Finite elements on polygon domains in $\mathbb{R}^2$

Alec Thériault
Stefan Dawydiak

May 2, 2013

Science One Program
The University of British Columbia
Vancouver, Canada

Abstract

Partial differential equations with Dirichlet boundary conditions are treated with a Finite Element Method over non-convex non-punctured polygon domains in $\mathbb{R}^2$. A program for generating and optimizing triangular meshes is developed. The model's error is measured against analytic solutions for certain heat and wave equation domains.

1 Introduction to the Finite Element Method

1.1 Overview

A scalar function f of more than one variable takes as its argument a point in at least two dimensional space and associates it with a particular value. For example, a function of two variables takes a position on the xy plane and associates that position with a height z above the position. This leads to the function describing a surface in $\mathbb{R}^3$. It should be plausible that if the surface was continuous and reasonably smooth (i.e. its first partial derivatives were somehow bounded above) that an approximate model surface can be constructed to lie roughly where the actual one does. The Finite Element Method (FEM) in two dimensions is a way of discretizing the xy plane and modelling the surface locally on each sub-domain. By uniting the sub-domains, the whole surface is approximated. This is essentially the Ritz-Gallerkin approximation method [1]. The FEM is a convenient implementation of this strategy, as it allows the same model function to be used on every sub-domain, each distorted and scaled to match the element shape in the xy plane.

1.2 Mathematical and Physical Principles

Let $\overline{\Omega} \in \mathbb{R}^2$ be a domain composed of an interior Ω and a boundary $\partial\Omega$, that is, $\overline{\Omega} = \Omega \cup \partial\Omega$.

Physical quantities of interest, such as temperature, energy, or small particles can be described with a flux $\boldsymbol{\sigma}$ [1]. Then, the flux through a small region of Ω, divided by the area of that region gives an approximate measurement of the extent to which that region is producing or consuming the quantity being described by the flux $\boldsymbol{\sigma}$. Mathematically, this measurement is the divergence of the flux $\nabla \cdot \boldsymbol{\sigma}$. Using this notation,

$$\nabla \cdot \boldsymbol{\sigma} = f(x, y) \tag{1}$$

where f describes sources or sinks at any $(x, y) \in \Omega$. Physically, this argument is at the root of any conservation law: the net change of any conserved quantity in a given region is a sum of the flows, taken over the boundary of the region. Mathematically, the argument is a form of Stoke's Theorem. We are claiming by equation (1) that, if nothing is being added or consumed in some region, as much flows in as flows out. The FEM, therefore, is plausibly suited to numerical approximation of both divergences $(\nabla \cdot u)$ and of gradients (∇u), and, by extension, to all differential operators.

1.3 Novel Meshes and Flexible Methods

Using shape functions ψ composed of Lagrangian polynomial families, we developed approximating functions u_h on arbitrary non-punctured domains in $\mathbb{R}^2$. Two largely separate steps are involved: the development of a mesh-generating discretization program and the implementation of the FEM on the resultant meshes.

The program *Simulacron*, which we designed from ground up to test our model, automatically creates the mesh and implements our modified matrix statement of the FEM on it. This allows us to approximate partial derivatives (and differential operators in general) over each element in the mesh. Special care is taken to preserve the generality of the triangulation algorithm, allowing treatment of domains containing cracks, chokes and regions of non-convexity. Several such meshes are presented at the end of §2 and in Appendix E.

2 Triangulation

2.1 Mesh Generation

The topic of mesh generation addresses the geometrical subdivision of a potentially irregular domain of interest $\Omega \in \mathbb{R}^2$ into smaller sub-domains Ω_T, triangles. The choice of triangles as basic shapes for the elements is motivated by the fact that a triangle is the only polygon whose sides can never intersect one another anywhere. (In this paper, we will treat the boundary of a polygon as being composed of a set of vertices and segments. Segments do not include the vertices at their extremities.) This ensures that any triangle sub-domain Ω_T generated is "valid". Extending the idea of validity to any particular way of subdividing the domain, we define a valid subdivision as one in which the union of all sub-domains Ω_i yields the initial domain Ω, and the intersection of any pair of sub-domains Ω_i and Ω_j is empty. In other words, we seek a set of subdivisions Ω_T that cover the initial domain once and only once (without holes or overlapping regions)[2]. This can be read as

$$\bigcup_{i=1}^{N} \Omega_i = \Omega \quad \text{and} \quad \bigcup_{1 \leq j \leq N} (\Omega_i \cap \Omega_j) = \emptyset. \tag{2}$$

Let us call such a valid subdivision of a domain Ω into triangles a triangulation $\mathcal{T}_i$. Ideal triangulations minimize the error in the final model. Unfortunately, since it is difficult and perhaps impossible to find the ideal triangulations in most scenarios, we rely instead on a set of general guidelines condensed into algorithm heuristics. Generally, a good triangulation $\mathcal{T}_i$ has triangles Ω_T that fulfil the following criteria:

1. Sufficiently small side length for Ω_T so that interpolation error is correspondingly small, and
2. An aspect ratio of roughly unity (approximately equilateral).

The program *Simulacron* triangulates domains bounded by valid polygons $\mathcal{P}$ by iteratively adding nodes η_i (triangle vertices) inside the domain, contracting the boundary polygon $\mathcal{P}_B$, and repeating the process. *Simulacron* ensures that any added node remains within the boundary polygon and that the triangles created by its addition don't invalidate the triangulation.

There are two types of documented invalidations: (a) points added that are not within the region bounded by $\mathcal{P}_B$ and (b) node connections added which form segments $S_i = \overline{(P_m P_n)}$ intersecting $\mathcal{P}_B$. The limited nature of invalidations is due to the iterative approach of slowly contracting $\mathcal{P}_B$. The two functions checking for these invalidations are discussed in Appendix B.

As a secondary consideration, *Simulacron* attempts to place the nodes in such a way that they satisfy the arbitrary conditions above for a good triangulation. In some situations, the boundary polygon may be split into two distinct boundary polygons. In these cases, both boundaries are dealt with independently.

The triangulation process, in a pseudo-code, is briefly formalized below. The names of the functions, which are each algorithms in themselves, are based on the names of biological enzymes with similar native purposes. The following arguments are needed to initiate triangulation:

1. A polygon $\mathcal{P}$ that encloses Ω. (In other words, the boundary $\partial\Omega = \mathcal{P}$); and

2. An ideal side length $\mathcal{S}$.

LOOP the following instructions while there are still boundary polygons: **IF** the number of vertices on any boundary polygon $\mathcal{P}_B$ is equal to 3, consider $\mathcal{P}_B$ to be a sub-domain Ω_T itself, and remove it from consideration in the following.

- PRIMASE is a function that scans all the sides of the boundary polygons and looks for segments that are too long. Upon finding such segments, points are added on the boundary polygon, dividing them. When all possible splits have been made the next step is undertaken.
- EXCISOR scans consecutive segments $\left(S_1 = \overline{P_{i-1}P_i}\right)$ and $\left(S_2 = \overline{P_iP_{i+1}}\right)$ of the polygon, looking for those which form acute angles ($\angle P_{i-1}P_iP_{i+1} < \pi/2$). When such angles are found, the center vertex P_i is removed from the boundary polygon, provided that the new connection formed between P_{i-1} and P_{i+1} does not invalidate the triangulation. If any points have been added, the loop is restarted.
- NUCLEASE scans every pair of non-consecutive nodes $P_m, P_n \in \mathcal{P}_B$ checking to see if any two such points are within a distance of $d < 1.5\mathcal{S}$ of one another. If there are such points, they are connected (provided their connection does not invalidate the triangulation), and two distinct boundary polygons are thus created. If any boundary polygon is split, the loop is restarted.
- POLYMERASE is the function that adds new nodes inside $\mathcal{P}_B$. It looks at a given set of three consecutive P_i, P_j, P_k on the boundary polygon $\mathcal{P}_B$ and calculates the number of new nodes $\mathbf{Q}_1, \mathbf{Q}_2, \ldots$ to add based on the angle ($\angle P_i, P_j, P_k = \angle A$). The number of nodes placed is evaluated as $n = \text{round}\left(\angle A \cdot 3/\pi\right) - 1$. The nodes are placed such that the lines passing through each of them and P_j differ by the regular angle $\theta = \angle A/(n+1)$. Their distance $||\mathbf{Q}_p - \mathbf{P}_j||$ from P_j is a weighted average of the two segments made of P_i, P_j, P_k and the prescribed side length $\mathcal{S}$. For the p^{th} node $\mathbf{Q}$, where the 1st node is closest to P_i, the distance is

$$||\mathbf{Q}_p - \mathbf{P}_j|| = \frac{1}{2}\left[\left(\frac{||\mathbf{P}_k - \mathbf{P}_i||p + ||\mathbf{P}_i - \mathbf{P}_k||(n-p)}{n}\right) + \mathcal{S}\right]. \tag{3}$$

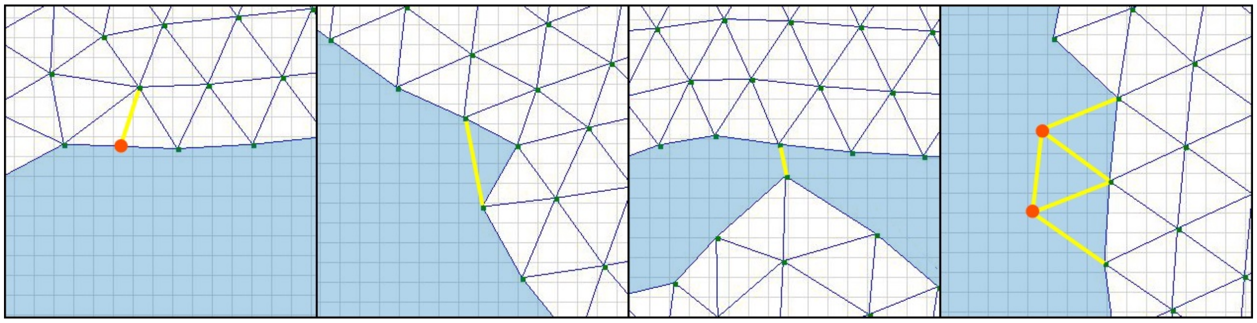


Figure 1: The four main functions involved in constructing the mesh. The interior of the boundary polygon $\mathcal{P}_B$ is shaded light blue. The segments and points added are shaded yellow and orange respectively. Going from left to right: PRIMASE adds points on segments that are too long, EXCISOR removes acute angles, NUCLEASE splits the boundary polygon into two new boundary polygons when two points are too close, and POLYMERASE adds points around a given point on the boundary polygon.

This algorithm, on which *Simulacron* is based, has been designed to triangulate any polygon $\mathcal{P}$ given that the prescribed side length $\mathcal{S}$ is smaller than the segments which make up the sides of $\mathcal{P}$. The rate of triangulation decays rapidly as the boundary polygon side increases, because the loop involves scanning every segment in the polygon at least once.

The program has been tested only up to triangulations $\mathcal{T}$ that generated slightly above 40,000 elements. These limit cases took 7 minutes to run. Triangulations of 6000 elements take under 20 seconds, while triangulations of under 500 elements are virtually instantaneous.

2.2 Mesh Optimization

The next step in triangulation is the optimization of the mesh: a process whereby individual nodes are displaced slightly from their initial positions to improve their aspect ratios [3]. The method used involves considering every node and its neighbours. A score is assigned to the set of triangle elements contained:

$$\text{score} = \frac{s_\theta}{\overline{\theta}} + \frac{s_A}{\overline{A}} \tag{4}$$

where s_θ and s_A are the standard deviations of all the angles or areas contained, and $\overline{\theta}$ and $\overline{A}$ are the average angle and area. The node center to this cluster is displaced slightly up, down, left, and right. Whichever offset yields the largest drop in score is the new position of the node. When none of the potential movements yielded a better score, the node is fixed into place and the algorithm applied to another node. The whole process ends only when the total sum of distance travelled by all the nodes, in one cycle, is less than an arbitrary pre-set threshold.

In *Simulacron* most triangulations (around 500 elements) take a fraction of a second to fully optimize. It has been noticed that the total distance the nodes move in a given cycle initially decreases quadratically. After reaching the vertex of the parabola, the distance decreased in an irregular manner. We do not know what promotes this behaviour. Rarely, in 4% of cases, the total distance measure of a given cycle is slightly higher than the total distance cycle of the previous cycle.

2.3 Local Mesh Refinement

Due to its generality, our method of triangulation lends itself efficiently to local refinement of the mesh. Since the side length $\mathcal{S}$ of the triangles in a given mesh is determined only through the mesh building functions above, different $\mathcal{S}$ values can be specified at different times during the triangulation process. Perhaps the most relevant way of utilizing this is by making $\mathcal{S}(x, y)$ a function of position (thus creating meshes with local refinements).

This type of refinement is highly useful in locally minimizing the error due created via the Euler method (see §3.2) in specific areas of inaccuracy (due to large derivative values) [4][5].

Unfortunately, there is some ambiguity in defining $\mathcal{S}(x, y)$ (since x and y arguments specified could be for any point on $\mathcal{S}$). In *Simulacron*, this argument point is given interchangeably as the start point, or, if the end point is also known, the average of the positions of the start and end points. Despite this drawback, *Simulacron*'s local refinement functionality is capable of generating meshes where the side length $\mathcal{S}$ is regulated as a function of position.

Another problem is that refinements of this sort cannot be optimized using the methodology presented in §2.2 because the algorithm would simply redistribute the nodes throughout the triangulated region, thus defeating the purpose of local refinements. Two seperate means of optimizing, none as efficient as §2.2, were developed catering to the nature of $\mathcal{S}(x, y)$.

1. If the function $\mathcal{S}(x, y)$ is continuous (or discontinuities are small), then only the score function (4) needs to be modified:
$$\text{score} = \frac{S_\theta}{\overline{\theta}} + \frac{1}{N}\sum_{i=1}^{N} \left(\mathcal{S}_i - \mathcal{S}(x, y)\right)^2 \tag{5}$$
2. In the case of local refinement with discrete mesh size choice function, the more appropriate method involves choosing a polygon that follows the boundary of the discontinuity. The nodes contained on this polygon are then kept fixed into place while the remaining nodes can be optimized using the methodology of §2.2.

More mesh samples are included in Appendix E.

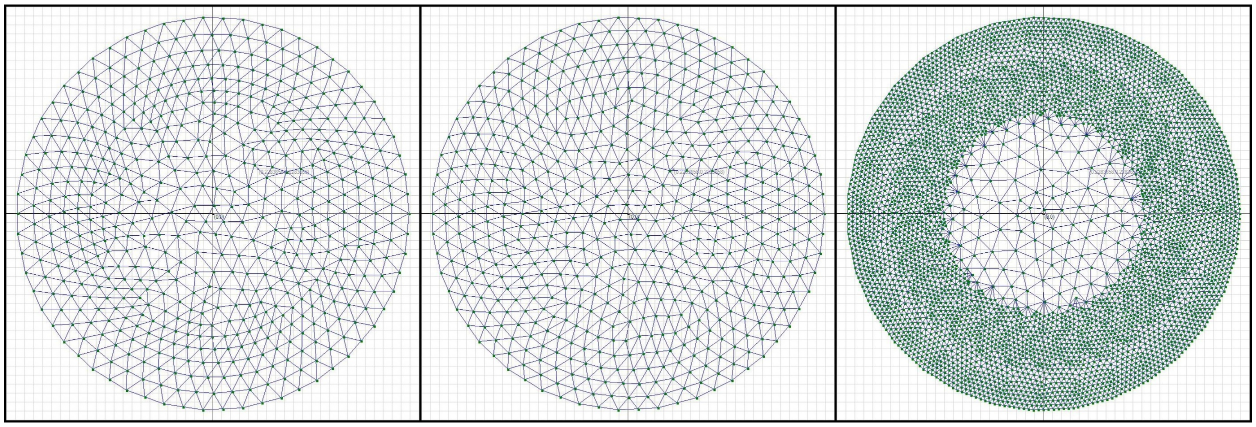


Figure 2: 31-sided polygon ("r"=1). Left: unoptimized, $\mathcal{S} = 0.1$, 546 nodes, 0.153 s. Center: optimized, $\mathcal{S} = 0.1$, 546 nodes, 0.234 s. Right: optimized, $\mathcal{S} = 0.1$ if $r < 0.5$ & $\mathcal{S} = 0.05$ if $r > 0.5$, 4556 nodes, 2.944 s.

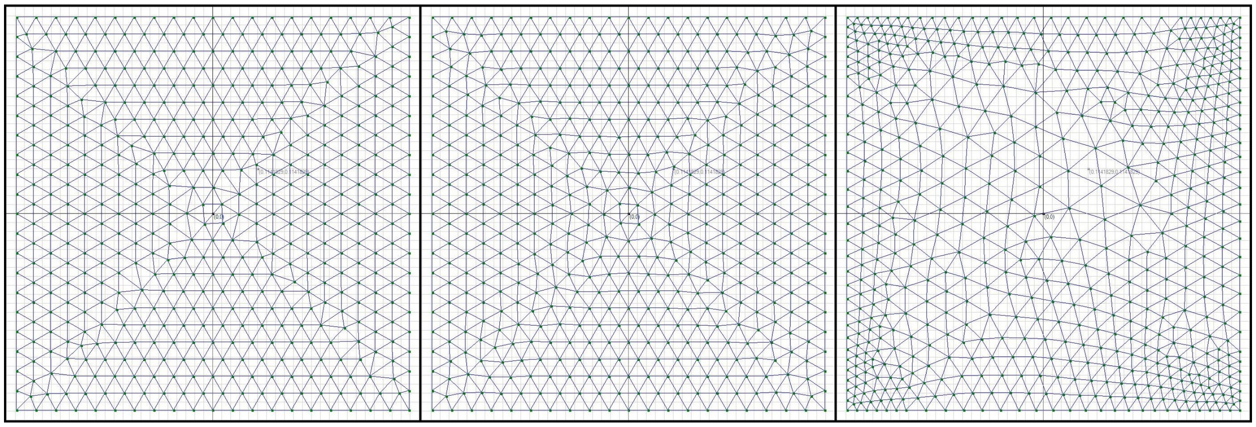


Figure 3: Meshes of a unit square. Left: unoptimized, $\mathcal{S} = 0.05$, 504 nodes, 0.086 s. Center: optimized, $\mathcal{S} = 0.05$, 504 nodes, 0.117 s. Right: optimized, $\mathcal{S}(x, y) = 0.05(0.375 - 2.5(x^2 + y^2))$, 604 nodes, 1.341 s.

3 Finite Element Implementation

In *Simulacron*, the partial differential equation (PDE) modelling is generalized in such a way that *any* system of coupled differential equations with explicit time derivatives can be advanced through time, given appropriate boundary conditions. Our method is not the same as the conventional approach: generally, linear approximations are used, in combination with a modified statement of the weak formulation. However, while this approach avoids many of the problems we encounter, it restricts the range of differential equation systems that can be dealt with.
An example of an equation that can be dealt with is the wave equation, which describes the propagation of energy:

$$u_{tt} = c^2\nabla^2 u \quad \leftrightarrow \quad \begin{bmatrix} u_t \\ v_t \end{bmatrix} = \begin{bmatrix} v \\ c^2\nabla^2 u \end{bmatrix} \quad \leftrightarrow \quad \frac{\partial}{\partial t}\mathbf{u}(x_0, y_0, t) = f\left(\mathbf{u}(x, y, t_0)\right) \tag{6}$$

The idea is to use a time-snapshot $\mathbf{u}(x, y, t_0)$ to evaluate $f\left(\mathbf{u}(x, y, t_0)\right)$. Then, since $\frac{\partial}{\partial t}\mathbf{u}(x_0, y_0, t) = f\left(\mathbf{u}(x, y, t_0)\right)$, we have the derivative with respect to time. This allows us to estimate $\mathbf{u}(x, y, t_1)$. It follows that the model, in order to easily evaluate the function f, it must be able to, at any time-constant snapshot, evaluate a local approximation of any differential operator used in the equation (most PDEs use some combination of the Laplacian $\nabla^2 u$, gradient ∇u, divergence $\nabla\cdot\mathbf{u}$, curl $\nabla\times\mathbf{u}$, and directional derivatives $\nabla_v u$). In order to do this, we consider a separate approximation for $\mathbf{u}(x, y, t_0)$ at each node in the triangulation $\mathcal{T}_i$.

The approach employed utilizes an element matrix and Lagrangian family polynomials to rapidly curve-fit a quadratic polynomial approximation to $\mathbf{u}(x, y, t_0)$. This approximation, being a polynomial, can then easily be manipulated to find the differential operators.

3.1 Determining u_h with Quadratic ψ

In the following we consider approximating $\mathbf{u}(x, y, t_0)$ into a piecewise combination of quadratic $u_{h,\eta}$.

Let us consider a set of quadratic functions with domain in $\mathbb{R}^2$ of the form $\psi(x, y) = A + Bx + Cy + Dx^2 + Exy + Fy^2$. Let us also consider 6 distinct domain points $P_1(x_1, y_1), P_2(x_2, y_2), \ldots, P_6(x_6, y_6)$. It is possible to solve for the coefficients of a quadratic function ψ which is 0 at every point except one, since this will yield us a system of 6 equations with 6 unknowns with a non-zero determinant. We may now consider 6 such functions $\psi_1, \psi_2, \ldots, \psi_6$ each obeying

$$\psi_i(P_j) = \delta_{ij} \text{ for } i, j \in \{1, 2, \ldots, 6\}. \tag{7}$$

Furthermore, we find that the coefficients of the functions can be solved with the following matrix equation:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 & x_5^2 & x_6^2 \\ x_1y_1 & x_2y_2 & x_3y_3 & x_4y_4 & x_5y_5 & x_6y_6 \\ y_1^2 & y_2^2 & y_3^2 & y_4^2 & y_5^2 & y_6^2 \end{bmatrix} \begin{bmatrix} A_1 & B_1 & C_1 & D_1 & E_1 & F_1 \\ A_2 & B_2 & C_2 & D_2 & E_2 & F_2 \\ A_3 & B_3 & C_3 & D_3 & E_3 & F_3 \\ A_4 & B_4 & C_4 & D_4 & E_4 & F_4 \\ A_5 & B_5 & C_5 & D_5 & E_5 & F_5 \\ A_6 & B_6 & C_6 & D_6 & E_6 & F_6 \end{bmatrix} = \begin{bmatrix} \delta_{11} & \cdots & \delta_{16} \\ \vdots & \ddots & \vdots \\ \delta_{61} & \cdots & \delta_{66} \end{bmatrix} \tag{8}$$

In condensed form, we can write this as

$$\mathbf{Q}_f\mathbf{M}_e = \mathbf{I} \implies \mathbf{M}_e = \mathbf{Q}_f^{-1} \tag{9}$$

Thus, simply by inverting the matrix $\mathbf{Q}_f$, which is purely dependent on our choice of $P_1, P_2, \ldots, P_6$, we can obtain the coefficients of all the quadratic functions. However, it is an interesting property of these functions ψ that if we want to curve-fit a quadratic surface $u_{h,\eta}$ and we know the values of the surface at the points $P_1, P_2, \ldots, P_6$, we can express the function $u_{h,\eta}$ as a linear combination of the ψ functions.

$$\text{If } \begin{cases} u_{h,\eta}(P_1) = \nu_1 \\ u_{h,\eta}(P_2) = \nu_2 \\ \vdots \\ u_{h,\eta}(P_6) = \nu_6 \end{cases} \text{ then we approximate } u_{h,\eta}(x, y) \approx \nu_1\psi_1(x, y) + \nu_2\psi_2(x, y) + \cdots + \nu_6\psi_6(x, y). \tag{10}$$

Alternatively, in matrix form, we have

$$u_{h,\eta}(x, y) \approx \underbrace{\begin{bmatrix} \nu_1 & \nu_2 & \nu_3 & \nu_4 & \nu_5 & \nu_6 \end{bmatrix}}_{\boldsymbol{\nu}^T} \underbrace{\begin{bmatrix} A_1 & B_1 & C_1 & D_1 & E_1 & F_1 \\ A_2 & B_2 & C_2 & D_2 & E_2 & F_2 \\ A_3 & B_3 & C_3 & D_3 & E_3 & F_3 \\ A_4 & B_4 & C_4 & D_4 & E_4 & F_4 \\ A_5 & B_5 & C_5 & D_5 & E_5 & F_5 \\ A_6 & B_6 & C_6 & D_6 & E_6 & F_6 \end{bmatrix}}_{\mathbf{M}_e} \underbrace{\begin{bmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{bmatrix}}_{\boldsymbol{\chi}} = \boldsymbol{\nu}^T \cdot \mathbf{M}_e \cdot \boldsymbol{\chi}. \tag{11}$$

This formulation is useful because it means that we can choose six points in our elements, and then compute an elemental matrix $\mathbf{M}_e$ which will be constant as long as the six points do not move (thus we can reuse the same $\mathbf{M}_e$ for calculations involving the same 6 points). Then, finding $u_{h,\eta}$ is a matter of matrix multiplication. We tried the three following approaches, the first of which was inherently flawed and the second which had instability:

1. The elements Ω_e where the triangles and the six points were chosen to be the vertices of the triangles, and their midpoints. Let there be N triangles that contain the node η. Then the model function at a given node η was the average of the functions which fitted elements 1 through N:

$$u_{h,\eta}(x,y) = \frac{\sum_{i=1}^{N} \boldsymbol{\nu}^T(t) \cdot M_e^{(i)} \cdot \boldsymbol{\chi}(x,y)}{N}. \tag{12}$$

   This model failed because of the following error: when material was flowing without divergence towards a node from one direction, and towards the same node from the opposite direction without divergence, that net material flow at the node was erroneously computed to have no divergence.

2. In the second version, the elements overlapped; each node was associated with an element Ω_e which contained the vertices of all the triangles that contained the given node (hence most elements were hexagons or pentagons). However, since there were often more than six points associated to each element, it was necessary to include cubic terms. This led to an incomplete cubic polynomial, with unpaired terms like x^2y or xy^2. These led to instability issues on very specific nodes; approximately 1 out of every 100 nodes would decrease continuously or increase continuously. Although infrequent, one such node was sufficient to completely overwhelm the remainder of the data points. The model function took the following form, in which χ and ν^T had as many entries N as there were of nodes in the element (M_e had N^2 entries).

$$u_{h,\eta}(x,y) = \boldsymbol{\nu}^T(t) \cdot M_e \cdot \boldsymbol{\chi}(x,y). \tag{13}$$

3. The elements Ω_e were chosen as in method 2, except that instead of expanding the polynomial to have as many terms as nodes in the element, only six nodes were selected in the approximation. The six nodes used were: (1) the center node and (2-6) the nodes in the element with the greatest difference in value from the center node. This means that more than one elemental matrix may exist for each element; they are calculated only when they are needed and subsequently stored.

$$u_{h,\eta}(x,y) = \boldsymbol{\nu}^T(t) \cdot M_e \cdot \boldsymbol{\chi}(x,y). \tag{14}$$

   This method is the one used for the remainder of this paper, since it is the only one that converged to the analytic solutions as the element diameter $h_e \to 0$.

3.1.1 Differential Operators

In equations (12) to (14), the function $u_{h,\eta}(x,y)$ is intentionally written so as to show that $\boldsymbol{\nu}$ is the only time-dependent component and $\boldsymbol{\chi}$ is the only position dependent component. This allows us to easily compute partial derivatives of $u_{h,\eta}(x,y,t)$ with respect to x and y, since only $\boldsymbol{\chi}(x,y)$ varies in terms of x and y. Furthermore, since all main function operators listed at the start of this section are based off of partial derivatives, they are clearly able to be computed. Here we will show how to derive only the Laplacian (as a prolongation of our analysis of the wave equation (6)), but the logic presented can easily be extended to other operators. The Laplacian of $u_{h,\eta}(x,y,t)$ is defined as

$$\begin{aligned}
\nabla^2 u_{h,\eta}(x,y,t) &= \frac{\partial^2}{\partial x^2}\left(\boldsymbol{\nu}^T(t) \cdot M_e \cdot \boldsymbol{\chi}(x,y)\right) + \frac{\partial^2}{\partial y^2}\left(\boldsymbol{\nu}^T(t) \cdot M_e \cdot \boldsymbol{\chi}(x,y)\right) \\
&= \left(\boldsymbol{\nu}^T(t) \cdot M_e\right) \cdot \left(\frac{\partial^2}{\partial x^2}\boldsymbol{\chi}(x,y) + \frac{\partial^2}{\partial y^2}\boldsymbol{\chi}(x,y)\right) \\
&= \left(\boldsymbol{\nu}^T(t) \cdot M_e\right) \cdot \left(\frac{\partial^2}{\partial x^2}\begin{bmatrix}1\\x\\y\\x^2\\xy\\y^2\end{bmatrix} + \frac{\partial^2}{\partial y^2}\begin{bmatrix}1\\x\\y\\x^2\\xy\\y^2\end{bmatrix}\right) = \boldsymbol{\nu}^T(t) \cdot M_e \cdot \begin{bmatrix}0\\0\\0\\2\\0\\2\end{bmatrix}.
\end{aligned} \tag{15}$$

3.2 Time Evolution

With our new method of evaluating the time-independent component of the equations, we turn our attention to advancing the system through time. Let us assume we know the scalar values ν_i for every node η_i in the mesh at a given time. Then, we can use the methods described above to solve numerically for the time-independent side of the differential equation system. This will yield a numerical evaluation of the time-dependent derivative.

For this part, we use the Euler step-method to advance the function through time. This approximation involves using the derivative and value of a function to find the value of the function a little while later[6]

$$\mathbf{u}(t + dt) \approx \mathbf{u}(t) + \mathbf{u}'(t) \cdot dt. \tag{16}$$

Then, using the new function values, we can calculate a new time-derivative $\mathbf{u}'(t)$ using the time-independent side of the equation. In this way we create a loop of the following form.
LOOP until the desired time has been reached: **FOREACH** node in the mesh:

- CALCULATE the time derivative of each function in differential equation system using the differential operators.
- CALCULATE the function values a short time later using (16).

4 Error

The unconventional way that we approach the FEM makes it difficult to apply traditional error theorems to our particular situation. Specifically, our choice of overlapping elements, and of using several elemental matrices, is inconvenient. For these reasons, as well as time constraints, we have not been able to establish a clear upper bound on the error in our model. However, based on the very scant information we have been able to collect, it is possible that error is still bound by some power of the step-size. This is discussed in Appendix D.

From practical tests, three main sources of error have been identified:

1. **Euler Method** The Euler Method error is generated by the approximation's inability to follow the second derivative well. Furthermore, as time progresses, the error accumulates at an increasing rate until it renders the simulation altogether meaningless.
2. **FEM** As discussed in part 3, our quadratic approximation of u often involves dynamically omitting or including neighbouring nodes into consideration.
3. **Rounding** As the model is advanced through time, the rounding error produced in every calculating propagates. However, this is probably a relatively insignificant aspect; the number values stored are floating points with 15 or 16 decimal places.

Very generally, it has been noticed that of these three errors, the first is by far the most significant. We were unable to come up with some bound for this error.

5 Results

5.1 Comparison with Analytic Solutions

In order to validate whether the simulation technique works, we need to compare *Simulacron* results to known solutions. We decided to use the heat PDE, since there are relatively simple analytic solutions for rectangular and circular plates with essential boundary conditions. These solutions are in the form of special infinite series, which are known as generalized Fourier series. The methodology for obtaining these solutions is expounded in sources [4], [11], and [12]. The continuous rainbow color scheme in the heat maps below is generated by adding gradients between the points.

5.1.1 Heat Diffusion on a Plate

The heat equation ($u_t = \alpha \nabla^2 u$) with $\alpha = 1$ on a 1×1 square plate, the edge held at constant temperature $u = 0$, with initial conditions of $u = 1$ inside the domain (and 0 on the edges) has an analytic solution in terms of a double Fourier series:

$$u(x, y, t) = \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} A_{m,n} \sin(m\pi x) \sin(n\pi y)\, e^{-\pi^2 (m^2+n^2) t}, \qquad A_{m,n} = \frac{16}{mn\pi^2} \text{ when } m, n \text{ are odd.}$$

The series was approximated with its 100th partial sum. The approximation appears to be accurate, with

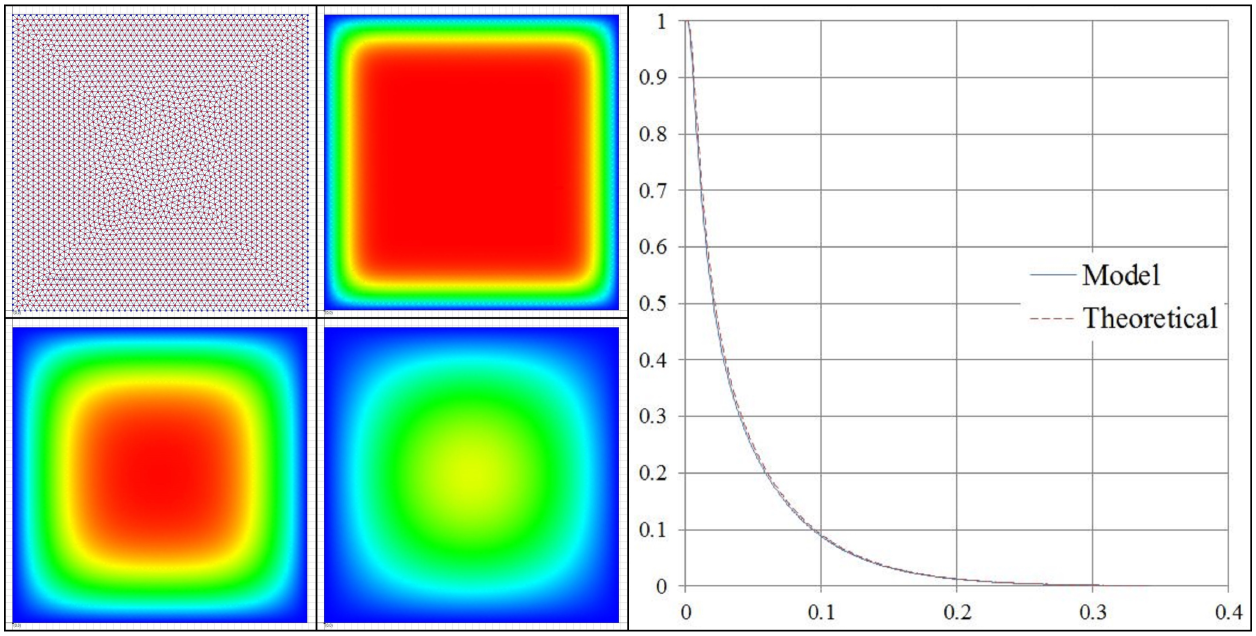


Figure 4: The above simulation is run on the mesh (top left) of a unit square, optimized, 2986 nodes, 3.498 s, $\mathcal{S} = 0.02$. Blue is $u = 0$ and red is $u = 1$. Top center left: heat map after 0.0025 s. Bottom left: heat map after 0.01 s. Bottom center left: heat map after 0.05 s. Right: graph of theoretical and modelled temperature as a function of time for a point of coordinates (0.2213, 0.2184).

the model generated on a reasonably fine mesh following the theoretical analytical solution very accurately. The error at any given time shown is less than 0.53% of the theoretical value. In this example, however, it is difficult to track the propagation of error since both values converge very rapidly towards 0. However, it appears that the model by *Simulacron* lags slightly behind the theoretical. This is a symptom of error generated by the Euler method for a decreasing concave upwards solution [6].

5.1.2 Heat Diffusion on a Disk

The heat equation ($u_t = \alpha \nabla^2 u$) with $\alpha = 1$ on a disk of radius 1, the edge held at constant temperature $u = 1$, with initial conditions of $u = 0$ inside the domain (and 1 on the edge) has an analytic solution in terms of a Fourier-Bessel series:

$$u(r, t) = 1 - 2 \sum_{n=1}^{\infty} \frac{J_0(\zeta_n r)}{\zeta_n J_1(\zeta_n)} e^{-\zeta_n^2 t},$$

approximated as its 100th partial sum. Since *Simulacron* deals only with polygons, we can approximate the disk using a many-sided polygon. The results, summarized in Figure 5, show that the disk is heating up almost exactly as predicted (the error at any given time shown is less than 0.85% of the theoretical value). Even then, the difference between the model and theoretical analytical solution after 1.3 seconds (2600 cycles of 0.0005 s), which amounts to 0.00406, is almost certainly due to the error in the Euler-method since the model lags behind the theoretical solution [6].

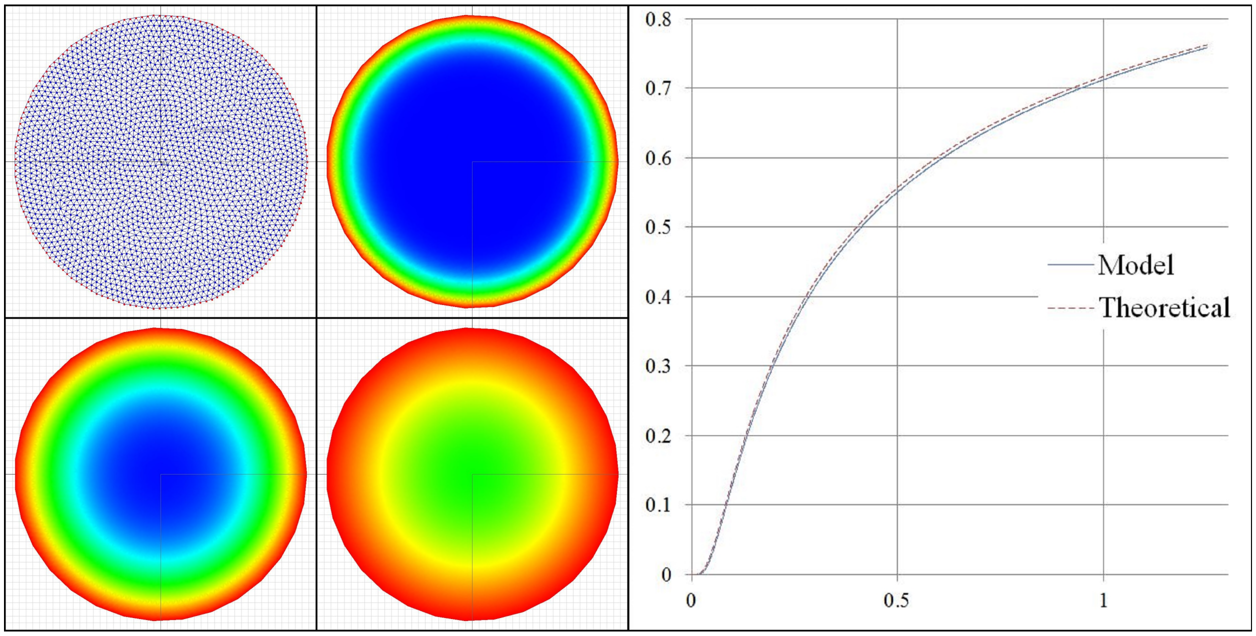


Figure 5: The above simulation is run on the mesh (top left) of a 31-sided polygon of “radius” 1, optimized, 2310 nodes, 3.767 s, $\mathcal{S} = 0.04$. Blue is $u = 0$ and red is $u = 1$. Top center left: heat map after 0.0025 s. Bottom right: heat map after 0.01 s. Bottom center right: heat map after 0.05 s. Right: graph of of theoretical and modelled temperature as a function of time for a point of coordinates (0.1738, -0.7706).

Conclusion

Based on the accuracy of our results, we claim that our model, as implemented in the program *Simulacron* is an accurate means of advancing differential equations through time. The innovation in this approach is double-pronged: the robust mesh generation mechanism on one side, and the easily handled quadratic polynomial approximation on the other.

Acknowledgements

We thank Wayne Nagata for his mentorship throughout the project, and Mark van Raamsdonk for providing us inspiration. We also thank Mr. Ron Xua for the provision of office space and for encouraging us to think like computer scientists. We thank James Charbonneau for moral support. We acknowledge the contributions of reviewers Dustin Woo and Ethan Liu.

References

[1] Carey, G. F. and Tinsley Oden, J. *Finite Elements: Volume I* 1983: Prentice-Hall, N.J.

[2] Shephard, M. S., Dey, S. and Georges, M. K. *Automatic Meshing of Curved Three-Dimensional Domains: Curving Finite Elements and Curvature-Based Mesh Control* in *IMA Volume 75: Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations* 1995: Springer-Verlag, N.Y.

[3] Brire de l'Isle, E. and Georges, P.L. *Optimization of Tetrahedral Meshes* in *IMA Volume 75: Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations* 1995: Springer-Verlag, N.Y.

[4] Demlow, A. *Numerical Solutions of Differential Equations* 2002 (notes).

[5] Haghighi, K. and Kang, E. *A Knowledge-Based Approach to the Adaptive Finite Element Analysis* in *IMA Volume 75: Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations* 1995: Springer-Verlag, N.Y.

[6] Boyce, W. and DiPrima, R. *Elementary Differential Equations and Boundary Value Problems* 2009: John Wiley and Sons, N.J.

[7] Carey, G. F. and Tinsley Oden, J. *Finite Elements: Volume II* 1983: Prentice-Hall, N.J.

[8] Carey, G. F. and Tinsley Oden, J. *Finite Elements: Volume IV* 1983: Prentice-Hall, N.J.

[9] Carstensen, C. and Brenner, S.C. *Finite Element Methods* in *Encyclopedia of Computational Mechanics* 2004: John Wiley and Sons.

[10] Betounes, D. *Partial Differential Equations for Computational Science* 1998: Springer-Verlag, N.Y.

[11] Daileda, R.C. *Partial Differential Equations* 2012 (notes).

[12] Weisstein, E. W. *Heat Conduction Equation – Disk.* From *MathWorld* – A Wolfram Web Resource.

A Example of weak forms and requirements of test functions

A natural question to ask is then whether the FEM could be used to determine u such that u solves the strong problem

$$\begin{aligned} &\text{Find } \mathit{u(x,y)} \text{ such that} \\ &-\nabla \cdot k(x,y)\nabla u(x,y) + b(x,y)u(x,y) = f(x,y), \\ &u = \hat{u}(x,y) \text{ on } \partial\Omega. \end{aligned} \tag{17}$$

Here, $k(x, y)$ is a property of the material we take to be constant in the questions we address explicitly, as can b for some problems. $\hat{u}$ describes the Dirichlet boundary conditions of the problem. These are "essential" boundary conditions; they are independent of u. [1].

The FEM turns our not be employable in solving the strong problem directly [1], but is well-suited to a variational statement, or weak formulation of problems like (18). As posing the weak problem is somewhat complex, we consider only a case in $\mathbb{R}$, but stated generally in the hopes of suggesting a multi-variable analog. From [1], we take the problem

$$\begin{aligned} &\text{Find } \mathit{u(x,y)} \text{ such that} \\ &u'' + u = x, \;\; 0 < x < 1, \\ &u(0) = u(1) = 0. \end{aligned} \tag{18}$$

Entertaining the existence of an set of integrable functions v, it follows that

$$\int_0^1 (u'' + u)\, v \; dx = \int_0^1 xv \; dx \implies \int_o^1 (u'' + u - x)\, v \; dx = 0 \;\; \forall v. \tag{19}$$

Integrating (19) by parts and using the boundary conditions similar to those given by (18) it can be shown that the integral of the right side of (19) is equivalent to

$$\int_0^1 u'v' \ dx. \tag{20}$$

Taking into account the possibility that $u' = v'$ might be true, we have the necessary conditions to define the aforementioned space to which v belongs: in one dimension $v \in H^1(\Omega)$ iff $\int_0^1 (v')^2 \ dx < \infty$ and v is continuous on an interval Ω. This is equivalent to the existence of the H^1 inner product for v, which is found when working from (19) to (20).

v itself is of little use from a numerical perspective as it is an infinite linear combination of other functions in H^1 [1]. It is possible, though, to take a finite dimensional subspace $H^{(N)}$ of H^1 in which v_N is a linear combination of finitely many basis functions $\phi_1, \phi_2, \phi_3, \ldots, \phi_N$ [1], [7], [9]. These ϕ_i are the surface-modeling functions from section 1.1. The finite-dimensional statement of (19) and (20) is therefore

$$\int_0^1 u_N' v_N' + u_N v_N \ dx = \int_0^1 u_N' v_N' \ dx \tag{21}$$

where

$$v_N(x) = \sum_{i=1}^{N} \beta_i \phi_i(x) \text{ and } u_N = \sum_{i=1}^{N} \gamma_i \phi_i(x).$$

If we stipulate that the coefficients γ_i of our finite dimensional function u_N modeling the property we are interested in are initial conditions, then it it logical to enforce that $\phi_i = 1$ on the ith node and be zero everywhere else on Ω. These conditions, and the condition $u_N \in H^{(N)} \subseteq H^1 \implies \phi_i \in H^1$ can be met by expressing each ϕ_i as a linear combination of shape functions ψ known to be square integrable and continuous on some interval in Ω. The end goal of a Finite Element Method is to determine these coefficients in order to obtain a piecewise continuous polynomial u_h approximating u.

As was demonstrated in greater detail in section 3.1, the coefficients γ_i can obtained by solving a matrix equation involving the shape functions ψ and the initial conditions. Such systems are linear and by (15) usually sparse [1], making them relatively easy to solve [1], [7].

B Invalidation Checking Algorithms

As discussed, the two types of documented invalidations are when (1) a node is placed outside of the polygon $\mathcal{P}$ and (2) when nodes are placed whose connecting segments intersect $\mathcal{P}$. Let us first consider the problem of determining the relative position of a point $P_0(x_0, y_0)$ to a segment $S = \overline{Q_1 Q_2}$. We define $\mathbf{v_1} = \langle x_0 - x_1, y_0 - y_1 \rangle$ and $\mathbf{v_2} = \langle x_2 - x_1, y_2 - y_1 \rangle$.

The cross product $\mathbf{v_1} \times \mathbf{v_2}$ will be positive if the point is on one side of the segment and negative if it is on the other. Thus, if we consider segments such that $x_0 \in [x_1, x_2]$ (where $x_2 > x_1$), we can determine whether the segment is above or below the point by checking the sign of $\mathbf{v_1} \times \mathbf{v_2}$. This reduces to a determinant

$$\mathbf{v_1} \times \mathbf{v_2} = \begin{vmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \\ v_{1x} & v_{1y} & v_{1z} \\ v_{2x} & v_{2y} & v_{2z} \end{vmatrix} = \begin{vmatrix} v_{1y} & v_{1z} \\ v_{2y} & v_{2z} \end{vmatrix} \hat{\mathbf{x}} - \begin{vmatrix} v_{1x} & v_{1z} \\ v_{2x} & v_{2z} \end{vmatrix} \hat{\mathbf{y}} + \begin{vmatrix} v_{1x} & v_{1y} \\ v_{2x} & v_{2y} \end{vmatrix} \hat{\mathbf{z}} = \begin{vmatrix} x_0 - x_1 & y_0 - y_1 \\ x_2 - x_1 & y_2 - y_1 \end{vmatrix} \hat{\mathbf{z}} \tag{22}$$

Thus, depending on the sign of the determinant above, and given the fact that $x_2 > x_1$, we can establish that a point P is above the segment given the above expression is positive. In order to determine whether a point is inside a polygon, it is only necessary to find all the segments S_x whose x range contains the x coordinate of the point (and analogously find all the segments S_y whose y range contains the y coordinate of the point). If there are an odd number of segments S_x below P and an odd number S_y to the left of P, the point is inside the polygon, otherwise it is outside.

Now let us consider the problem of finding whether a segment intersects a polygon. If two segments intersect, then it can be said that both pairs of their endpoints lie on opposite sides of the other segment. Thus, given two segments $S_1 = \overline{P_1P_2}$ and $S_2 = \overline{P_3P_4}$, they intersect iff

$$\begin{vmatrix} x_1 - x_2 & y_1 - y_2 \\ x_3 - x_2 & y_3 - y_2 \end{vmatrix} \cdot \begin{vmatrix} x_1 - x_2 & y_1 - y_2 \\ x_4 - x_2 & y_4 - y_2 \end{vmatrix} \leq 0 \text{ and } \begin{vmatrix} x_3 - x_4 & y_3 - y_4 \\ x_1 - x_4 & y_1 - y_4 \end{vmatrix} \cdot \begin{vmatrix} x_3 - x_4 & y_3 - y_4 \\ x_2 - x_4 & y_2 - y_4 \end{vmatrix} \leq 0 \qquad (23)$$

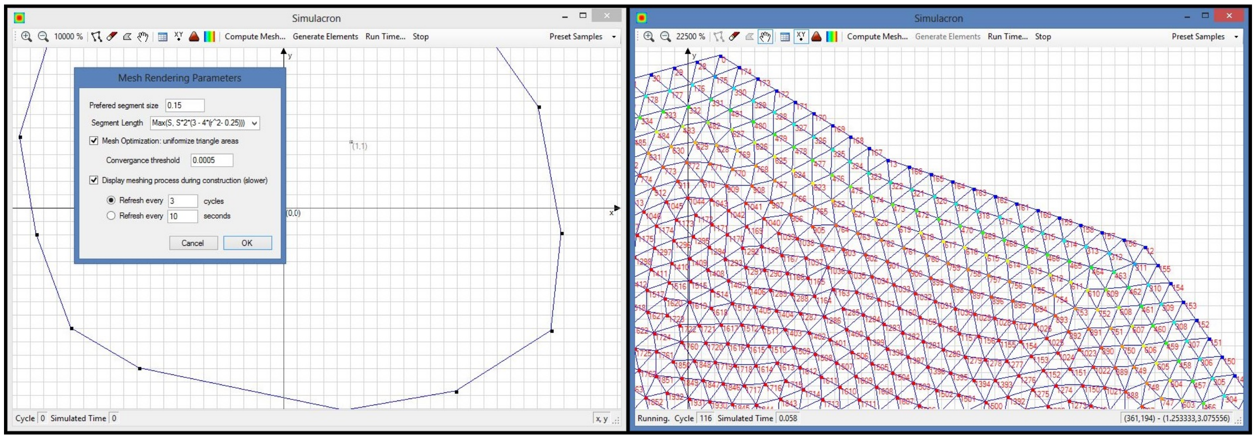
Note that if the endpoint of a segment is on the other segment, one of the two above products will be 0. However, we do not consider this an intersection. A test segment intersects a polygon if at least one of the constituent segments intersects the test segment.

C Programming Approach

Initially, this project was implemented using the *Python* programming language with the visual module. Later on, the program was translated to *C#*. This was done so as to take advantage of the very class-centered approach of *C#*, as well as the language's use of pointers. Pointers enable us to indirectly access data elements through references instead of direct data values; they are computer variables "pointing" to computer data variables. They allow us to involve the same data variables in different data structures. For example the same node instance (not copies) could be simultaneously referred by points, segments, triangles, polygons, and mesh data structures. When the node's value is modified in one of the structure it is immediately applies to all data structure in which it is involved.

The following classes (or object structures) were defined: 2D points, matrices, mesh nodes, and polygon points. The 2D point class was treated as a position vector, with addition, subtraction and scalar multiplication based off of vector operations. Mesh nodes were defined by a 2D point position, a list of adjacent mesh nodes, a list of elemental matrices, and function values. The polygon points were identified by the mesh nodes on which they were placed.

A matrix class was also developed, with the capacity to compute determinants, solve systems, find inverse matrices, and perform basic operations such as multiplication addition and subtraction.


Simulacron
10000 %
Compute Mesh... Generate Elements Run Time... Stop
Preset Samples
Mesh Rendering Parameters
Cancel
OK
y
x
(0,0)
(1,1)
Simulacron
22500 %

Prefered segment size 0.15
Segment Length Max(S, S*2*(3 - 4*(r^2- 0.25)))
☑ Mesh Optimization: uniformize triangle areas
Convergance threshold 0.0005
☑ Display meshing process during construction (slower)
● Refresh every 3 cycles
○ Refresh every 10 seconds

Cycle 0 Simulated Time 0

Running. Cycle 116 Simulated Time 0.058 (361,194) - (1.253333,3.075556)

Figure 6: Screen-shots taken of the program and its user interface. Left: the control for initializing the triangulation process; Right: a paused simulation of heat diffusion, without the color interpolation between nodes.

D Error in Finite Element Interpolations

No model is true, but some models are useful. Exactly how true a model is is found by measuring the *error* in the model. More specifically, we are interested in knowing the norm, or magnitude of the error; i.e., whether the approximation is above or below is secondary to how much the approximation is above or below by. With this motivation, the norm of the error $||e||$ in a finite element model is defined by [1] to be

$$||e|| \equiv ||u - u_h||. \tag{24}$$

Evaluation of this expression is possible only when u is known. Although analytic solutions exist for some domains and initial conditions for every equation treated in this paper, the development of a model is most useful when u is not known or does not exist. However, estimates of $||u - u_h||$ for a particular element in the form of upper bounds proportional to element size and the norm of u at that element are an established method for determining model quality [7].

Exactly which norm is used a matter of choice. As both u and u_h are in some Sobolev space, they can each by treated by some L_p norm [8], or the H^s norm itself [7], which measures error in both the approximation function u_h and its first derivative u_h'. The error in the derivative generally converges slower than the error in the function itself [4], leading to a more conservative estimate of what is already a pessimistic [4] estimate of the model's quality. The estimation of error using the H^1 norm for an FEM model with $\overline{\Omega} \in \mathbb{R}^2$ is

$$||u - u_h||_{H^1} \leq C h_e^k ||u||_{H^{1+k}}, \tag{25}$$

where h_e is the maximum distance between two points in Ω_e and k is the degree of the highest degree complete polynomial in ϕ [1]. The Constant C depends on the regularities of all the elements Ω_e in Ω.

D.1 A Priori Estimates

As mentioned, our mesh generates triangular elements that are nearly equilateral almost everywhere for reasonable Ω, there should be little hindrance in solution convergence due to C. At any point (x_0, y_0), any norm of u will be constant. Therefore, for temperature distributions involving small temperatures $\nu_1, \nu_2, \ldots, \nu_i$ on each element Ω_e and bounded, small magnitude position-dependent partial derivatives, we judge it reasonable to subsume this norm into C, leading to the *a priori* error estimate from [1]

$$||u - u_h||_{H^1} \leq C h_e^k. \tag{26}$$

This supports the conclusion that refinement of the mesh will allow convergence of u_h to converge u at $O(h^k)$. The maximum error in just the function values [4], measured with an L_∞ norm, converges somewhat faster, at $O(h^{k+1})$, giving by [1]

$$||u - u_h||_{L_\infty} \leq C h_e^{k+1}. \tag{27}$$

D.2 A Posteriori estimates and additional discussion

It is possible, substituting u_h for u, to develop an estimate for C by, for example, calculating $\frac{||u-u_h||_{H^1}}{||u_h||_{H^{k+1}}}$. However, statement of the H^k norm for $k \neq 1$ is beyond the scope of this paper. Other norms may be used in its place, and entirely other but not necessarily simpler estimators exist.

A key feature of our mesh generating program is its ability to efficiently handle domains with singularities, or regions where the interior angle between two vertexes very large, even nearly 2π. Error convergence in these areas is much slower than elsewhere in the domain, and usually converges at a fixed power of h_e regardless of the basis functions used [7]. However, it is acceptable [7], to define disks $\Omega_j \in \Omega$ for $j = 1, 2, \ldots, n$ of small enough radius to capture only one of the n singularities in Ω each. Then on the "nicer" sub-domain

$$\widetilde{\Omega} = \Omega \setminus \bigcup_{j=1}^{n} \Omega_j \tag{28}$$

the error behaves as previously described.

E Screen-shots from *Simulacron*

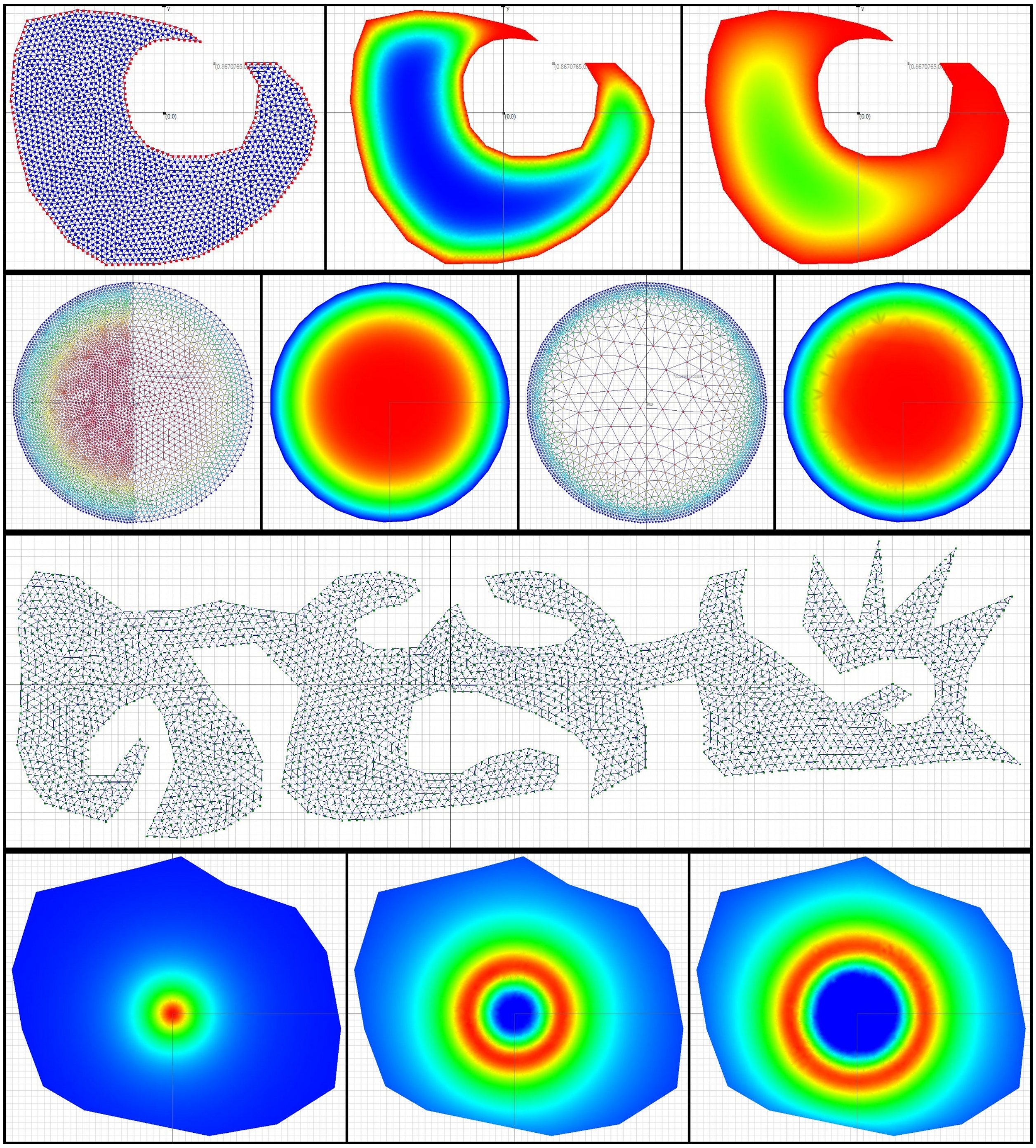


Figure 7: Rows from top to bottom: (1) non-convex crescent polygon diffusing; (2) diffusion on differently refined meshes and their color interpolations; (3) triangulation of a non-convex polygon; (4) propagation of a small disturbance at times 0 (red = 1), 0.3 (red = 0.28), and 0.5 (red = 0.23). Blue is always 0.