12-16-2015

Numerical Solutions to Two-Dimensional Integration Problems

Alexander Carstairs

NUMERICAL SOLUTIONS TO TWO-DIMENSIONAL INTEGRATION PROBLEMS

by

Alexander D. Carstairs

Under the Direction of Valerie Miller, PhD

ABSTRACT

This paper presents numerical solutions to integration problems with bivariate integrands. Using equally spaced nodes in Adaptive Simpson's Rule as a base case, two ways of sampling the domain over which the integration will take place are examined. Drawing from Ouellette and Fiume, Voronoi sampling is used along both axes of integration and the corresponding points are used as nodes in an unequally spaced degree two Newton-Cotes method. Then the domain of integration is triangulated and used in the Triangular Prism Rules discussed by Limaye. Finally, both of these techniques are tested by running simulations over heavily oscillatory and monomial (up to degree five) functions over polygonal regions.

INDEX WORDS: Delaunay Triangulation, Voronoi Sampling, Simpson's Rule, Adaptive Simpson's Rule, Quadrature

NUMERICAL SOLUTIONS TO TWO-DIMENSIONAL INTEGRATION PROBLEMS

by

Alexander D. Carstairs

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2015

NUMERICAL SOLUTIONS TO TWO-DIMENSIONAL INTEGRATION PROBLEMS

by

Alexander D. Carstairs

ACKNOWLEDGMENTS

I would like to first thank my thesis advisor, Valerie Miller, for everything she has done to help me complete this thesis. I would to also thank my committee members, Rebecca Rizzo and Michael Stewart, for their feedback during the editing process. I am also grateful for all of my classmates that have shared their wisdom and support throughout this process. Finally, I would like to thank my wife for her support and understanding while completing this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The subject of numerical integration is an essential topic in any numerical analysis course, and while there has been extensive research into numerical techniques, there is no unifying technique that works for all integrands. Choosing a technique can be based on many factors, but ultimately the decision will depend on the application. In the field of computer graphics, numerical integration is essential in determining the lighting of an object. In [9], Fiume and Ouellette outline several techniques for applications in computer graphics problems all of which must balance the numerical efficiency of the method with the aesthetics of the final image, i.e. sacrificing numerical accuracy for a better looking image. One of the methods outlined by Fiume and Ouellette is a one-dimensional Voronoi diagram-based sampling of the domain of integration. In this thesis this sampling technique will be expanded in two ways.

The first method is to perform the one-dimensional Voronoi sampling described in [9] along each axis of integration to get two sets of n points. Then the Cartesian product of those two sets is taken to create an $n \times n$ grid of nodes, which can then be used in a quadrature method. The second method will be to triangulate the domain of integration using a Delaunay triangulation. These methods will be described in greater detail along with some relevant background theory of each in Chapters 1 and 2, respectively. The Voronoi sampling will be implemented in an adaptive Newton-Cotes method of degree two, and the Delaunay triangulation will be implemented in the midpoint, trapezoidal and Simpson's rules described in [3]. These methods, along with adaptive Simpson's rule and Monte Carlo integration, will be used to integrate a variety of test functions described in [14] and the set of monomials given in [8]. The results of these numerical simulations will be discussed in Chapter 3.

1.1 Voronoi Sampling

In this section, the reader is introduced to the basics of Voronoi diagrams. The following definitions of the Voronoi diagrams are consistent with those given in [2] and [9]. The topic of Voronoi diagrams dates back to the 1600s to Descartes where he used the idea that a set S of *sites* (point *sites*) p in a space M are given and the *region* of a *site* p consists of all points s such that the *influence* of p is the strongest. This concept emerged independently in various fields of science such as biology to model and analyze plant competition; robotics to find an optimal path to avoid a set of obstacles; and meteorology to determine regional rainfall from a discrete set of rain gauges. In most of these cases, names particular to the field of study have been used, e.g. *Thiessen polygons* are used in meteorology. The mathematicians Dirichlet and Voronoi would be the first to formally introduce the concept while working with quadratic forms, and the resulting structure has been given the name *Dirichlet tessellation* or *Voronoi diagram* [2].

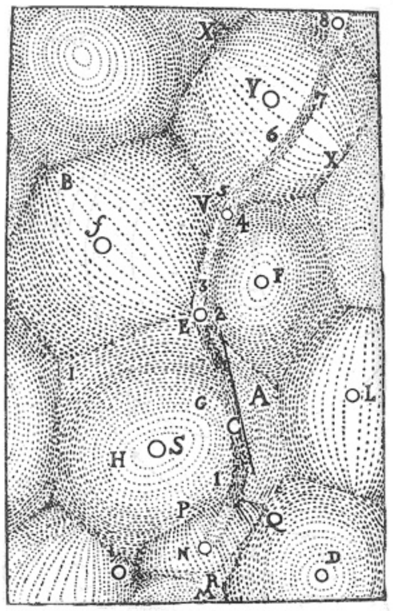


Figure 1.1. Descartes' diagram of the regions of *influence* for point *sites* [2]

Definition 1.1.1. Let $S \subset \mathbb{R}^2$ be a set of points $x_1, x_2, \ldots, x_n$ for $n \geq 3$ and $p \in \mathbb{R}^2$ with $d(x_i, p)$ given as some metric. For any $x_i, x_j \in S$ and $i \neq j$, let

$$B(x_i, x_j) = \{p \mid d(x_i, p) = d(x_j, p)\}$$

be the bisector of x_i and x_j, i.e. $B(x_i, x_j)$ is the perpendicular line through the center of the line segment connecting x_i and x_j. Thus, the bisector separates the halfplane

$$D(x_i, x_j) = \{p \mid d(x_i, p) < d(x_j, p)\}$$

containing x_i from the halfplane $D(x_j, x_i)$ containing x_j.

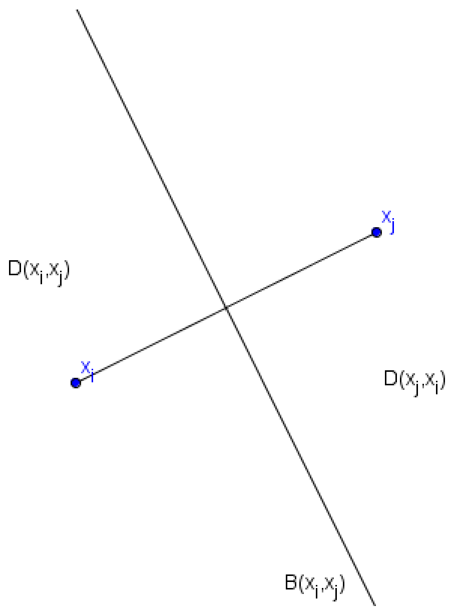


Figure 1.2. Dividing halfplanes with bisector.

Using the halfplane described above, the Voronoi diagram can now be defined.

Definition 1.1.2. Let the Voronoi region of x_i be given as

$$V_i = V(x_i, S) = \bigcap_{x_j \in S, i \neq j} D(x_i, x_j)$$

with respect to S where V_i is an open set in the topological sense. Then the *Voronoi Diagram* of S is defined as

$$V(S) = \bigcup_{x_i, x_j \in S, i \neq j} \overline{V_i} \cap \overline{V_j}$$

where $\overline{V}$ is the closure of set V, i.e. the open set V unioned with its boundary.

Scaling the above definitions down to a one-dimensional diagram creates a sampling method that iteratively selects points on an interval $[a, b]$. First, let $x_1 = a$ and $x_2 = b$ and have x_3 be

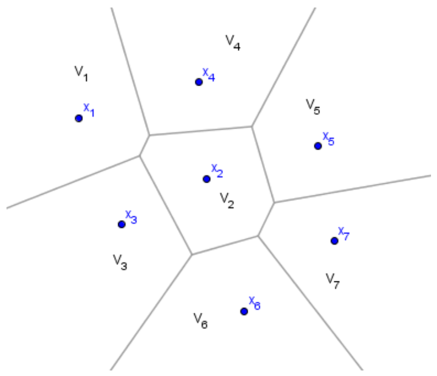


Figure 1.3. 2D Voronoi Diagram

a randomly chosen number from the uniform distribution over (a, b). Then the next n points are determined by constructing the Voronoi regions corresponding to the location of the sample points that already exist in the sequence. Let V_i represent the Voronoi region of x_i and be defined as

$$V_i = \{x \in [a, b] \mid |x - x_j| \leq |x - x_i|, \text{ for all } j \in [1, n+3]\}$$

for $i \in [1, n+3]$. Let V_M be the longest line segment as defined above with ties being broken randomly. The next sample point in the sequence would the midpoint of the line segment corresponding to V_M. The abbreviation V_m is used to denote a Voronoi sampling sequence of n additional points where $m = n+3$. In Fig 1.4, a quick example of the Voronoi sampling is given. Let $a = -2$, $b = 10$ and $x_3 = 5$. First, the midpoint (bisector) between the three points is found, which give us $m_1 = 1.5$ and $m_2 = 7.5$. Then each Voronoi region is formed: $V_1 = (-2, 1.5)$, $V_2 = (7.5, 10)$ and $V_3 = (1.5, 7.5)$. Since V_3 is the longest Voronoi region, the next point to be added to the sequence is $x_4 = 4.5$. The code written in MATLAB© is given in Appendix A.

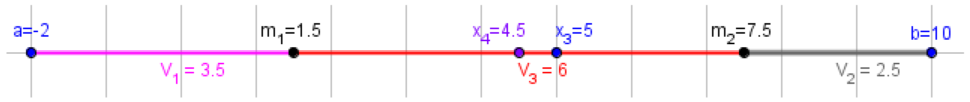


Figure 1.4. Voronoi Sampling

1.2 One-Dimensional Newton-Cotes Quadrature

Now it is necessary to derive a generic one-dimensional quadrature method for integrating over $[a,b]$. Given three arbitrary points a, m and b, where $a < m < b$, Lagrange interpolation is used to find a degree two polynomial to approximate our function $f(x)$. This polynomial is given by

$$p(x) = \sum_{j=1}^{3} f(x_j)L_j(x),$$

where $L_j(x)$ is the Lagrange polynomial

$$L_j(x) = \prod_{i=1, i\neq j}^{3} \frac{x - x_i}{x_j - x_i}, j = 1, 2, 3$$

Thus, $p(x)$ may be written as

$$p(x) = \sum_{j=1}^{3} f(x_j)L_j(x) = f(a)\frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m)\frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b)\frac{(x-a)(x-m)}{(b-m)(b-a)} = \tau \tag{1.2.1}$$

Assuming that $p(x)$ approximates some function $f(x)$ that needs to be integrated then

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = \sum_{j=1}^{3} w_j f(x_j) = w_1 f(a) + w_2 f(m) + w_3 f(b),$$

where the weights w_j are given by

$$w_j = \int_a^b L_j(x)dx.$$

All of these steps are similar to those for the derivation of an arbitrary Newton-Cotes method using equally spaced points and are given in almost any numerical analysis textbook. Indeed, a derivation of Simpson's rule (sometimes referred to as Simpson's 1/3 Rule) generally follows from 1.2.1 by exploiting the equal spacing of the points and gives a nice simplification.

Setting up a general spacing of points yields

$$b - a = \alpha + \alpha c = \alpha(1 + c)$$

$$b - m = \alpha$$
$$a - m = -c\alpha,$$

which simplifies the right hand side of Equation (1.2.1) to get

$$\tau = \frac{1}{\alpha^2 c(1+c)} \left[f(a)(x-m)(x-b) - f(m)(1+c)(x-a)(x-b) + cf(b)(x-a)(x-m)\right] \quad (1.2.2)$$

To find the weights w_1, w_2 and w_3 of $f(a)$, $f(m)$ and $f(b)$, respectively, the function $g(x) = (x-u)(x-v)$ is integrated, which yields

$$\int_a^b (x-u)(x-v)dx = \frac{1}{2}\left[(b-u)(b-v)^2 - (a-u)(a-v)^2\right] - \frac{1}{6}\left[(b-v)^3 - (a-v)^3\right] = \gamma.$$

The right hand side of the above equation, γ, can then be simplified further according to the following values of u and v:

(i) If $u = m, v = b$: $\gamma = \frac{1}{6}\alpha^3 (1+c)(2c-1)$

(ii) If $u = a, v = b$: $\gamma = -\frac{1}{6}\alpha^3(1+c)^3$

(iii) If $u = a, v = m$: $\gamma = -\frac{1}{6}\alpha^3(1+c)^2(c-2)$

Replacing the weights in (1.2.2) with the values above and simplifying gives

$$\sum_{j=1}^{3} f(x_i)L_j(x) = \frac{\alpha(1+c)}{6c}\left[(2c-1)f(a) + (1+c)^2 f(m) + c(2-c)f(b)\right].$$

This gives us our quadrature formula

$$I(f) = \int_a^b f(x)dx \approx \frac{\alpha(1+c)}{6c}\left[(2c-1)f(a) + (1+c)^2 f(m) + c(2-c)f(b)\right] = I(p).$$

For the error, the equation

$$R(x) = I(f) - I(p) = \int_a^b \frac{f'''(\alpha)}{3!}(x-a)(x-m)(x-b)dx$$

is examined, and the function f is assumed to be at least three times differentiable over $[a, b]$. Since the cubic polynomial function changes sign over the interval $[a, b]$, $R(x)$ is broken into two integrals in order to apply the weighted Mean-Value Theorem for Integrals:

$$R(x) = \frac{f'''(\alpha_1)}{6} \int_a^m (x-a)(x-m)(x-b)dx + \frac{f'''(\alpha_2)}{6} \int_m^b (x-a)(x-m)(x-b)dx,$$

where $\alpha_1 \in (a, m)$ and $\alpha_2 \in (m, b)$. It is now clear that only the general form of the integral above is needed in order to proceed with the error analysis. Thus,

$$\begin{aligned} \beta &= \int_u^v (x-a)(x-m)(x-b)dx \\ &= (v-u)\left[\frac{1}{4}(v+u)(v^2+u^2) - \frac{a+m+b}{3}(v^2+uv+u^2) + \frac{am+ab+bm}{2}(u+v) - amb\right]. \end{aligned}$$

Simplifying β according to the following values of u and v:

(i) If $u = a$ and $v = m$: $\beta = \frac{(m-a)^3}{12}[2b - m - a]$

(ii) If $u = m$ and $v = b$: $\beta = \frac{(b-m)^3}{12}[2a - m - b]$

gives the total error $R(x)$ to be

$$R(x) = \frac{f'''(\alpha_1)}{6}\frac{(m-a)^3}{12}[2b-m-a] + \frac{f'''(\alpha_2)}{6}\frac{(b-m)^3}{12}[2a-m-b]. \tag{1.2.3}$$

Assuming that f''' is essentially constant on $[a, b]$, then

$$\begin{aligned} R(x) &\approx \frac{A}{6}\frac{(m-a)^3}{12}[2b-m-a] + \frac{A}{6}\frac{(b-m)^3}{12}[2a-m-b] \\ &= \frac{A}{72}\left[(m-a)^3[2b-m-a] + (b-m)^3[2a-m-b]\right] \\ &= \frac{A}{72}(b-a)^3[2m-b-a], \end{aligned}$$

so $R(x) = 0$ if $f(x)$ is a polynomial of degree ≤ 2. This is to be expected since quadratic interpolation is only guaranteed to be exact if $f(x)$ is of degree ≤ 2. Using $c = 1$

($b - m = \alpha$ and $a - m = -\alpha$) for the above error analysis, the same degree of accuracy for Simpson's rule, but Simpson's rule should have an extra degree of accuracy. The extra degree of accuracy comes from the cancellation of the term containing f''' from the derivation of Simpson's rule using Taylor polynomials instead of Lagrange polynomials. To show this, Simpson's rule is derived using a Taylor expansion centered around the point $m = \frac{b+a}{2}$ to approximate $f(x)$. Integrating the Taylor expansion from a to b to yields

$$\begin{aligned}\int_a^b f(x)dx &= f(m)(b-a) + \frac{f'(m)}{2}\left[(b-m)^2 - (a-m)^2\right] + \frac{f''(m)}{6}[(b-m)^3 - (a-m)^3] \\ &\quad + \frac{f'''(m)}{24}\left[(b-m)^4 - (a-m)^4\right] + \frac{f^{(4)}(\xi)}{120}\left[(b-m)^5 - (a-m)^5\right] \\ &= f(m)(b-a) + \frac{f'(m)}{2}\left[(\alpha)^2 - (-\alpha)^2\right] + \frac{f''(m)}{6}\left[(\alpha)^3 - (-\alpha)^3\right] + \\ &\quad \frac{f'''(m)}{24}\left[(\alpha)^4 - (-\alpha)^4\right] + \frac{f^{(4)}(\xi)}{120}\left[(\alpha)^5 - (-\alpha)^5\right] \\ &= 2\alpha f(m) + \alpha^3\frac{f''(m)}{3} + \alpha^5\frac{f^{(4)}(\xi)}{60} \\ &= \frac{\alpha}{3}\left[f(a) + 4f(m) + f(b)\right] + \alpha^5\frac{f^{(4)}(\xi)}{60}.\end{aligned}$$

The f' and f''' terms cancel, and the additional degree of accuracy is gained for Simpson's rule.

The Voronoi sampling and quadrature are implemented in the function *vadapt3* and are given in Appendix B.

1.3 Delaunay Triangulation

According to Aurenhammer and Klein, Voronoi was the first to consider the *dual* of the Voronoi diagram where he stated that any two point sites are connected if their regions share a boundary. Later, Delaunay defined the *dual* in the following way:

Definition 1.3.1. Two point sites are connected if and only if the two sites lie on a circle whose interior contains no point in S where S is the set defined in Definition1.1.1.

For this, the *dual* was given the name *Delaunay tessellation* or *Delaunay triangulation*. Given a Voronoi diagram, one can easily construct a *Delaunay triangulation* by connecting the center x_i of a Voronoi region with the center of each adjacent Voronoi region for all $x_i \in S$ [2], but

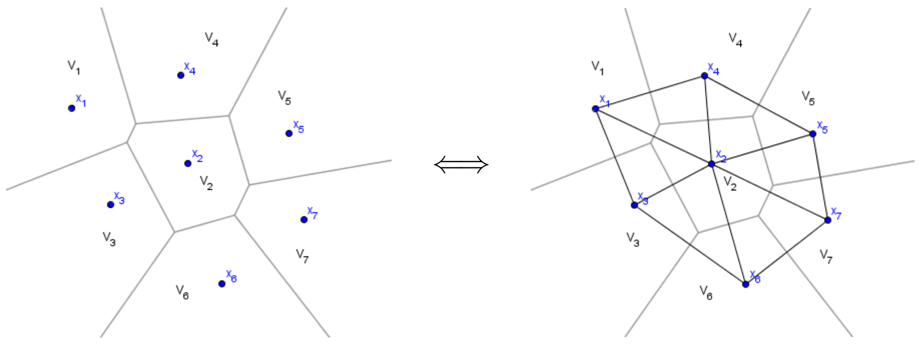


Figure 1.5. Creating a Delaunay Triangulation from Voronoi diagram

it is not necessary to have a Voronoi diagram first. Using an alternate definition of a Delaunay triangulation, a given triangulation can be checked to see if it is Delaunay.

Definition 1.3.2. A *circumcircle* is the circle that passes through the endpoints x_i and x_j for the edge x_ix_j and endpoints x_i, x_j and x_k of triangle $x_ix_jx_k$ for all combinations of i, j and k .

Definition 1.3.3. Let T be a triangulation with m triangles and a set of n points S where each element of S is a vertex of a triangle $t_i \in T$ for $i = 1, \ldots, m$. T is considered *Delaunay* if and only if the circumcircle of every t_i contains no other vertex in S.

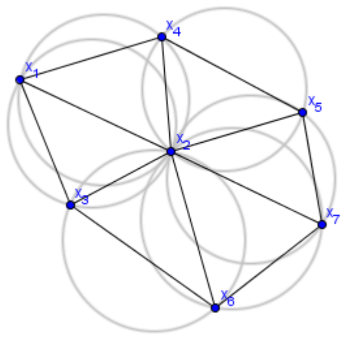
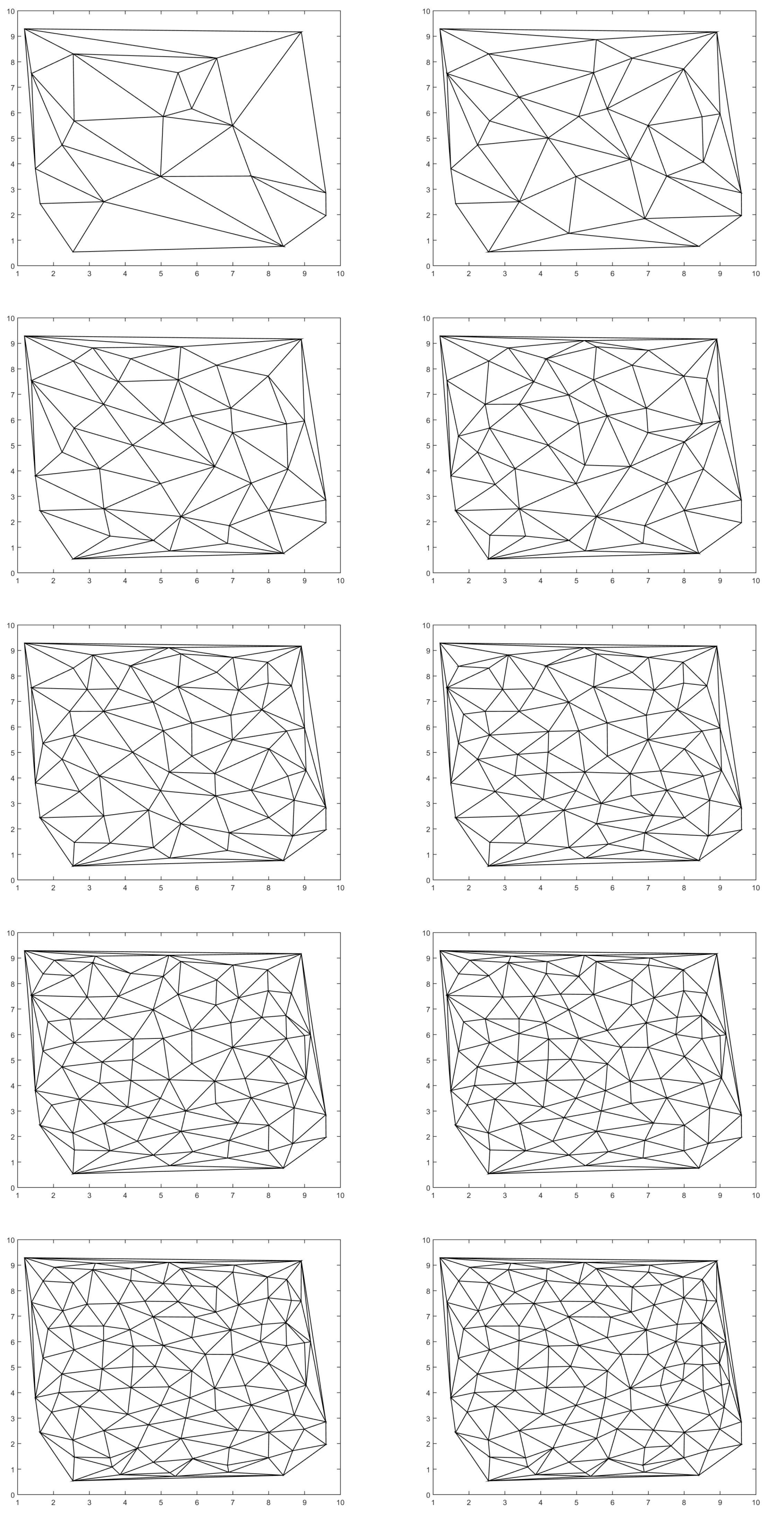


Figure 1.6. Checking the Delaunay criterion holds for each triangle.

Both definitions 1.3.1 and 1.3.3 are known as the Delaunay criterion or empty circle property for edges (1.3.1) and triangles (1.3.3), and are implemented in several algorithms used for creating Delaunay triangulations. Since an edge only has two points, it can have infinitely many circumcircles, but only one of the circumcircles has to be empty for the criterion to hold. On the other hand, triangles will have a unique circumcircle defined by its three vertices, so there is only a need to check that the one circle is empty. A main advantage of using a Delaunay triangulation is that it maximizes the minimum angle of all of the triangles within the triangulation of a given set of points, which helps avoid skinny triangles. As the number of triangles increases, the triangles appear more uniform in size as shown in Table 1.1. This reduces the risk of peaks on the function that is being integrated from being cut off by large skinny triangles, which improves the stability of the calculations performed on the mesh. In graphics, the uniformity of the triangles yields a nice mesh for the shading and texturizing of an image [15].

Table 1.1. Delaunay triangulations of 20 points through 110 points with the triangulation at every 10 points shown.



Chapter 2

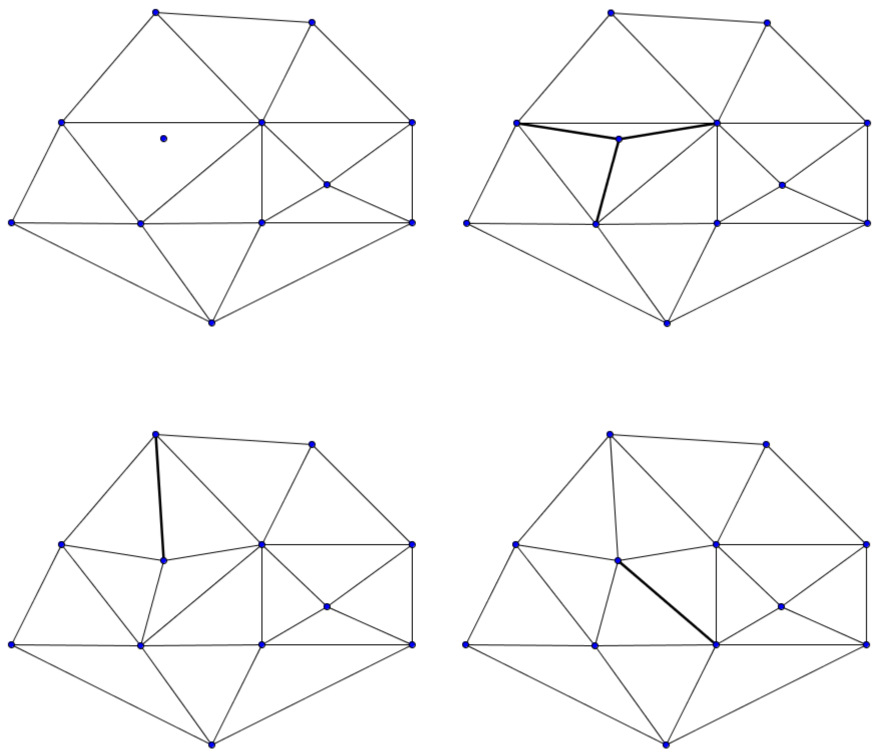
ALGORITHMS

There are three types of algorithms used in the construction of Delaunay triangulations: incremental insertion algorithms, divide-and-conquer techniques, and a sweepline techniques. The simplest are the incremental insertion algorithms, and they can be expanded to be used in higher dimensions. The algorithms that use the divide-and-conquer or sweepline techniques are faster than the incremental insertion techniques in two dimensions but are difficult to generalize (if at all) to higher dimensions. To construct the Delaunay triangulation in this thesis, two algorithms are combined: the method `dtris2` from the GEOMPACK package and the Bowyer-Watson algorithm. Both algorithms are incremental insertion algorithms, which means they maintain a Delaunay triangulation into which points are inserted [5]. First, `dtris2` is used to triangulate the set of initial points including the vertices along the boundary of integration and a randomly chosen point within the boundary. The centroid of the largest triangle is then inserted using the Bowyer-Watson algorithm. In the following sections, each algorithm is examined and shown how they are implemented into our integration problem.

2.1 Point Insertion Algorithms

The earliest incremental insertion algorithm was developed by Lawson [11] and is based on edge flips. When a vertex is inserted, the triangle that contains the new point is found, and the point is connected to the vertices of the containing triangle by inserting three new edges. (If the new point falls on the edge of an existing triangle, the edge is deleted, and the point is connected to the four vertices of the containing quadrilateral by inserting four new edges). The edges are placed into a stack and are tested to determine if they pass the Delaunay criterion. If not, then an edge flip is performed to remove the non-Delaunay edge. With each flip two new edges are added to the stack, and the algorithm ends when the stack is empty yielding a globally Delaunay triangulation. A pictorial representation of Lawson's algorithm is given below.

Table 2.1. Lawson's algorithm.



In 1981 A. Bowyer and D. Watson simultaneously presented an algorithm that does not depend on the use of edge flips and can easily be generalized to arbitrary dimensionality [11]. Our implementation of the Bowyer-Watson algorithm is given below and starts
with already having a Delaunay triangulation of n points with a new point, x_{n+1}, to be added.

1) Determine which triangle contains x_{n+1}. Delete this triangle and add its neighbors to a stack.

2) Pop a triangle off the stack and determine if the new point is within the circumcircle of the triangle. If yes, delete the triangle and add the neighboring triangles to the stack.

3) Repeat 2 until stack is empty.

4) Triangulate the deleted region (The method `dtris2` is used, which is discussed in the next section and our implementation is discussed in Chapter 3.).

5) Inserting the triangulation from 4 into the space that was voided by the deleted triangles provides the new Delaunay triangulation.

Figure 2.1 shows the insertion of a point using the Bowyer-Watson algorithm. The Bowyer-Watson algorithm can also be implemented from scratch with no preexisting triangulation. First, three points are chosen that created a bounding triangle that encloses all of the points that need to be triangulated. The algorithm as outlined above then follows. Once all of the points have been inserted, the bounding triangle is then deleted along with all of its connections to the inner triangulation.

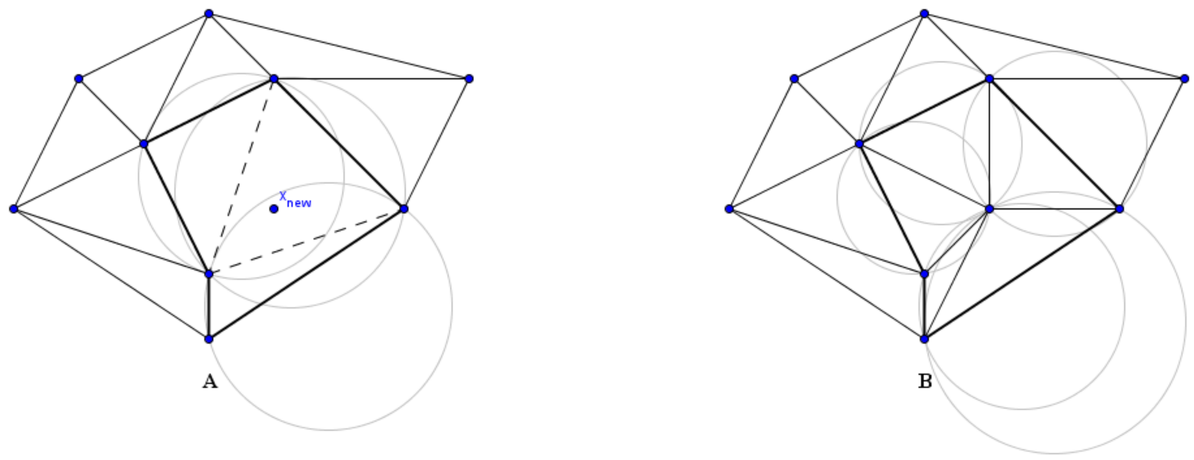


Figure 2.1. Bowyer-Watson Algorithm: **A**: Circumscribing circles that contain the new point with the edges to be deleted ; **B**: resulting triangulation

As stated above, this algorithm easily generalizes to higher dimensions. When the new point is inserted, the tetrahedron that contains the point is found, deleted and its neighbors are placed into a stack. The tetrahedra in the stack are then checked to determine if their circumsphere is empty. If the circumsphere is not empty, then the corresponding tetrahedron is deleted. Once the stack is empty, the empty polyhedron that is left is "triangulated" and the process is repeated until all points are inserted [11].

In its simplest form, this algorithm is not robust against roundoff error, though. A degenerate case can develop in which two triangles have the same circumcircle, but only one of them is deleted due to roundoff error, and the triangle that is not deleted is between the new vertex and the triangle that was not deleted. This gives an empty cavity that is not empty, and the resulting triangulation of the cavity would be"nonsensical" [11]. This problem can be avoided by using Lawson's algorithm instead. Lawson's algorithm is not absolutely robust to roundoff error, but failures occur much more sparingly compared to the simplest Bowyer-Watson algorithm. The

Bowyer-Watson algorithm can be implemented with a depth-first search of the containing triangle and will perform equally as robust as Lawson's algorithm. Another advantage of Lawson's is that it is slightly easier to implement due in part because the topological structure maintained throughout the process stays a triangulation [11]. The Bowyer-Watson was chosen due to the nice pairing that it has with the method `dtris2` below. The method keeps track of a neighbor matrix, which virtually negates the search time for the triangles that are effected by the new insertion point. The time complexity is discussed in further detail in Section 2.3.

There are many other methods that can be used for creating a Delaunay triangulation such as divide-and-conquer approaches. The first $\mathcal{O}(n \log n)$ algorithm to create a Delaunay triangulation was a divide-and-conquer approach that first created a Voronoi diagram then was dualized to form the Delaunay triangulation. Due to the unnecessarily complicated process, another divide-and-conquer approach was developed that directly constructed a Delaunay triangulation. In this approach, the existing set of points are recursively divided into two groups, each group is triangulated separately, and the groups are then merged together [7]. This algorithm proved to be as intricate and cannot easily be implemented in higher dimensions. The approach proved to not be as useful as Bowyer-Watson for our application for two reasons. First, this approach could have been used instead of the Bowyer-Watson algorithm, but it would triangulate the entire set of points after every point insertion as opposed to Bowyer-Watson, which only requires the triangulation of a small subset after each point insertion. Secondly, the divide-and-conquer approach could have been used in place of `dtris2`, but since `dtris2` is only used for small subsets of triangles, there was no added benefit to using the divide-and-conquer approach given its intricacy.

Another well-known approach is the sweepline method. This algorithm can also be implemented in $\mathcal{O}(n \log n)$, and it builds a triangulation by sweeping a horizontal line vertically across the plane with the triangulation accreting below the sweepline. When the sweepline collides with a vertex, a new edge is created connecting the vertex to the sweepline. Once the sweepline reaches the top of the circumcircle of a Delaunay triangle, the algorithm determines there is no other vertex inside that circumcircle; thus, the triangle is created. This method also has restrictions generalizing to higher dimensions and is not as easily implemented as the incremental insertion methods. Similar to the divide-and-conquer approaches, the sweepline algorithm assumes that the points to be triangulated are predetermined, so this method does not fit our particular application

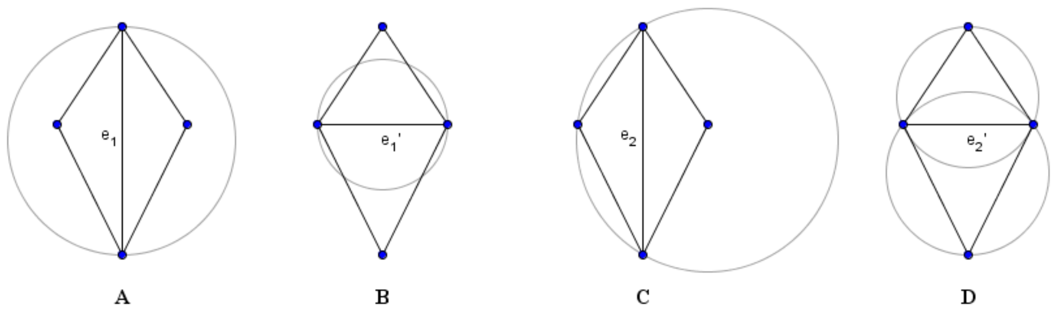


Figure 2.2. **A**: Edge e_1 is not locally Delaunay since there is no empty circumcircle, perform edge swap; **B**: e' is locally Delaunay; **C**: the left triangle does not have an empty circumcircle, so not Delaunay, perform edge swap; **D**: both triangles have empty circumcircles, so both are locally Delaunay

[7],[11].

2.2 Incremental Delaunay Triangulation Algorithm with Edge Flips

The method `dtris2` as described by Joe [5] is a variation of the algorithm given by Sloan [12]. Sloan's algorithm combines the techniques from the Bowyer-Watson and Lawson algorithms. First, a super triangle is created that encompasses all of the points to be triangulated. Then a point P is inserted into the triangulation. The triangle that contains P is found, and P is connected to the three vertices of the containing triangle to create three new triangles. The Lawson flip algorithm is then used to make sure the triangulation is still Delaunay. This process is repeated until all points have been inserted [12]. Joe uses the same outline for `dtris2` but disregards the initial bounding triangle and initially sorts the points lexicographically [5]. The algorithm in `dtris2` is outlined below by starting with a set of points S that needs to be triangulated.

1) Sort S using an ascending indexed heap sort to obtain the sorted set of indices S_s.

2) Take the first three points according to S_s and create the first triangle.

3) Add the next point according to S_s and connect the new vertex to the vertices that are "visible" to the new point.

4) Check to make sure the new triangles are Delaunay. If not, perform edge swaps until all triangles are Delaunay. These edge swaps are shown in Figure 2.2.

5) Repeat steps 3 and 4 until all points have been added to the triangulation.

The code for `dtris2` translated into MATLAB© is given in Appendix C. A crucial component of this algorithm is that edge swapping guarantees the new triangles created are both Delaunay. Welzl [13] provides the following proposition and proof that guarantees any four points that are not cocircular have exactly one Delaunay triangulation.

Proposition 2.2.1. *Given a set $P \subset \mathbb{R}^2$ of four points that are in convex position but not cocircular. Then P has exactly one Delaunay triangulation.*

Proof. Let $P = pqrs$ be a convex polygon (as shown in Figure 2.3). There are two triangulations of P: a triangulation $\mathcal{T}_1$ using the edge pr and a triangulation $\mathcal{T}_2$ using edge qs. Now consider the family $\mathcal{C}_1$ of circles through the edge pr, which contains the circumcircles $C_1 = pqr$ and $C_1' = rsp$ of the triangles in $\mathcal{T}_1$. By assumption s is not on C_1. If s is outside of C_1, then q is outside of C_1'. Consider the process of continuously moving from C_1 to C_1' in $\mathcal{C}_1$ (left image in Figure 2.3) then point q is "left behind" immediately when going beyond C_1 and only the final circle C_1' "grabs" the point s.

Similarly, consider the family $\mathcal{C}_2$ of circles through pq, which contains the circumcircles $C_1 = pqr$ and $C_2 = spq$, the latter belonging to a triangle in $\mathcal{T}_2$. As s is outside of C_1, it follows that r is inside C_2. Consider the process of continuously moving from C_1 to C_2 in $\mathcal{C}_2$ (right image in Figure 2.3). The point r is on C_1 and remains within the circle all the way up to C_2. This shows that $\mathcal{T}_1$ is Delauny, where as $\mathcal{T}_2$ is not.

The case that s is located inside C_1 is symmetric; just cyclically shift the roles of $pqrs$ to $qrsp$. □

2.3 Complexity

Now the time complexity of the `dtris2` method and the Bowyer-Watson algorithm are analyzed. Let T be the time it takes to triangulate a set of n additional points be given as

$$T = \sum_{k=1}^{n} T_k + S_k, \tag{2.3.1}$$

where T_k is the amount of time it takes to find the triangle that contains the new point and S_k the time it takes to find all of the triangles in the cavity. Since the neighbor relations are given

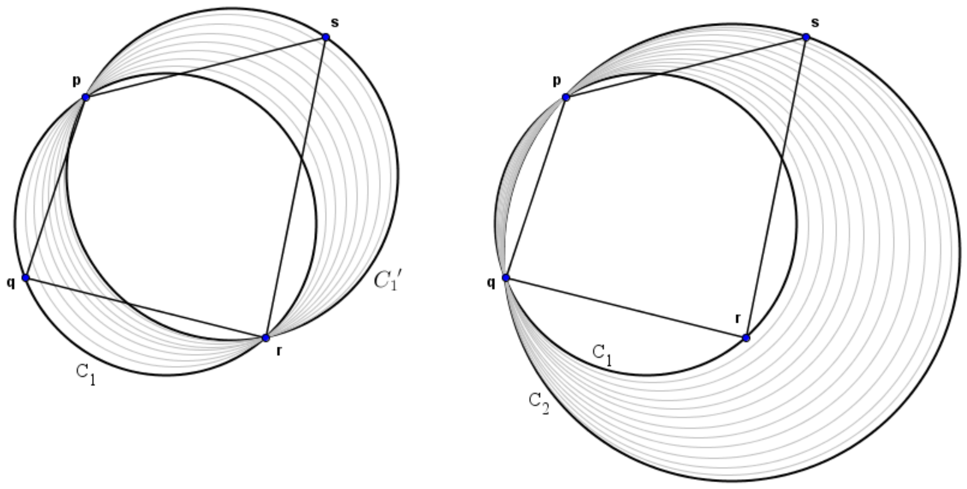


Figure 2.3. Circumcircles and containment for triangulations of four points [13]

in `dtris2`, $S_k = \mathcal{O}(1)$ because it is proportional to the number of triangles in the cavity not the number of points. The reason for this is that the search for the neighboring triangle becomes obsolete as the number of points in the triangulation increases. Experimentally, the number of triangles in the cavity per iteration is less than ten, so as n increases the number of triangles affected stays essentially constant; thus, making T_k the dominating factor. In the worst case the complexity of T_k is $\mathcal{O}(k)$, which gives us $\mathcal{O}(n^2)$ [1]. This worst case scenario happens when all existing triangles' circumcircles contain the new point at every point insertion. However, in the typical case, the number of triangles to be deleted at each point insertion does not depend on the number of existing triangles as described above. Combining an $\mathcal{O}(n \log n)$ multidimensional search for the triangle that contains the new point and the saved information of the neighbor relations between triangles, the Bowyer-Watson algorithm computes the Delaunay triangulation of n points in $\mathcal{O}(n \log n)$.

In [5], Joe discusses the time complexity of `dtris2` and determines it to also be $\mathcal{O}(m \log m)$. Since the method is only used for the initial set of points and the vertices along the hull of the cavity along with the new point, then as n increases, m stays relatively constant and is significantly smaller than n. Thus, the time complexity of `dtris2` and Bowyer-Watson together is still $\mathcal{O}(n \log n)$.

Whenever implementing a geometric algorithm, two problems always need to be addressed: geometric degeneracies and numerical errors. For the Delaunay triangulation, four or more co-circular points will result in a non-unique triangulation [6] as shown in Figure 2.3. In such a case `dtris2` will produce the triangulation on the left. Since it inserts points in lexicographically increasing order, triangle $p_1p_2p_3$ will be created first. The point p_4 will be added and triangle $p_2p_3p_4$ is created. It then checks to make sure that the triangulation is Delaunay, which it is, so the method will move to the next point without considering the triangulation on the right. Another geometric degeneracy that needs to be considered is if three or more points are too close to being co-linear. In this case, `dtris2` checks for a "healthy" triangle when inserting a new point. It determines this by checking if the third point is to the left or right of a directed ray between the initial two points of the triangle. If the third point is within a certain tolerance of the directed ray, the method will break and return a fatal error.

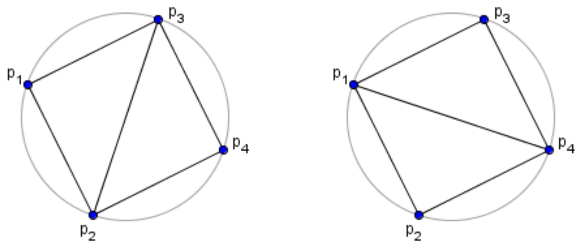


Figure 2.4. Two valid Delaunay triangulations with four co-circular points.

Numerical errors are more difficult to handle. As discussed in Section 2.1, there is a degenerate case for roundoff error in the Bowyer-Watson algorithm. Mavriplis also discusses this issue with round-off error in [7]. In general, the nature of the problem will determine the accuracy requirements of the inputs and outputs. For our implementation, double-precision arithmetic is used, and it proves to be very robust. An occasional error occurs when inserting several thousand points in `dtris2` that appears to happen when two points are too close to each other, which illustrates the round-off error problems described in Section 2.1. Since the error rarely occurred, the iteration was simply skipped but noted during the simulation process using a try/catch block.

2.4 Delaunay Integration

Now the triangulation of the integral domain is implemented in the triangular prism rules described in [3] to approximate the integral

$$\iint_D f(x,y)dxdy, \tag{2.4.1}$$

where D is the domain of integration that has been triangulated into n triangles. First, replace the function $f(x,y)$ above with a two variable polynomial function p_2 of total degree 2 whose values at the points (x_1, y_1), (x_2, y_2), and (x_3, y_3) and $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$, $(\frac{x_2+x_3}{2}, \frac{y_3+y_3}{2})$, and $(\frac{x_3+x_1}{2}, \frac{y_3+y_1}{2})$ are equal to the values at $f(x,y)$ at the same points. Then the "signed volume" under the surface given by $p_2(x,y)$ is the "volume" of the paraboloidal triangular prism with its base D_i, the lengths of the 3 parallel edges equal to $f(x_1,y_1)$, $f(x_2,y_2)$, $f(x_3,y_3)$, and the heights at the midpoints of the sides are equal to the values of $f(x,y)$ at those midpoints. This gives us a "cubature"[3] rule in two variables that is analogous to Simpson's rule in one-variable and is given by

$$\iint_{D_i} f(x,y)dxdy \approx \frac{Area(D_i)}{3}\left[f(\frac{x_1+x_2}{2},\frac{y_1+y_2}{2}) + f(\frac{x_2+x_3}{2},\frac{y_2+y_3}{2}) + f(\frac{x_3+x_1}{2},\frac{y_3+y_1}{2})\right], \tag{2.4.2}$$

where $D_i \in D$ for all $i = 1, 2, \ldots, n$. Although the 2D Simpson's rule is the main method in which our triangulation is implemented, the midpoint and trapezoidal equivalents described in [3] are also used for added performance comparison.

For the midpoint rule, let (s,t) be the centroid of D_i and replace the function $f(x,y)$ from 2.4.1 with the constant function $p_0(s,t)$. The "signed volume" under the surface given by p_0 is the "volume" of the triangular prism with the base D_i and height $f(s,t)$. This gives the "cubature" rule as

$$\iint_{D_i} f(x,y)dxdy \approx Area(D_i)f(s,t) = Area(D_i)f\left(\frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3}\right),$$

where $D_i \in D$ for all $i = 1, 2, \ldots, n$. Now replace the function $f(x,y)$ in 2.4.1 with a two variable polynomial function p_1 of total degree 1 whose value at (x_i, y_i) is equal to $f(x_i, y_i)$ for $i = 1, 2, 3$. The "signed volume" under the surface given by $p_1(x,y)$ is the "volume" of the obliquely cut triangular prism with base D_i, and the length of the three parallel edges are $f(x_1,y_1)$, $f(x_2,y_2)$ and $f(x_3,y_3)$. This gives us the two variable trapezoidal rule defined by

$$\iint_{D_i} f(x,y)dxdy \approx Area(D_i)\left[f(x_1,y_1) + f(x_2,y_2) + f(x_3,y_3)\right],$$

where $D_i \in D$ for all $i = 1, 2, \ldots, n$ [3].

2.5 Monte Carlo Integration

Monte Carlo methods are numerical methods that depend on taking random samples to approximate their results. Monte Carlo integration applies this process to the numerical estimation of integrals. In this section some of the fundamental properties of Monte Carlo integration as described by Jarosz are given. All of the definitions and descriptions below are consistent with those found in [4] but can be found in most sources that discuss probability and Monte Carlo methods.

Suppose random variable X, then the *cumulative distribution function*, or CDF, of X is defined as

$$cdf(x) = P\{X \leq x\} \tag{2.5.1}$$

The corresponding *probability density function*, or PDF, is defined as the derivative of CDF, i.e.

$$pdf(x) = \frac{d}{dx} cdf(x). \tag{2.5.2}$$

From 2.5.1 and 2.5.2, an important relationship forms that allows us to determine the probability that a random variable lies between two values:

$$P\{a \leq x \leq b\} = \int_a^b pdf(x)dx.$$

Now the expected values and variance of a random variable are investigated. Consider the random variable $Y = f(x)$ over a domain $\mu(x)$ then the *expected value* is defined as

$$E[Y] = \int_{\mu(x)} f(x)pdf(x)dx, \tag{2.5.3}$$

and the *variance* is defined as

$$\sigma^2[Y] = E\left[(Y - E[Y])^2\right], \tag{2.5.4}$$

where σ is the *standard deviation* and is the square root of the variance. From 2.5.3 and 2.5.4, it can easily be shown that the following will hold for any constant a:

$$E\left[aY\right] = aE\left[Y\right], \tag{2.5.5}$$

$$\sigma^2\left[aY\right] = a^2\sigma^2\left[Y\right].$$

Furthermore, the expected value of the sum of random variables Y_i is equal to the sum of their expected values:

$$E\left[\sum_i Y_i\right] = \sum_i E\left[Y_i\right]. \tag{2.5.6}$$

Combining these properties, 2.5.4 simplifies to the following:

$$\sigma^2\left[Y\right] = E\left[Y^2\right] - E\left[Y\right]^2.$$

Now, suppose $f(x)$ is to be integrated over $[a, b]$:

$$F = \int_a^b f(x)dx.$$

The integral, F, can then be approximated by averaging samples of the function f at random points from a uniform distribution between a and b. Given a set of n uniform random variables $X_i \in [a, b)$ with corresponding PDF of $\frac{1}{b-a}$, then the Monte Carlo estimator for F is

$$\langle F^n \rangle = (b-a)\,\frac{1}{n-1}\sum_{i=0}^{n} f(X_i). \tag{2.5.7}$$

Since $\langle F^n \rangle$ is a function of X_i, then it is a random variable as well, and this notation will be used to denote that $\langle F^n \rangle$ is an approximation of F using n samples. Intuitively, equation 2.5.7 can be viewed two ways: 1) the estimator in 2.5.7 computes the mean value of the function $f(x)$ over $[a, b]$ and multiplies by length of the interval $(b-a)$, or 2) by moving $(b-a)$ inside the summation, the estimator is choosing the height at a random evaluation of the function and averaging a set of rectangular areas computed by multiplying this height by the length of the interval $(b-a)$. It is now easy to show that the expected value of $\langle F^n \rangle$ is F:

$$
\begin{aligned}
E\left[\langle F^n\rangle\right] &= E\left[(b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(X_i)\right] \\
&= (b-a)\frac{1}{n}\sum_{i=0}^{n-1} E\left[f(X_i)\right] && \text{from eqns. 2.5.5 and 2.5.6} \\
&= (b-a)\frac{1}{n}\sum_{i=0}^{n-1}\int_a^b f(x)pdf(x)dx && \text{from eqn. 2.5.3} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\int_a^b f(x)dx && \text{since } pdf(x) = \frac{1}{b-a} \\
&= \int_a^b f(x)dx \\
&= F.
\end{aligned}
$$

As n increases, the estimator $\langle F^n\rangle$ becomes closer to F, and due to the *Strong Law of Large Numbers*, the exact solution is guaranteed in the limit:

$$P\left\{\lim_{n\to\infty}\langle F^n\rangle = F\right\} = 1.$$

For the one-dimensional case, the *convergence rate* is determined by looking at the convergence rate of the estimator's variance:

$$\sigma\left[\langle F^n\rangle\right] \propto \frac{1}{\sqrt{n}}.$$

Even though the convergence rate is slow compared to other one-dimensional techniques, it does not get exponentially worse like many other techniques as the dimension increases. For instance, a deterministic quadrature requires using n^d samples for a d-dimensional integral, but the Monte Carlo techniques provide the ability to choose any arbitrary number of points. The estimator, $\langle F^n\rangle$, can easily be extended to multiple dimensions by using random variables drawn from arbitrary PDFs and solving the following:

$$F = \int_\omega f(\bar{x})d\bar{x},$$

where $\bar{x} = x_1, \ldots, x_d$. Equation 2.5.7 now changes to

$$\langle F^n \rangle = \frac{1}{n} \sum_{i=0}^{n-1} \frac{f(X_i)}{pdf(X_i)}.$$

Similar to before when showing $E\left[\langle F^2 \rangle\right] = F$ for $pdf(x) = \frac{1}{b-a}$, the extended estimator has the correct expected value:

$$\begin{aligned}
E\left[\langle F^n \rangle\right] &= E\left[\frac{1}{n} \sum_{i=0}^{n-1} \frac{f(X_i)}{pdf(X_i)}\right] \\
&= \frac{1}{n} \sum_{i=0}^{n} E\left[\frac{f(X_i)}{pdf(X_i)}\right] \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \int_{\omega} \frac{f(\bar{x})}{pdf(\bar{x})} pdf(\bar{x}) d\bar{x} \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \int_{\omega} f(\bar{x}) d\bar{x} \\
&= \int_{\omega} f(\bar{x}) d\bar{x} \\
&= F.
\end{aligned}$$

As mentioned above the convergence rate stays constant at $\mathcal{O}(\frac{1}{\sqrt{n}})$ with the added dimensions, so

$$\sigma\left[\langle F^n \rangle\right] \propto \frac{1}{\sqrt{n}}.$$

The convergence rate can be improved by a variety of techniques that mainly deal with reducing the variance using more advanced sampling techniques, but for our purposes, the convergence rate at $\mathcal{O}(\frac{1}{\sqrt{n}})$ is satisfactory.

Chapter 3

IMPLEMENTATION AND NUMERICAL RESULTS

In this chapter the implementation of the algorithms and theory that was presented in the previous chapters is given. Then test functions are defined, and the numerical results given by testing our implementations versus other known techniques discussed in Chapter 2 are also discussed. First, the adaptive Newton-Cotes quadrature discussed in Section 1.2 is tested against adaptive Simpson's. Then adaptive Simpson's is compared to Simpson's rule over the Delaunay triangulation described in equation (2.4.2). Finally, the performance of all of the techniques are compared to each other simultaneously.

3.1 Implementation

In Sections 2.1 and 2.2, the Bowyer-Watson algorithm and the method `dtris2` were investigated. These methods were combined to create a hybrid method that is based mainly on the Bowyer-Watson algorithm. Starting with an array, P, that contains the four points representing the vertices of the rectangular integration domain and one randomly chosen point within the rectangle, the initial triangulation is constructed using the `dtris2` method. From this initial triangulation, `dtris2` produces three outputs: the number of triangles, a matrix `verts` that gives the vertices of each triangle and another matrix `nabes` that gives the neighbor relations of the triangles. The columns of each matrix refers to a triangle in the triangulation, i.e. column one refers to triangle T_1, column two refers to triangle T_2, etc. The values of `verts` are indices referencing the location of the point within P, and the values of `nabes` are positive or negative and reference a triangle

(> 0) or a boundary edge (< 0). For example, let

$$\texttt{verts} = \begin{bmatrix} 2 & 5 & 5 & 2 \\ 1 & 1 & 3 & 5 \\ 5 & 3 & 4 & 4 \end{bmatrix}$$

and

$$\texttt{nabes} = \begin{bmatrix} -7 & 1 & 2 & 1 \\ 2 & -10 & -14 & 3 \\ 4 & 3 & 4 & -3 \end{bmatrix}$$

then T_1 has vertices visited counterclockwise P_2, P_1 and P_5, and T_1 also neighbors triangles T_2 and T_4 along the edges P_1P_5 and P_5P_2 with edge P_2P_1 being a boundary edge.

Then the affected region is found by determining which triangles' circumcircles contain the new point. The boundary of the affected region is then stored in a temporary matrix with the newly inserted point. The region is then triangulated using `dtris2`. If the region is concave then the triangles that are created from bridging the concave vertices are deleted. If the region is convex then the triangulation is correct, and there is no need to delete any triangles.

Now that the affected region is triangulated, it needs to inserted back into the main triangulation. To do this, the referencing between the neighbor and vertex matrices of the affected region need to be inserted into the neighbor and vertex matrix for the main triangulation. First the point references in the vertex matrix are corrected. When triangulating the affected region with m points, `dtris2` labels the points $1, 2, \ldots, m$, so the references need to be changed to their original numbering from P that consists of n points. Similar to the vertex matrix for the affected region, the neighbor matrix is also updated to reflect the numbering of the whole triangulation. While correcting the numbering of `nabes`, the entries that are boundary edges for the affected region are set to 0 if they are not a boundary edge for the entire triangulation. If they are a boundary edge for both the affected region and larger triangulation, then the entry remains the same. Then the vertex and neighbor matrices for the affected region are merged with `verts` and `nabes` corresponding to the overall triangulation. This is done by first noticing that if m triangles were affected, then the new triangulation consists of $m + 2$ triangles, so each column in `nabes` is filled with one from the affected region's triangulation and the two extra triangles are placed onto the end. The corresponding columns in the vertex matrix for the affected region are added in the same manner to `verts`. The vertex matrix is now complete and describes the triangulation with the new point added. Lastly, all of the zeros in `nabes` are changed to their correct triangle references, and all of the negative entries are updated as well. The triangulation is now complete with correct vertex and neighbor matrices, `verts` and `nabes`, respectively. The code can be found in Appendix D.

The algorithm described above is used in the cubature rules described in Section 2.4. The implementation of each of the cubature rules follows the same general outline with the only difference being at what points the function is being evaluated as given by each rule. First, an initial triangulation is found using $dtris2$ along with the areas of each triangle. The areas are then placed array in increasing order, and a matrix containing the boundary edge information is also created. Then the functions is "integrated" using one of the three cubature rules described in [3] (Simpson's, midpoint and trapezoid). The error is then calculated to see if it is within the given tolerance. If the volume is not within the given tolerance then the triangle with largest area is selected for refinement, and its centroid is computed. This point becomes the new point to be inserted and the algorithm above is run to determine the new triangulation. After triangulating, the areas of the triangles affected during the triangulation are deleted, and the new ones are calculated and sorted in ascending order. The two arrays of areas are then merged together. This process repeats until the volume is within the given tolerance or a maximum number of iterations is reached. This method involving the cubature rules is not implemented adaptively, so the error at each step is compared to the previous step. However, the Voronoi Newton-Cotes method is implemented adaptively similar to our base case of the two-dimensional adaptive Simpson's Rule.

3.2 Integrands

In [14], Yu and Sheu examine solving the following double integral using the Mean-Value theorem for integrals:

$$\int_0^{2\pi} \int_0^R f(r, \theta, s, \phi, n) dr d\theta,$$

where $s, \phi, R \in \mathbb{R}$, $R > 0$, $n \in \mathbb{Z}^+$ and $f(r, \theta, s, \phi, n)$ is one of the following functions:

$$A(r,\theta,s,\phi,n) = r\exp\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\cos\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right], \tag{3.2.1}$$

$$B(r,\theta,s,\phi,n) = r\exp\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\sin\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right], \tag{3.2.2}$$

$$C(r,\theta,s,\phi,n) = r\sin\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\cosh\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right], \tag{3.2.3}$$

$$D(r,\theta,s,\phi,n) = r\cos\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\sinh\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right], \tag{3.2.4}$$

$$E(r,\theta,s,\phi,n) = r\cos\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\cosh\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right], \tag{3.2.5}$$

$$F(r,\theta,s,\phi,n) = r\sin\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\cos\left[(n-k)\phi+k\theta\right]\right]\sinh\left[\sum_{k=0}^{n}\binom{n}{k}s^{n-k}r^k\sin\left[(n-k)\phi+k\theta\right]\right]. \tag{3.2.6}$$

Even though n can be an any integer such that $n \geq 1$, it is only chosen to be between 1 and 3. When n is increased, the results would become quite large ($\geq 10^6$) most of the time, which made it more difficult to get a good graph and harder to determine what could be causing the inaccuracies.

The methods were also tested on functions of the form

$$\int_c^d \int_a^b x^i y^j \, dx dy$$

where $a, b, c, d \in \mathbb{R}$, $i, j \in \mathbb{Z}^+$ and $i + j \leq 5$.

Analytical solutions for each $f(r, \theta, s, \phi, n)$ is provided by Yu and Sheu, so the relative errors of each trial could easily be calculated. Similarly, the analytical solutions for the monomials can be found, so the relative error could easily be calculated.

3.3 Results

The Voronoi Newton-Cotes method is initially tested against Simpson's rule on the set of monomials described above. During the first several runs, the Voronoi sampling only chose 3 additional points between our boundaries at each step similar to how Simpson's rule finds the three midpoints (quartiles) between the boundaries. For all of the simulations shown in the table, a tolerance of $\epsilon = 0.0001$ is used, and the function is integrated over four randomly chosen points to create a rectangle with vertices $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$. The full results of one of the simulations are given in Table E.1 in Appendix E, and the graphs of the functions are given in Table F.1 in Appendix F. From Table 3.1, it is obvious that the Voronoi Newton-Cotes Method is exact (within machine epsilon) for polynomials of degree two or less but is not always exact for the polynomials of degree three. As given in Equation 1.2.3, this is expected since the interpolating polynomial is only exact through degree two, and there is no additional degrees of accuracy since the error term is only proportional to $f^{(3)}$. Also, it is clear that Simpson's rule is exact through degree three as expected as illustrated in Table 3.1.

Table 3.1. Condensed Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$ and $\epsilon = 0.0001$.

i	j	AS Time	AS Rel. Error	VNC Time	VNC Rel. Error
0	0	0.008355225	1.45465E-16	0.007077289	0
0	1	0.000445179	1.83618E-16	0.000605689	1.83618E-16
0	2	0.000444923	1.15589E-16	0.000636664	1.15589E-16
0	3	0.000717303	1.45154E-16	0.028169894	4.20143E-08
1	0	0.00050585	0	0.000657144	0
1	1	0.000572153	1.35526E-16	0.000752119	2.71052E-16
1	2	0.000714487	3.41259E-16	0.000930037	3.41259E-16
2	0	0.000655608	1.18479E-16	0.000882165	1.18479E-16
2	1	0.000450042	4.48665E-16	0.000632312	1.49555E-16
3	0	0.000457722	1.16218E-16	0.498908685	8.96198E-08

For the higher degree (≥ 4) polynomials shown in Table E.1 in Appendix E, the Voronoi Newton-Cotes method performs adequately giving four additional orders of accuracy for the given epsilon in many cases. However, there are three cases that notably stand out: $f(x, y) = x^3y^2$, $f(x, y) = x^4y^1$ and $f(x, y) = x^5$, which are examined further in Table 3.2. Originally the maximum number of iterations was set to 5000, and all three of those cases reached the maximum number, so the maximum iterations was increased to see how many it would take to achieve a desirable accuracy. When increasing the maximum iterations to 15000, the following results were found and are shown in Table 3.2. The relative errors again give us an additional four or five digits of accuracy

Table 3.2. 15000 Max Iteration Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$ and $\epsilon = 0.0001$.

i	**j**	**AS Time**	**AS Rel. Error**	**VNC Time**	**VNC Rel. Error**	**Iterations**
3	2	0.000457210	0	3.063991277	1.30530E-08	6497
4	1	0.116197996	4.03868E-08	4.288516134	2.32547E-09	9069
5	0	0.211611352	7.10947E-08	7.329204530	3.50050E-08	14869

and even outperform adaptive Simpson's rule on one of the runs. Unfortunately, the amount of time taken spiked drastically. Since the desired accuracy was finally achieved, the next step was to try to improve the speed of the method.

As stated above, the trials were initially run using the adaptive Voronoi Newton-Cotes with the Voronoi sampling only being used for three additional points along each axis. When examining the intermediate steps of both methods, the Voronoi Newton-Cotes had very long streaks of not adding any values to the total volume. This meant it was spending a lot of time finding an accurate enough approximation to be able to move on to the next quadrant. When looking at how the points were generally distributed between the two values, there were large gaps on the tails of the interval giving large areas to approximate over on the ends, which would then need more refinement. Since Simpson's rule uses the midpoints of each cell at every step, the empty space is filled much more evenly than with the Voronoi sampling; therefore, Simpson's rule was always using significantly fewer iterations. In an effort to correct this, more points were sampled at each step (19 additional for a total of 21 with the endpoints) but would only use the first, second and third quartiles of the sampling. As shown in Table 3.3, the guaranteed accuracy through degree two for Voronoi Newton-Cotes and degree three for Simpson's rule remains unchanged. In Table E.2, the times for the larger degree polynomials and did end up improving with the accuracy remaining roughly the same as before. Since increasing the number of points helped the speed of the algorithm and also gave us similar accuracy, the Voronoi sampling of 19 points over the three point method is used inn the rest of the trials described in this thesis. Even with the additional increase in speed, adaptive Simpson's provides both better accuracy and speed on these simple functions overall, though. Now both methods are tested on more complicated functions.

Table 3.3. Condensed Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Monomials with additional sampling and $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$ and $\epsilon = 0.0001$.

i	j	AS Time	AS Rel. Error	VNC Time	VNC Rel. Error
0	0	0.001667055	0	0.005648840	1.45465E-16
0	1	0.000532987	0	0.001388274	3.67237E-16
0	2	0.000424188	2.31179E-16	0.001337843	0
0	3	0.000500475	1.45154E-16	0.022388000	4.90503E-08
1	0	0.000477947	0	0.001335283	4.29461E-16
1	1	0.000468731	1.35526E-16	0.001387250	0
1	2	0.000577018	0	0.001409522	1.70629E-16
2	0	0.000573178	2.36958E-16	0.001332467	4.73917E-16
2	1	0.000468475	2.99110E-16	0.001327091	1.49555E-16
3	0	0.000555770	1.16218E-16	0.432514594	4.00847E-08

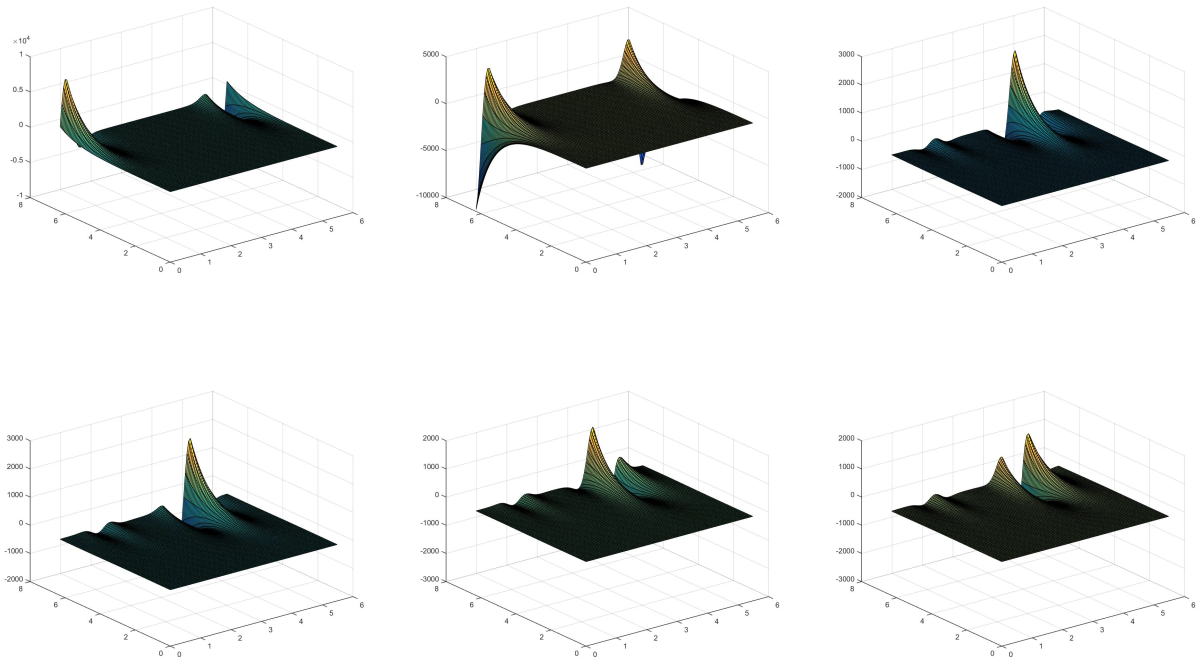
Next tests were run on the functions described above from [14] using Voronoi Newton-Cotes and Simpson's rule. For all of the simulations shown in Table 3.4, again a tolerance of $\epsilon = 0.0001$ is used, and the function is integrated over the rectangle given by $a = 0$, $R = 5.480255137$, $c = 0$, $d = 2\pi$ with R being a randomly chosen point. The parameters s, ϕ and n are randomly chosen to be $s = 2.444171059$, $\phi = 5.69125859039527$ and $n = 1$.

Table 3.4. Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Functions A-F with $a = 0$, $R = 5.480255137$, $c = 0$, $d = 2\pi$, $s = 2.444171059$, $\phi = 5.69125859039527$, $n = 1$ and $\epsilon = 0.0001$.

f Type	AS Time	AS Rel. Error	VNC Time	VNC Rel. Error
A	11.8374406	2.729767377	16.11166813	1.182096082
B	12.04885046	1.418534859	16.38065796	0.842777814
C	12.13505427	1.611736343	16.61918614	1.067504856
D	13.03777242	2.358085297	17.13948942	0.634025761
E	12.37735201	2.191400672	18.07931543	0.806867337
F	12.78167719	1.697312935	17.24927674	0.899201229

It is clear to see from Table 3.4 that neither Simpson's rule nor the Voronoi Newton-Cotes method performs well on the six functions. Looking at the graphs of these functions in Table 3.5, these functions have fairly sharp high and low peaks and are also oscillatory. Simpson's rule is known to break down with oscillatory functions, e.g. integrating $|\sin(x)|$ from $[0, 2\pi]$ using Simpson's rule arrives at an area of 0 when the true area should be 4. Considering most of the functions appear to have equally high and low peaks, a similar cancellation could be affecting the results. It is easy to see that the Voronoi Newton-Cotes method could also run into a similar problem given the right function and "midpoint."

Table 3.5. Graphs of Functions A-F with $a = 0$, $R = 5.480255137$, $c = 0$, $d = 2\pi$, $s = 2.444171059$, $\phi = 5.69125859039527$, $n = 1$ and $\epsilon = 0.0001$.



Similar to our simulations comparing the methods on monomials, the results displayed in Table 3.4 have a maximum number of iterations of 5000. As before, the maximum number of iterations was increased, this time all the way to 20000, but this did not improve the accuracy by a significant amount (rarely getting even one additional order of accuracy). Ignoring the accuracy and looking at the times it took to complete the simulation, Simpson's rule still trumps the Newton-Cotes method. Since both of the simulations are now executing the same number of iterations (5000), it is highly likely the time difference is attributed to the extra work the Voronoi Newton-Cotes has to do to perform the extra sampling.

Now that the Voronoi Newton-Cotes method and Simpson's rule have been compared against each other, Simpson's rule is now compared against the Simpson's rule analog using the Delaunay triangulation, which will be called Simpson's cubature rule, that was discussed in Section 2.4. Similar to the analysis performed above for the Voronoi Newton-Cotes and Simpson's rule, the performances of Simpson's rule and Simpson's cubature rule will first be compared over the monomials to make sure the expected guaranteed accuracies hold. Then they will be tested on the higher degree monomials. Both methods use a tolerance of $\epsilon = 0.0001$ over the rectangle $a = -1.62110966800282$, $b = -1.37432067059289$, $c = -3.3239379751915$, $d = -1.72003265166653$. The results for the entire simulation are given in Table E.3 in Appendix E. Looking at Table 3.6, both Simpson's rule and Simpson's cubature rule perform well through degree three and two polynomials, respectively. Simpson's cubature rule does not give us the extra third order of accuracy as shown earlier when comparing the Voronoi Newton-Cotes and Simpson's rule, but it does yield an additional two to three extra orders of accuracy, which is still quite adequate.

Table 3.6. Simpson's Cubature Rule (SC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -1.62110966800282$, $b = -1.37432067059289$, $c = -3.3239379751915$, $d = -1.72003265166653$ and $\epsilon = 0.0001$.

i	**j**	**SC Time**	**SC Rel. Error**	**AS Time**	**AS Rel. Error**
0	0	0.016313693	2.80482E-16	0.000848376	2.80482E-16
0	1	0.005531337	7.78505E-16	0.000447996	6.67290E-16
0	2	0.006637502	5.11924E-16	0.000467707	5.11924E-16
0	3	0.025780223	2.19831E-05	0.000461051	2.54077E-16
1	0	0.005419978	1.87274E-16	0.000468731	1.87274E-16
1	1	0.005155789	1.48513E-16	0.000481531	1.48513E-16
1	2	0.009143973	3.05773E-06	0.000441084	6.83606E-16
2	0	0.005003726	4.99029E-16	0.000488443	7.48543E-16
2	1	0.005474249	2.78244E-06	0.000503547	3.95743E-16
3	0	0.004566226	1.31798E-07	0.000488187	0

When looking at their differences in speed, Simpson's cubature rule is quite slow, even for functions for which it is exact, compared to Simpson's rule. Similar to the Voronoi Newton-Cotes rule, Simpson's cubature rule has to take the extra time to triangulate. The triangulation is slightly more time consuming than the Voronoi sampling, so this is why there is a larger gap on average between Simpson's cubature rule and Simpson's rule than the gap between Voronoi Newton-Cotes and Simpson's rule.

Looking at the higher degree (≥ 4) polynomials in Table E.3, Simpson's cubature rule performs about the same as it did for degree three in terms of accuracy. In the majority of cases, it gives at least one additional order of accuracy but does not perform as well as adaptive Simpson's. There are a couple exceptions where they perform equally as well or the Simpson's cubature performs better such as $f(x, y) = x^4$ and $f(x, y) = x^5$. Since Simpson's cubature is always slower or the same speed as Simpson's rule, it is easy to see that it does an adequate job, but Simpson's rule outperforms it in all categories for the simple functions, which is to be expected.

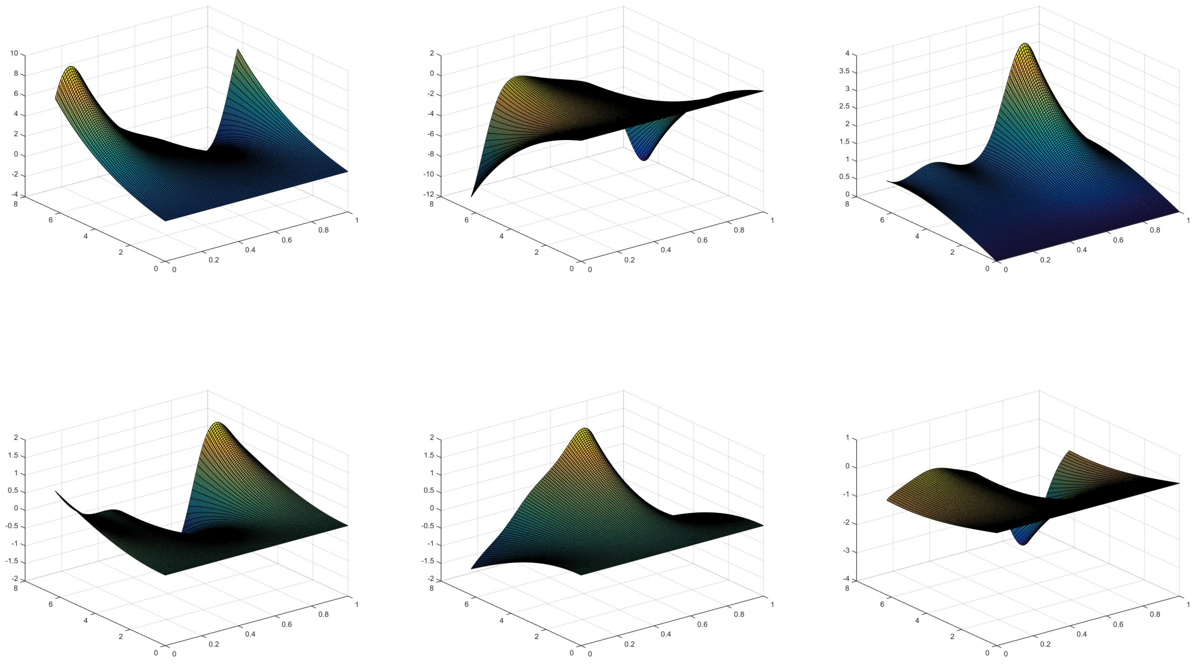
Table 3.7. Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Functions A-F with $a = 0$, $R = 1$, $c = 0$, $d = 2\pi$, $s = 1.83468664481796$, $\phi = 5.71912370455419$, $n = 1$ and $\epsilon = 0.0001$.

f Type	**SC Time**	**SC Rel. Error**	**AS Time**	**AS Rel. Error**
A	0.181542123	0.008030536	2.271972965	3.01322E-08
B	0.162510249	0.009020936	2.453636943	2.98341E-08
C	0.091698284	0.000826563	1.233137896	3.37518E-07
D	0.263396794	0.125639992	1.315914670	1.77746E-05
E	0.447086736	0.054360408	1.355982622	2.10402E-06
F	0.071517238	0.025073541	1.362870489	5.59632E-08

Simpson's cubature and Simpson's rule are now compared on the functions *A*-*F*. Table 3.7 shows that even though adaptive Simpson's gives much better accuracy, Simpson's cubature is either quicker or the same speed as Simpson's rule. Looking at the figures in Table 3.8, the graphs still have peaks and valleys like the previous example, but they are much less steep (only reaching as high as ten as opposed to several thousand in the previous example) and do not seem to be as oscillatory as the previous example. This definitely helps the accuracy of each method, but mainly helps Simpson's rule. Over the many simulations run, it was noticed that the convergence for these functions is quite slow. Since Simpson's cubature is not implemented adaptively, the method will stop when the current iteration and the previous iteration are within ϵ of each other. These two facts combined lead to the method terminating too early, which also explains its performance in speed.

Lastly, all of the methods presented in the previous chapters including Monte Carlo integration, midpoint rule, and trapezoid rule are compared against one another. Tables E.4 and E.5 in Appendix E contain the accuracy and run times for each method, and the graphs of each monomial are given in Table F.2 in Appendix F. Table 3.9 confirms that the midpoint Delaunay triangulation is accurate through constant functions (with a bonus of accuracy through degree one), and the trapezoidal Delaunay triangulation method is accurate through degree one, but both of their accuracies dip significantly as the polynomials have higher degree.

Table 3.8. Graphs of Functions A-F with $a = 0$, $R = 1$, $c = 0$, $d = 2\pi$, $s = 1.83468664481796$, $\phi = 5.71912370455419$, $n = 1$ and $\epsilon = 0.0001$.



The Monte Carlo method is also shown in Table 3.9 and was run for 50000 iterations. This number proved to be large enough to give a competitive accuracy without taking a significant amount of time. Since the Monte Carlo method is not a deterministic quadrature like the other methods, there is no guaranteed exactness for a specific degree (except when $f(x, y)$ is constant), so it does not perform as well with regards to accuracy for the lower degree polynomials when the other methods are exact. However, Table 3.9 illustrates that it still does a good job of approximating the functions and consistently provides three digits of accuracy.

For the lower degree (≤ 3) polynomials given in Table E.4, adaptive Simpson's still remains the superior choice in both time and accuracy. The Voronoi Newton-Cotes and Simpson's cubature rule perform giving several digits of accuracy and occasionally matching adaptive Simpson's rule. Both still lag behind in speed as shown in Table E.5, which is consistent with the analysis provided above.

Now the methods are compared on monomials with degree ≥ 4. Two interesting cases are examined further with their results presented in Tables 3.9 and 3.10 as well. When $f(x, y) = x^2y^2$ our Voronoi Newton-Cotes method and Simpson's rule provide exact solutions with Monte

Carlo and Simpson's cubature rule performing respectably giving three digits of accuracy. Even though Simpson's rule and the Voronoi Newton-Cotes provide the same level of accuracy, adaptive Simpson's rule is roughly three times faster than the Voronoi Newton-Cotes, so Simpson's rule is still a superior choice. On the other hand, Voronoi Newton-Cotes still provides better accuracy and speed than the other four methods.

When $f(x, y) = x^5$, it is clear that with respect to accuracy Simpson's rule, Simpson's cubature and Monte Carlo perform the best with the midpoint and trapezoid Delaunay triangulations performing about the same and Voronoi Newton-Cotes performing the worst. However, when reviewing each method's performance with respect to time, adaptive Simpson's rule is the second worst performer with Simpson's cubature rule and Monte Carlo performing the best. As discussed earlier with both Simpson's rule and the Voronoi Newton-Cotes method, additional iterations can be expensive with respect to time. The Voronoi Newton-Cotes and Simpson's rules are still capped at only 5000 iterations, but in this case, that amount is still too expensive. If needed, the iterations could be increased to gain more accuracy with Simpson's rule, but given its performance on this example, the added digits of accuracy could be extremely expensive with time. The Voronoi Newton-Cotes method performs the worst in both accuracy and speed, which could be improved by increasing the number of sampled points to greater than 19, but based on previous results, this could marginally increase the accuracy but not improve the speed at all.

Table 3.9. Condensed accuracy only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on Monomials with $a = 2.51778949114543$, $b = 5.67194769326589$, $c = -2.98410546965195$, $d = 5.22175955533465$ and $\epsilon = 0.0001$.

i	j	MDT Rel. Error	TDT Rel. Error	SC Rel. Error	AS Rel. Error	VNC Rel. Error	MC Rel. Error
0	0	1.37263E-16	1.37263E-16	0	1.37263E-16	2.74525E-16	0
0	1	1.22684E-16	1.22684E-16	1.22684E-16	0	1.22684E-16	0.005362692
1	0	1.34083E-16	0	1.34083E-16	0	0	0.001456460
2	2	0.041796026	0.391609471	0.003020576	2.90959E-16	2.90959E-16	0.006233964
5	0	0.022003814	0.087866804	0.001089610	0.009776169	0.888547317	0.004548625

Lastly, each of the methods is compared over the functions A-F. As seen previously and now in Table 3.11, none of the methods provide much accuracy on the oscillatory functions. Monte Carlo is a little bit of an exception since it provides two to three digits of accuracy for all but function A.

Table 3.10. Condensed time only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on Monomials with $a = 2.51778949114543$, $b = 5.67194769326589$, $c = -2.98410546965195$, $d = 5.22175955533465$ and $\epsilon = 0.0001$.

i	**j**	**MDT Time**	**TDT Time**	**SC Time**	**AS Time**	**VNC Time**	**MC Time**
0	0	0.264221362	0.048522268	0.021801254	0.006768572	0.017867854	0.352805694
0	1	0.010175386	0.009185188	0.005659591	0.000578810	0.004228822	0.308191995
1	0	0.005873861	0.00499195	0.005412554	0.000433404	0.001324019	0.305286168
2	2	0.172706884	0.024697353	0.053885158	0.000398588	0.001289971	0.303935782
5	0	0.159462088	0.113784976	0.038072452	1.719201267	6.147717949	0.30707687

From Table 3.11, Simpson's rule and the Voronoi Newton-Cotes both perform extremely poorly with regards to accuracy and time. Analyzing the graphs of the functions in Table 3.13, the functions again have fairly steep peaks and valleys with a couple of graphs maxing out in the low thousands. As discussed earlier, the high peaks and valleys that appear in all of the graphs could cause issues with the Newton-Cotes based methods.The maximum iterations could also be increased to greater than 5000, but this would only increase the run times of adaptive Simpson's and Voronoi Newton-Cotes, which are already extremely long compared to the other methods as shown in 3.12.

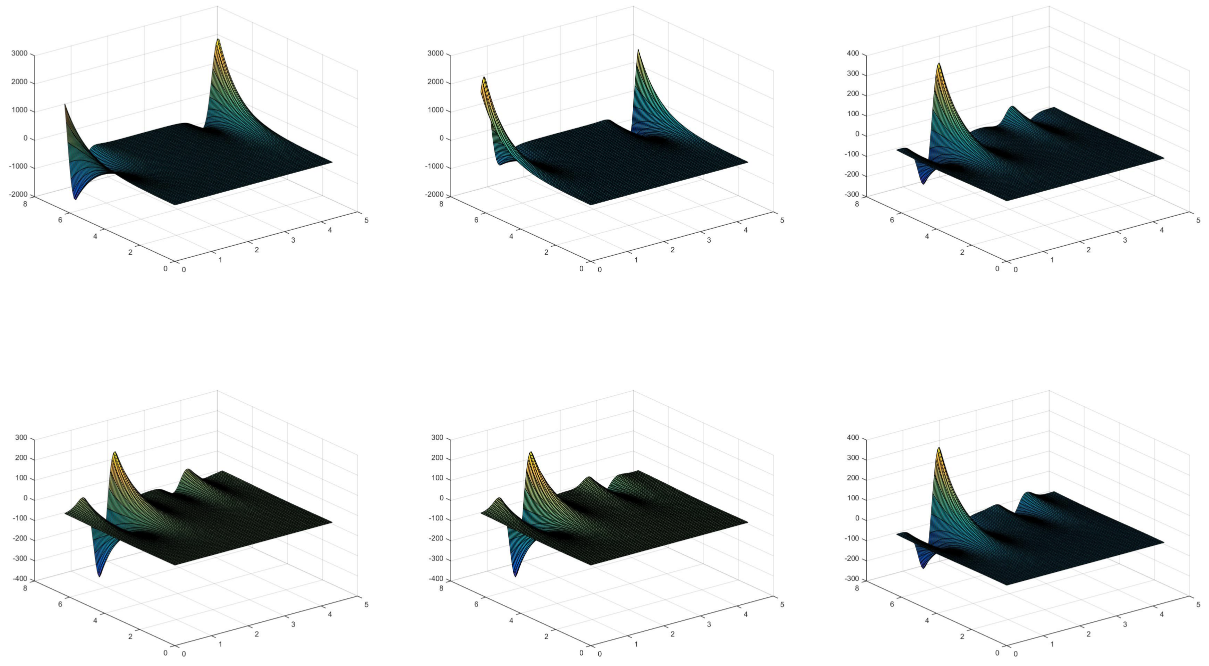
Table 3.11. Accuracy only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on functions *A*-*F* with $a = 0$, $R = 4.310689426030381$, $c = 0$, $d = 2\pi$, $s = 2.35651382285138$, $\phi = 0.387434275655817$, $n = 1$ and $\epsilon = 0.0001$.

f Type	**MDT Rel. Error**	**TDT Rel. Error**	**SC Rel. Error**	**AS Rel. Error**	**VNC Rel. Error**	**MC Rel. Error**
A	0.683714451	10.03800011	0.352786127	1.554339606	1.061044439	0.132485128
B	0.6990499	8.567750035	0.471139253	1.418966123	1.049486153	0.020218433
C	0.607982921	2.107507132	0.437277931	1.263018939	0.72749257	0.081858071
D	0.620570225	0.185445349	0.026569987	2.142429562	1.288711192	0.009542655
E	0.017144236	2.145806657	3.902967839	1.81321621	1.228863182	0.002194894
F	4.43223692	2.388462115	3.178847867	1.369362554	1.176581187	0.03489333

Table 3.12. Time only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on functions A-F with $a = 0$, $R = 4.310689426030381$, $c = 0$, $d = 2\pi$, $s = 2.35651382285138$, $\phi = 0.387434275655817$, $n = 1$ and $\epsilon = 0.0001$.

f Type	**MDT Time**	**TDT Time**	**SC Time**	**AS Time**	**VNC Time**	**MC Time**
A	0.155683822	0.160757179	0.173916472	12.23576955	17.99852238	2.672148681
B	0.240922523	0.183981267	0.16088441	12.23258725	17.25799038	2.696071642
C	0.268317065	0.204237321	0.124068137	12.82676935	17.25403164	2.747159004
D	0.476965478	0.105453539	0.230742016	12.74109161	17.56966932	2.702983829
E	0.296183923	0.241752978	0.029039326	12.81275861	17.46210201	2.725147064
F	0.085615529	0.193632883	0.052914928	12.95576287	17.38970056	2.761395790

Table 3.13. Graphs of Functions A-F with $a = 0$, $R = 4.310689426030381$, $c = 0$, $d = 2\pi$, $s = 2.35651382285138$, $\phi = 0.387434275655817$, $n = 1$ and $\epsilon = 0.0001$.



Chapter 4

CONCLUSIONS AND FUTURE WORK

This thesis presents two methods for solving a numerical integration problem: a second degree Newton-Cotes method combined with a Voronoi sampling technique and using a Delaunay triangulation to divide the integration domain into triangles to integrate over. These two methods are compared to a midpoint and trapezoid rule over triangles, adaptive Simpson's rule and Monte Carlo integration. In Chapter 3 the results are presented and show that the Voronoi Newton-Cotes method and Delaunay triangulation Simpson's rule perform adequately on simple functions such as monomials, but neither performs nearly as well as adaptive Simpson's with regards to accuracy and speed. When comparing their performances over more complicated functions such as those found in the first part of Section 3.2, all of the methods perform poorly in accuracy and run time and are not viable methods for solving these problems. In the end, the Voronoi Newton-Cotes and Delaunay triangulation methods can provide adequately accurate results most of the time, but adaptive Simpson's is still more reliable in both accuracy and speed.

There are a couple improvements that could be made to the Simpson's rule with Delaunay triangulation. To improve the accuracy of the Delaunay cubature rule, it could be implemented adaptively by comparing locally instead of globally after each step. In the hybrid algorithm the `dtris2` method and the Bowyer-Watson algorithm are combined. Since the triangulation puts the method at a disadvantage compared to adaptive Simpson's rule, the algorithm could be improved to attempt to reduce the time taken to triangulate. To do this, the Bowyer-Watson algorithm could be implemented using an object-oriented language such as Java and create a data structure that could hold all of the information for the triangle such as its vertices, neighbors and centroid. This would eliminate the use of the `dtris2` method entirely, which could improve the run time of the triangulation. The next step would be to perform a similar analysis in higher dimensions to see how our particular method handles the *curse of dimensionality*, which most methods struggle to handle.

REFERENCES

[1] C.A. Arens The Bowyer-Watson Algorithm: An efficient implementation in a database environment TU Delft, July 2002

[2] F. Aurenhammer and R. Klein. "Voronoi Diagrams," *Handbook of Computational Geometry*, Ed. J.R. Sack and J. Urrutia, North Holland, 2000, pp. 203-292.

[3] S. R. Ghorpade and B. V. Limaye. *A Course in Multivariable Calculus and Analyis*, pp. 346-361, Springer, 2010

[4] W. Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media* University of California, San Diego, 2008.

[5] B. Joe. GEOMPACK - A Software Package for the Generation of Meshes using Geometric Algorithms, *Advanced Engineering Software*, Vol. 13, No. 5/6, pp. 325-331, 1991.

[6] D. Lischinski Incremental Delaunay Triangulation, *Graphics Gems IV*, pp. 47-59, 1994.

[7] D.J. Mavriplis. Front Delaunay Triangulation Algorithm Designed for Robustness, Institute for Computer Applications in Science and Engineering, ICASE Report No. 92-49, October 1992.

[8] S.E. Mousavi, H. Xiao and N. Sukumar. Generalized Gaussian Quadrature Rules on Arbitrary Polygons, *International Journal for Numerical Methods in Engineering*, Vol. 82, Issue 1, pp. 99-113, 2010

[9] M. J. Ouellette and E. Fiume. On Numerical Solutions to One-Dimensional Integration Problems with Applications to Linear Light Sources, *ACM Transactions on Graphics*, Vol. 20, No. 4, pp. 232-279, 2001

[10] S. Rebay. Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm, *Journal of Computational Physics*, Vol. 106, pp. 125-138, 1993.

[11] J. R. Shewchuk. Lecture Notes on Delaunay Mesh Generation, Department of Electrical Engineering and Computer Science, UC Berkeley, September 1999.

[12] S. W. Sloan. A fast algorithm for constructing Delaunay triangulations in the plane, *Adv. Eng. Software*, Vol. 9, pp. 34-55, 1987.

[13] E. Welzl. Lecture Notes, Chapter 6: Delaunay Triangulations, Department of Computer Science, Swiss Federal Institute of Technology Zurich 2013

[14] C. Yu and S. Sheu. Using Mean Value Theorem to Solve Some Double Integrals, *Turkish Journal of Analysis and Number Theory*, Vol. 2, No. 3, pp. 75-79, 2014

[15] H. Zimmer. Voronoi and Delaunay Techniques, *Proceedings of Lecture Notes, Computer Sciences*, No. 8, 2005.

Appendices

Appendix A

VORONOI SAMPLING MATLAB CODE

```
function y = Voronoi6(a,b,no_of_pts)

% Voronoi6 samples points between a and b according to the midpoint of
% voronoi cells.

x(1,1) = 1;
x(1,2) = a;
x(2,1) = 3;
x(2,2) = a + (b a)*rand(1);
x(3,1) = 2;
x(3,2) = b;



n = 3;
numpts = 3;

for j = 0:no_of_pts 2
    V = zeros(n+1,1);
    V1 = zeros(n+1,1);
    x1 = sortrows(x,2);


% Find Midpoints
    for k = 1:n 1
         V(1) = a;
         V(k+1) = (x1(k+1,2)+x1(k,2))/2;
         V(n+1) = b;
    end

% Find Voronoi Cells
    for i = 1:n
         V1(1) = 0;
         V1(i+1) = V(i+1) V(i);

```

```
   end

    multiple = false;
    maxwidth = 0;
    maxrow = 2;

    % Determine where the max width of a V cell occurs
    for q = 2:n+1
       if abs(V(q) V(q 1))>=maxwidth
           if abs(V(q) V(q 1))>maxwidth
               maxwidth = abs(V(q) V(q 1));
               maxrow = q;
           else
               multiple = true;
           end
       end
    end

    newpt = (V(maxrow 1) + V(maxrow))/2;

    % Determine where the point will go
    for r = numpts: 1:2
        if newpt<x1(r,2) && newpt>x1(r 1,2)
            newrow = r;
        end
    end

%      newrow
    % Shift all the pts down to make place for new pt

    for s = numpts: 1:newrow
        x1(s+1,:) = x1(s,:);
    end

    % Add new pt in the vacated row
    n = n+1;
    x1(newrow,1) = numpts+1;
    x1(newrow,2) = newpt;
    x1(newrow,3) = 0;
    numpts = numpts+1;

    x = x1;
end

y = x(:,2)';

end
```

Appendix B

VORONOI NEWTON-COTES MATLAB CODE

```
function [approx,count] = vadapt3(a,b,c,d,tol,s,phi,n,exp1,exp2,
    funcID)

% Area of entire grid for epsilon purposes
A = (b a)*(d c);

level = 1;
NQ = 0;

approx = 0;
bool = 1;
count = 0;

coords = [1,3,1,3;
          3,5,1,3;
          3,5,3,5;
          1,3,3,5];

while bool == 1 && count < 15000
    count = count + 1;
%       a
%       b
%       c
%       d
    % Determining 3 new voronoi point for the particular quadrant
    x1 = Voronoi6(a,b,3);
    y1 = Voronoi6(c,d,3);

    x=x1;
    y=y1;
%       x = [x1(1),x1(6),x1(11),x1(16),x1(21)];
%       y = [y1(1),y1(6),y1(11),y1(16),y1(21)];
    % Area of quadrant to determine portion of entire grid
    A_hat = (b a)*(d c);


```

```
  % Simpson's over whole area then sum of simpson's over 4 equal quadrants
  Q = simp(x(1),x(3),x(5),y(1),y(3),y(5),s,phi,n,exp1,exp2,funcID);
  Q1 = simp(x(1),x(2),x(3),y(1),y(2),y(3),s,phi,n,exp1,exp2,funcID);
  Q2 = simp(x(3),x(4),x(5),y(1),y(2),y(3),s,phi,n,exp1,exp2,funcID);
  Q3 = simp(x(3),x(4),x(5),y(3),y(4),y(5),s,phi,n,exp1,exp2,funcID);
  Q4 = simp(x(1),x(2),x(3),y(3),y(4),y(5),s,phi,n,exp1,exp2,funcID);
  Q_hat = Q1+Q2+Q3+Q4;

  QQ = [abs(Q1),abs(Q2),abs(Q3),abs(Q4)];
  pushOrder = [1,2,3,4];
  [QQ1,pO1] = bubbleSort(QQ,pushOrder);

  epses(count) = (A_hat/A)*tol;
  Qs(count) = Q Q_hat;

  if(abs((A_hat/A)*tol) == 0)
      disp('Oops');
  elseif abs(Q Q_hat) > (A_hat/A)*tol
      level = level + 1;
      NQ = NQ + 1;

      % Pushing 4 corners of quadrants onto stack with level and epsilon
%         pile(NQ,:) = push(x(1),x(3),y(3),y(5),level,tol);
%         pile(NQ+1,:) = push(x(3),x(5),y(3),y(5),level,tol);
%         pile(NQ+2,:) = push(x(3),x(5),y(1),y(3),level,tol);
%         pile(NQ+3,:) = push(x(1),x(3),y(1),y(3),level,tol);
      for i4 = 1:4
          pile(NQ + i4   1,:) = push(x(coords(pO1(i4),1)),x(coords(pO1(i4),2)),...
                                  y(coords(pO1(i4),3)),y(coords(pO1(i4),4)),level,tol);
      end

      NQ = NQ + 3;
  else
      approx = approx + Q_hat;
  end

  % taking top quadrant off top of stack & reassign variables
  if NQ>0
      z_hat = pop(pile,NQ);
```

```


        a = z_hat(1);
        b = z_hat(2);
        c = z_hat(3);
        d = z_hat(4);
        level = z_hat(5);
        tol = z_hat(6);
        NQ = NQ   1;
    else
        bool = 0;

    end

    [epses;Qs];
end


end

function s1 = simp(alpha,v1,beta,gamma,w1,delta,s,phi,n,exp1,exp2,funcID)
    c1 = (v1 alpha)/(beta v1);
    mu = beta v1;

    h = mu*(1+c1)/(6*c1);
    s1 = h*((2*c1 1)*simp2(alpha,gamma,w1,delta,s,phi,n,exp1,exp2,funcID)+...
        (1+c1)^2*simp2(v1,gamma,w1,delta,s,phi,n,exp1,exp2,funcID)...
        +(2 c1)*c1*simp2(beta,gamma,w1,delta,s,phi,n,exp1,exp2,funcID));
end

function s2 = simp2(x_bar,gamma,w2,delta,s,phi,n,exp1,exp2,funcID)
    c2 = (w2 gamma)/(delta w2);
    nu = delta w2;

    k = nu*(1+c2)/(6*c2);
    s2 = k*((2*c2 1)*f1(x_bar,gamma,s,phi,n,exp1,exp2,funcID)+...
        (1+c2)^2*f1(x_bar,w2,s,phi,n,exp1,exp2,funcID)+(2 c2)*c2*f1(x_bar,delta,s,phi,n,exp1,exp2,funcID));
end

function z2 = push(a,b,c,d,level,tol)
    z2(1) = a;
    z2(2) = b;
    z2(3) = c;
    z2(4) = d;
    z2(5) = level;
    z2(6) = tol;
end

function z3 = pop(array1,nstuff)
    z3(1) = array1(nstuff,1);
    z3(2) = array1(nstuff,2);
    z3(3) = array1(nstuff,3);
    z3(4) = array1(nstuff,4);
    z3(5) = array1(nstuff,5);
    z3(6) = array1(nstuff,6);
end
```

Appendix C

DTRIS2 MATLAB CODE

```

function [ tri_num, tri_vert, tri_nabe ] = dtris2 ( point_num, p )

% DTRIS2 constructs a Delaunay triangulation of 2D vertices.
%
%  Discussion:
%
%    The routine constructs the Delaunay triangulation of a set of 2D vertices
%    using an incremental approach and diagonal edge swaps. Vertices are
%    first sorted in lexicographically increasing (X,Y) order, and
%    then are inserted one at a time from outside the convex hull.
%
%  Modified:
%
%    07 February 2005
%
%  Author:
%
%    Original FORTRAN77 version by Barry Joe.
%    MATLAB version by John Burkardt.
%
%  Reference:
%
%    Barry Joe,
%    GEOMPACK   a software package for the generation of meshes
%    using geometric algorithms,
%    Advances in Engineering Software,
%    Volume 13, pages 325 331, 1991.
%
%  Parameters:
%
%    Input, integer POINT_NUM, the number of vertices.
%
%    Input, real P(2,POINT_NUM), the vertices.
```

```
%
%    Output, integer TRI_NUM, the number of triangles in the triangulation;
%    TRI_NUM is equal to 2*POINT_NUM  NB  2, where NB is the number
%    of boundary vertices.
%
%    Output, integer TRI_VERT(3,TRI_NUM), the nodes that make up each triangle.
%    The elements are indices of P.  The vertices of the triangles are
%    in counter clockwise order.
%
%    Output, integer TRI_NABE(3,TRI_NUM), the triangle neighbor list.
%    Positive elements are indices of TIL; negative elements are used for links
%    of a counter clockwise linked list of boundary edges; LINK = (3*I + J 1)
%    where I, J = triangle, edge index; TRI_NABE(J,I) refers to
%    the neighbor along edge from vertex J to J+1 (mod 3).
%
  tri_num = 0;
  tri_vert = [];
  tri_nabe = [];

  tol = 100.0 * eps;
%
%  Sort the vertices by increasing (x,y).
%
  indx = r82vec_sort_heap_index_a ( point_num, p );

  p = r82vec_permute ( point_num, p, indx );

%
%  Make sure that the data points are "reasonably" distinct.
%
  m1 = 1;

  for i = 2 : point_num

    m = m1;
    m1 = i;

    k = 0;

    for j = 1 : 2

```

```
      cmax = max ( abs ( p(j,m) ), abs ( p(j,m1) ) );

      if ( tol * ( cmax + 1.0 ) < abs ( p(j,m)   p(j,m1) ) )
        k = j;
        break
      end

    end

    if ( k == 0 )
      fprintf ( 1, '\n' );
      fprintf ( 1, 'DTRIS2   Fatal error!\n' );
      fprintf ( 1, '  Fails for point number I = %d\n', i );
      fprintf ( 1, '  M = %d\n', m );
      fprintf ( 1, '  M1 = %d\n', m1 );
      fprintf ( 1, '  X,Y(M)  = %f  %f\n', p(1,m), p(2,m) );
      fprintf ( 1, '  X,Y(M1) = %f  %f\n', p(1,m1), p(2,m1) );
      error ( 'DTRIS2   Fatal error!' )
      return
    end

  end
%
%  Starting from points M1 and M2, search for a third point M that
%  makes a "healthy" triangle (M1,M2,M)
%
  m1 = 1;
  m2 = 2;
  j = 3;

  while ( 1 )

    if ( point_num < j )
      fprintf ( 1, '\n' );
      fprintf ( 1, 'DTRIS2   Fatal error!\n' );
      error ( 'DTRIS2   Fatal error!' )
      return
    end

    m = j;

    lr = lrline ( p(1,m), p(2,m), p(1,m1), p(2,m1), p(1,m2), p(2,m2), 0.0 );

    if ( lr ~= 0 )
      break
    end

    j = j + 1;

  end
%
%  Set up the triangle information for (M1,M2,M), and for any other
%  triangles you created because points were collinear with M1, M2.
%
  tri_num = j   2;

  if ( lr ==  1 )

    tri_vert(1,1) = m1;
    tri_vert(2,1) = m2;
    tri_vert(3,1) = m;
    tri_nabe(3,1) =  3;

    for i = 2 : tri_num

      m1 = m2;
      m2 = i+1;
      tri_vert(1,i) = m1;
      tri_vert(2,i) = m2;
      tri_vert(3,i) = m;
      tri_nabe(1,i 1) =  3 * i;
      tri_nabe(2,i 1) = i;
      tri_nabe(3,i) = i   1;

    end

    tri_nabe(1,tri_num) =  3 * tri_num   1;
    tri_nabe(2,tri_num) =  5;
    ledg = 2;
    ltri = tri_num;

  else

    tri_vert(1,1) = m2;
    tri_vert(2,1) = m1;
    tri_vert(3,1) = m;
    tri_nabe(1,1) =  4;

    for i = 2 : tri_num
      m1 = m2;
      m2 = i+1;
      tri_vert(1,i) = m2;
      tri_vert(2,i) = m1;
      tri_vert(3,i) = m;
      tri_nabe(3,i 1) = i;
      tri_nabe(1,i) =  3 * i    3;
      tri_nabe(2,i) = i    1;

    end

    tri_nabe(3,tri_num) =  3 * tri_num;
    tri_nabe(2,1) =  3 * tri_num    2;
    ledg = 2;
    ltri = 1;

  end
%
%  Insert the vertices one at a time from outside the convex hull,
%  determine visible boundary edges, and apply diagonal edge swaps
   until
%  Delaunay triangulation of vertices (so far) is obtained.
%
  top = 0;

  for i = j+1 : point_num

    m = i;
    m1 = tri_vert(ledg,ltri);

    if ( ledg <= 2 )
      m2 = tri_vert(ledg+1,ltri);
    else
      m2 = tri_vert(1,ltri);
    end

    lr = lrline ( p(1,m), p(2,m), p(1,m1), p(2,m1), p(1,m2), p(2,m2
   ), 0.0 );

    if ( 0 < lr )
      rtri = ltri;
      redg = ledg;
      ltri = 0;
    else
      l =  tri_nabe(ledg,ltri);
      rtri = floor ( l / 3 );
      redg = mod(l,3) + 1;
    end

    [ ltri, ledg, rtri, redg ] = vbedg ( p(1,m), p(2,m), point_num,
    p, ...
      tri_num, tri_vert, tri_nabe, ltri, ledg, rtri, redg );

    n = tri_num + 1;
```

```
l =  tri_nabe(ledg,ltri);

while ( 1 )


  t = floor ( l / 3 );
  e = mod ( l, 3 ) + 1;
  l =  tri_nabe(e,t);
 m2 = tri_vert(e,t);

  if ( e <= 2 )
    m1 = tri_vert(e+1,t);
  else
    m1 = tri_vert(1,t);
  end

  tri_num = tri_num + 1;
  tri_nabe(e,t) = tri_num;
  tri_vert(1,tri_num) = m1;
  tri_vert(2,tri_num) = m2;
  tri_vert(3,tri_num) = m;
  tri_nabe(1,tri_num) = t;
  tri_nabe(2,tri_num) = tri_num   1;
  tri_nabe(3,tri_num) = tri_num + 1;
  top = top + 1;

  if ( point_num < top )
    fprintf ( 1, '\n' );
    fprintf ( 1, 'DTRIS2   Fatal error!\n' );
    fprintf ( 1, '  Stack overflow.\n' );
    error ( 'DTRIS2   Fatal error!' )
  end

  work(top) = tri_num;

  if ( t == rtri && e == redg )
    break
  end

end

tri_nabe(ledg,ltri) =  3 * n   1;
tri_nabe(2,n) =  3 * tri_num   2;
tri_nabe(3,tri_num) =  l;
ltri = n;
ledg = 2;

[ top, ltri, ledg, tri_vert, tri_nabe ] = swapec ( ...
      m, top, ltri, ledg, point_num, p, tri_num, tri_vert, tri_nabe
   , work );

  end
%
%  Now account for the sorting that we did.
%
  for i = 1 : 3
    for j = 1 : tri_num
      tri_vert(i,j) = indx ( tri_vert(i,j) );

    end
  end

  indx = perm_inverse ( point_num, indx );
  p = r82vec_permute ( point_num, p, indx );

  return
end
```

Appendix D

BOWYER-WATSON ALGORITHM MATLAB CODE

```
function [newNumTris,finalVert,finalNabes,boundEdg,cycle,trisLinkList,...
    convTriAreas,convTriAreaIndex,index,conversionInd] = insertNew8(P,verts,...
    new_pnt,numTris,nabes,boundEdg,cycle,trisLinkList,maxTriIndex)

% Determine which circumcircles contain new_pnt
index = maxTriIndex;
numYes = 1;
triStack = 0;
trisTested = maxTriIndex;
countTest = 1;
countStack = 0;
for i2 = 1:3
    if(nabes(i2,maxTriIndex)>0)
        countStack = countStack + 1;
        triStack(countStack) = nabes(i2,maxTriIndex); % push onto stack
    end
end

while(countStack ~= 0)
    triIndTest = triStack(countStack); % pop off stack
    triStack(countStack) = [];
    countStack = countStack  1;
    countTest = countTest + 1;
    trisTested(countTest) = triIndTest;
    A = zeros(2,3);
    for i3 = 1:3
        A(:,i3) = P(:,verts(i3,triIndTest));
    end
    if(inCircle(A,new_pnt) == 1)
        numYes = numYes + 1;
        index(1,numYes) = triIndTest;
        for i4 = 1:3
            if(nabes(i4,triIndTest)>0 && isempty(intersect(trisTested,
    nabes(i4,triIndTest))))
                countStack = countStack + 1;
                triStack(countStack) = nabes(i4,triIndTest); % push
    onto stack
            end
        end
    end
end
% index = sort(index,'ascend');

% Determine hull of space containing new_pnt
n = length(index);
pntIdxs = zeros(1,3*n);
for k = 1:n
    for s = 1:3
    pntIdxs(3*(k 1)+s) = verts(s,index(k));
    end
end

hullInd = unique(pntIdxs);
m = length(hullInd);
hullPts = zeros(2,m);
for q = 1:m
    hullPts(:,q) = P(:,hullInd(q));
end

newP = [hullPts,new_pnt];
[~,cols] = size(P);
numPnts = length(newP);
[newNumTris,newVert,newNabes] = dtris2(numPnts,newP);
[newTriAreas,newTriAreaIndex] = findAreas(newVert,newP);

% Check for convexity of hull, if concave remove ghost triangles
numNo = 0;
numGhost = 0;
for g = 1:newNumTris
    if(contains(newVert(:,g),numPnts) == 1)
        numNo = numNo + 1;
        convexVert(:,numNo) = newVert(:,g);
        convexNabe(:,numNo) = newNabes(:,g);
    else
        numGhost = numGhost + 1;
        ghostTris(numGhost) = g;
        ghostAreaInd(numGhost) = g;
    end

end
% Get rid of ghost tri areas and indices
```

```
convTriAreaIndex = zeros(1,newNumTris  numGhost);
convTriAreas = zeros(1,newNumTris  numGhost);
for g1 = 1:numGhost
    delete = ghostTris(g1)==newTriAreaIndex;
    newTriAreaIndex(delete) = 0;
    newTriAreas(delete) = 0;
end
newTriAreaIndex(newTriAreaIndex==0) = [];
newTriAreas(newTriAreas==0) = [];
convTriAreaIndex = newTriAreaIndex;
convTriAreas = newTriAreas;


if(numGhost ~=0)
    % Set reference that was deleted to zero
    if(numNo < newNumTris)
        for h = 1:length(ghostTris)
            convexNabe(convexNabe == ghostTris(h)) =  1;
        end
    end

    % Correct referencing of triangles that fall after one that was
    deleted
    sub = 0;
    if(numGhost == 1)
        y1 = convexNabe > ghostTris(1);
        convexNabe(y1) = convexNabe(y1)   1;
        y2 = convTriAreaIndex > ghostAreaInd(1);
        convTriAreaIndex(y2) = convTriAreaIndex(y2)   1;
    else
        for idx = 1:numGhost 1
            first1 = ghostTris(idx);
            second1 = ghostTris(idx+1);
            sub = sub + 1;
            z1 = convexNabe > first1;
            o1 = convexNabe < second1;
            y1 = o1==z1;
            convexNabe(y1) = convexNabe(y1)   sub;

            % Fix area indexing for those falling after ghost
   triangles
            first2 = ghostAreaInd(idx);
            second2 = ghostAreaInd(idx+1);
            z2 = convTriAreaIndex > first2;
            o2 = convTriAreaIndex < second2;
            y2 = o2==z2;
            convTriAreaIndex(y2) = convTriAreaIndex(y2)   sub;
        end
```

```

            sub = sub + 1;
            y1 = convexNabe > ghostTris(numGhost);
            convexNabe(y1) = convexNabe(y1)   sub;
            y2 = convTriAreaIndex > ghostAreaInd(numGhost);
            convTriAreaIndex(y2) = convTriAreaIndex(y2)   sub;
    end
end

newNumTris = newNumTris   numGhost;

% % Graph circumcircles
% for f = 1:length(index)
%       cor = [P(:,verts(1,index(f))),P(:,verts(2,index(f))),P(:,verts(3,index(f)))];
%       [r,cc]=circumcircle(cor,1);
%       hold on
% %       pause
% end
% % Graph triangles affected with the new triangulation
% for e = 1:newNumTris
%       triangle(:,1)=newP(:,convexVert(1,e));
%         triangle(:,2)=newP(:,convexVert(2,e));
%         triangle(:,3)=newP(:,convexVert(3,e));
%         triangle(:,4)=triangle(:,1);
%         plot(triangle(1,:),triangle(2,:),' ','color','m','Linewidth',1.5);
%         hold on
%         % pause
% end
% % pause(.05)
% % pause
convexConVert = convertVert(hullInd,convexVert,cols+1);

% wholeP = [P,new_pnt];
% [numTest,vertTest,nabeTest]=dtris2(length(wholeP),wholeP);
% Graph DT as if we did it on original points plus the new pnt
% for f = 1:numTest
%       triangle(:,1)=wholeP(:,vertTest(1,f));
%         triangle(:,2)=wholeP(:,vertTest(2,f));
%         triangle(:,3)=wholeP(:,vertTest(3,f));
%         triangle(:,4)=triangle(:,1);
%         plot(triangle(1,:),triangle(2,:),' ','color','g','Linewidth',1.5);
%         hold on
%         %% pause
% end

```

```matlab

% Determine which tris were broken up then replace/add new ones to verts
nnn=length(index);
finalVert = verts;
conversionInd = zeros(1,newNumTris);
for jjj = 1:nnn
    finalVert(:,index(jjj)) = zeros(3,1);
end

[~,cc] = size(verts);
numAdded = 0;
iii = 1;
numSplits = zeros(2,nnn);

checkBound = zeros(1,2);
% index
numOfNumSplits = 0;
while(iii <= nnn)

    splitTris = detNewTris(convexConVert,index(iii),verts);

    if((length(splitTris) == 1 && splitTris~=0) || length(splitTris) == 2)

        numSplits(1,iii) = length(splitTris);
        finalVert(:,index(iii)) = convexConVert(:,splitTris(1));
        conversionInd(splitTris(1)) = index(iii);

        if(numSplits(1,iii)==2)
            numAdded = numAdded + 1;
            newInd = cc+numAdded;
            finalVert(:,newInd) = convexConVert(:,splitTris(2));
            conversionInd(splitTris(2)) = newInd;
            numOfNumSplits = numOfNumSplits + 1;
            numSplits(2,iii) = numOfNumSplits;
        end

        iii = iii + 1;

    % If splitTris comes back as zero, save that index, run like normal
    % until another double split is hit. Fill in the first splitTris
    % like normal, but istead of tacking splitTris(2) onto end, replace
    % previous empty slot
    elseif(splitTris == 0)
```

```

        numSplits(1,iii) = 0;
        check = index(iii);
        checkInd = iii;

        % The last index(iii) gave splittris of 0
        if(iii == nnn)
            prevInd = find(numSplits(1,:)==2,1,'last');
            previous = index(prevInd);
            maxInd = max(conversionInd);
            finalVert(:,check) = finalVert(:,maxInd);
            conversionInd(conversionInd==maxInd) = check;
            finalVert(:,maxInd) = [];
            numSplits(1,checkInd) = numSplits(1,checkInd) + 1;
            numSplits(1,prevInd) = numSplits(1,prevInd)   1;
            numSplits(2,prevInd) = 0;
            numOfNumSplits = numOfNumSplits   1;
            checkBound = [check,previous];
            break;
        else
            while(1)
                % Start at next index(iii) and go until you find another
                iii = iii + 1;
                splitTris = detNewTris(convexConVert,index(iii),verts);
                numSplits(1,iii) = length(splitTris);
                finalVert(:,index(iii)) = convexConVert(:,splitTris(1));
                conversionInd(splitTris(1)) = index(iii);
                if(length(splitTris)==2)
                    finalVert(:,check) = convexConVert(:,splitTris(2));
                    conversionInd(splitTris(2)) = check;
                    numSplits(1,checkInd) = numSplits(1,checkInd) + 1;
                    numSplits(1,iii) = numSplits(1,iii)   1;
                    checkBound = [check,index(iii)];
                    iii = iii + 1;
                    break;
                end
                % Do not encounter another splittris of 2
                if(iii == nnn)
                    prevInd = find(numSplits(1,:)==2,1,'last');
                    previous = index(prevInd);
                    maxInd = max(conversionInd);
                    finalVert(:,check) = finalVert(:,maxInd);
                    conversionInd(conversionInd==maxInd) = check;
                    finalVert(:,maxInd) = [];
                    numSplits(1,checkInd) = numSplits(1,checkInd) + 1;
                    numSplits(1,prevInd) = numSplits(1,prevInd)  1;
                    numOfNumSplits = numOfNumSplits  1;
                    numSplits(2,prevInd) = 0;
                    checkBound = [check, previous];
                    break;
                end
            end
        end
    elseif(length(splitTris)==3)
        numSplits(1,iii) = 3;
        finalVert(:,index(iii)) = convexConVert(:,splitTris(1));
        conversionInd(splitTris(1)) = index(iii);
        finalVert(:,cc+1) = convexConVert(:,splitTris(2));
        conversionInd(splitTris(2)) = cc+1;
        finalVert(:,cc+2) = convexConVert(:,splitTris(3));
        conversionInd(splitTris(3)) = cc+2;
        numOfNumSplits = numOfNumSplits + 1;
        numSplits(2,iii) = numOfNumSplits;
        break;
    end
end

% Fix convTriAreaIndex to represent correct triangle references
for i6 = 1:length(convTriAreaIndex)
    convTriAreaIndex(i6) = conversionInd(convTriAreaIndex(i6));
end
[trisWithBE,trisWithNoBE,finalNabes] = convertConvexNabe(convexNabe,...
    conversionInd,nabes,index,verts,convexConVert,numSplits,checkBound,trisLinkList,boundEdg);

% Update LL with new tri references and edg numbers
if(trisWithBE ~=0)
    for i = 1:length(trisWithBE)
        newE = find(finalNabes(:,trisWithBE(i))<0);
        x = finalVert(newE,trisWithBE(i));
        oldE = find(boundEdg(1,:) == x);
        trisLinkList(1,oldE) = trisWithBE(i);
        trisLinkList(2,oldE) = newE;
    end
end

% Find corrections to tri references to triangles that were boundaries to
% affected area
leng = length(index);
nabesAff = zeros(3,leng);
for i5 = 1:leng
    nabesAff(:,i5) = nabes(:,index(i5));
end

[mmm,nnn] = size(nabesAff);
hits = 0;
for j5 = 1:mmm
    for k5 = 1:nnn
        if(nabesAff(j5,k5)>0)
            hits = hits+1;
            boundTris(hits) = nabesAff(j5,k5);
        end
    end
end

boundTris = setdiff(boundTris,index);
num = length(boundTris);

for t1 = 1:num
    checkTri = boundTris(t1);
    checkBound = intersect(finalNabes(:,checkTri),conversionInd);
    for t2 = 1:length(checkBound)
        checkInd1 = find(checkBound(t2)==finalNabes(:,checkTri));
        checkInd2 = 1+mod(checkInd1,3);
        checkEdg = [finalVert(checkInd1,checkTri);finalVert(
   checkInd2,checkTri)];
        for t3 = 1:length(conversionInd)
            cI = conversionInd(t3);
            checkCIVert = finalVert(:,conversionInd(t3));
            finalCheck = intersect(checkCIVert,checkEdg);
            if(length(finalCheck) == 2)
                finalNabes(checkInd1,checkTri) = conversionInd(t3);
                break;
            end
        end
    end
end

% Update all negative values to correct numbers using (3*I + J 1)
nums = length(cycle);
orderedBE = zeros(size(boundEdg));
orderedTrisLL = zeros(size(trisLinkList));
for Q = 1:nums 1
    A = cycle(Q);
    orderedBE(:,Q) = boundEdg(:,ismember(boundEdg(1,:),A));
    orderedTrisLL(:,Q) = trisLinkList(:,ismember(boundEdg(1,:),A));
end

orderedTrisLL = [orderedTrisLL,orderedTrisLL(:,1)];
M = length(orderedTrisLL);

for R = 1:M 1
    inds = orderedTrisLL(:,R);
    checkInds = orderedTrisLL(1,R+1);
    J = orderedTrisLL(2,R+1);
    finalNabes(inds(2),inds(1)) =  (3*checkInds+J 1);
end

% Find neighbor on edge that was ghost boundary and update nabe accordingly
W = length(trisWithNoBE);
tempVert = [finalVert;finalVert(1,:)];

for V = 1:W
    colTriInd = trisWithNoBE(V);
    ind1 = find(finalNabes(:,colTriInd)==0);
    bbb = tempVert(ind1,colTriInd);
    aaa = tempVert(ind1+1,colTriInd);
    for X = 1:num
        if(length(intersect([aaa,bbb],finalVert(:,boundTris(X))))==2)
            finalNabes(ind1,trisWithNoBE(V)) = boundTris(X);
            break;
        end
    end
end

newNumTris = numTris+2;

end

function yes_no = contains(vect,pntCheck)

a = vect(1);
b = vect(2);
C = vect(3);

if(a == pntCheck)
    yes_no = 1;
elseif(b == pntCheck)
    yes_no = 1;
elseif(C == pntCheck)
    yes_no = 1;
```

```
else
    yes_no = 0;
end

end

function convVert = convertVert(hullInd,newVert,numPnts)

vect = [hullInd,numPnts];
[m,n] = size(newVert);
convVert = zeros(m,n);
for ii = 1:m
    for jj = 1:n
        convVert(ii,jj) = vect(newVert(ii,jj));
    end
end

end

function splittris = detNewTris(convexConVert,index,verts)
[~,n] = size(convexConVert);
splits = 0;
splittris = 0;
for j = 1:n
    a = verts(:,index);
    b = convexConVert(:,j);
    ints = intersect(a,b);
    if(length(ints) == 2)
        splits = splits + 1;
        splittris(splits) = j;
    end
end

end

function [triWithBound,triWithNoBound,finalNabes] =
    convertConvexNabe(newNabes,...
    conversionInd,nabes,index,verts,convConVert,numSplits,
    checkBound,trisLL,boundEdg)

[rows,cols] = size(newNabes);
finalNabes = zeros(rows,cols);
% Convert from subset triangle references to whole set triangle
    references
for i = 1:cols
    for j = 1:rows
        if(newNabes(j,i)<0)
            finalNabes(j,i) = newNabes(j,i);
            else
                finalNabes(j,i) = conversionInd(newNabes(j,i));
            end
        end
end

% Add nabe of subset into nabe of whole set
n = length(conversionInd);
newFinalNabes = nabes;
for k = 1:n
    newFinalNabes(:,conversionInd(k)) = finalNabes(:,k);
end
finalNabes = newFinalNabes;
% Determine which triangles should have boundary edges
m = length(index);
[~,ccc] = size(nabes);
triWithBound = zeros(1,1);
numYes = 0;
numHits = 0;

for L = 1:m
    if(length(find(nabes(:,index(L))<0)) == 1)
        numYes = numYes + 1;
        triWithBound(numYes) = index(L);
    elseif(length(find(nabes(:,index(L))<0)) == 2)
        numYes = numYes + 1;
        triWithBound(numYes) = index(L);
        numYes = numYes + 1;
        numHits = numHits + 1;
        triWithBound(numYes) = ccc + numHits;
    end
end
% Determine if a triangle was split, that it still has the boundary
    edge or
% if it was the added triangle, and also check that for a split
   triangle had
% its split replaced one that got absorbed
verts1 = [verts;verts(1,:)];
inters1 = intersect(triWithBound,trisLL(1,:));

if(~all(checkBound==0))
    for k1 = 1:length(checkBound)
        ind3 = conversionInd==checkBound(k1);
        lengInt = length(intersect(convConVert(:,ind3),boundEdg));
        if(lengInt ==2)
            triWithBound(k1) = checkBound(k1);
        end
    end
```

```matlab
end

if(~isempty(inters1))
    for k2 = 1:length(inters1)
        a2 = inters1(k2)==index;
        if(numSplits(1,a2) == 2)
            a3 = convConVert([1,2],conversionInd==inters1(k2));
            b3 = convConVert([2,3],conversionInd==inters1(k2));
            c3 = [convConVert(3,conversionInd==inters1(k2)), convConVert(1,conversionInd==inters1(k2))];
            interA = intersect(a3',boundEdg','rows');
            interB = intersect(b3',boundEdg','rows');
            interC = intersect(c3,boundEdg','rows');
            if(length(interA) ~= 2 && length(interB) ~=2 && length(interC) ~= 2)
                triWithBound(inters1(k2)==triWithBound) = ccc + numSplits(2,a2);
            end
        elseif(numSplits(1,a2) == 3)
            for k3 = 1:length(conversionInd)
                a3 = convConVert([1,2],k3);
                b3 = convConVert([2,3],k3);
                c3 = [convConVert(3,k3),convConVert(1,k3)];
                interA = intersect(a3',boundEdg','rows');
                interB = intersect(b3',boundEdg','rows');
                interC = intersect(c3,boundEdg','rows');
                if(length(interA) == 2 || length(interB) ==2 || length(interC) == 2)
                    triWithBound(inters1(k2)==triWithBound) = conversionInd(k3);
                end
            end
        end
    end
end
triWithBound;
% Determine which triangles should not have boundary edges and set neg
% entry to zero
triWithNoBound = setdiff(conversionInd ,triWithBound);
for K = 1:length(triWithNoBound)
    for Z = 1:3
        if(finalNabes(Z,triWithNoBound(K)) < 0)
            finalNabes(Z,triWithNoBound(K)) = 0;
        end
    end
end

```

```
end
```

Appendix E

RESULTS TABLES

Table E.1. Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$ and $\epsilon = 0.0001$.

i	**j**	**AS Time**	**AS Rel. Error**	**VNC Time**	**VNC Rel. Error**
0	0	0.008355225	1.45465E-16	0.007077289	0
0	1	0.000445179	1.83618E-16	0.000605689	1.83618E-16
0	2	0.000444923	1.15589E-16	0.000636664	1.15589E-16
0	3	0.000717303	1.45154E-16	0.028169894	4.20143E-08
0	4	0.005256895	3.11878E-08	0.116979043	3.11815E-08
0	5	0.018198816	9.64697E-09	0.484475327	5.47195E-09
1	0	0.00050585	0	0.000657144	0
1	1	0.000572153	1.35526E-16	0.000752119	2.71052E-16
1	2	0.000714487	3.41259E-16	0.000930037	3.41259E-16
1	3	0.000505338	2.14273E-16	0.025437127	1.05478E-07
1	4	0.005495996	9.21119E-09	0.130504124	1.02836E-09
2	0	0.000655608	1.18479E-16	0.000882165	1.18479E-16
2	1	0.000450042	4.48665E-16	0.000632312	1.49555E-16
2	2	0.000574457	0	0.000780278	1.88292E-16
2	3	0.000786678	1.18226E-16	0.03968156	7.85136E-08
3	0	0.000457722	1.16218E-16	0.498908685	8.96198E-08
3	1	0.000620536	0	1.49217904	5.65809E-08
3	2	0.00045721	0	2.354864375	0.006500369
4	0	0.121101104	4.03868E-08	1.890572353	3.33763E-07
4	1	0.116197996	4.03868E-08	2.345452395	0.028265236
5	0	0.211611352	7.10947E-08	2.349108286	0.184686147

Table E.2. Voronoi Newton-Cotes (VNC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$ and $\epsilon = 0.0001$.

i	j	AS Time	AS Rel. Error	VNC Time	VNC Rel. Error
0	0	0.001667055	0	0.005648840	1.45465E-16
0	1	0.000532987	0	0.001388274	3.67237E-16
0	2	0.000424188	2.31179E-16	0.001337843	0
0	3	0.000500475	1.45154E-16	0.022388000	4.90503E-08
0	4	0.001801966	3.11878E-08	0.082919108	3.34052E-08
0	5	0.007804850	9.64697E-09	0.432353059	2.63289E-09
1	0	0.000477947	0	0.001335283	4.29461E-16
1	1	0.000468731	1.35526E-16	0.001387250	0
1	2	0.000577018	0	0.001409522	1.70629E-16
1	3	0.000557306	2.14273E-16	0.031654084	1.24042E-08
1	4	0.004736465	9.21119E-09	0.124463397	6.37093E-09
2	0	0.000573178	2.36958E-16	0.001332467	4.73917E-16
2	1	0.000468475	2.99110E-16	0.001327091	1.49555E-16
2	2	0.000460539	1.88292E-16	0.001327347	5.64876E-16
2	3	0.000472827	3.54679E-16	0.036043672	2.80358E-08
3	0	0.000555770	1.16218E-16	0.432514594	4.00847E-08
3	1	0.000464379	1.46700E-16	0.974934778	8.84301E-08
3	2	0.000470011	1.84698E-16	1.876273619	6.47114E-08
4	0	0.118157412	4.03868E-08	1.116981616	1.10403E-05
4	1	0.117699177	4.03868E-08	2.882673811	1.24821E-07
5	0	0.203343378	7.10947E-08	3.297244264	2.83738E-08

Table E.3. Simpson's Cubature Rule (SC) v. Adaptive Simpson's Rule (AS) on Monomials with $a = -1.62110966800282$, $b = -1.37432067059289$, $c = -3.3239379751915$, $d = -1.72003265166653$ and $\epsilon = 0.0001$.

i	**j**	**SC Time**	**SC Rel. Error**	**AS Time**	**AS Rel. Error**
0	0	0.016313693	2.80482E-16	0.000848376	2.80482E-16
0	1	0.005531337	7.78505E-16	0.000447996	6.67290E-16
0	2	0.006637502	5.11924E-16	0.000467707	5.11924E-16
0	3	0.025780223	2.19831E-05	0.000461051	2.54077E-16
0	4	0.005628104	0.000980107	0.008406700	2.76362E-07
0	5	0.025711359	0.000504705	0.030613710	7.71966E-08
1	0	0.005419978	1.87274E-16	0.000468731	1.87274E-16
1	1	0.005155789	1.48513E-16	0.000481531	1.48513E-16
1	2	0.009143973	3.05773E-06	0.000441084	6.83606E-16
1	3	0.027169265	0.000164893	0.000472827	1.69643E-16
1	4	0.024940551	0.000247897	0.031728579	1.72726E-08
2	0	0.005003726	4.99029E-16	0.000488443	7.48543E-16
2	1	0.005474249	2.78244E-06	0.000503547	3.95743E-16
2	2	0.004514515	3.36712E-05	0.000525819	1.06261E-15
2	3	0.009087397	0.000268450	0.000467451	7.91086E-16
3	0	0.004566226	1.31798E-07	0.000488187	0
3	1	0.006894267	1.19500E-05	0.000463611	0
3	2	0.012163975	1.11948E-05	0.000596730	2.01799E-16
4	0	0.005361354	4.93102E-07	0.000466939	3.78813E-07
4	1	0.006172866	3.78024E-05	0.000483579	3.78813E-07
5	0	0.005350859	7.47998E-07	0.000449276	1.87723E-06

Table E.4. Accuracy only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on Monomials with $a = 2.51778949114543$, $b = 5.67194769326589$, $c = -2.98410546965195$, $d = 5.22175955533465$ and $\epsilon = 0.0001$.

i	**j**	**MDT Rel. Error**	**TDT Rel. Error**	**SC Rel. Error**	**AS Rel. Error**	**VNC Rel. Error**	**MC Rel. Error**
0	0	1.37263E-16	1.37263E-16	0	1.37263E-16	2.74525E-16	0
0	1	1.22684E-16	1.22684E-16	1.22684E-16	0	1.22684E-16	0.005362692
0	2	0.031254337	0.312565299	3.20000E-16	3.20000E-16	0	0.001030788
0	3	0.079913262	0.071087959	0.000673081	0	0.288846650	0.000739701
0	4	0.085085066	0.349330409	0.000541126	0.000170947	0.832750855	0.005001589
0	5	0.164348054	0.250576017	0.005319045	0.482681292	0.929758393	0.003924581
1	0	1.34083E-16	0	1.34083E-16	0	0	0.001456460
1	1	0.000343677	0.001646319	0	1.19842E-16	1.19842E-16	0.002003265
1	2	0.030968972	0.071611407	0.000628227	1.56293E-16	1.56293E-16	0.001598617
1	3	0.046836029	0.097188218	0.003659738	0	0.552294340	0.007234298
1	4	0.087317344	0.247612927	0.004180156	0.439828056	0.910577689	0.011503918
2	0	0.004460387	0.012879597	1.24805E-16	2.49611E-16	3.74416E-16	0.001587096
2	1	0.002454287	0.019768922	4.02520E-05	1.11550E-16	3.34651E-16	0.002064355
2	2	0.041796026	0.391609471	0.003020576	2.90959E-16	2.90959E-16	0.006233964
2	3	0.052590909	0.075861068	0.000576465	0	0.764040384	0.014456091
3	0	0.013892921	0.024434128	0.000445991	1.11416E-16	5.39003E-09	0.001646359
3	1	0.005542517	0.016485043	0.001896887	1.99165E-16	2.22670E-08	0.002018411
3	2	0.055330400	0.135658417	0.002939728	0	0.299282761	0.005762480
4	0	0.020762319	0.083940060	0.000124715	1.34399E-10	0.594516070	0.004245154
4	1	0.016455419	0.021512025	0.001381476	1.15998E-10	0.367237377	0.005712344
5	0	0.022003814	0.087866804	0.001089610	0.009776169	0.888547317	0.004548625

Table E.5. Time only for Midpoint Delaunay triangulation (MDT), Trapezoid Delaunay triangulation (TDT), Simpson's cubature (SC), Adaptive Simpson's (AS), Vorono Newton-Cotes (VNC) and Monte Carlo (MC) on Monomials with $a = 2.51778949114543$, $b = 5.67194769326589$, $c = -2.98410546965195$, $d = 5.22175955533465$ and $\epsilon = 0.0001$.

i	**j**	**MDT Time**	**TDT Time**	**SC Time**	**AS Time**	**VNC Time**	**MC Time**
0	0	0.264221362	0.048522268	0.021801254	0.006768572	0.017867854	0.352805694
0	1	0.010175386	0.009185188	0.005659591	0.000578810	0.004228822	0.308191995
0	2	0.212448951	0.038722941	0.004221398	0.000389372	0.001326323	0.304616223
0	3	0.069382987	0.323772511	0.068109400	0.000453883	6.1294469	0.305803539
0	4	0.108419526	0.111599014	0.054363361	1.694423275	6.034388905	0.305586709
0	5	0.076242439	0.180678132	0.028078824	1.599314592	5.991035482	0.305959953
1	0	0.005873861	0.00499195	0.005412554	0.000433404	0.001324019	0.305286168
1	1	0.112953752	0.094920268	0.004018392	0.000424956	0.001363698	0.300387145
1	2	0.168781675	0.225280823	0.028803296	0.000450556	0.001321203	0.303793959
1	3	0.094807373	0.209948624	0.052410613	0.000420092	6.003799002	0.30607208
1	4	0.106156252	0.135796660	0.079992545	1.650970269	5.999080969	0.305938450
2	0	0.089998973	0.106370010	0.004385236	0.000401660	0.001311475	0.355044391
2	1	0.189325214	0.094818125	0.057345731	0.000575738	0.001439474	0.303675688
2	2	0.172706884	0.024697353	0.053885158	0.000398588	0.001289971	0.303935782
2	3	0.129739761	0.298044512	0.071032123	0.000481275	5.997509913	0.307859966
3	0	0.087928978	0.188436391	0.019048514	0.000416508	3.934356623	0.303384875
3	1	0.160555453	0.220046699	0.032886456	0.000463099	5.764459568	0.305458710
3	2	0.162005423	0.195561824	0.100761618	0.000406780	5.976562154	0.323290212
4	0	0.178353931	0.080940248	0.004267989	0.483936032	6.037025678	0.313836483
4	1	0.115505535	0.355869727	0.046673454	0.454968130	5.989358442	0.309507822
5	0	0.159462088	0.113784976	0.038072452	1.719201267	6.147717949	0.30707687

Appendix F

SURFACE FIGURES

Table F.1. Graphs of Monomials with $a = -0.00884120840760527$, $b = 2.71855632151155$, $c = 2.88900981641759$, $d = 3.44868288240732$.

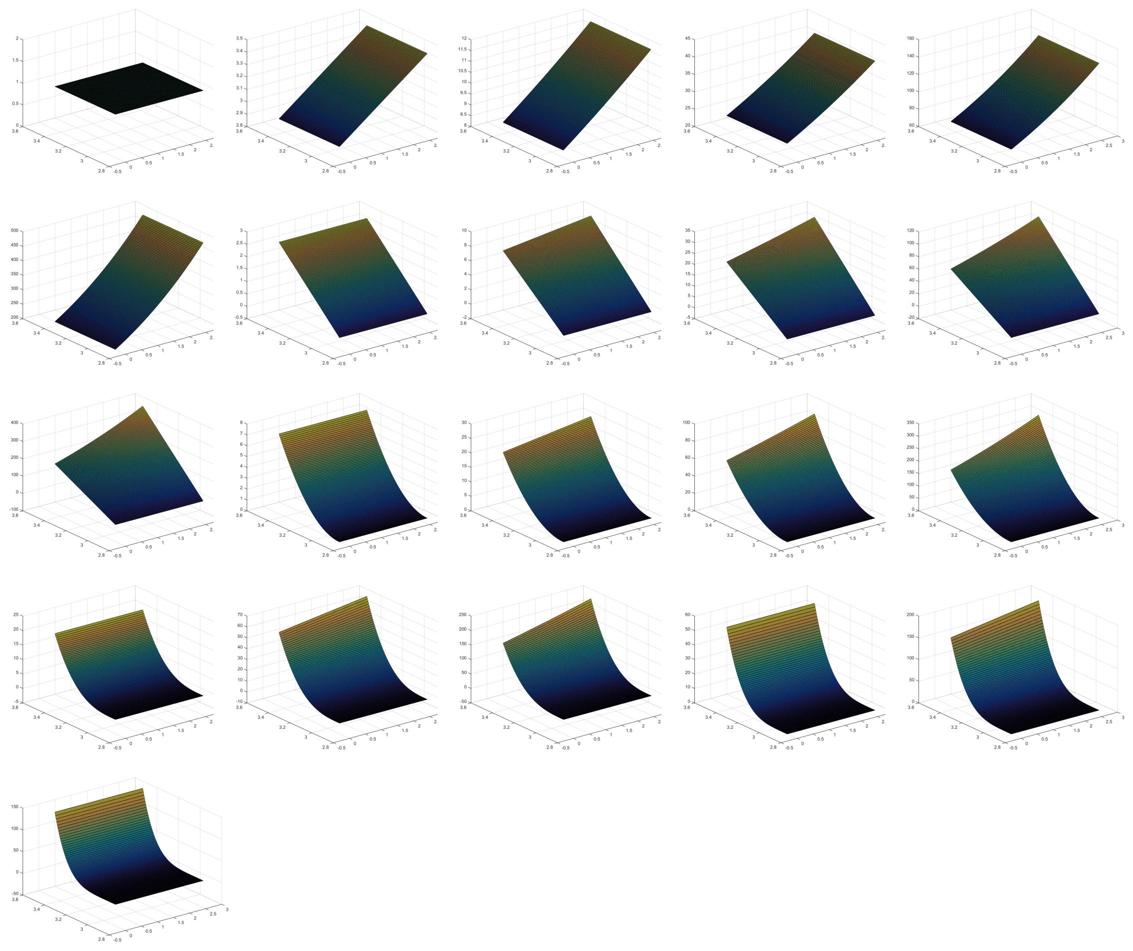
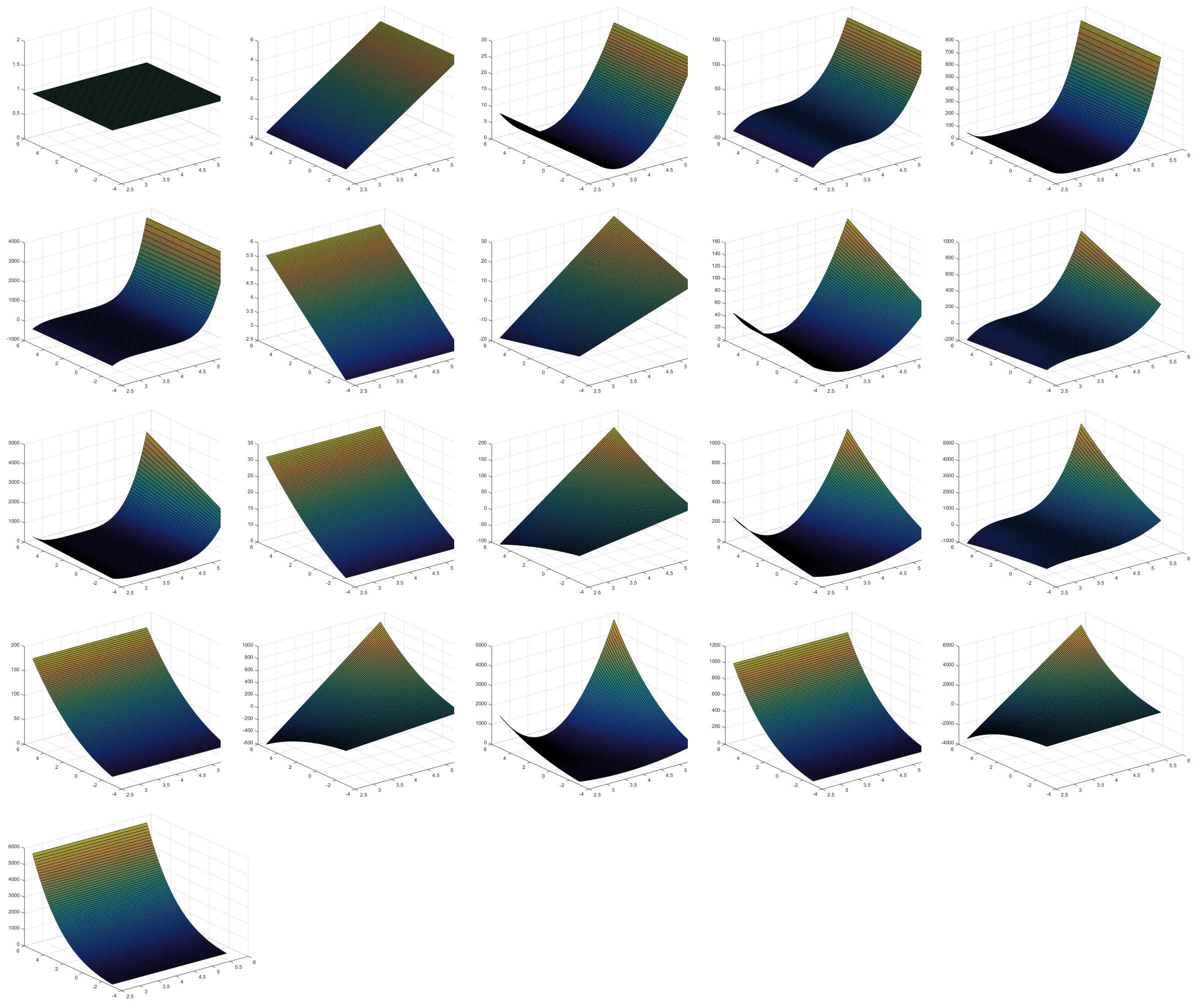


Table F.2. Graphs ofMonomials with $a = 2.51778949114543$, $b = 5.67194769326589$, $c = -2.98410546965195$, $d = 5.22175955533465$.



Appendix G

MONTE CARLO MATLAB CODE

```
function [volume,its] = monteCarlo(a,b,c,d,numPnts,s,phi,n,exp1,exp2,funcID)

sum = 0;
Area = (b  a)*(d  c);
for i = 1:numPnts
    samplePnt = point(a,b,c,d);
    sum = sum + f1(samplePnt(1),samplePnt(2),s,phi,n,exp1,exp2,funcID);
end

volume = Area*sum/numPnts;
its = numPnts;
end

function sampPnt = point(lox,hix,loy,hiy)

pointx = lox + (hix  lox)*rand;
pointy = loy + (hiy  loy)*rand;

sampPnt = [pointx,pointy];
end
```