ARBITRARY GEOMETRY CELLULAR AUTOMATA FOR ELASTODYNAMICS

A Thesis
Presented to
The Academic Faculty

by

Ryan Hopman

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in Mechanical Engineering in the
School of Mechanical Engineering

Georgia Institute of Technology
August 2009

ARBITRARY GEOMETRY CELLULAR AUTOMATA FOR ELASTODYNAMICS

Dedicated to my wife:
Peyton

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Michael Leamy, who has been a superb advisor—helping me select classes, get an assistantship, and manage graduate school, in addition to providing the topic for my thesis work and helping me in the development and writing. He has proven to be an invaluable asset as I have attended Georgia Institute of Technology, always seeming to have what I need to get to the next step.

I also would like to thank the secretaries in the Mechanical Engineering Graduate office who have helped me negotiate the many forms and formalities necessary to graduate. They, along with Dr. Wayne Whiteman, have always been there to greet me and help me when I didn't know where to find the answer.

I would like to thank Dr. Aldo Ferri and Dr. Karim Sabra for taking time out of their schedule to serve on my Master's committee and their willingness to provide feedback.

I am deeply grateful to my parents and in-laws for all the support (both financial and otherwise) that they have given through the years. Without their guidance, assistance, and praise, I would have never made it this far and persisted this long.

Lastly, but most importantly, I would like to thank my wife, to whom I dedicate this work. It is for her that I came to graduate school and through her continued support that I have completed it. She has always been there to provide any assistance I should need and shown great patience when projects have come due.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

t	time
u	displacement
v	velocity
n	normal direction
s	tangential direction
T	Target Cell
N	von Neumann Neighbor
M	Moore Neighbor
σ	stress
ε	strain
E	Young's modulus
υ	Poisson's ratio
ρ	density
λ	Lame's first parameter
μ	shear modulus(Lame's second parameter)
CA	Cellular Automata
OO	Object Oriented
FD	Finite Difference
FV	Finite Volume
FE	Finite Element
FDTD	Finite Difference Time Domain
CSM	Computational Solid Mechanics
CFD	Computational Fluid Dynamics

SUMMARY

This study extends a recently-developed [1] cellular automata (CA) elastodynamic modeling approach to arbitrary two-dimensional geometries through development of a rule set appropriate for triangular cells. The approach is fully object-oriented (OO) and exploits OO conventions to produce compact, general, and easily-extended CA classes. Meshes composed of triangular cells allow the elastodynamic response of arbitrary two-dimensional geometries to be computed accurately and efficiently. As in the previous rectangular CA method, each cell represents a state machine which updates in a stepped-manner using a local “bottom-up” rule set and state input from neighboring cells. The approach avoids the need to develop partial differential equations and the complexity therein. Several advantages result from the method’s discrete, local and object-oriented nature, including the ability to compute on a massively-parallel basis and to easily add or subtract cells in a multi-resolution manner. The extended approach is used to generate the elastodynamic responses of a variety of general geometries and loading cases (Dirichlet and Nuemann), which are compared to previous results and/or comparison results generated using the commercial finite element code, COMSOL. These include harmonic interior domain loading, uniform boundary traction, and ramped boundary displacement. Favorable results are reported in all cases, with the CA approach requiring fewer degrees of freedom to achieve similar or better accuracy, and considerably less code development.

CHAPTER 1

INTRODUCTION

Numerical methods for computing wave propagation in elastic solids present challenges due to difficulties associated with accurate and stable tracking of discontinuous wave fronts. In seismic problems involving regular domains, the staggered grid variant of the finite difference time domain (FDTD) approach is the *de facto* standard due to several characteristics: 1) the field variables are represented by what amounts to a discontinuous discretization, 2) boundary conditions can be formulated in a straight-forward manner, and 3) problems can be efficiently parallelized [2]. However, the finite difference approach does not adequately accommodate arbitrarily-shaped domains, and thus in the analysis of engineering structures, the finite element method has been more commonly applied to study wave propagation in computational solid mechanics (CSM).

In a recent paper [1], a cellular automata (CA) approach has been developed for modeling wave propagation in elastic media. The approach shares an idea central to all cellular automata modeling, which is domain discretization using uniform cells (usually rectangular or hexagonal) whose state is updated via simple rules. The rules typically operate on state information collected from local, neighbor cells. The elastodynamic CA approach is discontinuous like the finite difference method, and thus accurately captures propagating wave fronts. In fact, for a uniform rectangular grid, the method reproduces the same update equations as the central difference FDTD method[1]. Furthermore, the

CA approach has the potential to compute with cells of varying shapes, to include triangular cells. With a triangular rule set, the CA approach could employ cell assembly to compute on arbitrary geometries, much like the finite element method, while conceivably retaining the advantages of the FDTD method in resolving wave fronts. This would mean a very versatile method capable of complex geometries that makes FE a dominant tool in solid mechanics while retaining the accuracy that makes FD a dominant tool in wave modeling.

With the motivation established, this research sets out to extend the CA approach to compute arbitrary two-dimensional geometries using triangular cells. The extension requires 1) formulating a rule set for triangular-shaped cells and 2) implementing an object-oriented simulation approach for cell assembly and state calculations. Results from the new simulator are compared to earlier CA results [1], and to results generated using the commercial finite element package, COMSOL.

Literature Review

In the CA method, a domain is divided into discrete automatons, or cells, each holding a state. This state is discrete and evolves during steps in time via, typically, simple rules which take into account neighbor interactions. In other simulations, such as FD, only space and time are discrete, yet "in CA the space, time, and state of the physical system are discretized" [3]. CA is well-established for studying system dynamics in a diverse variety of disciplines. The first CA simulations proposed and implemented are often credited to John von Neumann [4]. A familiar example is Conway's Game of Life [5], in which each cell in the domain perpetuates or dies depending on the state of its eight neighbors. Although von Neumann imagined a deterministic rule set, a probabilistic rule set is also possible and able to be implemented [6]. Some CA models constrain the state variables to a fixed set (sometimes referred to as totalistic CA) limiting the cell to program specific characteristic, such as alive or dead in the Game of Life, where as other implementations may adopt a continuous range of values[7] (continuous CA).

CA has been used in a variety of modeling applications because of the elegant manner in which formulation of a local set of simple interactions can solve for complex global behavior [8] — this global behavior is often termed ‘emergent.’ Such rules are derived from known relationships (such as geometrical constraints, conservation of energy, etc.) and statistical relationships/models as well as other intuitive relationships. Such intuitive relationships and statistical examples include the rate at which HIV infection affects T-cell count [9], spread of brushfires [10], and traffic models [11]. In the case of HIV infection an intuitive relationship was found and the spread properly modeled with 4 simple rules and 4 possible states. In this case the authors observed for the first time a correct simulation of T-cell infection in which new parameters were not introduced during the running of the simulation. The brushfires were modeled via CA to account for a very heterogeneous landscape in which each quadrant could have individually specified burn rates and fuel abundance.

Applications of known relationships include modeling earthquakes. In a paper by Olami [12] an extremely simple model is used to produce results that are acceptable and agree with data, and this method is in some ways advantageous over the complex differential equations typically used. Yet another paper [13] used CA to show that the earthquakes are due to a critical point system and are in some instances predictable; instead of a critically stressed system, which doesn't allow for predictions. CA has also been used in thermal modeling [14], carbon nanotubes via energy minimization [15], and EM wave propagation [16] which has Boolean state variables, indicating that while a finer mesh is required even binary state variables can provide accurate modeling of wave propagation. Common uses for CA involve modeling grain growth in recrystallization of metals [3, 17-20] and computational fluid dynamics (CFD) with the implementation of Lattice Boltzmann techniques [21, 22] and Navier-Stokes [23]. CA is popular in CFD for its ability to handle highly complex fluid flow. Both techniques, CFD and grain growth, have been combined with FE for its CSM modeling capabilities to model compound systems such as blood flowing through a heart [22] (because of the high Reynolds number and fine time step needed) and stress related to grain boundary growth [19, 20, 24].

Benefits of CA

A principle reason to use CA in simulations is its efficiency and speed [3, 8, 15, 25, 26]. The domain is able to be updated at a fast rate due to CA's massively parallel nature [27-29]. For example, in [8], a one-to-one mapping of 16384 cells to 16384 processors is used to demonstrate that CA allows for simultaneous updating in large-scale problems. This is possible "since information processing in [CA] systems is intrinsically parallel they are naturally amenable to implementation on massively parallel computers with minimal loss in efficiency for coordination requirements" [28]. This is in contrast to FE methods in which, while they can be broken down into sub-domains, there is a diminishing return since more work is required for any increase in number of sub-domains. Computers have seen an increase in processors and parallelizability making CA an attractive option since parallel computing, coupled with simplicity of models, means simulations can be run on desktop computers [30]. Furthermore, CA computational cost is linear with respect to the number of cells [15, 24, 31]. By comparison [31] shows (implicit) FE has a quadratic relationship to the number of cells/nodes and [15] argues Molecular Dynamics has an exponential relationship; hence the value in showing the viability of CA in carbon nanotubes, since multi-walled tubes would be cost prohibitive with a Molecular Dynamic scheme.

Another principle advantage that CA has is the ability to model heterogeneity [6, 10, 27]. Since CA is based on a local rule set and each cell is autonomous, it "…is extremely flexible in adapting to anything singular, discontinuous or even inhomogeneous" [23]. This provides for accurate modeling of inhomogeneous and anisotropic models without the need for smoothing. This is unlike FD which requires smoothing of material properties [8] which can lead to instability in inhomogeneous cases [26]. This also allows CA to perform well in contact modeling [30, 32] which is typically a smoothed partial differential equation [30].

In addition to being able to avoid smoothing of functions and to allow for heterogeneities and discontinuities, CA is able to assure local continuity [25]. This is because CA amounts to a fictitious micro world and not a continuous analytical model [23]. Each cell is connected and part of the domain, so local continuity of the system is preserved;

however, since there is no global equation, no governing analytical equation spanning the domain, there can be discontinuities and variety in parameters and state variables. In an optimization in the orientation of composite fibers [25] the CA model was able to assure continuity in the fiber orientation which was not achievable by just looking at principle stress with FE. Since CA is discrete and can 1) represent or 2) be linked with a continuum, it can model complex systems that break, crumble, and move. Implementation of movable CA based on stress-strain relationships allowed for the accurate modeling of fracture [33]. A limitation of CA can be the challenge in forming the proper governing rule set for the local area.

The combination of these two advantages can lead to the development and handling of complex systems in a timely matter, making real time implementation possible. In [32] a haptic system relying on the mechanical modeling of soft tissue deformation is created. CA is used because mass-spring and FE are not able to handle the nonlinearities of deformation and material properties, since the system supports anisotropic and inhomogeneous materials. Also, the heat equation and weighting factors are included. Results show that the cells are able to preserve continuous nature in addition to nonlinearities and inhomogeneities of the system while achieving the required amount of resolution in real time.

Cellular Automata in Solid Mechanics

A limitation of CA can be the difficulty in the governing rule sets; due to this there has been limited use with CA in CSM. For example, in the modeling of earthquakes both spring-block [12] and a power-law-time-to-failure function [13] were used to derive rule sets. Most rely on a uniform grid, typically square cells [3, 8, 16-18, 24-26, 28, 31], although some simulations apply node-truss implementation [14, 29] sometimes adapting it to triangular elements [14], polar geometry blocks [28], and hexagons [20, 24, 34]. Hexagonal meshing can benefit from better isotropy which translates into a lower stability requirement [34], as opposed to square cells which constrain the calculation to travel strictly in two directions. While irregular geometry can be modeled with a node centered cell with truss connectivity of elements, turning a truss that connects to a cell outside the domain off, this still is based on a regular grid and does not account for truly arbitrary configurations used in FE analysis. Thus far no known implementation has been able to handle the meshed domains that make FE so popular in CSM.

One of the driving factors in applying CA to CSM is the draw of a computational method that is popular for solving solid mechanics and fluid dynamics. Many FV papers [35-38] discuss the prominence of FV in CFD and the desire to extend the method to CSM with the end goal of being able to model a complex problem that has fluid and solids interacting. A common solver could reduce the problem of boundary and numerical tool interaction to merely how boundaries are treated. FV is implemented in both node centered and cell centered schemes although [35] claims that cell centered yields a higher solution quality. While the implementation breaks the domain into finite volumes it still does not have the parallelizability of CA (it is still matrix dependent), it does not have the autonomous nature that CA has, and does not have the emergent characteristics of CA.

Some attempts to adapt CA to solid mechanics have included derivation of a rule set via energy minimization [28], the use of fluid CA with extremely high viscosity [23] to account for shear, this method which begins to show potential but lacks accurate response. [31] Presents a method which shows good results but is limited to a membrane analysis due to inability to properly account for Poisson effects. The node truss technique is often a Jacobi and Gauss Seidel implementation; these are relaxation methods and attempt to model deformation [14]. Some CA inspired CSM models have been linked to other solvers such as FD and FE [31], showing promising results.

In a paper that links FD with CA [39], a boundary interaction was looked at and showed no reflection of the wave as it passed through a shared interface, indicating successful joining of the two methods into a continuous domain. The implementation involved two different scales of CA with respect to the FD grid, indicating that CA is capable of multi-resolution [39]. CA has had application in mesoscale modeling [7, 33] and in multi-scale schemes [17, 19] where CA provides grain growth on the micro scale and interacts with FE which provides the stress on the macro scale. CA has also provided the sound interaction response with and FE analysis of a sound barrier to determine the system response [26]. Some attempts to join FE and CA have gone so far as to mesh the two methods into one hybrid system [7, 14]. Yet, in these applications rule sets are derived from a numerical approximation of the global equations [6-8, 14, 28, 31], the same top-down approach used in FE, FD, and FV as opposed to the bottom-up approach based on local rules that makes CA so adaptable, and none are able to adapt to an arbitrary environment.

Extending Cellular Automata

This work develops a novel CA-based triangular cell class, together with specialized boundary cells, without reusing or reinterpreting any of the previous analysis techniques (FD, FE, or FV). As is well known, this shape is useful for representing domains of arbitrary geometry (see Figure 1) which in part accounts for the popularity of FE. Since each cell is an autonomous state machine, it can be represented as a single object. This lends itself well to object-oriented (OO) programming practices, which has the added benefits of straight-forward implementation using a modern programming language (C++, Java, etc.) and compatibility with parallel processing. The state held by the cell includes such information as material properties, cell geometry, neighbor lists, external forces, and centroidal displacements. Neighbors are easily referenced via pointers so that the order in which the cells are stored is unimportant, eliminating the grid dependency that was used in the previous study [1]. The order in which cells update is also unimportant, furthering the compatibility with parallel computing and multi-resolution analysis. Chapter 2 develops the rule set and computational aspects of the project, Chapter 3 then compares the new formulation and algorithm to the previous work and to a commercial FE package COMSOL to evaluate the codes implementation and accuracy.

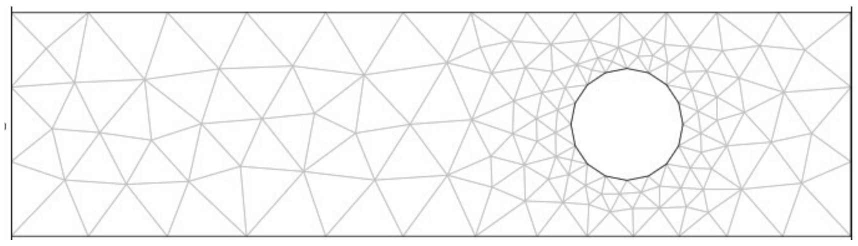


Figure 1: Arbitrary triangular mesh of a two-dimensional domain with an included hole.

CHAPTER 2

METHOD DEVELOPMENT

In this section, the previous rectangular approach is extended to triangular elements via development of an appropriate rule set. Figure 2 demonstrates that, contrary to the square case, which has highly-regular geometry, triangular cells require additional considerations. First, each triangle has a unique set of face lengths and angles. In addition, a line connecting two cell centroids might not be normal to the face the cells share. This complicates the evaluation of the normal and shear strains. Finally, the triangular mesh may produce many neighbors sharing a single vertex, whereas in the rectangular case this was limited to three. The importance of this latter consideration becomes evident when attempting to evaluate the Type I and Type II strains previously identified in the rectangular approach.

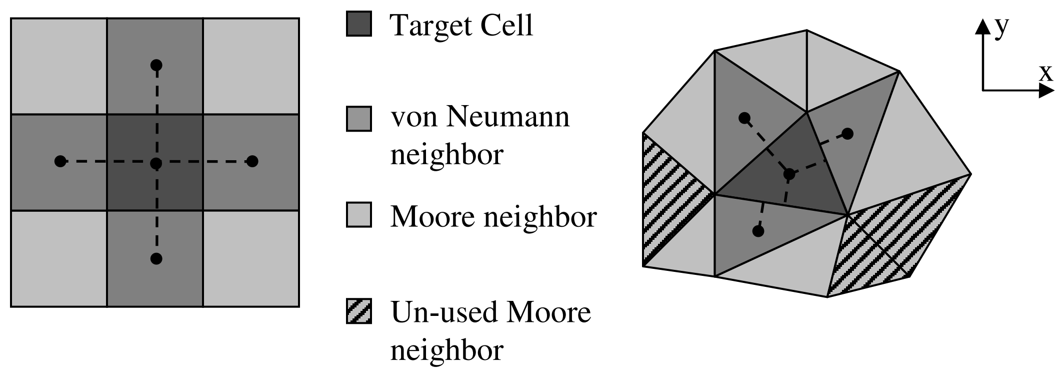


Figure 2: Illustration of geometrical differences between regular cell and arbitrary triangular cells. With square cells all lines perpendicular to the face pass through adjacent centroids, whereas the lines are discontinuous for triangular cells.

As mentioned, in CA each cell stores its own unique state. For this simulation the state variables are displacement, velocity, applied stress (*u, v, σ*), other stored parameters are material properties (*ρ, μ, λ*), cell geometry (centroid, face angles and lengths, area), and neighbor references. All variables are stored relative to a global x-y axis. The state variables of material properties and cell geometry are fixed; they are defined for each cell and stored at setup. The displacement and applied stress are determined at every step of the simulation and updated based on the rule set using the cell's previous state, the von Neumann neighbors, and the immediate Moore neighborhood. As illustrated in Figure 2 not all Moore neighbors are used in this computation; only those that are also von Neumann neighbors of the target cell's von Neumann neighbors.

An overview of the method in which each cell updates its state is now given with a detailed derivation following. Each step begins by transforming the *x* and *y* displacement components into tangential and normal components along one of three faces using the stored face angle. Strains are obtained by numerically estimating derivatives of the displacements across the face; the displacements are obtained by using the target cell and local neighbors. Similar to the previous work on rectangular cells, these derivatives are classified as either Type I or Type II, as described below. Using Hooke's law, the stresses on the face are then calculated and stored; these stress values, when multiplied by the area of the face, yield forces that can be transformed back to the underlying *x,y* coordinate system. This allows for the determining of the overall forces on the cells. After new stresses for each face have been obtained, application of Newton's Second Law leads to semi-discrete differential equations governing the cell's velocity change. A suitable temporal discretization of these equations yields the final rule set. The new displacements are stored until all cells have been updated in what amounts to a double buffering technique suitable for parallelization.

Arbitrary Geometry

As mentioned, each face has a unique orientation that the strains are calculated for. To facilitate strain computations in the two-dimensional case, an angle θ is set as the angle between the x-axis and the face normal ($-\pi \leq \theta \geq \pi$) as shown in Figure 3. The direction perpendicular (normal) to the face (defined as $\boldsymbol{e_n}$) is in the θ direction and subscripts identify the three faces. The direction parallel (tangent) to the face ($\boldsymbol{e_t}$) is in the $\theta + {}^{\pi}/_{2}$ direction. Note that the calculation for the strains and stresses are like those of a square element as per typical mechanics; however, each face is independent. Hence, the shear stress along the face does not relate to the shear stress along any other face. However, in order to satisfy equilibrium, it is a requirement that the shear and normal stresses on a face be the same for both neighbors that share the face.

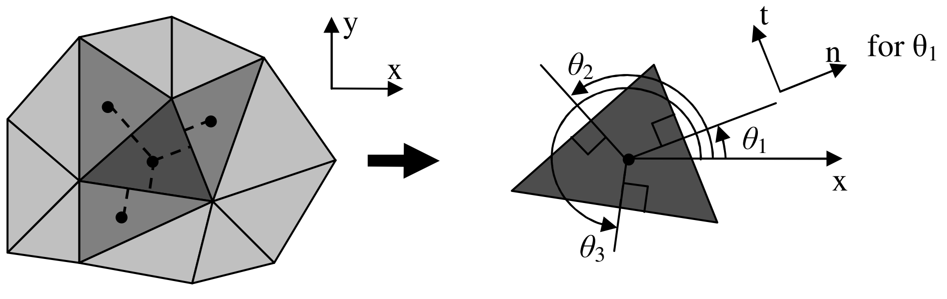


Figure 3: θ for each face is determined by the line from centroid and perpendicular to the face relative to the x-axis. From this the rotation tranformation is determined for the normal and tangent directions.

Strain calculations require displacements in the $\boldsymbol{e_n}$ and $\boldsymbol{e_t}$ directions, which are denoted by u_n and u_t, obtained from the stored cell state by a simple rotation transformation:

$$\begin{bmatrix} u_n \\ u_t \end{bmatrix} = R \begin{bmatrix} u_x \\ u_y \end{bmatrix} \qquad \text{where,} \qquad R = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{1}$$

Once u_n and u_t have been computed, the strains can be calculated according to the definition of strain, a geometrical relationship used to formulate the wave equation:

$$\varepsilon_{nn} = \left.\frac{\partial u_n}{\partial n}\right|_\theta , \varepsilon_{tt} = \left.\frac{\partial u_t}{\partial s}\right|_\theta , \varepsilon_{nt} = \left(\left.\frac{\partial u_t}{\partial n}\right|_\theta + \left.\frac{\partial u_n}{\partial s}\right|_\theta \right) \tag{2}$$

Evaluation of these expressions requires numerical differentiation in either the normal (∂n) or tangential (∂s) directions. As in the previous work [1], the derivatives are defined as Type I (across the face) and Type II (parallel to the face). Figure 4 shows that Type I models tensile stress and direct shear, whereas Type II models the interactions from Poisson effects and indirect shear, things that happen along the face not just across it. It should be noted that these interactions are representative of what the cells are movements would be if they were unattached. Since the model is of a continuum and there is no fracture or cracks allowed the cells are not displacing with respect to one another. The visualization aides in the physical interpretation of the strains but it should be kept in mind that this is the not the actual deformation; especially in the case of the rotation of cells, as rotation doesn't cause shear, but a difference in the displacement across the top and bottom of a face would cause shear along the face as if the cell were trying to shear away from its neighbor.

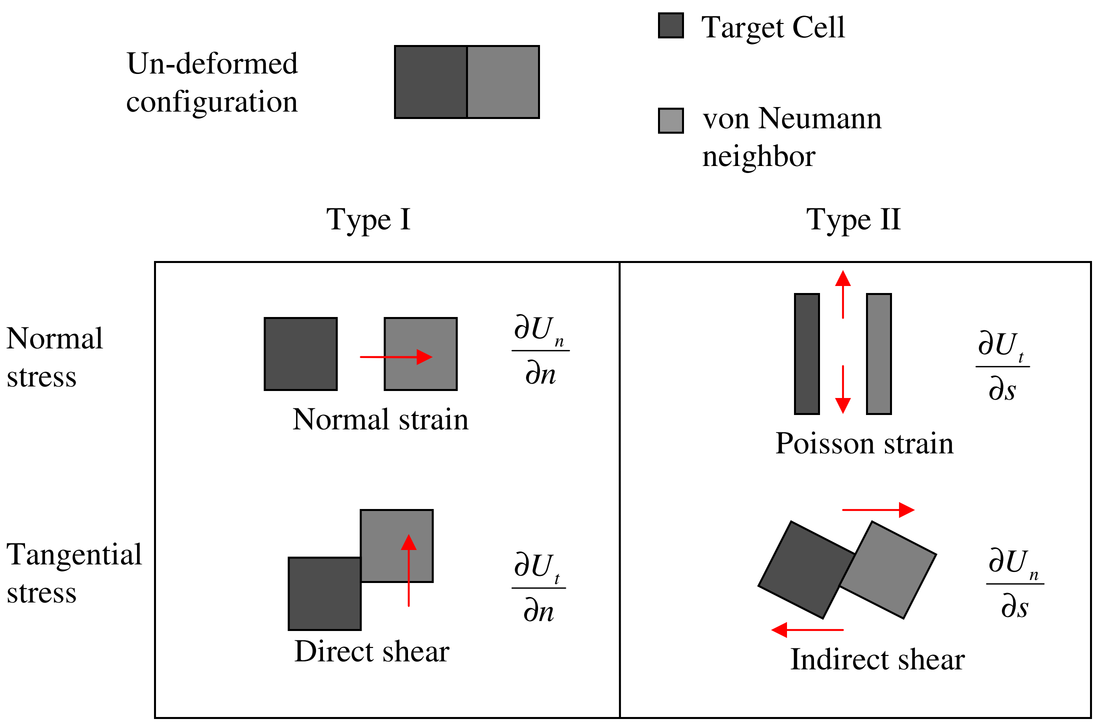

Un-deformed configuration
Target Cell
von Neumann neighbor
Type I
Type II
Normal stress
Tangential stress
$\frac{\partial U_n}{\partial n}$
Normal strain
$\frac{\partial U_t}{\partial s}$
Poisson strain
$\frac{\partial U_t}{\partial n}$
Direct shear
$\frac{\partial U_n}{\partial s}$
Indirect shear

Figure 4: Visualization of strains. Type I strains involve translation where as Type II strains come from such things as elongation or rotation of cells.

For evaluation of the Type I derivatives the von Neumann neighbor is adequate; however, Type II evaluation requires inclusion of Moore neighbors. This is because knowledge of the cell's deformation requires more than just information about the cell's displacement; it is more dependant upon displacements of the cells around it. Therefore, Type II calculations require information about the domain above and below the face, not just on either side of it. Type II calculations are made by combining and averaging four Type I calculations, as shown in Figure 5. Type I calculations to the other von Neumann and Moore neighbors will give a balanced approximation of the Poisson effects which account for the stretching or rotating of an element.

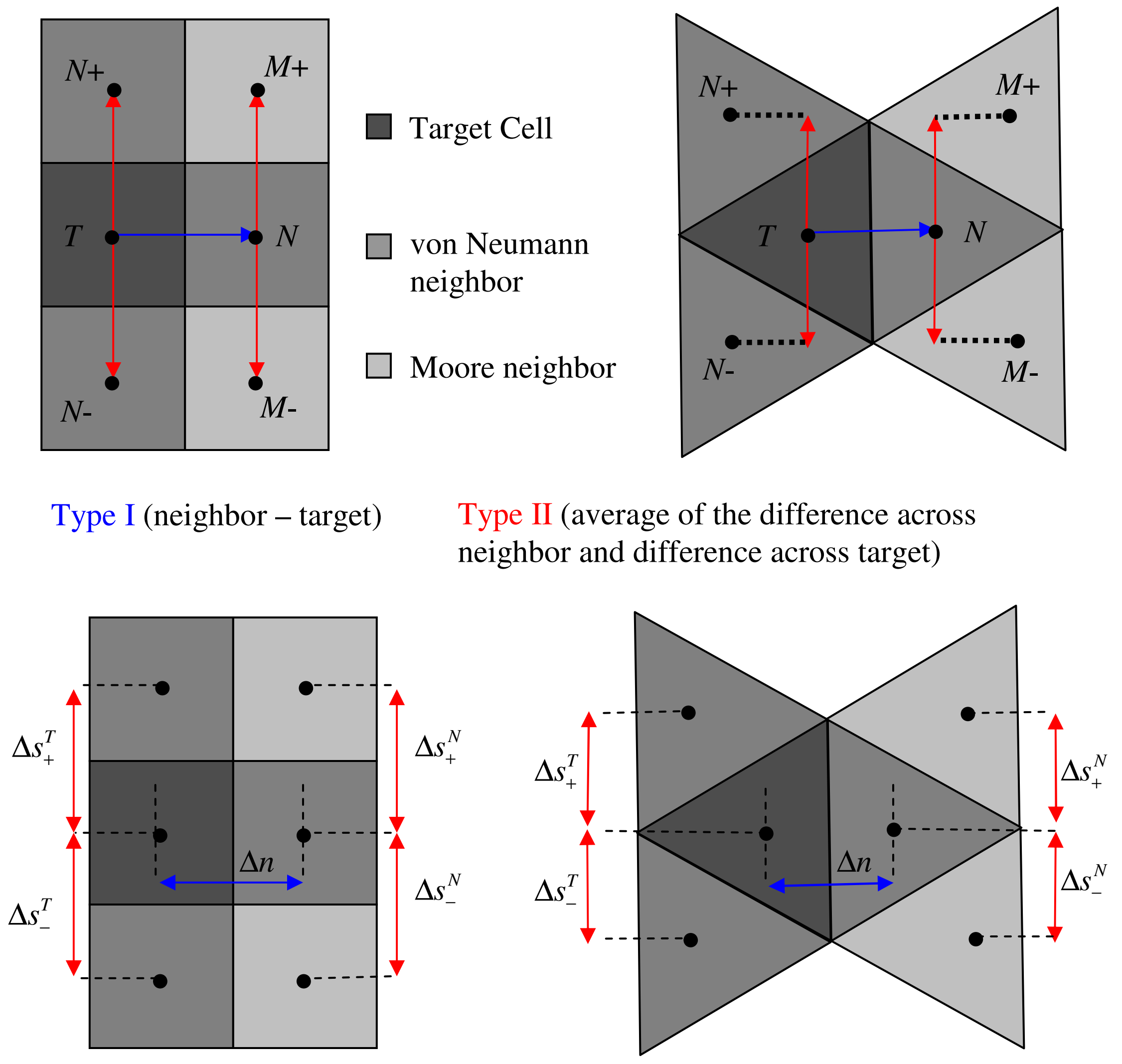


Figure 5: Illustration of the Type I and Type II strain calculations for both the square and triangular cases across a particular face. Also neigbors and terms used are identified.

In order to further define the interactions some additional notation will need to be introduced. Superscripts refer to which cell the calculation pertains to whereas subscripts will refer to direction, such as: *n* normal, *t* tangential + positive theta – negative theta. Figure 5 shows the neighbor labeling for a particular face with the target cell (T) the von Neumann neighbors (N) and Moore neighbors (M). The positive and negative theta directions are in regard to the target cell because each cell is autonomous, thus the frame of reference is always the target cell. Using the Δ*n* and Δ*s* as the distance between the centroids of the neighbors in the normal and tangential directions respectively as shown in Figure 5, a discrete rule set can now be written where the Type I calculations become:

$$\left.\frac{\partial u_t}{\partial n}\right|_{\theta} \Rightarrow \frac{u_t^N - u_t^T}{\Delta n} \qquad \text{similarly,} \qquad \left.\frac{\partial u_n}{\partial n}\right|_{\theta} \Rightarrow \frac{u_n^N - u_n^T}{\Delta n} \qquad \text{(3a, 3b)}$$

For the Type II calculations, two Type I calculations are combined to span the difference across either a neighbor or a target. These calculations are then averaged to find the difference in the tangential direction at the face. The Type II equations take the form:

$$\left.\frac{\partial u_t}{\partial s}\right|_{\theta} \Rightarrow \frac{1}{2}\left(\frac{\left(u_t^{N+} - u_t^T\right) - \left(u_t^{N-} - u_t^T\right)}{\Delta s_+^T + \Delta s_-^T} + \frac{\left(u_t^{M+} - u_t^N\right) - \left(u_t^{M-} - u_t^N\right)}{\Delta s_+^N + \Delta s_-^N}\right) \tag{4}$$

This can be simplified since the target cell's value cancels so the Type II equation becomes

$$\left.\frac{\partial u_t}{\partial s}\right|_{\theta} \Rightarrow \frac{1}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_t^{M+} - u_t^{M-}}{\Delta s_+^N + \Delta s_-^N}\right) \tag{6}$$

Substituting the Type II calculation into equation (2) yields

$$\varepsilon_{nn} = \frac{u_n^N - u_n^T}{\Delta n} \tag{7a}$$

$$\varepsilon_{tt} = \frac{1}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_t^{M+} - u_t^{M-}}{\Delta s_+^N + \Delta s_-^N}\right) \tag{7b}$$

$$\varepsilon_{nt} = \left(\frac{u_t^N - u_t^T}{\Delta n} + \frac{1}{2}\left(\frac{u_n^{N+} - u_n^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_n^{M+} - u_n^{M-}}{\Delta s_+^N + \Delta s_-^N}\right)\right) \tag{7c}$$

The stresses σ_{nn}, σ_{tt}, and σ_{nt} are evaluated according to Hooke's Law: $\sigma = E^*\varepsilon$. The reactions at the face only include the normal stress σ_{nn} and shear stress σ_{nt}; σ_{tt} is not present. For isotropic elastic materials the equations are:

$$\sigma_{nn} = (\lambda + 2\mu)\varepsilon_{nn} + \lambda\varepsilon_{tt} \tag{8a}$$

$$\sigma_{nt} = \mu\varepsilon_{nt} \tag{8b}$$

The stress is multiplied by the surface area to obtain forces. Since σ_{tt} is not present, there are only two forces defined as— $F_n = \sigma_{nn} w * l$ and $F_t = \sigma_{nt} w * l$ where w is the width in the z direction and l is the length. Now that the derivatives and relationship have been defined the equations can be further manipulated to obtain the rule set that the computer implements. It will also be shown that when using square geometry the arbitrary rule set recovers the equations from the previous work. The first step in implementing the solution is substitution to find the equation for the forces. This involves using Hooke's Law, a material relationship used in deriving the wave equation.

$$\begin{bmatrix} F_n \\ F_t \end{bmatrix} = \begin{bmatrix} \sigma_{nn} wl \\ \sigma_{nt} wl \end{bmatrix} = wl \begin{bmatrix} (\lambda + 2\mu)\varepsilon_{nn} + \lambda\varepsilon_{tt} \\ \mu\varepsilon_{nt} \end{bmatrix} \tag{8}$$

$$F_n = wl\left((\lambda + 2\mu)\frac{u_n^N - u_n^T}{\Delta n} + \frac{\lambda}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_t^{M+} - u_t^{M-}}{\Delta s_+^N + \Delta s_-^N} \right) \right) \tag{9a}$$

$$F_t = wl\mu\left(\frac{u_t^N - u_t^T}{\Delta n} + \frac{1}{2}\left(\frac{u_n^{N+} - u_n^{N-}}{\Delta s_+^T + \Delta s_-^T} + \frac{u_n^{M+} - u_n^{M-}}{\Delta s_+^N + \Delta s_-^N} \right) \right) \tag{9b}$$

During the setup the computer reads in a list of nodes and mapping of which nodes make up an element. From this information centroids, face lengths and angles, and neighbors can be determined. However, since the vectors linking centroids are not always parallel to the line perpendicular to the face, the offset must be taken into account as seen in Figure

6. Since the deformation of the cells is assumed negligible by the small strain approximation, this calculation is only performed at setup when an adjusted *r* matrix is created and stored.

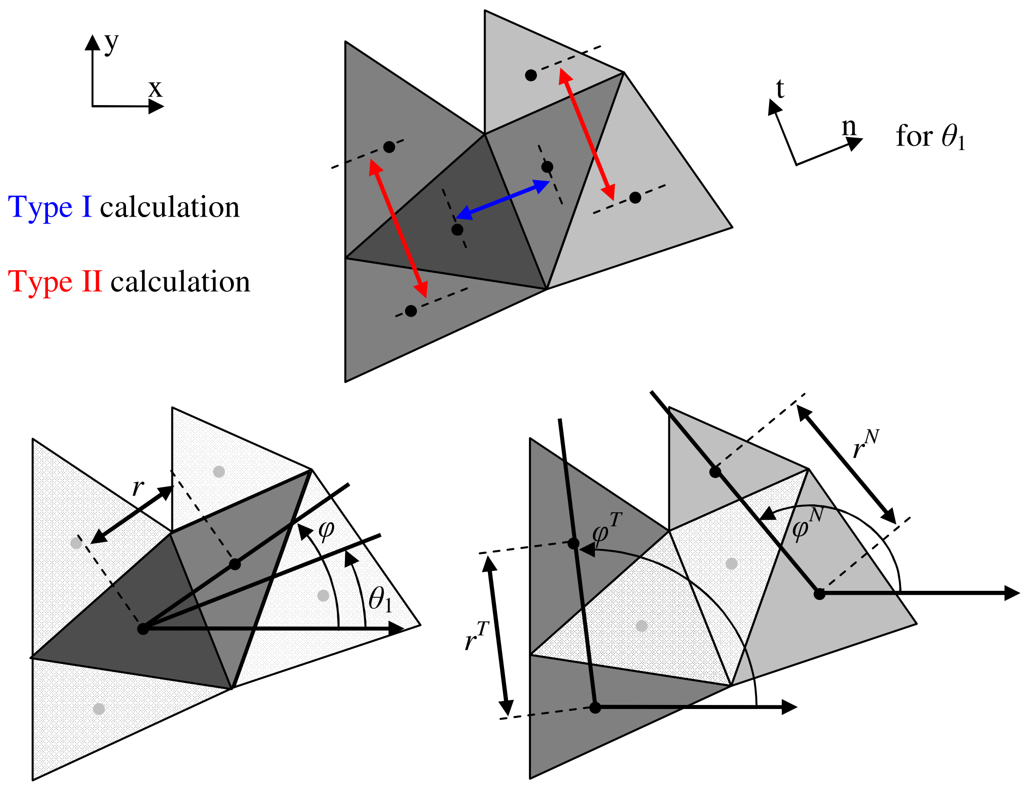


Figure 6: Illustration of the parameters to get Δn and Δs from centroid and face angle information.

The vector connecting cell centers has a magnitude r and angle φ (x-axis datum).The Δn becomes r*cos(φ-θ). However, for the Type II scenario, Δs^+ and Δs^- combine to become r^N*cos(φ^N-θ-$^\pi/_2$) or r^T*cos(φ^T-θ-$^\pi/_2$).

$$F_n = wl\left((\lambda+2\mu)\frac{u_n^N - u_n^T}{r\cos(\phi-\theta)} + \frac{\lambda}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{r^T \cos\left(\phi^T - \theta + \pi/2\right)} + \frac{u_t^{M+} - u_t^{M-}}{r^N \cos\left(\phi^N - \theta + \pi/2\right)} \right)\right) \tag{10a}$$

$$F_t = wl\mu\left(\frac{u_t^N - u_t^T}{r\cos(\phi-\theta)} + \frac{1}{2}\left(\frac{u_n^{N+} - u_n^{N-}}{r^T \cos\left(\phi^T - \theta + \pi/2\right)} + \frac{u_n^{M+} - u_n^{M-}}{r^N \cos\left(\phi^N - \theta + \pi/2\right)} \right)\right) \tag{10b}$$

This formulation takes into consideration the discontinuities inherent in the triangular formulation without interpolation which could be detrimental in the analysis of wave propagation. The forces in the *x* and *y* directions are obtained by the reverse of the earlier rotation transformation, which is the transverse of the rotation matrix, giving

$$\begin{bmatrix} F_x \\ F_y \end{bmatrix} = R^T \begin{bmatrix} F_n \\ F_t \end{bmatrix} \tag{11}$$

Once this has been accomplished for each face the forces acting on a cell are once again in the global coordinate system. Since the directional method takes into account the direction of the force, the force balance in the *x*-direction is $F_x^{Total} = \Sigma F_x + F_{x\text{-}load}$, where $F_{x\text{-}load}$ is any external loading applied to the cell. This a mechanical relationship used in formulation of the wave equation. Using forward Euler as a first order temporal update approximation for the balance of momentum, the velocity update equations become:

$$v_x^{k+1} = v_x^k + \frac{1}{d\rho wA} F_x^{Total} \tag{12a}$$

$$v_y^{k+1} = v_y^k + \frac{1}{d\rho wA} F_y^{Total} \tag{12b}$$

Where *d* is the discrete number of steps per unit time, *ρ* is the density, *w* is the width of the domain and *A* is the area of a cell. The superscripts *k* naturally represent an iteration of the simulation. Displacement can be obtained from velocity using the same approximation yielding:

$$u_x^{k+1} = u_x^k + \frac{1}{d} v_x^{k+1} \quad \text{and,} \quad u_y^{k+1} = u_y^k + \frac{1}{d} v_y^{k+1} \qquad (13a,13b)$$

Equations (10-13) formulate the rule set by which each cells state is updated; solving for applied stress, velocity, and displacement of each cell. This rule set was derived using the three fundamental relationships (geometrical, material, mechanical) in formulating the wave equation for elastodynamics but in a bottom-up formulation, never deriving the global analytical equation that top-down formulation requires.

Verification of Arbitrary Rule Set

Now the rules by which all the state variables (*u*, *v*, *σ*) are updated have been completely defined for an irregular cell with arbitrary orientation. Since this is an extension of previous work when applied to the square case, this formulation should perfectly recover the equations set forth therein. For the square case, $\cos(\phi - \theta + \pi/2) = \cos(\phi - \theta - \pi/2) = 1$ since the cell centers where displacements are stored are at $\theta - \pi/2$. The r values are all the same in the square case due to the uniform geometry. This simplifies the equations considerably, and they can be written as:

$$F_n = wl\left((l+2m)\frac{u_n^N - u_n^T}{r} + \frac{l}{2}\left(\frac{u_t^{N+} - u_t^{N-}}{2r} + \frac{u_t^{M+} - u_t^{M-}}{2r} \right)\right) \qquad (14a)$$

$$F_t = wl\mu\left(\frac{u_t^N - u_t^T}{r} + \frac{1}{2}\left(\frac{u_n^{N+} - u_n^{N-}}{2r} + \frac{u_n^{M+} - u_n^{M-}}{2r} \right)\right) \qquad (14b)$$

The above expression holds for any face in the square case. A face-by-face analysis is derived in appendix A utilizing a grid indexing scheme that yields the resulting *x*-direction forces:

$$F_x^{right} = w\Delta y\left(\left(\lambda + 2\mu\right)\frac{_{(i+1,j)}u_x - {}_{(i,j)}u_x}{\Delta x} + \frac{\lambda}{2}\left(\frac{_{(i,j+1)}u_y - {}_{(i,j-1)}u_y}{2\Delta y} + \frac{_{(i+1,j+1)}u_y - {}_{(i+1,j-1)}u_y}{2\Delta y} \right) \right) \quad (15a)$$

$$F_x^{left} = w\Delta y\left(\left(\lambda + 2\mu\right)\frac{_{(i-1,j)}u_x - {}_{(i,j)}u_x}{\Delta x} + \frac{\lambda}{2}\left(\frac{_{(i,j-1)}u_y - {}_{(i,j+1)}u_y}{2\Delta y} + \frac{_{(i-1,j-1)}u_y - {}_{(i-1,j+1)}u_y}{2\Delta y} \right) \right) \quad (15b)$$

$$F_x^{top} = w\Delta x\mu\left(\frac{_{(i,j+1)}u_x - {}_{(i,j)}u_x}{\Delta y} + \frac{1}{2}\left(\frac{_{(i+1,j)}u_y - {}_{(i-1,j)}u_y}{2\Delta x} + \frac{_{(i+1,j+1)}u_y - {}_{(i-1,j+1)}u_y}{2\Delta x} \right) \right) \quad (15c)$$

$$F_x^{bottom} = w\Delta x\mu\left(\frac{_{(i,j-1)}u_x - {}_{(i,j)}u_x}{\Delta y} + \frac{1}{2}\left(\frac{_{(i-1,j)}u_y - {}_{(i+1,j)}u_y}{2\Delta x} + \frac{_{(i-1,j-1)}u_y - {}_{(i+1,j-1)}u_y}{2\Delta x} \right) \right) \quad (15d)$$

These forces are now in the same notation and format as the formulation used in [1]. Presentation of the previous grid based rule set follows to allow for verification of recovery of regular geometry implementation.

$$F_x^{right} = \left(w\Delta y\right)\left[\frac{\lambda + 2\mu}{\Delta x}\left(_{i+1,j}u_x - {}_{i,j}u_x \right) + \frac{\lambda}{2}\left(\frac{_{i+1,j+1}u_y - {}_{i+1,j-1}u_y}{2\Delta y} + \frac{_{i,j+1}u_y - {}_{i,j-1}u_y}{2\Delta y} \right) \right] \quad (16a)$$

$$F_x^{left} = \left(w\Delta y\right)\left[\frac{\lambda + 2\mu}{\Delta x}\left(_{i,j}u_x - {}_{i-1,j}u_x \right) + \frac{\lambda}{2}\left(\frac{_{i-1,j+1}u_y - {}_{i-1,j-1}u_y}{2\Delta y} + \frac{_{i,j+1}u_y - {}_{i,j-1}u_y}{2\Delta y} \right) \right] \quad (16b)$$

$$F_x^{top} = \left(w\Delta x\right)\mu\left[\frac{_{i,j+1}u_x - {}_{i,j}u_x}{\Delta y} + \frac{1}{2}\left(\frac{_{i+1,j+1}u_y - {}_{i-1,j+1}u_y}{2\Delta x} + \frac{_{i+1,j}u_y - {}_{i-1,j}u_y}{2\Delta x} \right) \right] \quad (16c)$$

$$F_x^{bottom} = \left(w\Delta x\right)\mu\left[\frac{_{i,j}u_x - {}_{i,j-1}u_x}{\Delta y} + \frac{1}{2}\left(\frac{_{i+1,j-1}u_y - {}_{i-1,j-1}u_y}{2\Delta x} + \frac{_{i+1,j}u_y - {}_{i-1,j}u_y}{2\Delta x} \right) \right] \quad (16d)$$

Notice that when the comparison is done for left and bottom faces, the directional method differs from the earlier derivation by a negative sign. This is because the previous study computes the force balance with the understanding of the stress directions, so the force balance is $F_x^{Total} = F_x^{right} + F_x^{top} - F_x^{left} - F_x^{bottom}$. However, the directional method takes into account the direction of the force, so the force balance is $F_x^{Total} = \Sigma F_x$. Hence, the two methods are equivalent.

Object Oriented Approach

In addition to expanding the cells to accommodate arbitrary geometry, the CA approach was also adapted to a truly OO environment. Since every cell in CA is autonomous, a cell object can be created to store all of the parameters. This object is included in a list of objects that makes up the domain. As long as the neighbor cells are referenced properly the order of the cells is not important, the order of the computation is not important, and while each cell must be a standard format (storing all essential information) it can be completely different with respect to the properties. OO programming allows the basic cell to be sub classed into different types, such as domain cells or boundary cells. This can be taken further in later work with CA to accommodate other shapes, types, loading configurations, or boundary interactions such as fluid – solid interactions or crack growth treatment.

The developed simulation has a basic Cell class which stores the material properties (ρ, μ, λ), cell geometry (centroid, face angles and lengths, area), and state (displacement, velocity, applied stress). The generic Cell class also has methods which govern the updating and sharing of these parameters. The Cell is sub-classed into domain cells, both triangular and quadrilateral, as well as boundary cells (discussed later). The generic class is abstract, meaning it cannot be instantiated. This guarantees that each cell has the same methods yet must be of a certain type; a cell cannot be vague and undetermined.

Since each cell has the same methods in spite of specific parameters or type, this allows the simulation to run smoothly, as each cell is called in the same way. Since each cell must be created and its parameters must be set, the initial setup is a bit computationally intensive. However, this same domain list can be stored. This is analogous to LU decomposition in that the factoring of the matrix requires more work up front but the same setup can be saved for later runs, reducing the cost overall. A further advantage to CA and OO implementation is that since the cells are autonomous and individual the process is parallelizable, both in setup and during the simulation.

A breakdown of the simulation is included in Figure 7. The simulation begins with a setup method calling for a list of Cells. This could be read from a file of a saved list or, as shown, a mesh file. From the element and node information the setup class determines each cell's geometry, then its immediate neighbors, and the program then is able to organize the neighbors and establish matrices of pointers and r values. The r values are the distances between centroids needed for the computation. For cells which lie on a boundary and do not have a full compliment of neighbors, boundary conditions are applied via the addition of boundary cells which are discussed shortly. Each cell, including boundary cells, is saved into an array that is passed back to the main simulation.

During the analysis stage each cell is told to run its step method which finds all the applied stress, either by computing the face stress via or receiving it from a neighbor. This is simply done by checking to see if the neighbor cell has been updated already and, if so, the target updates by using the negative of the neighbor’s stress vectors, allowing for the calculation to be performed only once per face. If a load is applied the setup determines which cell is loaded and the simulation computes what the load is, thereby making the same domain applicable to a variety of load cases. If a load is applicable it is passed to the cell to be used in conjunction with the face stresses to update its displacement. The new displacement is stored and the cell is now set to “updated” so other cells can use the stresses it calculated. Once all the cells have been updated the code stores the new displacement as its current displacement. This is a standard double buffer routine ensuring old and new data aren’t being mixed. Output is stored based on user defined frequency or time steps.

```
Simulation
setupCell()
        CALLS ⇒
                        MeshInput Class
                                OPENS file
                                        READS node and element matrices
                                CREATES Triangular elements
                                        FOR each element uses nodes to:
                                            Determine cell geometry (face angles and
                                                lengths, centroid, area)
                                            Determine neighbors (if no neighbor assigns
                                                a boundary cell based on orientation)
                                            Develop matricies of pointers and distance to
                                                centroids which are determined by cell
                                                geometry
                                            SAVES cell information and material
                                                properties to an array of Cell objects
                                DETRMINES which cell(s) is loaded
                      ⇐ RETURNS  array of Cells

run()
        While t < $t_{final}$
          FOR each Cell
            CALLS ⇒
            passes force    Cell step() routine
            if a load cell     Computes face stress as specified by equations 14 a,b
                               unless faces stress has already been computed by neighbor
           FOR each Cell
            CALLS ⇒ Cell buffer() routine

           IF criteria is met
                OUTPUTS data files
        End While
        CLOSES files
```

Figure 7: Logic path of OO approach

Boundary Treatment

Any cell that lacks a neighbor is assigned a boundary cell depending on orientation. A regular triangle is added with the side lengths matching the domain cell as in Figure 8; centroids and other parameters are calculated accordingly. To accommodate boundary conditions two types of boundary cells are used: Neumann, which models applied displacement, and Dirichlet, which models applied stress.

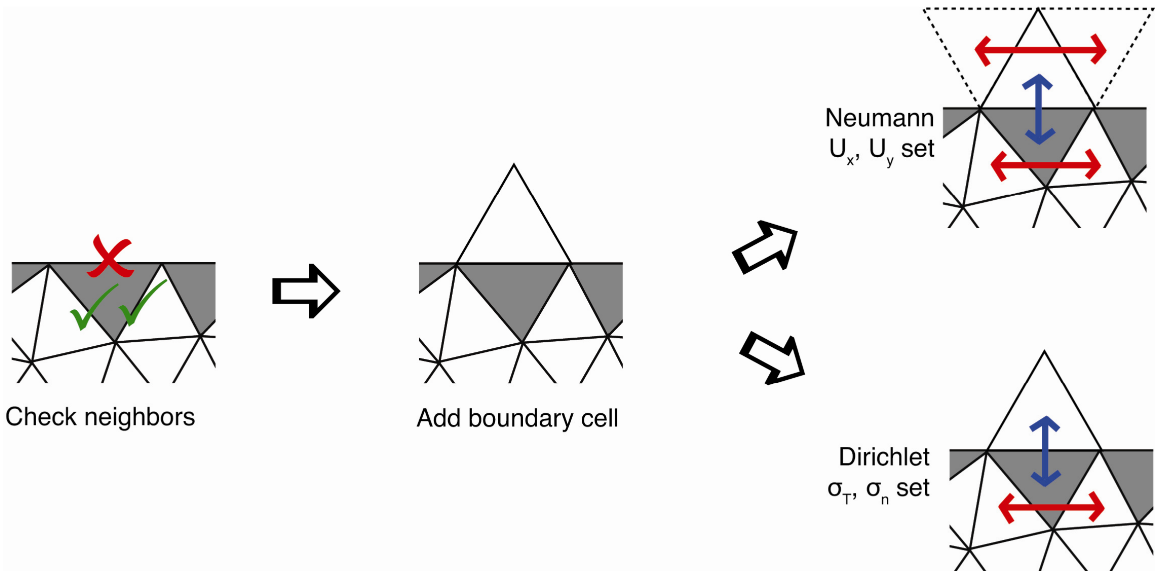


Figure 8: Method of applying a boundary cell. Any cell that fails a neighbor check gets a cell added to it with parameter set reflecting either a displacement fixed (Neumann) or stress fixed (Dirichlet).

The Neumann cells account for the fixed boundary requirement and also provide the initial displacement condition. When the cell is initiated it also requires a displacement: <0, 0> is fixed, <1, 0> is unit displacement in the x, etc. When the program runs, the cell's displacement never updates, since it is fixed and the cell never computes the boundary stress. The computation also needs two additional cells for the Type II calculation, but as indicated in Figure 8 the necessary geometry is calculated, yet the cells are never created. This can occur because any additional boundary cells would also not need to be updated and share the same material properties, so the pointers for the Type II calculation refer back to the base boundary cell.

The Dirichlet cells allow for no stress or traction at the surface and are initialized accordingly. Since the stress is set when initiating the cell, the parameter that stores whether the cell has computed face stresses is set to true, and it doesn't require imaginary neighbors. This is due to the domain cell adjacent to it obtaining the face stress and therefore never goes through the Type II (or Type I) calculation for that face. The Dirichlet cell is involved in a Type II calculation; thus it requires neighbors to update position as the program runs. The calculation is as outlined in previous CA work [1]; when the notation is modified to match the arbitrary geometries and loading parameters the displacement equations become:

$$u_n^{dirichlet} \cong u_n^N - \frac{\sigma_n * \Delta n}{(\lambda + 2\mu)} + \frac{\lambda * \Delta n}{(\lambda + 2\mu) * \left(\Delta s_+^N + \Delta s_+^N\right)} \left(u_t^{M+} - u_t^{M-}\right) \tag{17a}$$

$$u_t^{dirichlet} \cong u_t^N - \frac{\sigma_t * \Delta n}{\mu} + \frac{\Delta n}{\Delta s_+^N + \Delta s_-^N} \left(u_n^{M+} - u_n^{M-}\right) \tag{17b}$$

Where σ_n and σ_t denote the applied normal and tangential tractions, respectively. This now accounts for boundary treatment in a straightforward manner similar to how the domain cells are loaded. The implementation is simple in OO programming which CA is well suited for. The algorithm is now suitably developed and the accuracy and versatility of the method can be compared to other codes for solving waves in elastodynamics.

CHAPTER 3

VALIDATION AND DISCUSSION

To verify the accuracy of the results the program is compared to results obtained using previous CA results [1] and finite element package COMSOL. The first test case compares the updated algorithm to that obtained from a previous non-OO cellular automata. This test uses the same parameter values and loading—differentiated Gaussian pulse—as discussed in the previous work, but the new results utilize full OO implementation and directional derivatives. The same load scenario is then looked at again with triangular elements to judge the effects that irregular geometry has on the response. Arbitrary meshes generated by COMSOL are used to evaluate the accuracy of harmonic loading and boundary traction. The same mesh was used by both the COMSOL and CA. Further comparisons to COMSOL are made with regard to loading and numerical approximations.

Rectangular Recovery

In [1] the CA code use the parameters of a 200*100 grid with 10 steps per unit time with a differentiated Gaussian pulse loaded on the center square on the top row of cells. Square cells are used and a free boundary condition is applied to the top surface including the corners. All other boundary conditions are set as fixed. The results were compared at step 160. For consistency the same loading, cell geometry, boundaries, and step time will be applied in the new implementation and looked at here. Both programs output results to a file that Matlab could then read, and the *imagesc* command was used to create a color scale plot for easy visualization. The displacements in both the x and y directions are shown and compared in Figures 9 and 10 respectively.

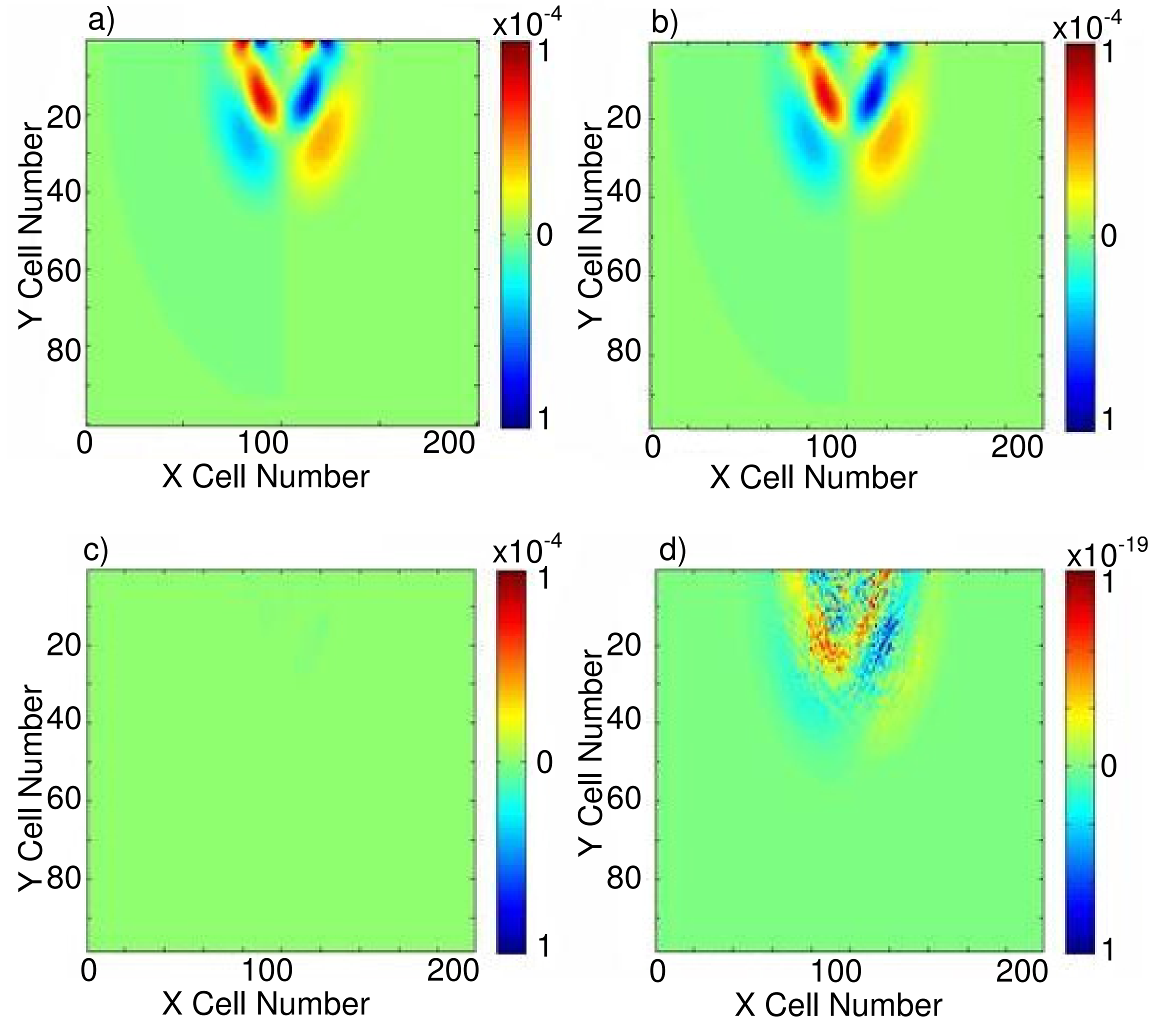


Figure 9: Comparison of x-displacement output. a) is output of OO implementation b) is output of grid based implementation c) is the difference of the two outputs on the same scale as the response d) is the difference scaled to visible range

The differences are presented at two different scales: 1) the Matlab default scale maximum and minimum values are bounds, 2) adjusted scales to match the order of magnitude of the displacement. A percent error calculation indicated that the maximum percent error is 3.3e^{-9}. It is hypothesized that the error is due to the precision of the stored values. While it was hoped the calculation would be exact, this validates the OO approach as a mathematically sound way of doing the CA approach set forth in (Leamy, 2008) since the square case yields the same equations and results.

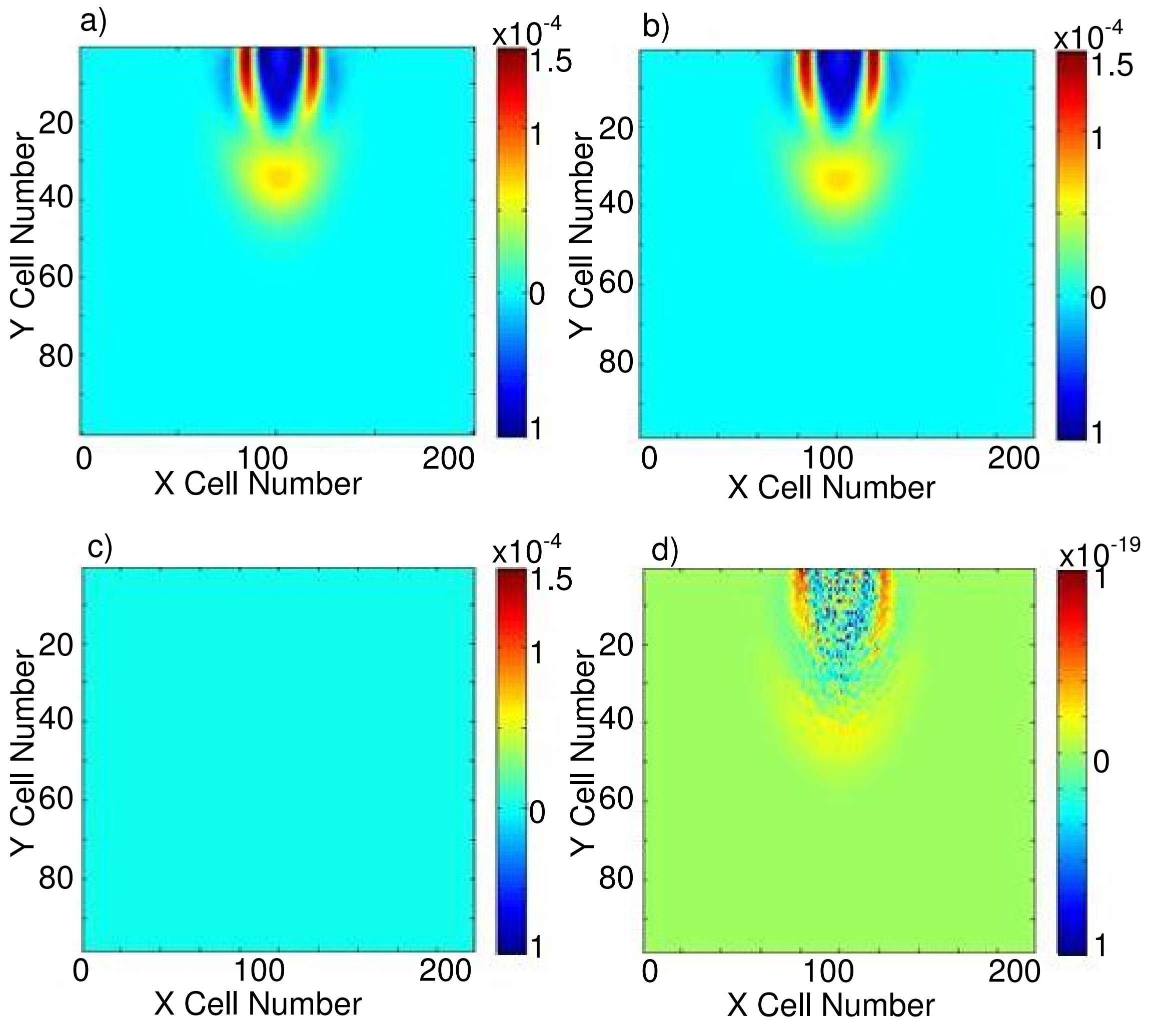


Figure 10: Comparison of y-displacement output. a) is output of OO implementation b) is output of grid based implementation c) is the difference of the two outputs on the same scale as the response d) is the difference scaled to visible range

Boundary Condition Sensitivity

This section details what was learned as the code was modified to reproduce the reviewed conditions exactly. Since the purpose was to match the output, an in depth analysis of the boundary conditions will not be presented, yet in the process of matching the output of the two codes interesting examples appeared showing just how sensitive the simulation is depending on exactly how the conditions are applied. For this reason the output recorded during the debugging process is presented here.

A difference in *when* the boundary conditions are calculated is significant. The original [1] code calculated the free boundary cells' displacement between time steps, as compared with the OO code which calculated the boundary conditions as part of the space. Because the CA method deals with, in the square case, the square's eight neighbors, the area of cells that experience the pulse grows by one layer of cells with each step the code takes. With the boundary calculation between steps the top surface grew by two cells at each step. Figure 11 shows that the difference is now on the order of 10^{-6} instead of 10^{-20}. While small, the magnitude of the change in difference between old and new is still significant. This perhaps can be better seen in the percent error of the plots of the old and new methods. When the boundary is updated separately from the domain up to 20% error is common, and at certain points the error is larger than 100%, although this occurs where displacement is close to zero due to transitioning from positive to negative values. This demonstrating how sensitive the computation is to small differences in the procedure. The codes were adjusted to update in the same manner (during the step).

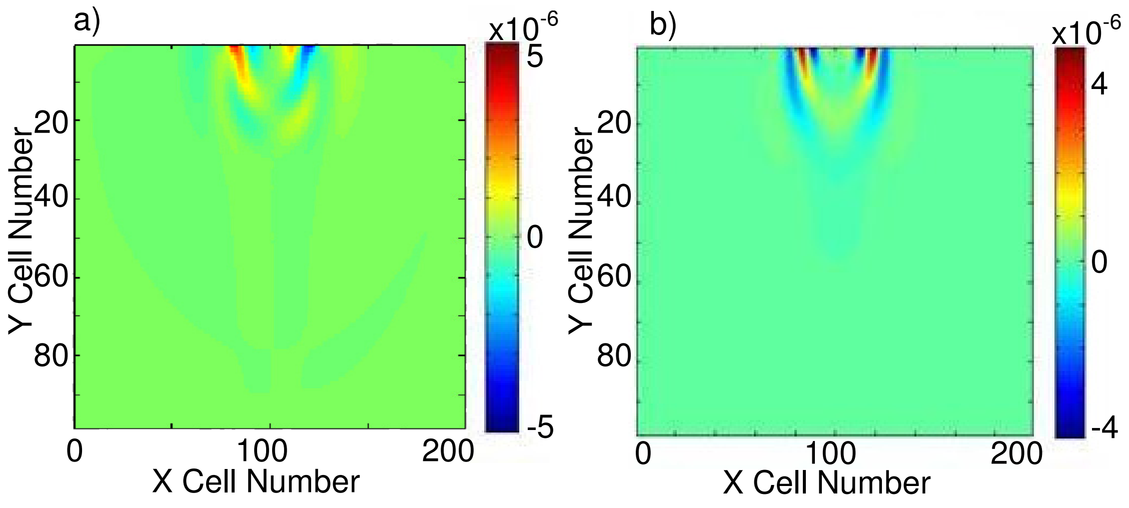


Figure 11: The difference in the a) x -displacement and b) y-displacement when the boundary and domain cells update in different steps.

Another subtle difference that depends on the boundary conditions at the corners was found. In the original [1] code the top corners along with the top surface were free while initially in the OO code the corners were all fixed. This did not have much impact on step 160 as the wave had not yet traveled to and reflected off the sides. Taking this into consideration the program was run to step 500 to guarantee a visible impact of different corner boundary cells. Figure 12 gives an idea of what the displacement field looks like at step 500. The original code output is shown; in the interest of space the OO output is not shown but was used in finding the difference between the boundary conditions. The difference is significant and noticeable. While small it does indicate that there are cumulatively significant impacts depending on very small changes to the domain. Again large percent errors are noticeable yet occur in regions where the values approach zero.

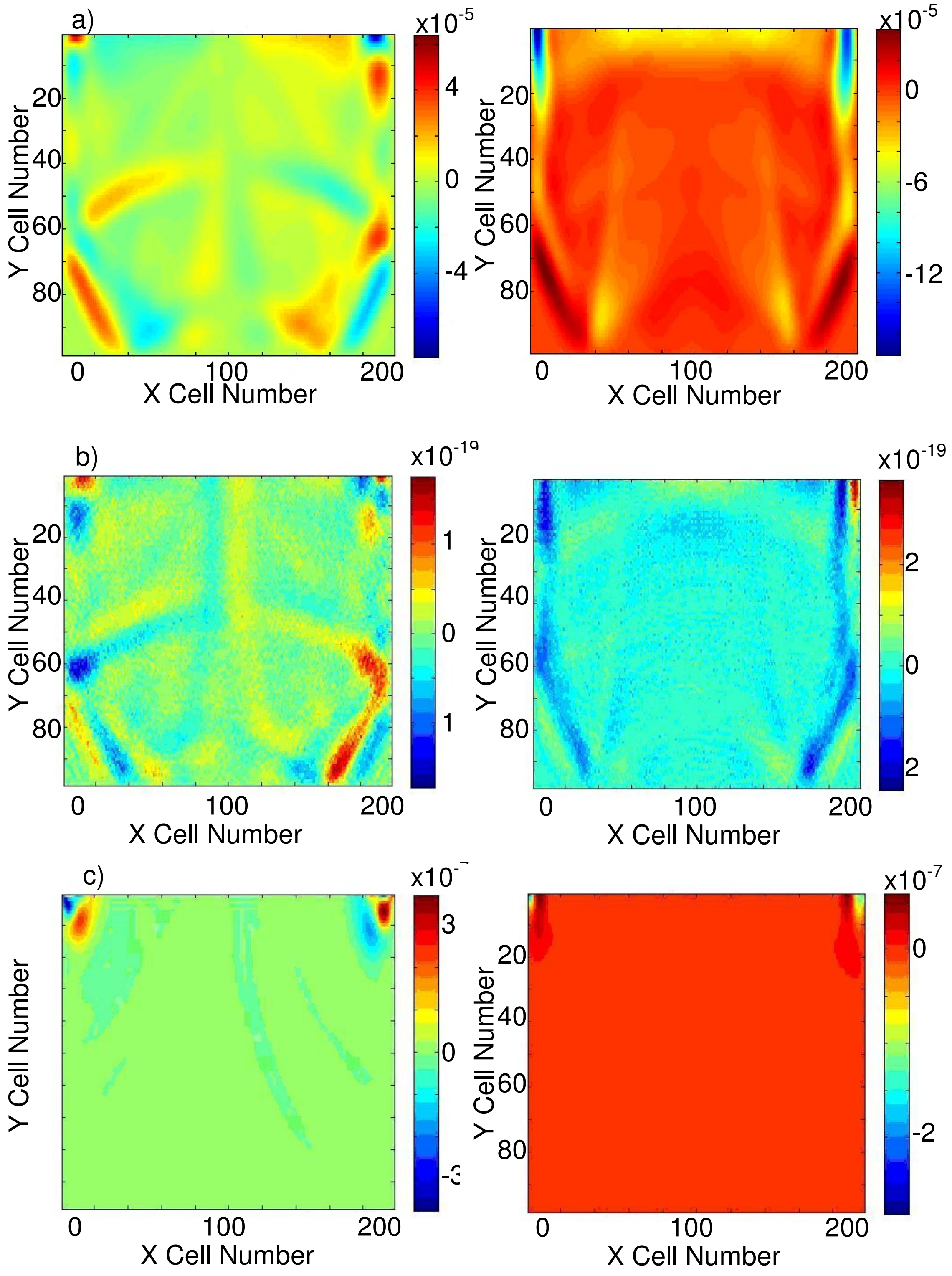


Figure 12: Difference in output based on corner cell agreement in *x* (left) and *y* (right) displacements. a) response at step 500 (t = 50 s) b) difference in response when corners are in agreement c) difference when corners are fixed in OO and free in grid-based.

There is still debate on how boundary conditions should be applied, especially at the corners, and when they should be computed. A common difficulty in modeling often deals with exactly how to apply boundary conditions. The presentation here was to demonstrate the sensitivity and the importance of matching the exact conditions that were used previously. This also could aid in understanding later problems and discrepancies in the codes; since boundary conditions effect a relative difference of approximately 1% results varying by that amount can be considered accurate with respect to the algorithm. This gives perspectives as to how significant later variance between results is. Once the codes were matched, the output thereof shown previously, no significant differences were present and the error can be attributed to other numerical analysis issues such as underflow and machine epsilon.

Triangular Geometry

In order to evaluate the effectiveness of the code and to determine the effect of trianglar compared to rectangular elements, the first comparison utilizes the same 2-D domain with differentiated Gaussian pulse applied on a free surface. The triangular elements are created by dividing a square cell in two along alternating diagonals as seen in Figure 13. Also shown are the relevant neighbors of an element. This allows for a grid dependent system that has uniform yet irregular geometry. Simple geometry is used to obtain all the cell properties and, to define the neighbor relationships, based on the grid relationship $(i,j+1)$, $(i,j-1)$, and $(i+f(i,j),j)$ where:

$$f(i,j)=\left(\frac{\mathrm{i}^{j}}{\left|\mathrm{i}^{j}\right|}\right)(-1)^{i} \tag{17}$$

This relationship is very dependent on this geometry and does not occur when the preprocessing is computed based on a meshed domain.

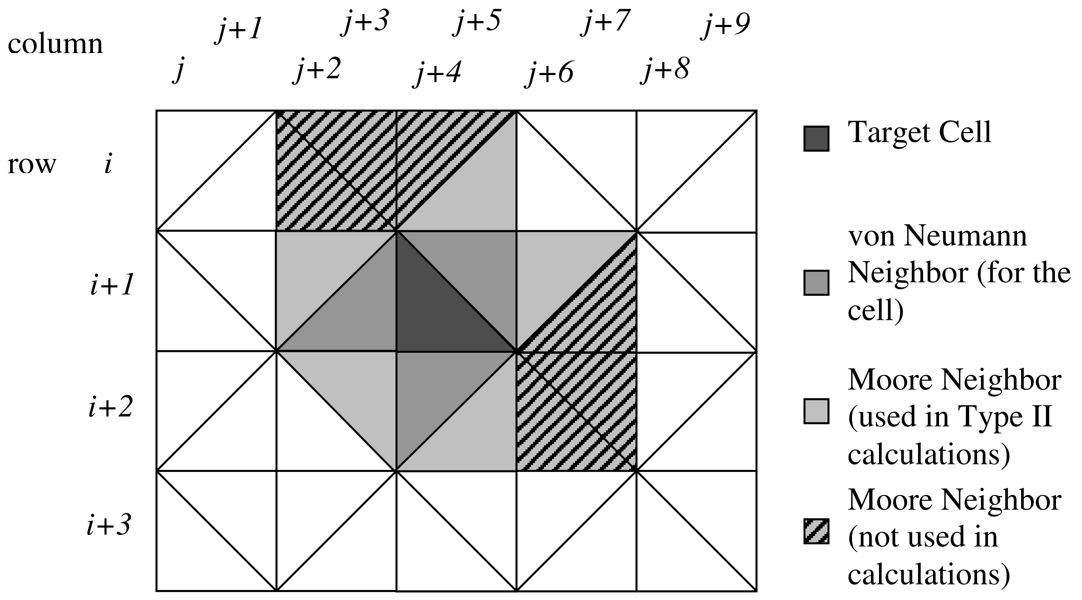

column
j
j+1
j+2
j+3
j+4
j+5
j+6
j+7
j+8
j+9
row i
i+1
i+2
i+3
Target Cell
von Neumann Neighbor (for the cell)
Moore Neighbor (used in Type II calculations)
Moore Neighbor (not used in calculations)

Figure 13: Demonstration of the repeating geometry used to test the triangular cell case.

Results from both cases (squares and triangles) are compared using Matlab by outputting a matrix and utilizing the *imagesc* command to plot the results. In order to compare values the triangular elements are averaged back into square cells allowing for the comparison of absolute and relative error. Figure 14 shows the results are very similar yet there is a noticeable difference. Some symmetry is lost, which was expected due to a loss of symmetry in the division of the domain; a visible difference can be seen in the expanded portion in the y displacement plots.

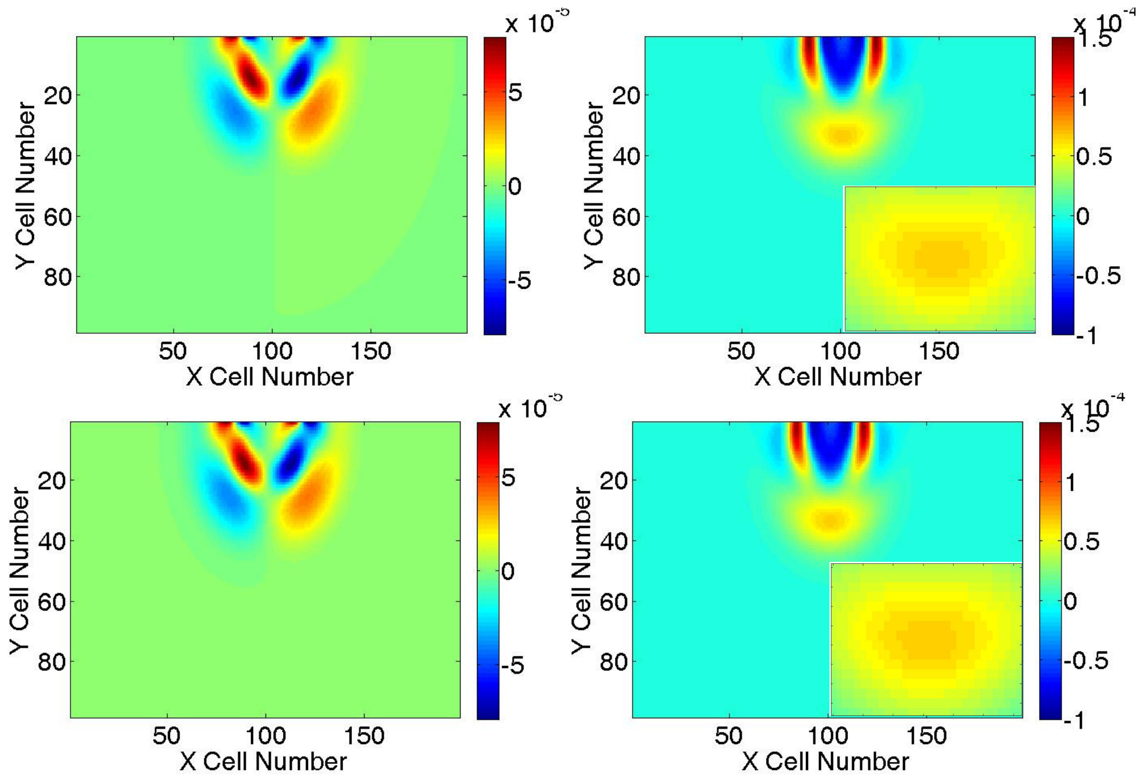


Figure 14: Comparison of displacements of square cells (top) to triangular cells (bottom) with respect to x direction (left) and y direction (right).

Results were generated with a variety of loading cases to explore the effects of loading and resolution. The loading cases involved loading one triangular cell with the full load, splitting the load between two triangular cells that made up the corresponding square cell, or splitting the load between two adjacent triangular cells that were reflections of one another's positions. All cases yielded equivalent results with differences being an order of magnitude smaller (Figure 15). When symmetrical loading was applied, symmetry in the domain was recovered. When the loading is applied to the two triangular cells that create the load square the best agreement occurs between those cases. Also of note is when the cells were created as rectangles and averaged back into squares. The differences were on the same order of magnitude but the computational time step had to be lowered indicating that triangles are more stable with respect to the computational wave speed than to the propagated wave speed.

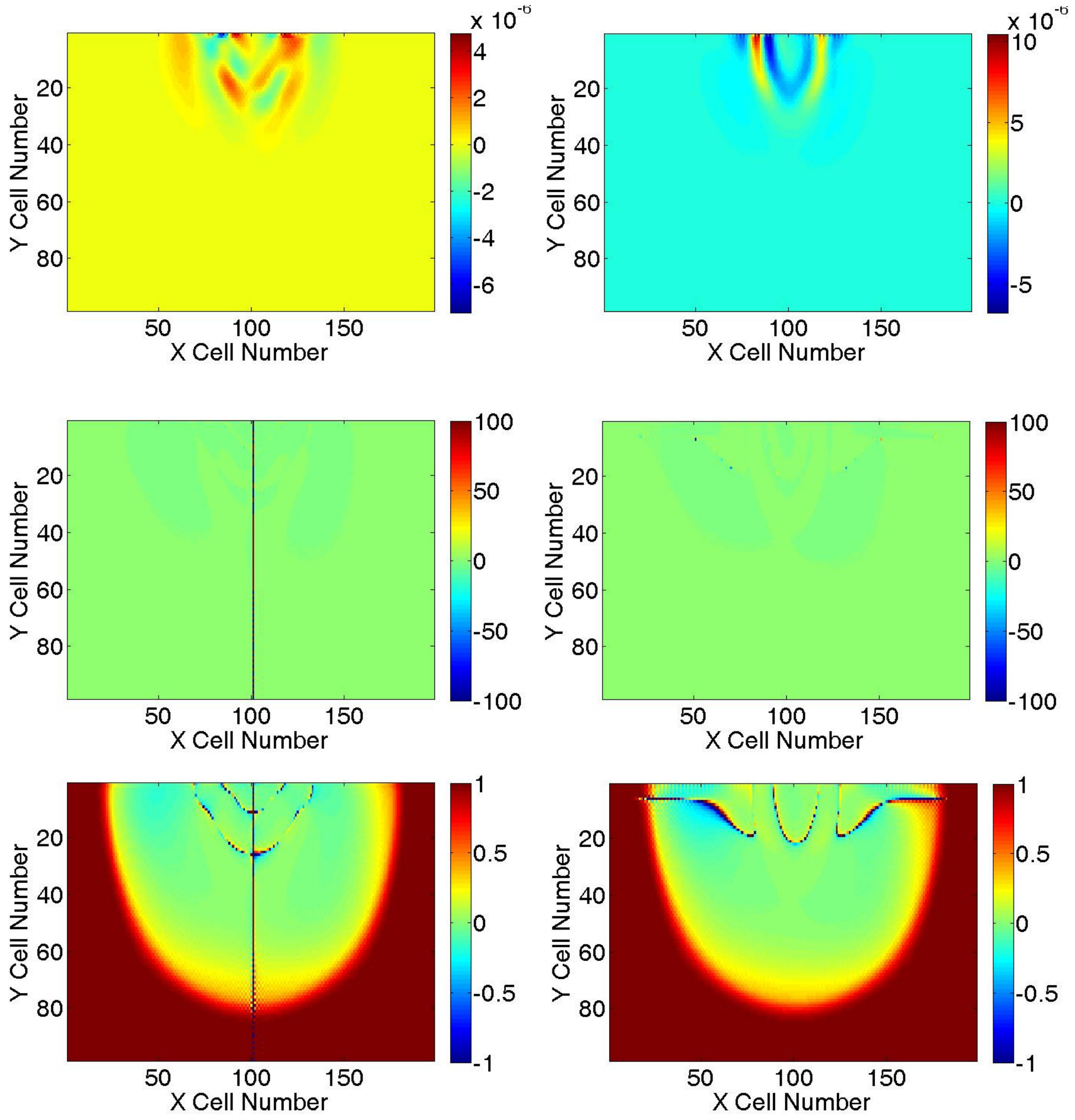


Figure 15: The absolute error on the order of magnitude of the output(top), the relative error from -100% to 100%(middle), the relative error from -1% to 1% (bottom).

The absolute error shown in Figure 15 indicates very good agreement between the squares case and the triangles case, as does the relative error. The majority of the difference is less than 1%, particularly in the area of the blast. Now it can clearly be seen that the largest discrepancies occur where values are transitioning though zero. This is not surprising since precision can affect big discrepancies in the small numbers, the need to average, and the loss of symmetry in both the loading and the domain. Larger error on the left side of the plots than on the right indicates a loss of symmetry. Interestingly, the greatest region of large difference is where the waveform has yet to propagate. Because the percent error was calculated by subtracting the averaged triangles from the square case, the plot indicates that the square case has higher values than the triangular case, indicating that the triangular case is closer to the true value. This is likely to be due to resolution rather than geometry. The most important thing is that the results show that there is adequate agreement with non-regular triangular cells and that the method could be adapted with confidence to an arbitrary mesh. Loading cases and resolution both show sensitivity, but this is something all models and modelers must take into account.

CA versus Finite Element

While the triangular case illustrated that object-oriented triangular cells gave accurate results, the case was still based on a domain being broken up into a uniform grid of cells. To look at arbitrarily shaped cells a COMSOL mesh was read into the CA code, which can handle arbitrary geometry because the object-oriented approach requires no dependence on grid location, as is depends instead on determining neighbors. COMSOL also generated the benchmark results, allowing the results to be based on the same mesh. CA computed and stored each element as a triangular cell along with its cell properties, references to neighboring cells, and distance to neighbors. Load cells were determined by whether a point was found inside the boundaries of a cell; thus exact agreement with COMSOL cannot be obtained since it requires the load to be at a node. Note that while the cells collectively covered the domain, the domain parameters themselves were never stored.

COMSOL and CA results were post-processed in Matlab. The centroid of the cell was output along with the *x* (or *y*) displacement, though Matlab requires a uniform mesh to plot a 3-D color mapping of the results. The uniform mesh was created using COMSOL dimensions and user defined spacing with linear interpolation of points. The actual data points were then also plotted in green. Loading parameters are given in Table 1 for both harmonic and boundary loading.

Table 1: Loading parameters for COMSOL vs. CA comparisons

Loading parameters	Harmonic Loading	Boundary Loading
E	200	2e8
ν	.25	.33
ρ	4000	7850
load location	middle of domain (0,-2.5)	left boundary ($\theta = \pi$)
load	100 sin(.6*t)	$\sigma = 1000$
time of comparison	t = 16	t = .02

Harmonic Loading

There is good agreement in shape and magnitude shown in Figure 16 as values from COMSOL are .0712 m (max) and -.0713 m (min), while CA yields .0721 m and -.0733

m. The relative difference in the results is 2.81%. There is a very minor loss of symmetry due in large part to the point loading. COMSOL has improved symmetry but also has many more degrees of freedom for which it solves: 85546 compared to the 42472 elements used in CA, double the amount of precision.

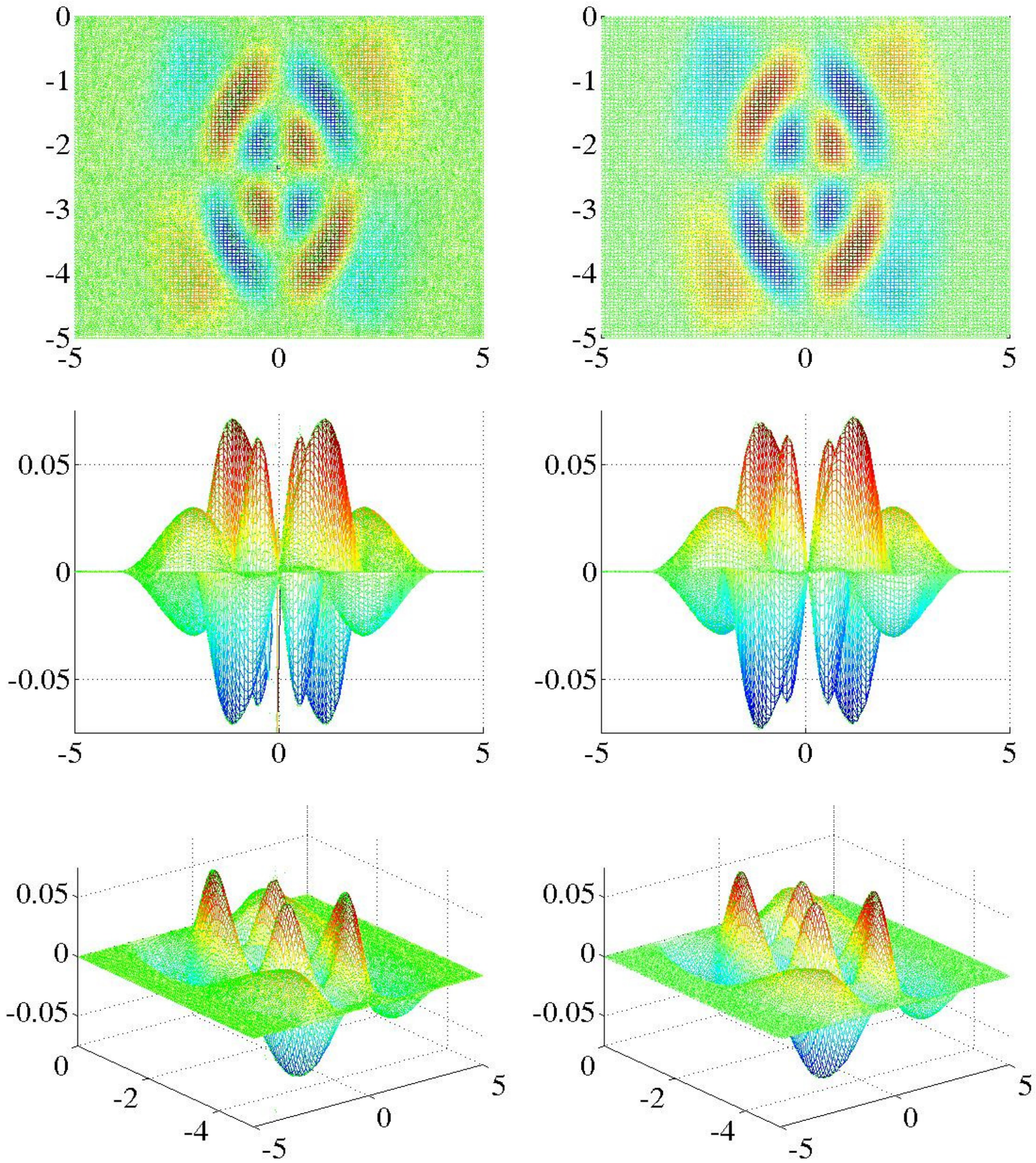


Figure 16: Visual comparisons of COMSOL FE (left) vs. CA (right) for harmonic middle of domain point load showing three different orientations of the output.

Initial Boundary Stress Response

For the boundary traction load case a rectangular domain with a circular hole was used. This allows for comparison of how CA solves irregular geometries which require an irregular mesh and thus an arbitrary rule set. The same loading condition can be set in both COMSOL and CA and is applied at the boundary. Opposite the loaded boundary is a fixed boundary with all other boundaries being free. The mesh size used was 1010 elements which corresponds to 4244 degrees of freedom that COMSOL solves for.

Plots of the results shown in Figure 17 illustrate very good agreement between CA and COMSOL results. Comparisons of the maximum values, 1.518 e-4 m for COMSOL and 1.515e-4 m for CA, yield less than 1% relative difference. The slope of the plot and the variance in the displacement show very good agreement. Of primary interest is the area around the hole, since uniform grid analysis does not allow for that shape. Due to the more complex geometry there is a gradual increase in resolution, which is also not something that can be done readily with a grid based scheme. However, it can be seen that CA does a good job of modeling this with arbitrary geometry. Plots with meshes of varying degrees were also compared, but due to the simplicity of the load case little difference was seen. The most apparent difference is the number of data points solved seen; COMSOL, in addition to solving for more degrees of freedom, also interpolates values for plotting purposes, so it outputs approximately 8x the number of data points.

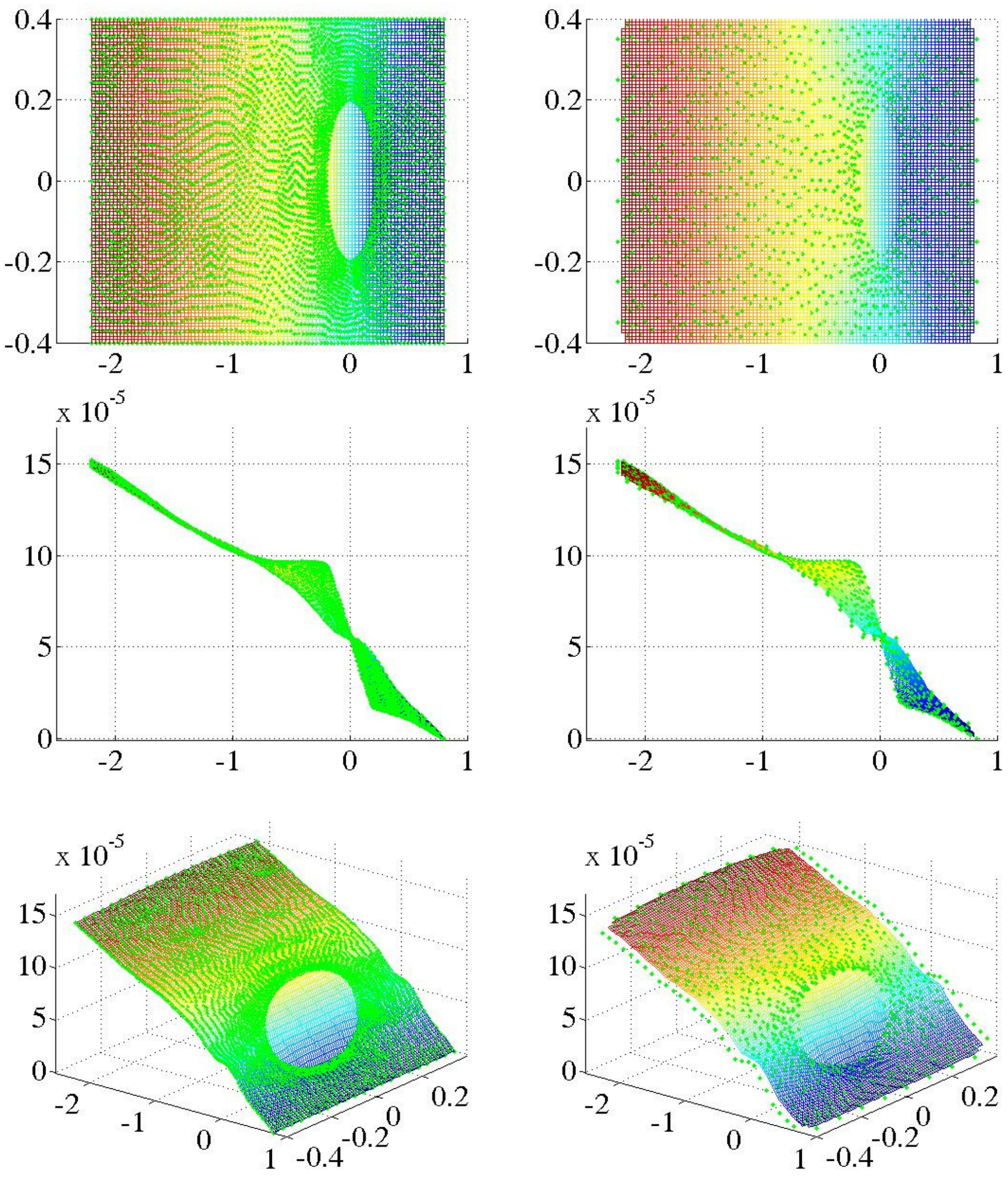


Figure 17: Views of boundary stress response at t= 0.02 for COMSOL FE (left) and CA (right)

Further discussion of CA vs. COMSOL

The comparisons of the two methods yielded favorable results and brought to light some additional considerations including the role of numerical damping in the system, the regularity of the mesh, and the handling of sharp discontinuities.

For stability purposes COMSOL includes some algorithmic damping; this can be adjusted by the user. Since there was no damping in the CA algorithm, material damping or otherwise, damping had to be adjusted in COMSOL. Even with material damping turned off COMSOL includes some algorithmic damping for stability purposes; this can also be adjusted by the user. The default value is set so that when working with the wave equation the algorithm is unconditionally stable. A finer time step and higher degree of the interpolating polynomial will decrease the effect of numerical damping.

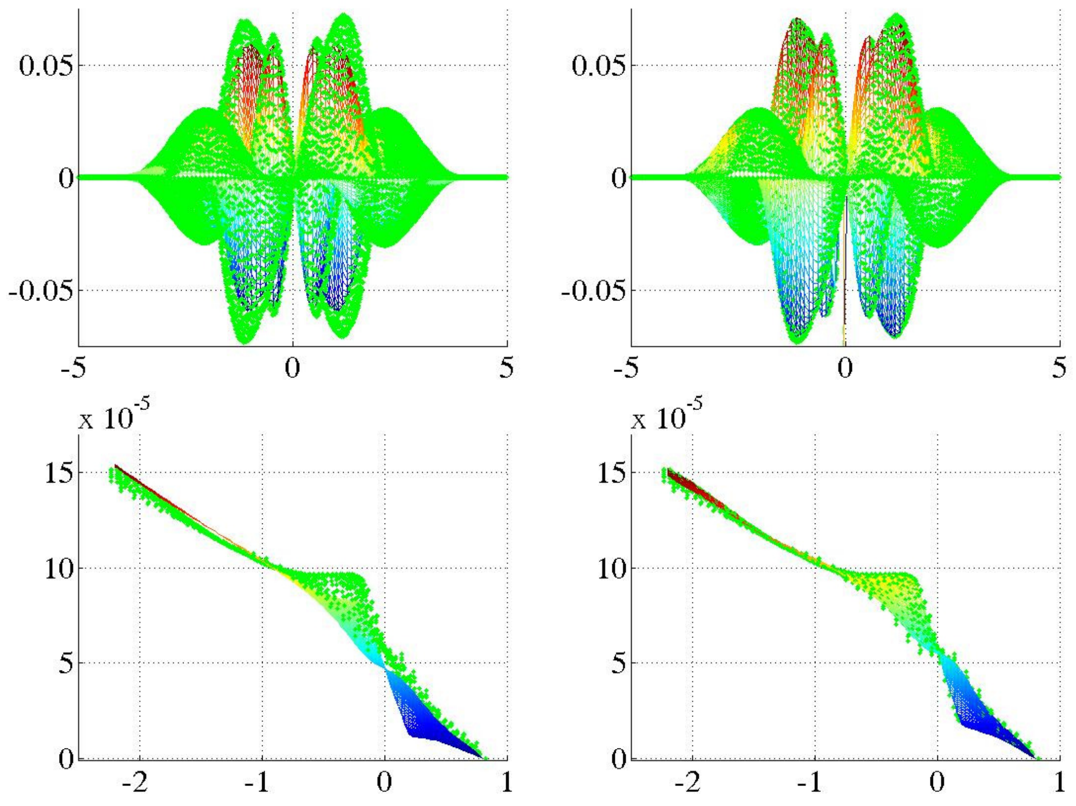


Figure 18: CA data points superimposed over COMSOL FE mesh with numerical damping (left) without numerical damping (right) for both harmonic (above) and boundary loading (below)

The comparison in Figure 18 is with CA data points superimposed over a colored mesh based on COMSOL data, specifically to compare CA with the algorithmic damping in

COMSOL unadjusted. As can be seen the damping has a significant effect on the solution. The shape of the response isn't affected in the harmonic case but a large difference in the magnitude is seen. This visualization also helps to indicate the agreement of the two methods when damping isn't present. For the boundary loading case in addition to the magnitude, there is a difference in shape. Algorithmic damping indicates the end experiences uniform displacement, whereas with less damping more variance is present in the end. CA and COMSOL show good agreement with respect to displacement along the domain as well. This did cause a bit of instability in COMSOL. For the same resolution in time stepping the finest mesh COMSOL allows in the harmonic loading case began to exhibit some instability and a less refined mesh had to be used in the boundary loading case. While an in depth stability analysis was not performed for CA, and this was deliberately forcing COMSOL to allow the likelihood of instability, this result does indicate that CA has the potential to be as stable as certain finite element methods with respect to the wave equation.

Another thing to consider is the regularity of the mesh itself, which is illustrated in Figure 19. The green dots indicate cell centroids, and the squares are the Matlab-generated regular mesh. COMSOL solves using an interpolation scheme which benefits from regularity and uniform elements but does not require them. CA has been shown to have improved results with more regularity in its elements. The red ovals in Figure 19 indicate areas with a low concentration of centroids, and the blue ovals indicate high concentrations. Many squares have three to four centroids per location, while other squares have zero to one. The mesh is perfectly regular, while the dispersion of centroids is not. The symmetry and accuracy of the model would improve with an improved meshing algorithm that could enforce more regularity in the cell geometry. Even with a non ideal mesh, CA exhibits very good performance with good agreement while having less resolution and simpler algorithms than COMSOL.

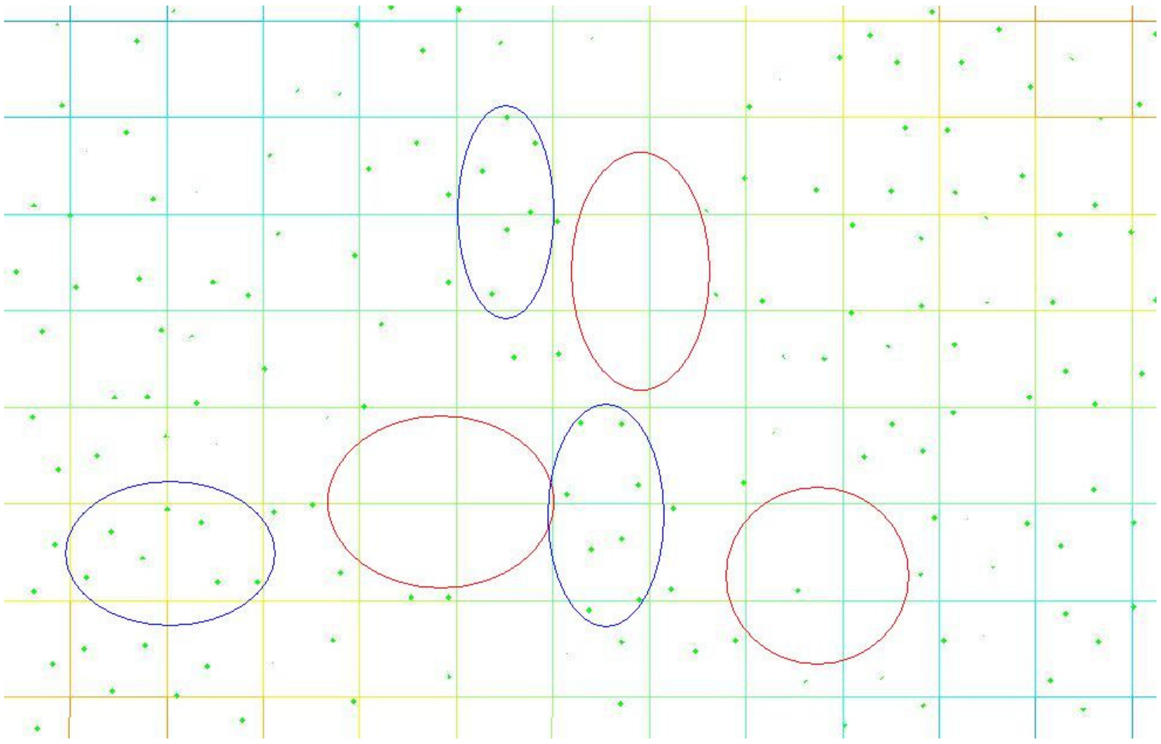


Figure 19: Expanded version of the centroids of the elements compared with a regular mesh created by Matlab.

The other type of boundary condition that can be applied is displacement. This differs greatly in that now a sharp discontinuity occurs at the boundaries. It can be argued that this is an unnatural application, with one end starting out at unit displacement while points immediately adjacent start out at zero. However, this application is also representative of how wave propagation works, in that a particle will have become part of a wave yet a neighbor particle will experience no displacement at all until the wave front has reached it. This is the problem with interpolating wave propagation as it anticipates the wave and does not allow for discontinuities across the domain.

The loading parameters used to generate the results in these figures are the same as those used in the initial stress boundary condition with the exception of the loading, which is changed to x = 1 (unit displacement). Several time steps were looked at, including the first time step at *t* = .0001 seconds shown in Figure 20. COMSOL shows a large negative value, which intuitively is incorrect since a positive displacement will not yield a negative displacement immediately next to it. CA does not yield negative values since the computation does not rely on smoothness; instead it can be considered as a piecewise continuous algorithm.

Further analysis shows that the negative values that occur in COMSOL are not uniquely due to interpolation of data points as illustrated in Figure 20 during later time steps. At time .008 negative values are still visible. The shapes of the above waves differ slightly since in this case the numerical damping was not minimized in order to maintain stability; the complexity of the plot is due to the impulse of the excitation exciting all frequencies. The results indicate that the CA approach is able to handle at once the discontinuities of wave propagation without the ringing that occurs in other methods and the complex geometry that requires a non-uniform mesh. This occurrence of not being able to handle sharp discontinuities without overshoot is known as Gibbs phenomenon and is a common problem in FE modeling especially with wave propagation. While other results show good agreement between CA and other solving tools Figure 20 demonstrates why preference would be given to the CA method. This is again with fewer degrees of freedom than solved for with COMSOL.

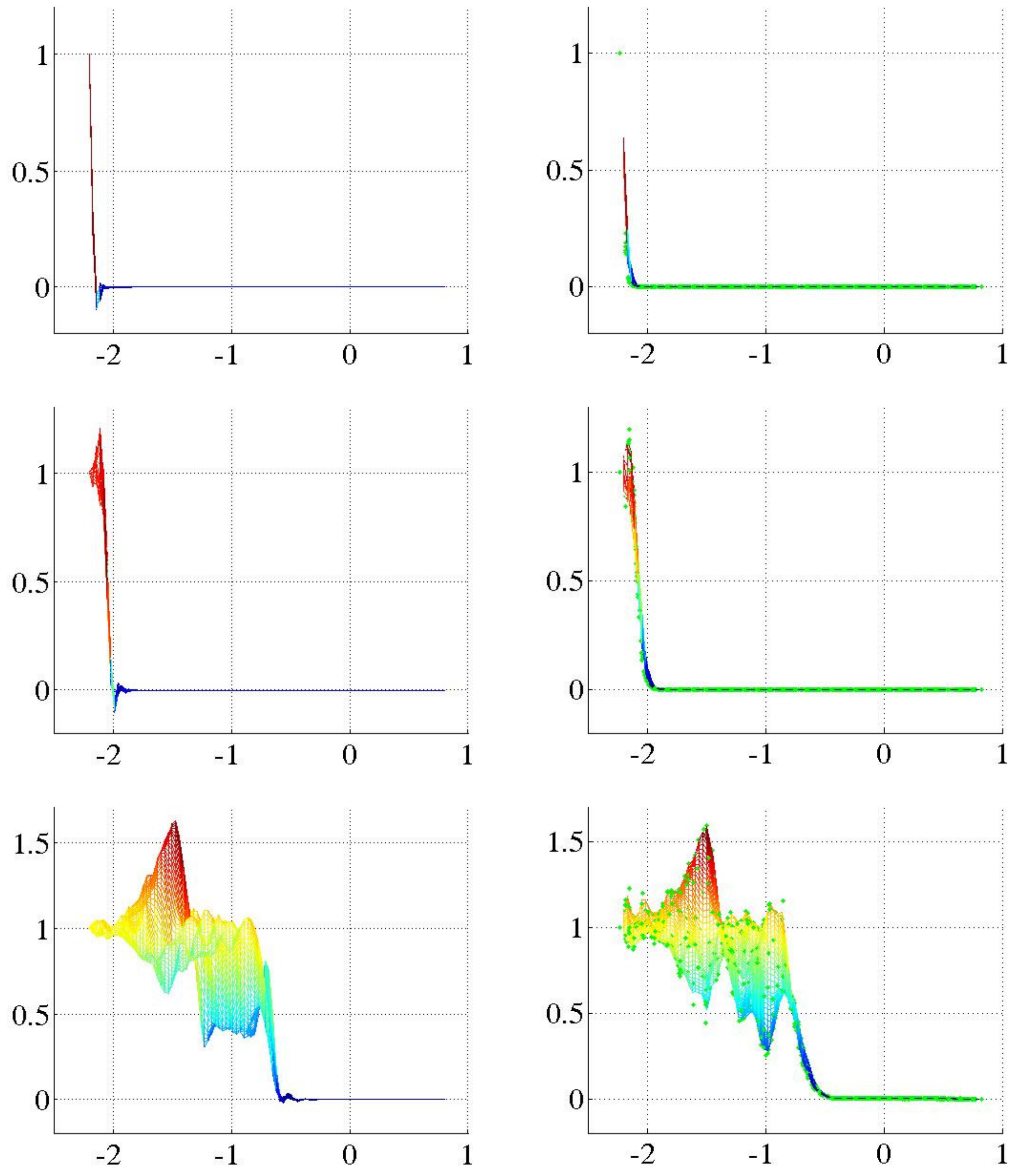


Figure 20: Unit boundary displacement response from COMSOL FE (left) and CA (right) at t = .0001 s (top), t = .0008 s (middle), t = .008 s (bottom).

While COMSOL is just one of many FE codes the results do indicate favorable results from CA. Given that the method was run on a mesh created for use with a FE method, solving for fewer degrees of freedom, it produced comparable results without the use of algorithmic damping and avoided errors resulting from Gibbs phenomenon; not universal problems in FE but common ones.

CHAPTER 4

CONCLUSIONS

The CA method is an extremely viable and adaptable method for analyzing elastodynamics. Elements work well as a cell class in OO programming that allows for the storing of the local cells' state and no global domain parameters. OO implementation also makes it possible to adapt to arbitrary geometries and non-uniform elements that are not grid-dependent using the formulations and equations derived for irregular cell shapes. The more general equations also recover perfectly the formulation for rectangular shapes. When the domain was broken up into triangular shapes the results were on the order of 1% relative error with only slight loss of symmetry due to geometry.

When compared to COMSOL, the CA implementation was able to use the same mesh and obtain favorable results with less complex equations and fewer degrees of freedom. The preliminary qualitative comparison indicates that both the harmonic load case and the applied boundary stress yield results that agree in shape and magnitude. This indicates correct formulation of the domain cell and properly applied loads at the boundaries. Interpolation is overcome by an increase in resolution, and symmetry is recovered. This translates to improved response regarding discontinuities inherent in wave propagation. CA does not exhibit the Gibbs phenomenon for suddenly applied displacements and does not anticipate the waves like interpolation. The accuracy of the results coupled with the efficiency available through parallelization and adaptability of the formulation to arbitrary geometries and possible multi-resolution domains makes CA an attractive option as a computational solver in the realm of elastodynamics.

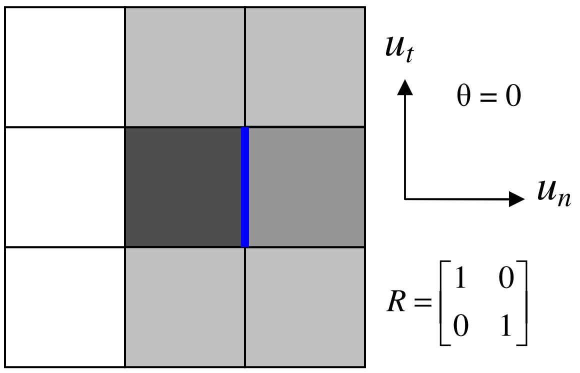
Future work

To further the utility and application of the CA method work could be done to make the code able to model multi resolution, incorporate fatigue, fracture including crack growth, and nonlinearities. Also within the scope of the work would be to include CA code to solve fluid dynamics so that a complex system involving fluid-structure interaction could be solved via the running of one code. This could easily be implemented via boundary cells and boundary cell interactions. Also possible is expanding the code to three dimensional capabilities.

APPENDIX A

RECOVERING OF GRID NOTATION

The following is a face by face analysis deriving the force equations when the rule set for arbitrary geometry is applied to a square index notation.



$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Right Face:

$$\begin{bmatrix} F_x \\ F_y \end{bmatrix} = \begin{bmatrix} F_n \\ F_t \end{bmatrix} \text{ and } \begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} u_x \\ u_y \end{bmatrix}$$

$T = (i, j), N = (i+1, j),$

$M_{T+} = (i, j+1), M_{T-} = (i, j-1),$

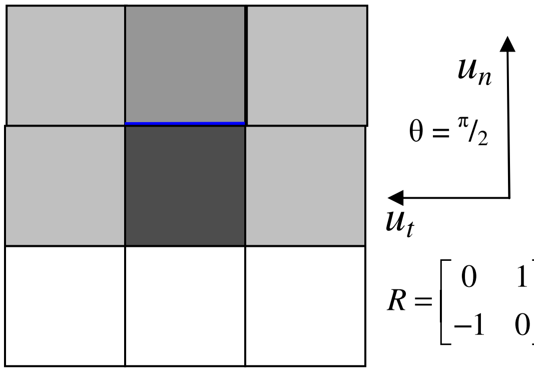
$M_{N+} = (i+1, j+1), M_{N-} = (i+1, j-1)$

$l = \Delta y, r_{typeI} = \Delta x, r_{typeII} = \Delta y$

$$F_x = w\Delta y\left((\lambda + 2\mu)\frac{{}_{(i+1,j)}u_x - {}_{(i,j)}u_x}{\Delta x} + \frac{\lambda}{2}\left(\frac{{}_{(i,j+1)}u_y - {}_{(i,j-1)}u_y}{2\Delta y} + \frac{{}_{(i+1,j+1)}u_y - {}_{(i+1,j-1)}u_y}{2\Delta y} \right)\right)$$

$$F_y = w\Delta y\frac{\mu}{2}\left(\frac{{}_{(i+1,j)}u_y - {}_{(i,j)}u_y}{\Delta x} + \frac{1}{2}\left(\frac{{}_{(i,j+1)}u_x - {}_{(i,j-1)}u_x}{2\Delta y} + \frac{{}_{(i+1,j+1)}u_x - {}_{(i+1,j-1)}u_x}{2\Delta y} \right)\right)$$

Top Face:



$$R = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} F_x \\ F_y \end{bmatrix} = \begin{bmatrix} -F_t \\ F_n \end{bmatrix} \text{ and } \begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} u_y \\ -u_x \end{bmatrix}$$

$T = (i, j), N = (i, j+1),$

$M_{T+} = (i-1, j), M_{T-} = (i+1, j),$

$M_{N+} = (i-1, j+1), M_{N-} = (i+1, j+1)$

$l = \Delta x, r_{typeI} = \Delta y, r_{typeII} = \Delta x$

$$F_x = -w\Delta x\frac{\mu}{2}\left(\frac{{}_{(i,j+1)}{-u_x} - {}_{(i,j)}{-u_x}}{\Delta y} + \frac{1}{2}\left(\frac{{}_{(i-1,j)}u_y - {}_{(i+1,j)}u_y}{2\Delta x} + \frac{{}_{(i-1,j+1)}u_y - {}_{(i+1,j+1)}u_y}{2\Delta x} \right)\right)$$

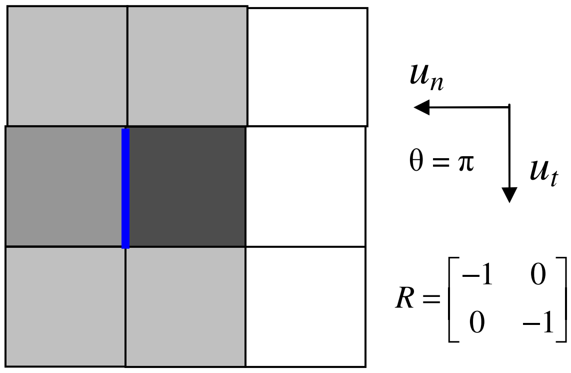
$$F_y = w\Delta x\left((\lambda + 2\mu)\frac{{}_{(i,j+1)}u_y - {}_{(i,j)}u_y}{\Delta y} + \frac{\lambda}{2}\left(\frac{{}_{(i-1,j)}{-u_x} - {}_{(i+1,j)}{-u_x}}{2\Delta x} + \frac{{}_{(i-1,j+1)}{-u_x} - {}_{(i+1,j+1)}{-u_x}}{2\Delta x} \right)\right)$$

This becomes:

$$F_x = w\Delta x \frac{\mu}{2}\left(\frac{{}_{(i,j+1)}u_x - {}_{(i,j)}u_x}{\Delta y} + \frac{1}{2}\left(\frac{{}_{(i+1,j)}u_y - {}_{(i-1,j)}u_y}{2\Delta x} + \frac{{}_{(i+1,j+1)}u_y - {}_{(i-1,j+1)}u_y}{2\Delta x} \right)\right)$$

$$F_y = w\Delta x \left((\lambda + 2\mu) \frac{{}_{(i,j+1)}u_y - {}_{(i,j)}u_y}{\Delta y} + \frac{\lambda}{2}\left(\frac{{}_{(i+1,j)}u_x - {}_{(i-1,j)}u_x}{2\Delta x} + \frac{{}_{(i+1,j+1)}u_x - {}_{(i-1,j+1)}u_x}{2\Delta x} \right)\right)$$

Left Face:



$$R = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

$\begin{bmatrix} F_x \\ F_y \end{bmatrix} = \begin{bmatrix} -F_n \\ -F_t \end{bmatrix}$ and $\begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} -u_x \\ -u_y \end{bmatrix}$

$T = (i, j), N = (i-1, j),$

$M_{T+} = (i, j-1), M_{T-} = (i, j+1),$

$M_{N+} = (i-1, j-1), M_{N-} = (i-1, j+1)$

$l = \Delta y, r_{typeI} = \Delta x, r_{typeII} = \Delta y$

$$F_x = -w\Delta y \left((\lambda + 2\mu) \frac{{}_{(i-1,j)}{-u}_x - {}_{(i,j)}{-u}_x}{\Delta x} + \frac{\lambda}{2}\left(\frac{{}_{(i,j-1)}{-u}_y - {}_{(i,j+1)}{-u}_y}{2\Delta y} + \frac{{}_{(i-1,j-1)}{-u}_y - {}_{(i-1,j+1)}{-u}_y}{2\Delta y} \right)\right)$$

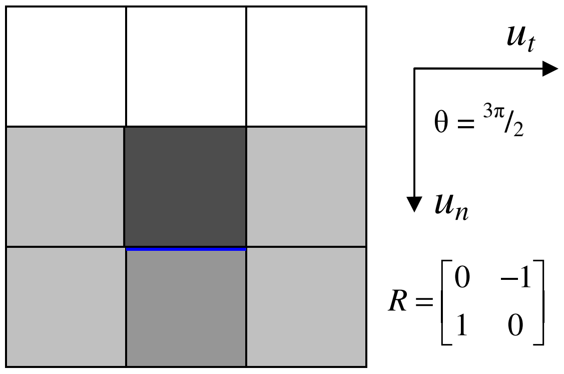
$$F_y = -w\Delta y \frac{\mu}{2}\left(\frac{{}_{(i-1,j)}{-u}_y - {}_{(i,j)}{-u}_y}{\Delta x} + \frac{1}{2}\left(\frac{{}_{(i,j-1)}{-u}_x - {}_{(i,j+1)}{-u}_x}{2\Delta y} + \frac{{}_{(i-1,j-1)}{-u}_x - {}_{(i-1,j+1)}{-u}_x}{2\Delta y} \right)\right)$$

This becomes

$$F_x = w\Delta y \left((\lambda + 2\mu) \frac{{}_{(i-1,j)}u_x - {}_{(i,j)}u_x}{\Delta x} + \frac{\lambda}{2}\left(\frac{{}_{(i,j-1)}u_y - {}_{(i,j+1)}u_y}{2\Delta y} + \frac{{}_{(i-1,j-1)}u_y - {}_{(i-1,j+1)}u_y}{2\Delta y} \right)\right)$$

$$F_y = w\Delta y \frac{\mu}{2}\left(\frac{{}_{(i-1,j)}u_y - {}_{(i,j)}u_y}{\Delta x} + \frac{1}{2}\left(\frac{{}_{(i,j-1)}u_x - {}_{(i,j+1)}u_x}{2\Delta y} + \frac{{}_{(i-1,j-1)}u_x - {}_{(i-1,j+1)}u_x}{2\Delta y} \right)\right)$$

Bottom Face:



$$R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

$\begin{bmatrix} F_x \\ F_y \end{bmatrix} = \begin{bmatrix} F_t \\ -F_n \end{bmatrix}$ and $\begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} -u_y \\ u_x \end{bmatrix}$

$T = (i, j), N = (i, j-1),$

$M_{T+} = (i+1, j), M_{T-} = (i-1, j),$

$M_{N+} = (i+1, j-1), M_{N-} = (i-1, j-1)$

$l = \Delta x, r_{typeI} = \Delta y, r_{typeII} = \Delta x$

$$F_x = w\Delta x \frac{\mu}{2}\left(\frac{{}_{(i,j-1)}u_x - {}_{(i,j)}u_x}{\Delta y} + \frac{1}{2}\left(\frac{{}_{(i+1,j)}{-u_y} - {}_{(i-1,j)}{-u_y}}{2\Delta x} + \frac{{}_{(i+1,j-1)}{-u_y} - {}_{(i-1,j-1)}{-u_y}}{2\Delta x}\right)\right)$$

$$F_y = -w\Delta x\left(\left(\lambda + 2\mu\right)\frac{{}_{(i,j-1)}{-u_y} - {}_{(i,j)}{-u_y}}{\Delta y} + \frac{\lambda}{2}\left(\frac{{}_{(i+1,j)}u_x - {}_{(i-1,j)}u_x}{2\Delta x} + \frac{{}_{(i+1,j-1)}u_x - {}_{(i-1,j-1)}u_x}{2\Delta x}\right)\right)$$

This becomes

$$F_x = w\Delta x \frac{\mu}{2}\left(\frac{{}_{(i,j-1)}u_x - {}_{(i,j)}u_x}{\Delta y} + \frac{1}{2}\left(\frac{{}_{(i-1,j)}u_y - {}_{(i+1,j)}u_y}{2\Delta x} + \frac{{}_{(i-1,j-1)}u_y - {}_{(i+1,j-1)}u_y}{2\Delta x}\right)\right)$$

$$F_y = w\Delta x\left(\left(\lambda + 2\mu\right)\frac{{}_{(i,j-1)}u_y - {}_{(i,j)}u_y}{\Delta y} + \frac{\lambda}{2}\left(\frac{{}_{(i-1,j)}u_x - {}_{(i+1,j)}u_x}{2\Delta x} + \frac{{}_{(i-1,j-1)}u_x - {}_{(i+1,j-1)}u_x}{2\Delta x}\right)\right)$$

APPENDIX B

PSEUDOCODE

Key:

BOLD CAPS	class
↳	calls a method from same class
➪	calls method from a different class
-	new method
CAPS	programming call out(i.e. for, if, else)
()	what is passed to the method

CELL

```
        Stores all cell parameters (ID, updated, $\lambda$, $\mu$, $\rho$, area, width, previous and new
        displacement, previous and new Velocity, stress, angles, centroid, verticies, face
        length, neighbors, r values).
    -   step(  $\Delta$t, (and external force) )
            ↳ calculateNeigborForces()
            IF applicable adds in external force
            $v_n = v_p + F/\rho A\Delta t$
            $d_n = d_p + v/\Delta t$
            *updated* = true
    -   buffer()
            $v_p = v_n$                 Switches old and new displacements
            $d_p = d_n$
            *updated* = false
```

- calculateTotalNeighborForces()

    IF: neighbor is updated

        $\sigma_t = - \sigma_t$

        $\sigma_n = - \sigma_n$

    ELSE: performs stress calc found in paper (obtains neighbors displacements, rotates them, calculates strains, stresses, and in turn forces)

- get/set functions for parameters()

DIRICHLET BC extends **CELL**

Stress set during initialization

Updated always = true

- step() *override*

    ↳ calculateNewDisplacementAndVelocity()

- buffer() *override*

    same except updated = true

- calculateNewDisplacementAndVelocity()

    performs update calculation set forth in boundary discussion

NEUMANN BC extends **CELL**

Displacement set on initialization

- step() *override*

    does nothing (displacement not dependent on neighbors never calculates or stores stress)

TRIANGULAR CELL extends **CELL**

Restricts cells to three faces (all parameter matrices are sized accordingly)

MESH INPUT class

```
    Material parameters $\lambda$, $\mu$, $\rho$
    loadCellIndex
    SetupList
    filename
-   setCellList()
        ↳ comsolMeshFileReader ()
        ↳ createTriangularCellElements ()
        ↳ determineLoadCell ()
        ↳ writeBoundaryVisualization ()
        return SetupList
-   getLoadCellIndex()
        returns loadCellIndex
-   comsolMeshFileReader()
        open “filename”.mphtxt
        parses file to get node matrix and element matrix
-   createTriangularCellElements ()
        FOR: each element
            gets nodal information
            using nodal information find centroid, area, ↳ faceAngles (), facelengths
            sets geometry and cell properties $\lambda$, $\mu$, $\rho$
            adds Cell to SetupList
        ↳ determineNeighborsAndBoundaryConditions ()
        ↳ setupNeighbors ()
        ↳ determineRvalues ()
-   calculateFaceAngles ()
        calculates face angles from centroid and node information
-   determineNeighborsandBoundaryConditions ()
        For: each element
            sorts through element matrix and adds as a neighbor any cell that shares 2
            nodes
            IF no cell shares 2 nodes ↳ addBoundaryCell
-   addBoundaryCell()
        adds a cell based on certain parameters specified by user
            ↳  addNeumann ($u_x$, $u_y$)
                -or-
            ↳ addDirichlet ($\sigma_t$, $\sigma_n$)
-   addNeumann ($u_x$, $u_y$)
        creates and adds a equilateral triangle cell setting centroid, face angles,
        lengths, cell properties – stores displacement
-   addDirichlet ($\sigma_t$, $\sigma_n$)
        creates and adds a equilateral triangle cell setting centroid, face angles,
        lengths, cell properties – stores face stress
```

- inAmbit(targetValue, actualValue)

  method that allows for slight variations in computation of angles

- setupNeighbors ()

  passes a matrix of pointers that includes all cells needed for calculation obtained based on neighbor type and the order of face angles which is determined via↳ sort ()

- sort ()

  arranges cells in order based on face angle

- next ()

  0 to 1 to 2 to 0 (increments through the three neighbors in the forward direction used in sorting and setting up neighbors)

- prev ()

  2 to 1 to 0 to 2 (increments through the three neighbors in the reverse direction used in sorting and setting up neighbors)

- determineRValues ()

  sets R values based on cell centroids (now that pointer matrix has been stored) and primary neighbor cell type (boundary cells are treated special since they are created only as von Neumann neighbors and not Moore neighbors)

- determineLoadCell ()

  evaluates load cell based on if load point is within boundaries of cell

- writeBoundaryVisualization ()

creates a file for Matlab to which stores nodes, elements, and color scheme (which is based on cell type) and provides location of centroids for plotting. Visualization is commonly used for debugging

- writeElementMatrix ()

  writes nodes and elements for visualization based on displacement. (color scheme is created by simulation but element and node matrices are from setup)

RUN_TRIANGLE_SIMULATION

```
    iteration, maxIterations, stepsPerUnitTime
    loadingParameters, cellList, loadCellIndex
    openOutputFiles
-   main()
        ↳ SetupCellArray ()
        ↳ run()
-   setupCellArray()
    ➪ setCellList() from MESHINPUT
    ➪ getLoadCellIndex() from MESHINPUT
-   run()
        while < maxIterations
            ↳ stepThroughCells ()
            ↳ updateValues ()
            ↳ output() – at intervals specified by user
        ↳closeFiles()
```

```
-   stepThroughCells()
        FOR: every cell
            IF: loaded
                    ↳ externalLoad() ➪   step($\Delta t$, externalLoad) from **CELL**
            ELSE:
                    ➪   step($\Delta t$) from **CELL**
-   output()
        Various output profiles depending on user specified visualization
        ↳ mesh3D ()
        ↳ surface()
        ↳ displacement ()
        ↳ velocity()
-   surfacePlot()
        interprets displacements into color vizualization from Red (minimum) to
        Green (average) to Blue (maximum value)
        ↳ writeElementMatrix () from **MESHINPUT**
        RGB files
-   meshout3D()
        writes centroid and $u_x$ or $u_y$ to file
-   closeOutputFiles()
        Closes any output files
-   updateValues()
        ➪   buffer() from **CELL**
```

- externalLoad()

  calls specific loading scenario specified by user

  ↪ harmonic ()/gaussianPulse()/impulse()

- impulseLoad()

  Force suddenly applied and removed at 1 or more iterations

- differentiatedGaussianPulse()

  recreates load case in previous CA paper

- harmonicPulse()

  $F_x = \sin(\omega t/\text{stepsPerUnitTime})$

- displacementWriter()

  time

$$\begin{bmatrix} & j \rightarrow & \\ i & u_x / u_y & \\ \downarrow & & \\ & & \end{bmatrix}$$

  writes a file with values separate in a matrix format for comparisions with previous grid based CA results

- velocityWriter()

  same as above but with velocity

- writeForcingProfile()

  Time

  $F_x \quad F_y$

REFERENCES

1. Leamy, M.J., *Application of cellular automata modeling to seismic elastodynamics.* International Journal of Solids and Structures, 2008. **45**(17): p. 4835-4849.

2. Schroeder, C.T., *On the interaction of elastic waves with buried land mines: an investigation using the finite-difference time-domain method*, in *School of Electrical and Computer Engineering*. 2001, Georgia Institute of Technology: Atlanta.

3. Raghavan, S. and S.S. Sahay, *Modeling the grain growth kinetics by cellular automaton.* Materials Science and Engineering a-Structural Materials Properties Microstructure and Processing, 2007. **445**: p. 203-209.

4. von Neumann, J., *Theory of Self-reproducing Automata*. 1966, Univ. of Illinois Press: Urbana, IL.

5. Gardner, M., *Fantastic Combinations of John Conways New Solitaire Game Life.* Scientific American, 1970. **223**(4): p. 120-&.

6. Raabe, D., *Cellular automata in materials science with particular reference to recrystallization simulation.* Annual Review of Materials Research, 2002. **32**: p. 53-76.

7. Khvastunkov, M.S. and J.W. Leggoe, *Adapting cellular automata to model failure in spatially heterogeneous ductile alloys.* Scripta Materialia, 2004. **51**(4): p. 309-314.

8. Delsanto, P.P., et al., *Connection machine simulation of ultrasonic wave propagation in materials. II: The two-dimensional case.* Wave Motion, 1994. **20**: p. 9.

9. dos Santos, R.M.Z. and S. Coutinho, *Dynamics of HIV infection: A cellular automata approach.* Physical Review Letters, 2001. **8716**(16): p. -.

10. Green, D.G., A. Tridgell, and A.M. Gill, *Interactive Simulation of Bushfires in Heterogeneous Fuels.* Mathematical and Computer Modelling, 1990. **13**(12): p. 57-66.

11. Schreckenberg, M., et al., *Discrete Stochastic-Models for Traffic Flow.* Physical Review E, 1995. **51**(4): p. 2939-2949.

12. Olami, Z., H.J.S. Feder, and K. Christensen, *Self-Organized Criticality in a Continuous, Nonconservative Cellular Automaton Modeling Earthquakes.* Physical Review Letters, 1992. **68**(8): p. 1244-1247.

13. Mora, P. and D. Place, *Stress correlation function evolution in lattice solid elasto-dynamic models of shear and fracture zones and earthquake prediction.* Pure and Applied Geophysics, 2002. **159**(10): p. 2413-2427.

14. Honma, T. and N. Tosaka, *Autonomous decentralized finite element method and its applications.* International Journal for Numerical Methods in Engineering, 2003. **57**(6): p. 853-874.

15. Ryoo, J., et al., *Estimation of Young's modulus of single-walled carbon nanotube using cellular automata.* Advances in Engineering Software, 2007. **38**(8-9): p. 531-537.

16. Simons, N.R.S., et al., *Cellular-Automata as an Environment for Simulating Electromagnetic Phenomena.* Ieee Microwave and Guided Wave Letters, 1994. **4**(7): p. 247-249.

17. Zheng, C.W., et al., *Microstructure prediction of the austenite recrystallization during multi-pass steel strip hot rolling: A cellular automaton modeling.* Computational Materials Science, 2008. **44**(2): p. 507-514.

18. Yang, B.J., L. Chuzhoy, and M.L. Johnson, *Modeling of reaustenitization of hypoeutectoid steels with cellular automaton method.* Computational Materials Science, 2007. **41**(2): p. 186-194.

19. Das, S., et al., *A combined neuro fuzzy-cellular automata based material model for finite element simulation of plane strain compression.* Computational Materials Science, 2007. **40**(3): p. 366-375.

20. Lan, Y.J., et al., *Mesoscale simulation of deformed austenite decomposition into ferrite by coupling a cellular automaton method with a crystal plasticity finite element model.* Acta Materialia, 2005. **53**(4): p. 991-1003.

21. Bernsdorf, J., F. Durst, and M. Schafer, *Comparison of cellular automata and finite volume techniques for simulation of incompressible flows in complex geometries.* International Journal for Numerical Methods in Fluids, 1999. **29**(3): p. 251-264.

22. Krafczyk, M., et al., *Two-dimensional simulation of fluid-structure interaction using lattice-Boltzmann methods.* Computers & Structures, 2001. **79**(22-25): p. 2031-2037.

23. Eugenio, A. and M. Rasetti, *A cellular automaton for elasticity equations.* International Journal of Modern Physics B, 1996. **10**(2): p. 203-218.

24. Rappaz, M. and C.A. Gandin, *Probabilistic Modeling of Microstructure Formation in Solidification Processes.* Acta Metallurgica Et Materialia, 1993. **41**(2): p. 345-360.

25. Setoodeh, S., Z. Gurdal, and L.T. Watson, *Design of variable-stiffness composite layers using cellular automata.* Computer Methods in Applied Mechanics and Engineering, 2006. **195**(9-12): p. 836-851.

26. Hirsekorn, M., et al., *Elastic wave propagation in locally resonant sonic material: Comparison between local interaction simulation approach and modal analysis.* Journal of Applied Physics, 2006. **99**(12): p. -.

27. Rothman, D.H., *Modeling Seismic P-Waves with Cellular Automata.* Geophysical Research Letters, 1987. **14**(1): p. 17-20.

28. Hajela, P. and B. Kim, *On the use of energy minimization for CA based analysis in elasticity.* Structural and Multidisciplinary Optimization, 2001. **23**(1): p. 24-33.

29. Slotta, D.J., et al., *Convergence analysis for cellular automata applied to truss design.* Engineering Computations, 2002. **19**(7-8): p. 953-969.

30. Abdellaoui, M., A. El Jai, and M. Shillor, *Cellular automata model for a contact problem.* Mathematical and Computer Modelling, 2002. **36**(9-10): p. 1099-1114.

31. Kwon, Y.W. and S. Hosoglu, *Application of lattice Boltzmann method, finite element method, and cellular automata and their coupling to wave propagation problems.* Computers & Structures, 2008. **86**(7-8): p. 663-670.

32. Zhong, Y.M., et al., *A cellular neural network methodology for deformable object simulation.* Ieee Transactions on Information Technology in Biomedicine, 2006. **10**(4): p. 749-762.

33. Psakhie, S.G., et al., *Movable cellular automata method for simulating materials with mesostructure.* Theoretical and Applied Fracture Mechanics, 2001. **37**(1-3): p. 311-334.

34. Fabero, J.C., A. Bautista, and L. Casasus, *An explicit finite differences scheme over hexagonal tessellation.* Applied Mathematics Letters, 2001. **14**(5): p. 593-598.

35. Xia, G.H., et al., *A 3D implicit unstructured-grid finite volume method for structural dynamics.* Computational Mechanics, 2007. **40**(2): p. 299-312.

36. Wenke, P. and M.A. Wheel, *A finite volume method for solid mechanics incorporating rotational degrees of freedom.* Computers & Structures, 2003. **81**(5): p. 321-329.

37. Taylor, G.A., C. Bailey, and M. Cross, *A vertex-based finite volume method applied to non-linear material problems in computational solid mechanics.* International Journal for Numerical Methods in Engineering, 2003. **56**(4): p. 507-529.

38. Slone, A.K., C. Bailey, and A. Cross, *Dynamic solid mechanics using finite volume methods.* Applied Mathematical Modelling, 2003. **27**(2): p. 69-87.

39. Psakhie, S.G., et al., *Modeling the behavior of complex media by jointly using discrete and continuum approaches.* Technical Physics Letters, 2004. **30**(9): p. 712-714.