be removed, whether they were placed there by the user or via the bracketing technique. If arrays have been used for tracking the points, it may be convenient to remove all points for all intersections, and then reorder the array to avoid the cost of deleting points at each operation.

Once intersection points have been placed for all intersections, it is time to verify the grid, to assure that no points are interfering with the line boundaries. Technically, this technique need not be applied until the finalization process, when the user decides to save or utilize the finished grid. However, as a matter of good practice, it should be applied immediately, so that the user has an accurate picture of what the grid will look like.

The addition of fractures after the original grid operation has been performed will force at least a partial repetition of the task above, especially if the new fractures intersect old and previously gridded fractures. Validation of user added points can also be delayed, however, it will probably be necessary for the user to see if his newly-added point has caused the addition of several more points along fractures to maintain grid integrity. The user must also be prohibited from adding a point closer to a fracture than the distance used for offsetting the bracketing points from the fracture line, as it would be impossible for the verification method to guarantee the grid boundaries' alignment in that case. If the user is to be permitted to carry out such an action, it will be necessary to re-grid at least the fracture he is distorting, utilizing the new offset distance.

Similar restrictions apply to the user's ability to add points near to an intersection. In that case, the existing intersection points must be removed and the intersection redone with the tighter radius. The relatively large amount of verification required for each user-added point coupled with the restrictions on adding points in areas of interest means that the use of this technique favors automatically generated grids. Rather than bother the user with the frustration of attempting to add a single point and grid block only to

find the verification technique adds several, the automatic generation of the grid even in areas away from the fracture is desirable. A completely or near completely automatic grid generation procedure also allows for the reduction of Voronoi blocks with excessive numbers of sides, which could cause problems with the matrix solver, and removes a great deal of complication from the concern of the engineer using it.

CONCLUSION

The techniques described in this thesis allow for the construction of a set of grid blocks ready for simulation using either a black-oil or compositional simulator, with the standard flow orthogonal flow equations. From a simulation standpoint, once the grid is built, the only real adaptation needed to the equations used for conventional grids is the adjusted matrix solver.

Any standard Voronoi generation technique can be used, as only the position and number of grid block points is altered. The techniques themselves require only relatively simple techniques of computational geometry and allow for the production of a grid that helps to maximize accuracy and minimize simulation time. While building a dynamic grid can be more computationally intensive to build than a standard grid, grid building is by nature a much faster operation than the numeric simulations that follow, which may use the same grid for countless runs of the actual simulation.

The DFN technique is still in its infancy, due to the relatively recent introduction of better fracture data sources as well as increasing computational power. Given the difficulty of simulating individual fractures for fracture intensive reservoirs with little anisotropy, it is likely that variations of the Dual Porosity model will be used even if DFN is fully adopted. Indeed, there's no reason that both techniques can not be used, with only the major fractures being modeled, or specific flow affecting fracture groups being modeled as discrete pseudofractures.

Experimental confirmation of the DFN idea is already underway using the CT scanner to analyze flow through a core with a single discrete fracture. Once experimental results produce confirmation of the technique and an increased understanding of fractured flow, the use of DFN will expand to larger studies, and in time, take its place in the numerical simulation field as new tool for increasing simulation accuracy. Fractured reservoirs have long been a source of uncertainty when it comes to forecasting future production, and this technique, coupled with improved

imaging of fractures and analysis thereof will help reduce that uncertainty, to the benefit of operators of many fractured reservoirs.

The Voronoi grid provides an improved platform over the Delaunay tessellation, with low block count and greater versatility for aligning itself with directional flow. While adoption of Voronoi polygons for simulation has been slow due to the rather fragmentary nature of the technique's adoption by the sciences, new advances in both the computational formulation and sparse matrix solvers and matrix reducers can now bring this technique into the repertoire of the reservoir engineer.

REFERENCES

1. Kazemi, H., Merrill, L.S. Jr., Porterfield, K.L., and Zeman, P.R.: "Numerical Simulation of Water-Oil Flow in Naturally Fractured Reservoirs," paper SPE 5719 presented at the SPE-AIME Fourth Symposium on Numerical Simulation of Reservoir Performance, Los Angeles, California, February 19-20, 1976.

2. Gilman, James R., and Kazemi, Hossein: "Improvements in Simulation of Naturally Fractured Reservoirs," paper SPE 10511 presented at the 1982 SPE Reservoir Simulation Symposium in New Orleans, Louisiana, January 31 – February 3.

3. Yamamoto, Roy H., Ford, Wayne T., and Boubeguira, Ahmed: "Compositional Reservoir Simulator for Fissured Systems—The Single-Block Model," paper SPE 2666 presented at the SPE 44th Annual Fall Meeting, Denver, Colorado, September 28th – October 1st, 1969.

4. Karimi-Fard, M., and Firazoobadi, A.: "Numerical Simulation of Water Injection in 2D Fractured Media Using Discrete-Fracture Model," paper SPE 71615 presented at the 2001 SPE Annual Technical Conference and Exhibition in New Orleans, Louisiana, September 30th – October 3rd.

5. Karimi-Fard, M., Durlofsky, L.J., and Aziz, K.: "An Efficient Discrete Fracture Model Applicable for General Purpose Reservoir Simulators," paper SPE 79699 presented at the SPE Reservoir Simulation Symposium in Houston, Texas, February 3rd – 5th 2003.

6. Quandalle, Philippe: "Eighth SPE Comparative Solution Project: Gridding Techniques in Reservoir Simulation," paper SPE 25263 presented at the 12th SPE Symposium on Reservoir Simulation in New Orleans, Louisiana, February 28th – March 3rd, 1993.

7. Schechter, D.S., Putra, E., Baker, R.O., Knight, W.H., McDonald, W.P. *et al.*: "CO2 Pilot Design and Water Injection Performance in the Naturally Fractured Spraberry Trend Area, West Texas," paper SPE 71605 presented at the SPE Annual Technical Conference and Exhibition, 30 September-3 October 2001, in New Orleans, Louisiana.

8. Weisstein, Eric W., "Voronoi Diagram," http://mathworld.wolfram.com/VoronoiDiagram.html, 1999.

9. Wang, C., and Schubert L.: "An Optimal Algorithm for Constructing the Delaunay Triangulation of a Set of Line Segments," *Proc.*, Third Annual Symposium on Computational Geometry, Waterloo, Ontario, Canada (1987) 223 – 232.

10. Wang, C.A., and Chin, F.: "Finding the Constrained Delaunay Triangulation and Constrained Voronoi Diagram of a Simple Polygon in Linear-Time," *Proc.*, Algorithms – ESA Third Annual European Symposium (1995) 280 – 294.

11. Fortune, Steve: "Voronoi Diagrams and Delaunay Triangulations," *Computing in Euclidean Geometry*, D. Z. Du, F.Hwang (eds.), World Scientific Publ., 1992, 193-223.

12. Hoff, Kenneth E. III, Culver, Tim, Keyser, John, Lin, Ming, and Manocha Dinesh.: "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware," presented at SIGGRAPH '99 in Los Angeles, California August 8^{th} – 13^{th}, 1999.

13. Shah, P.C.: "A Nine-Point Finite Difference Operator for Reduction of the Grid Orientation Effect," paper SPE 12251 presented at the Reservoir Simulation Symposium in San Francisco, California November 15^{th} – 18^{th}, 1983.

14. Sharpe, H.N., and Anderson, D.A.: "A New Adaptive Orthogonal Grid Generation Procedure for Reservoir Simulation," paper SPE 20744 presented at the 65^{th} Annual Technical Conference and Exhibition of SPE in New Orleans, Louisiana, September 23^{rd} – 26^{th}, 1990.

15. Leventhal, S.H., Klein, M.H., and Culham, W.E.: "Curvilinear Coordinate Systems for Reservoir Simulation," paper SPE 11253 presented at the 1982 SPE Annual Technical Conference and Exhibition in New Orleans, Louisiana, September 26th – 29th.

16. Chong, Emeline E.: "Development of a 2-D Black-Oil Reservoir Simulator Using a Unique Grid-Block System," Master's Thesis, Texas A&M University, (2004). (Unpublished.)

17. Ding, Y., Lemonnier, P.: "Use of Corner Point Geometry in Reservoir Simulation," paper SPE 29933 presented at the International Meeting on Petroleum Engineering in Beijing, PR China, November 14th – 17th, 1995.

18. Okabe, Atsuyuki, Boots, Barry, Sugihara, Kokichi, and Nok Chu, Sung: *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd Ed*, Chichester, England, 2000.

19. El Gindy, H.A. and D. Avis: "A Linear Algorithm for Computing the Visibility Polygon from a Point," *Journal of Algorithms*, (1981), **2**, 186-197.

20. Dijksta, E.: "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, **1**, 269-271.

21. Bourke, Paul: "Intersection Points of Two Lines," astronomy.swin.edu.au/~pbourke/geometry/lineline2d/, April 1989.

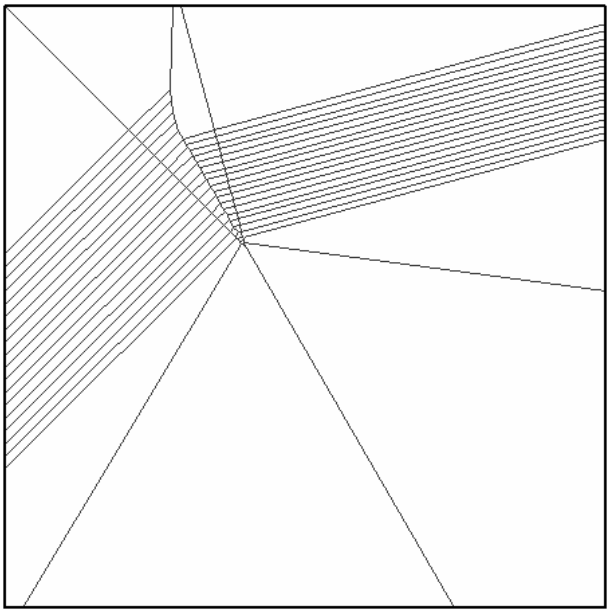
22. Byers, John A.: "Correct Calculation of Dirichlet Polygon Areas," *Journal of Animal Ecology*, (1996), **65**, 528-529.

23. "Harstad, H., "Characterization and Simulation of Naturally Fractured Frontier Sandstone, Green River Basin, Wyoming," Master's Thesis, New Mexico Institute of Mining and Technology, 1995.

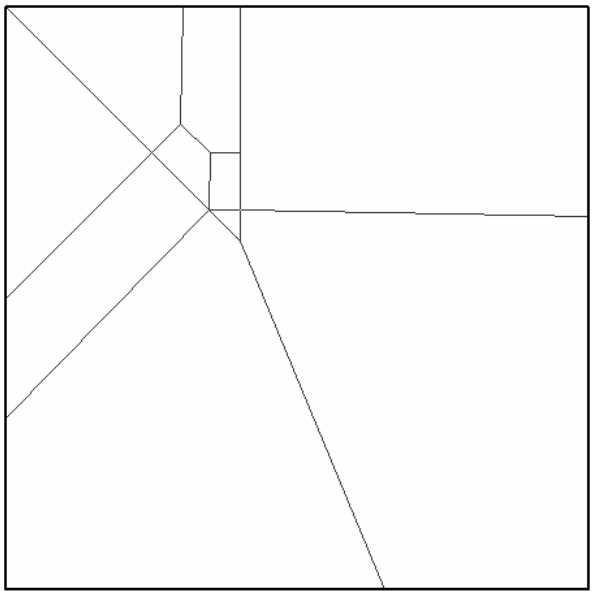
24. Ferraiolo, Jon, Fujisawa, Jun, and Jackson Dean: "Scalable Vector Graphics (SVG) 1.1 Specification," http://www.w3.org/TR/SVG11/, January 2003.

APPENDIX I

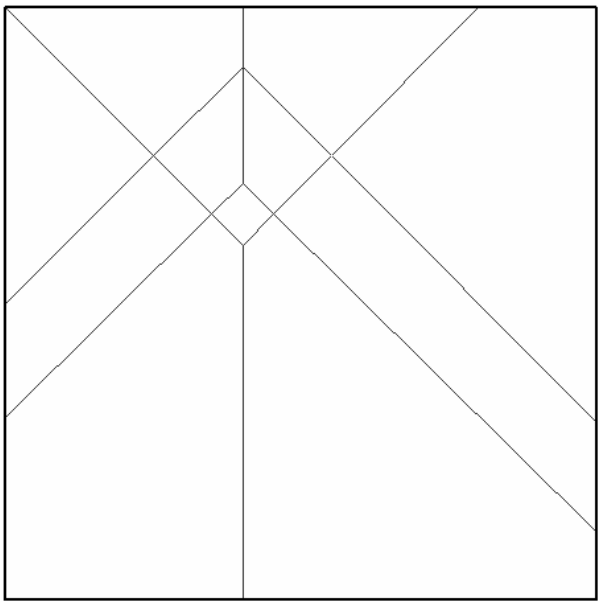
This appendix presents a demonstration of each technique at various angles, to show that computational methods to produce the desired results. All grids were produced using the SmartSim gridding program, parts of which are provided in code in Appendix II.



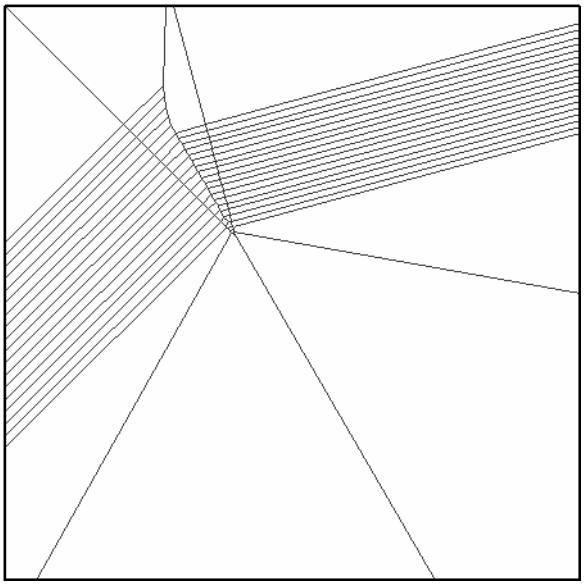
Two fractures intersecting at 30^0, single point technique



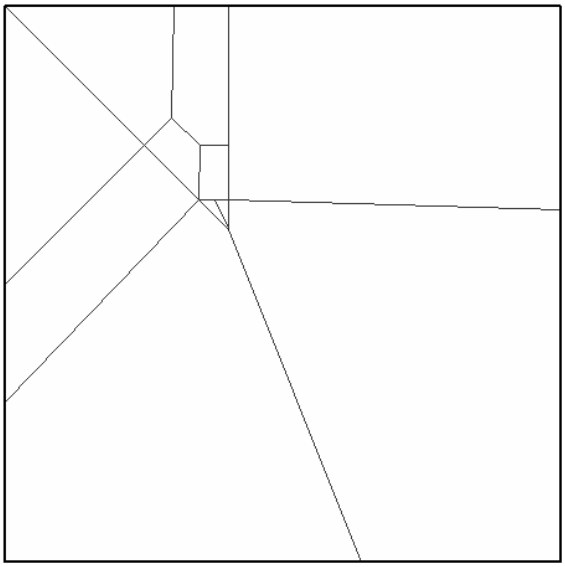
Two fractures intersecting at 45^0, single point technique



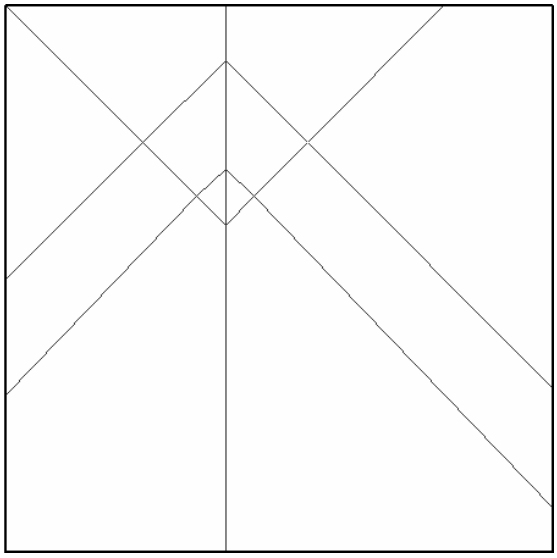
Two fractures intersecting at 90^0, single point technique



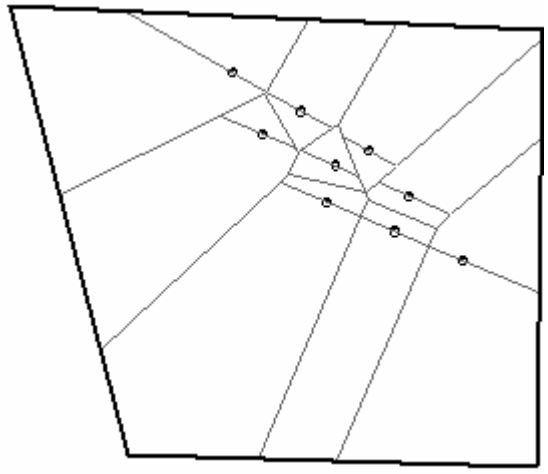
Two fractures intersecting at 30^0, multi-point technique



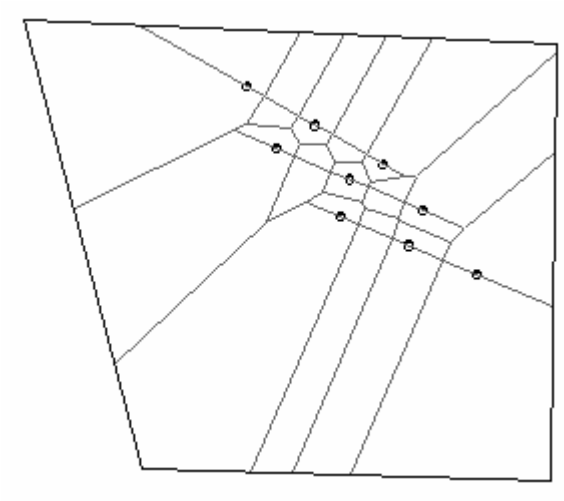
Two intersecting fractures at 45^0, multi-point technique



Two intersecting fractures at 90^0, multi-point technique



Lines in close proximity, without the use of the validation technique



The same fractures, after use of the validation technique

APPENDIX II

The code which performs the primary gridding task is broken into three subroutines. The primary routine, Intersections(), parses the fractures into line segments, determines intersections, and then passes intersection control to BuildMultiCircle(), which then passes it further to LowAngleCircle() if a two line intersection with a small angle is detected.

Once control has resumed upon return from the circle drawing procedures, Intersections() performs grid validation and a quick check to assure no duplicate points have been placed before returning.

```
Sub Intersections()
'This subroutine handles all the automatic grid generation work with the exception of the actual
'drawing of voronoi polygons
Dim A, B, C, F As Integer, G As Integer, H As Integer, i As Integer, j As Long
Dim D, E, RunningX As Double, RunningY As Double, PointsEachSide As Integer, AddedON As Integer
Dim TempS As String, Inversion As Boolean
Dim TempX As Double, TempY As Double, OSlope As Double, FinalX As Double, FinalY As Double
Dim SideArr(4, 100) As Integer 'Stores indexes into our point arrays for each side
Dim SideInn(4) As Integer 'Index

PointsEachSide = 3 'essentially a constant

'First, find all the intersections

C = 0
For A = 0 To (FLine - 2)
   For B = A To (FLine - 1)
      Call IntersectPoint(A, B, RunningX, RunningY)
   'Now we need to install modification for dealing with intersections
   If RunningX <> -65535 Then
      G = 0
      For F = 1 To C
         If (Abs(InterArr(F, 1) - RunningX) < 0.001) And (Abs(InterArr(F, 2) - RunningY) < 0.001) Then G = 1
      Next F
      If G = 0 Then 'Now we're sure it's not a dupe.
         C = C + 1
         InterArr(C, 1) = RunningX: InterArr(C, 2) = RunningY
         InterArr(C, 3) = A: InterArr(C, 4) = B
      End If
   End If
   If C >= 1000 Then 'Too many intersections
      MsgBox ("Too many intersections (> 1000)")
      B = FLine - 1 'End the loops
      A = FLine - 2
   End If
   Next B
Next A

'Now, break the lines down into line segments

B = 0
```

```
For A = 0 To (FLine - 1)
   'Don't add lines that aren't real
   If (Form1.FracLine(A).x1 <> Form1.FracLine(A).x2) Or (Form1.FracLine(A).y1 <> Form1.FracLine(A).y2) Then
      B = B + 1
      LSegment(B, 1) = Form1.FracLine(A).x1: LSegment(B, 2) = Form1.FracLine(A).y1
      LSegment(B, 3) = Form1.FracLine(A).x2: LSegment(B, 4) = Form1.FracLine(A).y2
      'LSegment(B, 5) = A 'Parent Line
   End If
Next A

'All lines loaded, now start splitting them.

'Note, this assumes there are no duped entries in Interarr
'If there are, we could end up with cloned line segments.

For A = 1 To C 'Examine each intersection
         H = B 'B will be altered
         For G = 1 To H
            D = Point2LineB(InterArr(A, 1), InterArr(A, 2), LSegment(G, 1), LSegment(G, 2), LSegment(G, 3), LSegment(G, 4))
            If (D < 0.00001) And (CheckLine(G, InterArr(A, 1), InterArr(A, 2)) = False) Then
                  B = B + 1
                  LSegment(B, 1) = LSegment(G, 1): LSegment(B, 2) = LSegment(G, 2)
                  LSegment(B, 3) = InterArr(A, 1): LSegment(B, 4) = InterArr(A, 2)
                  'Now, alter the original entry.
                  LSegment(G, 1) = InterArr(A, 1): LSegment(G, 2) = InterArr(A, 2)
                  'Leave the x2 and y2 in place.
               End If
         Next G
Next A

LSNum = B

Thickness = 0.06
CurrentLine = 0 'Start Up

PointCount = LSNum * PointsEachSide * 2

'First pass done, number in PointCount

OldNNN = NNN
j = NNN + PointCount 'Allocate space for new points
If j > ArraySize Then Call ReAllocArray(j)

NNN = j

CurrentLine = 1

While (CurrentLine <= LSNum)
   OrigThickness = dis1(LSegment(CurrentLine, 1), LSegment(CurrentLine, 2), LSegment(CurrentLine, 3), LSegment(CurrentLine,
4))
   spacing = OrigThickness / (PointsEachSide - 1) 'Each line should have 6 points on each side
   'Evaluate all line segments
   If (LSegment(CurrentLine, 1) <> LSegment(CurrentLine, 3)) Or (LSegment(CurrentLine, 2) <> LSegment(CurrentLine, 4)) Then
      'It's not a point, rather than a line
      LLength = OrigThickness
      AmountDone = 0
      If LLength >= spacing Then
         'At least one set of points to place.
         OpNumber = Round(LLength / spacing, 0) + 1
         For LoopVar = 1 To OpNumber
            Distance = spacing
            Call LineOperation(LSegment(CurrentLine, 1), LSegment(CurrentLine, 2), LSegment(CurrentLine, 3),
LSegment(CurrentLine, 4), AmountDone, TempX, TempY)
            'TempX:TempY holds a point, spaced on the fracture, but we want it perpendicular.
            If Abs(LSegment(CurrentLine, 1) - LSegment(CurrentLine, 3)) < 0.001 Then
                  OSlope = -65535
```

```
                Else
                    If Abs(LSegment(CurrentLine, 2) - LSegment(CurrentLine, 4)) < 0.1 Then
                        OSlope = 0
                        Else
                            OSlope = (LSegment(CurrentLine, 4) - LSegment(CurrentLine, 2)) / (LSegment(CurrentLine, 3) -
LSegment(CurrentLine, 1))
                    End If
                End If
            Call FinalPoint(TempX, TempY, Thickness, OSlope, FinalX, FinalY, False)
            'Now we have the FinalX:FinalY of our new point, but is it valid?
                aX(OldNNN) = FinalX: aY(OldNNN) = FinalY
                Xorg(OldNNN) = FinalX: Yorg(OldNNN) = FinalY
                ad(OldNNN) = dis1(0, 0, FinalX, FinalY)
                AO(OldNNN, 1) = CurrentLine: AO(OldNNN, 2) = 0
                Form1.Picture1.Circle (aX(OldNNN), aY(OldNNN)), 0.2, RGB(255, 255, 255)
                OldNNN = OldNNN + 1

            'Second point on other side
            Call FinalPoint(TempX, TempY, Thickness, OSlope, FinalX, FinalY, True)
                aX(OldNNN) = FinalX: aY(OldNNN) = FinalY
                Xorg(OldNNN) = FinalX: Yorg(OldNNN) = FinalY
                ad(OldNNN) = dis1(0, 0, FinalX, FinalY)
                AO(OldNNN, 1) = CurrentLine * -1 'neg number, to show it is on the other side
                AO(OldNNN, 2) = 0

                Form1.Picture1.Circle (aX(OldNNN), aY(OldNNN)), 0.2, RGB(255, 255, 255)
                OldNNN = OldNNN + 1

            AmountDone = AmountDone + spacing
            If AmountDone > LLength Then AmountDone = LLength
        Next 'Done with this segment
    End If
  End If
  CurrentLine = CurrentLine + 1 'Do for all segments of the fracture
Wend 'End of While

'MsgBox (C)

D = 10000
For F = 1 To (C - 1)
  For E = (F + 1) To C
    H = dis1(InterArr(F, 1), InterArr(F, 2), InterArr(E, 1), InterArr(E, 2))
  If (H < D) And (H > 0) Then D = H
  Next E
Next F

'Technically, we should make sure this isn't beyond the length of any line segment
If (D = 0) Or (D > 6) Then D = 6
For F = 1 To C
  Call BuildMultiCircle(InterArr(F, 1), InterArr(F, 2), InterArr(F, 3), InterArr(F, 4), D) 'Evaluate all intersection points
Next F

AddedON = 1 'This is the validation trigger, and is enabled, at the moment.

While ((AddedON > 0) And (LSNum > 1))

AddedON = 0
'Now, we need to verify that the spacing is correct

For A = 1 To 100
  SideArr(1, A) = 0: SideArr(2, A) = 0: SideArr(3, A) = 0: SideArr(4, A) = 0
Next A

For A = 1 To LSNum
  'Evaluate every fracture
  SideInn(1) = 0: SideInn(2) = 0
```

```
For B = 1 To LSNum '(A + 1) To LSNum
   If (A <> B) Then

   'Build SideArr
   For F = 1 To 4
      SideInn(F) = 0
   Next F

   For C = 0 To (NNN - 1)
      D = 0
      If (AO(C, 1) = A) Then D = 1
      If (AO(C, 1) = A * -1) Then D = 2
      If (AO(C, 1) = B) Then D = 3
      If (AO(C, 1) = B * -1) Then D = 4
      If D > 0 Then
         SideInn(D) = SideInn(D) + 1
         SideArr(D, SideInn(D)) = C
      End If

      D = 0
      If (AO(C, 2) = A) Then D = 1
      If (AO(C, 2) = A * -1) Then D = 2
      If (AO(C, 2) = B) Then D = 3
      If (AO(C, 2) = B * -1) Then D = 4
      If AO(C, 1) = AO(C, 2) Then D = 0 'Do not make duplicates!
      If D > 0 Then
         SideInn(D) = SideInn(D) + 1
         SideArr(D, SideInn(D)) = C
      End If
   Next C

   'Now we have all the relevant points for the other line
   'Here we take advantage of the fact that points were added in order
   'Thus, the first and second points on each side are adjacent in the array

   RunningX = dis1(aX(SideArr(1, 1)), aY(SideArr(1, 1)), aX(SideArr(3, 1)), aY(SideArr(3, 1)))
   RunningY = dis1(aX(SideArr(2, 1)), aY(SideArr(2, 1)), aX(SideArr(3, 1)), aY(SideArr(3, 1)))
   If (RunningX < RunningY) Then G = 1 Else G = 2 'Which side we'll be using
   RunningX = dis1(aX(SideArr(3, 1)), aY(SideArr(3, 1)), aX(SideArr(G, 1)), aY(SideArr(G, 1)))
   RunningY = dis1(aX(SideArr(4, 1)), aY(SideArr(4, 1)), aX(SideArr(G, 1)), aY(SideArr(G, 1)))
   If (RunningX < RunningY) Then H = 3 Else H = 4 'The same
   Call SortPoints(SideInn(), SideArr()) 'Sort the points
   For C = 1 To SideInn(G)
      'For each point on one of our lines
      For D = 1 To (SideInn(H) - 1) 'Note: This technique assumes points are in order
         'For each point on the opposing line
         'Slope between the two points on the line
         'RunningX:RunningY will be one endpoint, FinalX:FinalY the other
         OSlope = 1
         If aY(SideArr(H, D)) - aY(SideArr(H, D + 1)) = 0 Then
            'Horz line, perpendicular will be vertical
            RunningX = aX(SideArr(G, C)): FinalX = aX(SideArr(G, C))
            RunningY = aY(SideArr(G, C)) + 10: FinalY = aY(SideArr(G, C)) - 10
            OSlope = 0
         End If
         If aX(SideArr(H, D)) - aX(SideArr(H, D + 1)) = 0 Then
            'Vert line, perpendicular will be horizontal
            RunningY = aY(SideArr(G, C)): FinalY = aY(SideArr(G, C))
            RunningX = aX(SideArr(G, C)) + 10: FinalX = aX(SideArr(G, C)) - 10
            OSlope = 0
         End If
         If OSlope <> 0 Then
            RunningX = aX(SideArr(G, C)) + 10
            FinalX = aX(SideArr(G, C)) - 10
            OSlope = (aY(SideArr(H, D)) - aY(SideArr(H, D + 1))) / (aX(SideArr(H, D)) - aX(SideArr(H, D + 1)))
            'Perpendicular slope
```

```
            OSlope = -1 / OSlope
            'Now, we need to make a line through our 3rd point
            RunningY = OSlope * (RunningX - aX(SideArr(G, C))) + aY(SideArr(G, C))
            FinalY = OSlope * (FinalX - aX(SideArr(G, C))) + aY(SideArr(G, C))
         End If

         Call IntersectPointB(FinalX, FinalY, RunningX, RunningY, aX(SideArr(H, D)), aY(SideArr(H, D)), aX(SideArr(H, D +
1)), aY(SideArr(H, D + 1)), TempX, TempY)
         'Now TempX:TempY should hold the closet point on our line to the 3rd point
         'Distance to end point #1
         RunningX = dis1(aX(SideArr(G, C)), aY(SideArr(G, C)), aX(SideArr(H, D)), aY(SideArr(H, D)))
         'Distance to end point #2
         RunningY = dis1(aX(SideArr(G, C)), aY(SideArr(G, C)), aX(SideArr(H, D + 1)), aY(SideArr(H, D + 1)))
         'Distance to "Closest" point
         FinalX = dis1(aX(SideArr(G, C)), aY(SideArr(G, C)), TempX, TempY)
         If (TempX <> -65535) And (FinalX < RunningX) And (FinalX < RunningY) Then
            'There is an intersection
            'Now, we'll need three temp variables to evaluate if we have a problem or not.
            FinalX = dis1(aX(SideArr(G, C)), aY(SideArr(G, C)), TempX, TempY) 'From point to middle
            RunningY = dis1(aX(SideArr(H, D)), aY(SideArr(H, D)), TempX, TempY) 'From one side
            RunningX = dis1(aX(SideArr(H, D + 1)), aY(SideArr(H, D + 1)), TempX, TempY) 'The other
            If ((FinalX < RunningY) Or (FinalX < RunningX)) And (RunningX > 0.1) And (RunningY > 0.1) Then
               'We need to take corrective action
               TempS = Str(RunningX) + "," + Str(RunningY)
               TempX = (aX(SideArr(H, D + 1)) + aX(SideArr(H, D))) / 2
               TempY = (aY(SideArr(H, D + 1)) + aY(SideArr(H, D))) / 2
               AddOn = AddOn + 1
               If (NNN + 2) > ArraySize Then Call ReAllocArray(NNN + 2)

               NNN = NNN + 2
               'First, make space to insert them in the proper spot in the main array.
               'Make sure we don't insert it after an intersection point, if possible
               If SideArr(H, D) > SideArr(H, D + 1) Then j = SideArr(H, D + 1) Else j = SideArr(H, D)
               'J holds the point we need to insert afterwards of

               i = NNN 'Our array, plus two slots at the end

               While (i >= (j + 2))
                  AO(i, 1) = AO(i - 2, 1): AO(i, 2) = AO(i - 2, 2)
                  ad(i) = ad(i - 2)
                  aX(i) = aX(i - 2): aY(i) = aY(i - 2)
                  Xorg(i) = Xorg(i - 2): Yorg(i) = Yorg(i - 2)
                  i = i - 1
               Wend

               'i = j + 1 'The spot after our last good point
               If H = 3 Then
                  i = j + 1
                  AO(i, 1) = B
               Else
                  i = j + 2
                  AO(i, 1) = B * -1
               End If
               aX(i) = TempX: aY(i) = TempY
               AO(i, 2) = 0
               ad(i) = dis1(0, 0, TempX, TempY)
               'Now we need the other line, we still have the perpendicular slope
               'This next trick can be done due to the ordering of the points
               If H = 3 Then Inversion = True Else Inversion = False
               Call FinalPoint(TempX, TempY, Thickness * 2, (-1 / OSlope), RunningX, RunningY, Inversion)

               'i = i + 1

               'Form1.Picture1.Circle (aX(i), aY(i)), 0.4, RGB(155, 50, 155)
               If H = 3 Then
                  i = j + 2
```

```
                AO(i, 1) = B * -1
              Else
                i = j + 1
                AO(i, 1) = B
              End If
              aX(i) = RunningX: aY(i) = RunningY
              AO(i, 2) = 0
              ad(i) = dis1(0, 0, RunningX, RunningY)
              'Now we need to rebuild our side arrays

              For TempX = 1 To 4
                SideInn(TempX) = 0
              Next TempX

              For TempX = 0 To (NNN - 1)
                F = 0
                If (AO(TempX, 1) = A) Then F = 1
                If (AO(TempX, 1) = A * -1) Then F = 2
                If (AO(TempX, 1) = B) Then F = 3
                If (AO(TempX, 1) = B * -1) Then F = 4
                If F > 0 Then
                  SideInn(F) = SideInn(F) + 1
                  SideArr(F, SideInn(F)) = TempX
                End If

                F = 0
                If (AO(TempX, 2) = A) Then F = 1
                If (AO(TempX, 2) = A * -1) Then F = 2
                If (AO(TempX, 2) = B) Then F = 3
                If (AO(TempX, 2) = B * -1) Then F = 4
                If AO(TempX, 1) = AO(TempX, 2) Then F = 0 'Do not make duplicates!
                If F > 0 Then
                  SideInn(F) = SideInn(F) + 1
                  SideArr(F, SideInn(F)) = TempX
                End If
              Next TempX

              Call SortPoints(SideInn(), SideArr())

              AddedON = 1
              'We only care about the point added on our side of interest
              'Note: Because of the way we made our new point the next point,
              'We'll look at it again to make sure it doesn't need further correction
            End If
          End If 'There was an intersection
        Next D
      Next C
    End If
    Next B
Next A

Wend 'Done with verification

For F = 0 To FLine - 1
  Form1.FracLine(F).Visible = False 'Hide it so we can see grid results
Next F


'Now, graph up the intersections

B = NNN
A = DuplicatePoints
TempS = "Old Points: " + Str(B) + " New Points: " + Str(NNN) + " Dupes: " + Str(A)
MsgBox (TempS) 'Duplicate points should only be present in the event of errors in gridding.

End Sub
```

```
Sub BuildMultiCircle(X As Double, y As Double, LineA As Double, LineB As Double, ByVal CInter As Double)

'This builds a circle of points around an intersection.
'LineA and LineB are only used if just 2 line segments intersect

Dim MinimumD  As Double, Thickness As Double, A As Integer, LCount As Integer, B As Double, C As Integer, D As Integer, E As Long
Dim Lines(20) As Integer  'As many as 20 lines allowed through one point
Dim Skip As Boolean, TempMax As MAXMIN, NewX As Double, NewY As Double, FinalX As Double, FinalY As Double, OSlope As Double, TempX As Double, TempY As Double
Dim TS As String


Thickness = 0.1: Skip = False
LCount = 0 'Number of lines involved in intersection
MinimumD = (CInter / 2.1) / 2 'To assure neighboring intersection circles do not overlap

'Detect true number of lines involved
For A = 1 To LSNum
      If ((X = LSegment(A, 1)) And (y = LSegment(A, 2))) Or ((X = LSegment(A, 3)) And (y = LSegment(A, 4))) And (B = 0) Then
            LCount = LCount + 1
            Lines(LCount) = A
      End If
Next A

PointNum = NNN: D = NNN

If LCount = 2 Then
   'Send to low angle handler
   TempMax = AngleTRI(LSegment(Lines(1), 1), LSegment(Lines(1), 2), LSegment(Lines(1), 3), LSegment(Lines(1), 4), LSegment(Lines(2), 1), LSegment(Lines(2), 2), LSegment(Lines(2), 3), LSegment(Lines(2), 4))
   NewX = TempMax.MIN 'Now we have the smallest angle between the two segments
   If NewX <= 45 Then
      Call LowAngleCircle(Lines(1), Lines(2), LSegment(Lines(1), 1), LSegment(Lines(1), 2), LSegment(Lines(1), 3), LSegment(Lines(1), 4), LSegment(Lines(2), 1), LSegment(Lines(2), 2), LSegment(Lines(2), 3), LSegment(Lines(2), 4), X, y, MinimumD)
      Skip = True
   End If
End If

If Skip = False Then 'Alternative method

For A = 0 To (NNN - 1)
      B = dis1(X, y, aX(A), aY(A)) 'Distance to center point
      If (B < (CInter / 2)) Then
         aX(A) = -65535: aY(A) = -65535 'Mark for deletion
      End If
Next A

A = CleanArray() 'Delete points that were too close.

D = NNN '+1 or over?
E = NNN + (LCount * 2)

If (E > ArraySize) Then Call ReAllocArray(E)

NNN = E

'Make sure we don't go beyond the end of any portion

B = 10000
For A = 1 To LCount
      'Choose which end point is NOT on the intersection
      If (LSegment(Lines(A), 1) = X) And (LSegment(Lines(A), 2) = y) Then
         TempX = LSegment(Lines(A), 3): TempY = LSegment(Lines(A), 4)
      Else
```

```
        TempX = LSegment(Lines(A), 1): TempY = LSegment(Lines(A), 2)
      End If
      C = dis1(TempX, TempY, X, y)
      If (C < B) And (C > 0) Then B = C
Next A

If (B > MinimumD) Then B = MinimumD
If (B < Thickness) Then
   MsgBox ("Line Segment shorter than Thickness, will not render correctly.")
   B = MinimumD
End If

'MsgBox (B)

For A = 1 To LCount
   'Choose which end point is NOT on the intersection
   If (LSegment(Lines(A), 1) = X) And (LSegment(Lines(A), 2) = y) Then
      TempX = LSegment(Lines(A), 3): TempY = LSegment(Lines(A), 4)
      Else
         TempX = LSegment(Lines(A), 1): TempY = LSegment(Lines(A), 2)
   End If

   Call LineOperation(TempX, TempY, X, y, B, NewX, NewY)
   If Abs(X - TempX) < 0.001 Then
      OSlope = -65535
         Else
            If Abs(TempY - y) < 0.001 Then
               OSlope = 0
                  Else
                     OSlope = (y - TempY) / (X - TempX)
            End If
      End If
  Call FinalPoint(NewX, NewY, Thickness, OSlope, FinalX, FinalY, False)
  'Now we add the points
  aX(D) = FinalX: aY(D) = FinalY: Xorg(D) = FinalX: Yorg(D) = FinalY
  ad(D) = dis1(0, 0, aX(D), aY(D))
  AO(D, 1) = Lines(A): AO(D, 2) = Lines(A)
  D = D + 1

  Call FinalPoint(NewX, NewY, Thickness, OSlope, FinalX, FinalY, True)
  aX(D) = FinalX: aY(D) = FinalY: Xorg(D) = FinalX: Yorg(D) = FinalY
  ad(D) = dis1(0, 0, aX(D), aY(D))
  AO(D, 1) = Lines(A) * -1: AO(D, 2) = Lines(A) * -1
  D = D + 1
Next A
End If
```

End Sub

```
Sub LowAngleCircle(LA As Integer, LB As Integer, ax1 As Double, ay1 As Double, ax2 As Double, ay2 As Double, bx1 As Double, by1 As Double, bx2 As Double, by2 As Double, X As Double, y As Double, MinimumD As Double)
'Deals with kinks in the fracture that are < 45 degrees.

Dim cx1 As Double, cy1 As Double, cx2 As Double, cy2 As Double
Dim dx1 As Double, dy1 As Double, dx2 As Double, dy2 As Double
Dim SideArr(4, 100) As Integer 'Stores indexes into our point arrays for each side
Dim SideInn(4) As Integer 'Index

Dim E As Double, F As Double, A As Integer, TempX As Integer, G As Double, H As Double, TempY As Integer, i As Double, j As Double
Dim x1 As Double, y1 As Double, x2 As Double, y2 As Double
Dim Dist As Double, TempS As String, PointNum As Integer, RunningX As Double, RunningY As Double, FinalX As Integer

For TempX = 1 To 4
   SideInn(TempX) = 0
Next TempX

For TempX = 0 To (NNN - 1)
                TempY = 0
                If (AO(TempX, 1) = LA) And (AO(TempX, 2) = 0) Then TempY = 1
                If (AO(TempX, 1) = (LA * -1)) And (AO(TempX, 2) = 0) Then TempY = 2
                If (AO(TempX, 1) = LB) And (AO(TempX, 2) = 0) Then TempY = 3
                If (AO(TempX, 1) = (LB * -1)) And (AO(TempX, 2) = 0) Then TempY = 4
                If (TempY = 0) Then
                   If (AO(TempX, 2) = LA) Then TempY = 1
                   If (AO(TempX, 2) = (LA * -1)) Then TempY = 2
                   If (AO(TempX, 2) = LB) Then TempY = 3
                   If (AO(TempX, 2) = (LB * -1)) Then TempY = 4
                   'We need to handle intersection points (Those with Two AO() entries) specially.
                   'Thanks to our technique, we're guaranteed that all intersections points are
                   'higher on the array that fracture gridding points.
                   If TempY > 0 Then
                      'An intersection point
                      RunningX = dis1(aX(TempX), aY(TempX), aX(SideArr(TempY, 1)), aY(SideArr(TempY, 1)))
                      RunningY = dis1(aX(TempX), aY(TempX), aX(SideArr(TempY, SideInn(TempY))), aY(SideArr(TempY, SideInn(TempY))))
                      If (RunningX < RunningY) Then
                         'Put a front of list
                         SideInn(TempY) = SideInn(TempY) + 1
                         FinalX = SideInn(TempY)
                         While (FinalX > 0)
                            SideArr(TempY, FinalX) = SideArr(TempY, FinalX - 1)
                            FinalX = FinalX - 1
                         Wend
                         SideArr(TempY, 1) = TempX: TempY = 0
                         Else
                            SideInn(TempY) = SideInn(TempY) + 1
                            SideArr(TempY, SideInn(TempY)) = TempX: TempY = 0
                         End If
                      End If 'TempY > 0
                   End If 'TempY = 0
                If (TempY <> 0) Then
                   SideInn(TempY) = SideInn(TempY) + 1
                   SideArr(TempY, SideInn(TempY)) = TempX
                   End If 'TempY <> 0
                Next TempX 'Done loading all points for our fracture


   RunningX = dis1(aX(SideArr(1, 1)), aY(SideArr(1, 1)), aX(SideArr(3, 1)), aY(SideArr(3, 1)))
   RunningY = dis1(aX(SideArr(2, 1)), aY(SideArr(2, 1)), aX(SideArr(3, 1)), aY(SideArr(3, 1)))
   If (RunningX < RunningY) Then G = LA Else G = -1 * LA 'Which side we'll be using
   If G = LA Then i = 1 Else i = 2
   RunningX = dis1(aX(SideArr(3, 1)), aY(SideArr(3, 1)), aX(SideArr(i, 1)), aY(SideArr(i, 1)))
   RunningY = dis1(aX(SideArr(4, 1)), aY(SideArr(4, 1)), aX(SideArr(i, 1)), aY(SideArr(i, 1)))
   If (RunningX < RunningY) Then H = LB * -1 Else H = LB 'The same
```

```
Call SortPoints(SideInn(), SideArr())

x1 = ax1: y1 = ay1
x2 = bx1: y2 = by1
If (ax1 = X) And (ay1 = y) Then 'Chose the point besides the intersection
   x1 = ax2: y1 = ay2
End If

If (bx1 = X) And (by1 = y) Then 'Chose the point besides the intersection
   x2 = bx2: y2 = by2
End If

E = dis1(X, y, x1, y1)
Call LineOperation(x1, y1, X, y, MinimumD, cx1, cy1)
x1 = cx1: y1 = cy1
E = dis1(X, y, x2, y2)
Call LineOperation(x2, y2, X, y, MinimumD, dx1, dy1)
x2 = dx1: y2 = dy1

'So now we have two points, equidistant from the intersection.

If (ay2 - ay1) = 0 Then
   'Horizontal Line
   cy2 = cy1 + 100
   If cy1 > dy1 Then cy2 = cy1 - 100
   cx2 = cx1
   Else
      If (ax2 - ax1) = 0 Then
         'Vertical Line
         cx2 = cx1 + 100
         If cx1 > dx1 Then cx2 = cx1 - 100
         cy2 = cy1
            Else
            i = (ay2 - ay1) / (ax2 - ax1) 'Slope of first line
            i = -1 / i 'Perpendicular slope
            cx2 = cx1 + 100
            If cx1 > dx1 Then cx2 = cx1 - 100
            cy2 = i * (cx2 - cx1) + cy1
         End If
   End If

If (by2 - by1) = 0 Then
   'Horizontal Line
   dy2 = dy1 + 100
   If dy1 > cy1 Then dy2 = dy1 - 100
   dx2 = dx1
   Else
      If (bx2 - bx1) = 0 Then
         'Vertical Line
         dx2 = dx1 + 100
         If dx1 > cx1 Then dx2 = dx1 - 100
         dy2 = dy1
            Else
               j = (by2 - by1) / (bx2 - bx1) 'Slope of second line
               j = -1 / j 'Perpendicular slope
               dx2 = dx1 + 100
               If dx1 > cx1 Then dx2 = dx1 - 100
               dy2 = j * (dx2 - dx1) + dy1
            End If
      End If

'Now we have two perpendicular lines that should intersect, provided that our angles are acute.
Call IntersectPointB(cx1, cy1, cx2, cy2, dx1, dy1, dx2, dy2, E, F)

For A = 0 To (NNN - 1)
   cx1 = dis1(X, y, aX(A), aY(A)) 'Distance to center point
```

```
    If (cx1 < MinimumD) Then
      Form1.Picture1.Circle (aX(A), aY(A)), 0.2, RGB(0, 255, 0)
      aX(A) = -65535: aY(A) = -65535 'Mark for deletion
    End If
  Next A

  A = CleanArray()
  PointNum = NNN + 3

  If PointNum > ArraySize Then Call ReAllocArray(PointNum)

  'e,f holds the intersection point.
  'The intersection point will be our first point added.

  aX(NNN) = E: aY(NNN) = F: Xorg(NNN) = E: Yorg(NNN) = F: AO(NNN, 1) = G: AO(NNN, 2) = H

  cx1 = (x1 - E) 'Change in X, reversed direction
  cx1 = cx1 + x1 'Add it to the original X position
  cy1 = i * (cx1 - x1) + y1

 aX(NNN + 1) = cx1: aY(NNN + 1) = cy1: Xorg(NNN + 1) = cx1: Yorg(NNN + 1) = cy1: AO(NNN + 1, 1) = G * -1
  AO(NNN + 1, 2) = G * -1

  cx1 = (x2 - E) 'Change in X, reversed direction
  cx1 = cx1 + x2 'Add it to the original X position
  cy1 = j * (cx1 - x2) + y2

  aX(NNN + 2) = cx1: aY(NNN + 2) = cy1: Xorg(NNN + 2) = cx1: Yorg(NNN + 2) = cy1: AO(NNN + 2, 1) = H * -1
  AO(NNN + 2, 2) = H * -1

  Form1.Picture1.Circle (aX(NNN), aY(NNN)), 0.2, RGB(0, 0, 255)
  Form1.Picture1.Circle (aX(NNN + 1), aY(NNN + 1)), 0.2, RGB(0, 0, 255)
  Form1.Picture1.Circle (aX(NNN + 2), aY(NNN + 2)), 0.2, RGB(0, 0, 255)

  NNN = PointNum

End Sub
```

VITA

Name: Matthew Edward Gross

Address: 2253 Barrington Court
Auburn, Alabama, 36830

Email Address: mgross@tamu.edu

Education: B.S., Petroleum Engineering, Texas A&M University, 2003.