5-9-2005

Computing Isodose Curves for Radiotherapy Treatment Plans

Ryan Acosta
Trinity University

Computing Isodose Curves for Radiotherapy Treatment Plans

Ryan Acosta

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for Graduation.

April 11, 2005

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Computing Isodose Curves for Radiotherapy Treatment Plans

Ryan Acosta

Abstract

Radiation therapy increasingly means Intensity Modulated Radiation Therapy (IMRT) and with it a trend towards inverse treatment planning. The metrics of a treatment consist of important conformality indices such as the Ian Paddick Conformality Index (IPCI), Dose Volume Histograms (DVH), as well as the conformance isodose lines. This final metric, which offers spatial information of radiation dose that the others do not, shows the results of simulating the treatment plan on the CAT scan images. A physician is able to examine this image and ascertain which portions of the anatomy are the recipients of different levels of radiation dose.

Computing isodose curves is no simple task, especially when the set of CAT scan images becomes large. Our approach is to first interpolate a surface over a finite mesh of scalar data representing the dose distribution in that plane, then find the set of points which constitute that isodose level by using a variation on a an approach called the Hit-and-Run algorithm. We examine the behavior of our algorithm in dealing with sample radiotherapy plans as well as constructed examples to determine the effectiveness of this approach.

Computing Isodose Curves for Radiotherapy Treatment Plans

Ryan Acosta

April 11, 2005

Acknowledgments

I want to thank Dr. Allen Holder for introducing me to "the cancer problem" which allowed me to uncover this interesting aspect of that problem. His help with generating ideas to try as well as his knowledge of the LaTeX typesetting system have proved invaluable. I would also like to express my gratitude to Dr. Maurice Eggen for allowing me to do this project and helping to ensure I got the things I needed to turn in done on time.

Contents

List of Figures

Chapter 1

Introduction

Cancer is a disease, or group of diseases, characterized by uncontrolled cell division. It is possible for this uncontrolled proliferation of cells to extend beyond their original tissue and invade other parts of the body and organs and in doing so become life threatening. While the idea that cancer is a deadly disease is a generalization of all types of cancer, it has been the leading cause of death in the United States since 2002 [11], and in the United Kingdom since the late 1990s [3]. Because of its obvious effect on human life, there has been much research into both the disease processes as well as treatments for the disease.

1.1 Radiation Therapy

Radiation therapy is one of the three main approaches to providing treatment for cancer patients. The three primary methodologies are: radiation therapy, chemotherapy, and surgery. The choice of therapy depends on the type and location of the tumor, as well as the stage of the disease. Our focus is on radiation therapy and how to improve the effectiveness of treatments. Because there are numerous technologies, there are many different possibilities of treatment. This

fact leads us to the different techniques that medical physicists use to design a treatment. Because there are varying approaches and different patient desires, the sense of optimal is not fixed. Moreover, there are certain metrics commonly used to evaluate a treatment and assess whether or not it should be used on an actual patient. One such metric, with which we are mainly concerned, is the conformance of "isodose" lines to the tumor within the patient's anatomy. Isodose lines show the contour of the radiation levels in the anatomy and provide spatial information about the deposition of radiation. In turn, these curves help physicians to evaluate potential treatments.

There are two methods of treating cancer that use radiation. The first is called brachytherapy and involves surgically implanting radioactive pellets into the tumor. This type of treatment is often used to treat prostate cancer because the arrangement of critical structures in this region makes it difficult to use the second type of therapy, external beam therapy, and avoid causing damage to those organs. The second method of radiation therapy uses an external radiation source and forms a beam that is aimed at the patient according to a specified plan, which combines different beams to deliver a uniform dose to the tumor. Radiation therapy's approach takes advantage of what is called a therapeutic advantage. When a patient is treated with radiation, the beam causes DNA damage to cells. Healthy cells repair themselves over time, but cancerous cells are not capable of repairing themselves. This means that it is possible to deliver enough radiation so that the healthy cells recover (in about 24 hours) yet the cancerous cells accumulate the damage. This is referred to as the fractionation of a treatment and patients often receive a small dose of radiation each day for several weeks in order to exploit the therapeutic advantage afforded by the cellular repair. Sometimes great effort is made to ensure that the patient receives the radiation from the same position each day. In some cases involving tumors

in the brain, this even requires inserting screws into the patient's skull to ensure the treatment is delivered as planned.

Figure 1.1: IMRT Radiation Therapy [1]

A common form of external beam radiation therapy is known as Intensity Modulated Radiation Therapy (IMRT). The radiation beam is formed by accelerating photons that creates a high-energy, ionizing particle beam, which is focused on a patient by a gantry. The gantry head includes a device called a collimator that has a number of tungsten leaves, which can move back and forth in order to effectively shape the beam. Controlling the shape in turn controls how we deposit radiation from that angle. The gantry head moves in a great circle on a single axis of rotation about a couch that the patient lies on, which has a rotation of 180 degrees. The combination of these two rotations allows medical physicists to treat from almost any angle on a sphere around the patient. The system is designed so that from each combination of couch and gantry angles,

the radiation beam is focused on a point called the isocenter. Generally the isocenter is placed in the tumor so that all angles are focused on the tumor. Because of the rotation of the system, the tumor appears differently depending on where the gantry is located. This is called the beams-eye-view and the collimator adjusts to the appropriate setting for each gantry angle. Additionally, because the beams from each angle work together on the overall treatment, it may be necessary to "turn off" parts of the beam from a certain angle because that section of the tumor receives enough dose from the other gantry angles. This allows the physician to ensure that healthy tissue absorbs no more radiation than is absolutely necessary. In combination, the rotation of the couch and gantry, coupled with our ability to shape the beam, gives us a great measure of flexibility when designing treatments.

1.2 Treatment Planning and Optimization

There are two paradigms to designing a radiotherapy treatment. The first is known as forward planning. In forward planning a physicist or dosimetrist (a dosimetrist is a certified professional who designs radiotherapy treatment plans) selects angles and exposure times for each angle and lets the computer calculate the amount of radiation that would be deposited in the anatomy. If this results in an acceptable treatment, they are finished, but often they adjust and readjust the treatment and simulate again. In this way, forward planning is a trial-and-error method and depends heavily on the planner's knowledge and experience. The second approach is called inverse planning. In this method, the physicist chooses the gantry angles and sets limits on absorbed dose. The computer then calculates a set of exposure times for those angles that satisfies the "prescription". Inverse planning relies on optimization models to efficiently find an optimal treatment.

There are a variety of optimization models in existence, but they all operate with the same goal: attempt to satisfy the "prescription" in the best way possible. Some models use volumetric constraints, meaning that a certain percentage of the tumor volume must receive a given lower bound of radiation and that a volumetric percentage of the critical structures must receive less than some upper bound. Other models attempt to maximize the difference of the dose to the tumor and the dose to the critical structures. Additionally, optimization models can be linear or nonlinear (most often quadratic) and different models describe an optimal treatment differently.

The optimization model that we use, because of our familiarity with it, is described in Holder and Salter [9]. The specifics of this model are not important; it is more valuable to understand the general idea of how the model works. Based on the "prescription" that the dosimetrist or physicist gives us there are ideal treatments that satisfy all the physician's requests perfectly. However, in the real world the geometry of the system usually produces obstacles so that an ideal treatment is not possible. In this case, we sacrifice some aspect of an ideal treatment to approach reality. This model is called elastic because we allow the bounds to stretch yet tend to the requested limits. In effect, we minimize the deviation from an ideal treatment by measuring how far the bounds of the prescription must be moved to attain a possible treatment.

1.3 Treatment Evaluation

In the summer of 2004, on a grant from the National Cancer Institute, the author worked on a team to develop a software system to compute optimal radiotherapy treatments. This software was challenged with choosing gantry angles. In order to evaluate treatments, we used the three common metrics in the field. One manner was Ian Paddick Conformality Indices (IPCI), which measure

the ratio of volume within a certain isolevel compared with the total volume of the corresponding structure. An isolevel is a region that receives at least a certain level T of radiation (notice that if $T_1 \leq T_2$ then $\text{isolevel}_{T_1} \subseteq \text{isolevel}_{T_2}$). The Over Treatment Ratio (OTR) is a measure of how much volume is within a certain isolevel that is not part of the target volume (or tumor). The Under Treatment Ratio (UTR) measures how much of the target volume is within an isolevel compared to the total volume of that isolevel. IPCI is a combination of the two such that

$$\begin{aligned} OTR_T &= \frac{|TV \cap IV_T|}{|IV_T|}, \\ UTR_T &= \frac{|TV \cap IV_T|}{|TV|}, \text{ and} \\ IPCI_T &= OTR \times UTR = \frac{(|TV \cap IV_T|)^2}{|TV| \times |IV_T|}, \text{ where} \\ TV &= \text{volume represented by the tumor, and} \\ IV_T &= \text{volume inside the isodose level } T. \end{aligned}$$

Loosely speaking, these conformality indices represent how well we "colored within the lines." A perfect treatment has an OTR and UTR value of 1, and consequently an IPCI value of 1. While the terminology may seem misleading, OTR is really a measure of how little volume is over-treated and UTR is a measure of exactly how much of the tumor receives the radiation level of isolevel T. Using this metric, we have a way to quantitatively choose between treatments. Some optimization models use these conformality indices as their objective function, and the model attempts to maximize the IPCI. One such model is described by Cheek, Holder, Salter, and Fuss [5].

The second metric commonly used is a graph called a Dose Volume Histogram (DVH), which plots the amount of radiation deposited against the volume of tissue that receives that dose. Figure 1.2 shows an example of a DVH for

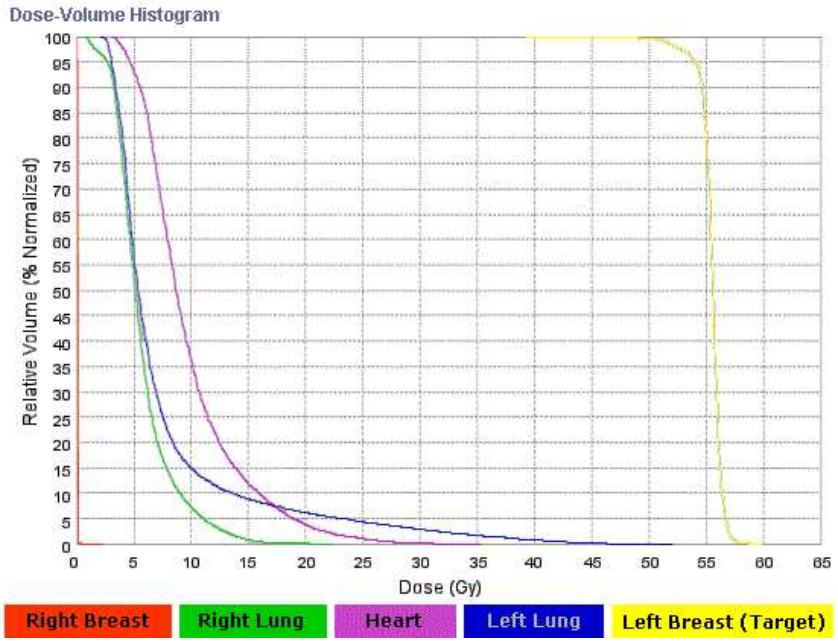


Figure 1.2: Dose Volume Histogram [2]

a treatment plan where the left breast holds the tumor. We easily see that the yellow line that represents the target volume has nearly 100% of its volume receiving 50 Gy of radiation. We also see that the critical structures in the region on average receive between 5 and 10 Gy, with less than 5% of any structures volume receiving 25 Gy. This example shows the kind of separation that an optimal treatment attempts to achieve. We desire the target's curve to achieve the upper right corner of the chart while we simultaneously attempt to push all other curves to the bottom axes. This allows us to easily determine if any part of a treatment is problematic by visual inspection; however, this metric does not indicate where the problem is. For instance, if the curve representing the normal tissue has a small section of the line that passes the upper bound in the prescription, then there may be a "hot spot" in the anatomy that is receiving too much radiation (more radiation than the tumor); however, we cannot pin-

point the location of the hot spot. Additionally, we cannot know immediately if that volume that passes the upper bound is a contiguous region, and we only classify the volume as a hot spot if it is contiguous. So we may not even have a problem.

Earlier we mentioned that some optimization models use volumetric dose constraints to describe the prescription. Dose Volume Histograms show how this kind of constraint can be useful. Essentially, a dosimetrist can look at a DVH and find the places in the graph to change and thereby effectively adjust the dose-volume constraints.

An isodose curve is the third metric commonly used to assess a treatment plan. An isodose curve is simply a curve drawn on a CAT Scan that shows a level of radiation. In essence, a series of isodose curves shows the contour of the deposited radiation within the anatomy. This tool allows the physicist to observe whether or not radiation "leaks out" of the target or, likewise, if there is a section of the target that does not receive the prescribed dose. Figure 1.2 shows an example of how isodose curves show the deposition of radiation.

In examining Figure 1.3, the PTV stands for Prescribed Target Volume, which is the tumor (shown in blue), and we see that the 100% isodose line, which is the target dose level for this prescription, does not include the PTV and hence the PTV is under irradiated. It is immediately clear that this metric provides the spatial information not found in the other metrics. In some sense this gives us a visible presentation of what a treatment means to the patient's anatomy, and we see how a treatment attempts to cope with the geometry of the anatomy.

Our approach for computing isodose curves is to first interpolate a smooth surface over the data using a B-spline surface and then to find the level curve corresponding to an isolevel by using the hit-and-run algorithm, which ensures

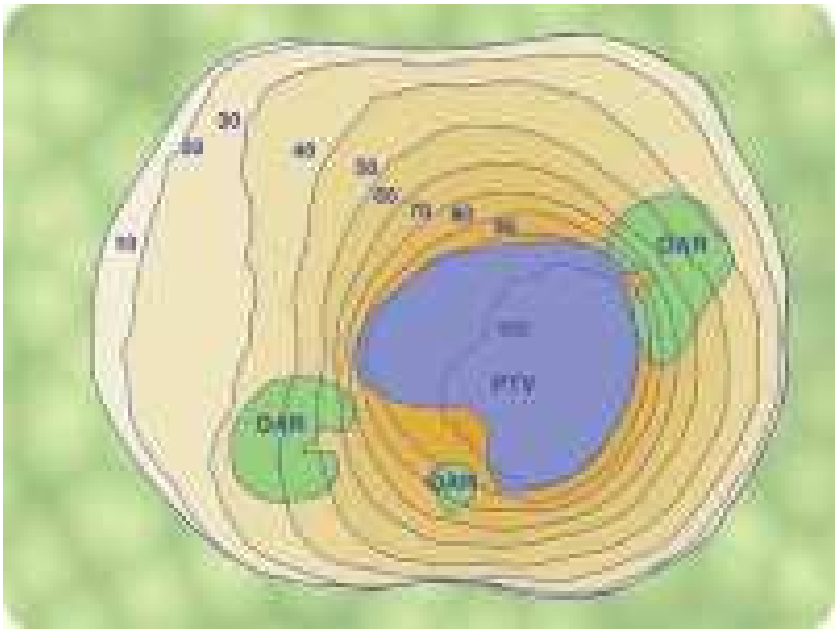


Figure 1.3: Isodose curves [12]

that we uniformly sample the isodose curve [4] (if the isoregion is convex, otherwise practical application still tells us that hit-and-run performs well). Computationally, we process each image separately and use parallel techniques to compute the isodose curves for each CAT Scan.

Chapter 2

Curve and Surface Fitting

The input data is the deposited data on a regular grid of sampled dose points, to which an interpolating surface is fit. We first consider methods for interpolating curves to data in the plane and then look at the similar methods used for interpolating surfaces in three dimensions.

The term spline, in a mathematical sense, is a piecewise polynomial function that is typically of a simple form that has smooth properties, which make it ideal for modeling arbitrary functions in computer graphics. The term spline, in a general sense, has been around much longer than this mathematical definition. The first mention of a spline occurs in 1752 and refers to a tool used in shipbuilding [7]. In this case a spline is a piece of wood that is bent with the use of metal weights that allows for the drawing of a smooth curve. Another type of tool used for this same purpose is called a French curve and consists of pieces of conics and spirals. Curves can be drawn by tracing appropriate parts of the French curve to piecewise construct the curve. The splines we think of today combine these two ideas in that we attempt to draw a smooth piecewise curve that passes through or near specified data points.

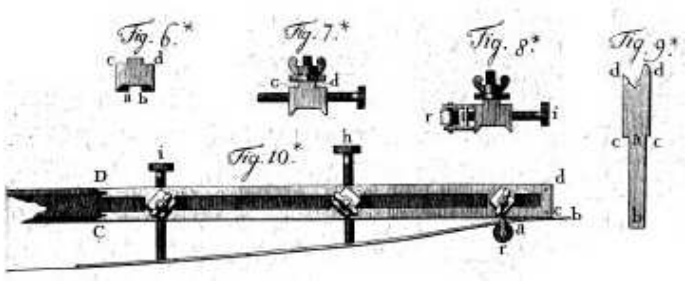


Figure 2.1: Mechanical Spline from the 1700s [7]

2.1 Cubic Splines

For the purposes of interpolating data, there are a variety of methods. A naive approach is to simply "connect the dots" with a piece-wise linear interpolation. However, data is seldom linear, and higher order polynomials are typically more appropriate. Lagrangian Polynomials and Divided Differences construct the unique polynomial of degree n that passes through $n+1$ data points, but here again how can we be sure that a single polynomial of a high degree (n) correctly interpolates the data? We use splines which consist of pieces of lower degree polynomials glued together to interpolate data in such a way that we do not induce extreme behavior in the curve. For the purpose of completeness we introduce cubic splines and show, through their derivation, how we construct a smooth curve that passes through our data points yet maintains the data trend beyond our data set. Our development follows the one found in [8], and interested readers should consult a standard Numerical Analysis text for further details. A cubic spline is a piecewise function consisting of $n-1$ third-order polynomials that match not only on the data points, but also agree in slope and curvature (first and second derivatives) where the splines meet. In this way our interpolating curve is smooth through all the points and what's more, should be more accurate than a simple exact fit polynomial.

The cubic spline function is of the form

$$g(x) = g_i(x) \text{ if } x \in [x_i, x_{i+1}], \text{ for } i = 0, 1, \ldots, n-1,$$

where each $g_i(x)$ has the equation

$$g_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i.$$

We desire

$$\begin{aligned} g_i(x_i) &= y_i, & i = 0, 1, \ldots, n-2; & \quad (2.1)\\ g_i(x_{i+1}) &= g_{i+1}(x_{i+1}), & i = 0, 1, \ldots, n-2; & \quad (2.2)\\ g_i'(x_{i+1}) &= g_{i+1}'(x_{i+1}), & i = 0, 1, \ldots, n-2; & \quad (2.3)\\ g_i''(x_{i+1}) &= g_{i+1}''(x_{i+1}), & i = 0, 1, \ldots, n-2. & \quad (2.4) \end{aligned}$$

From Eq. (2.1) we see that $d_i = y_i$, $i = 0, 1, \ldots, n-1$.

Now, looking at Eq. (2.2)

$$y_{i+1} = a_i(x_{i+1} - x_i)^3 + b_i(x_{i+1} - x_i)^2 + c_i(x_{i+1} - x_i) + y_i.$$

If we let $h_i = x_{i+1} - x_i$, we write

$$y_{i+1} = a_i h_i^3 + b_i h_i^2 + c_i h_i = y_i, i = 0, 1, \ldots, n-1.$$

Because $y_{i+1} = g_i(x_{i+1})$, we differentiate to find

$$\begin{aligned} g_i'(x) &= 3a_i h_i^2 + 2b_i h_i + c_i & i = 0, 1, \ldots, n-1;\\ g_i''(x) &= 6a_i h_i + 2b_i & i = 0, 1, \ldots, n-1. \end{aligned}$$

Notice that the second derivative is linear in h_i; we want to try to write the other equations in terms of the second derivative.

Let $S_i(x_i) = g_i''(x_i)$, for $i = 0, 1, \ldots, n-1$, and $S_n = g_{n-1}''(x_n)$. Then,

$$\begin{aligned} S_i(x_i) &= 6a_i(x_i - x_i) + 2b_i = 2b_i, \text{ and} \\ S_i(x_{i+1}) &= 6a_i h_i + 2b_i. \end{aligned}$$

So,

$$\begin{aligned} b_i &= \frac{S_i}{2} \\ a_i &= \frac{S_{i+1} - S_i}{6h_i}. \end{aligned}$$

Substituting these and solving for c_i, we have that

$$\begin{aligned} y_{i+1} &= \left(\frac{S_{i+1} - S_i}{6h_i}\right) h_i^3 + \frac{S_i}{2} h_i^2 + c_i h_i + y_i \text{ and} \\ c_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{2h_i S_i + h_i S_{i+1}}{6}. \end{aligned}$$

We ensure that the slopes of the cubics that join at points (x_i, y_i) are the same,

$$y_i' = 3a_i(x_i - x_i)^2 + 2b_i(x_i - x_i) + c_i = c_i.$$

Noticing that in the previous interval we had

$$\begin{aligned} y_i' &= 3a_{i-1}(x_i - x_{i-1})^2 + 2b_{i-1}(x_i - x_{i-1}) + c_{i-1} \\ &= 3a_{i-1}h_{i-1}^2 + 2b_{i-1}h_{i-1} + c_{i-1}. \end{aligned}$$

We find upon substituting for a_i, b_i, c_i and d_i that

$$\begin{aligned} y_i' &= \frac{y_{i+1}-y_i}{h_i} - \frac{2h_iS_i + h_iS_{i+1}}{6} \\ &= 3\left(\frac{S_i - S_{i-1}}{6h_{i-1}}\right)h_{i-1}^2 + 2\left(\frac{S_{i-1}}{2}\right)h_{i-1} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{2h_{i-1}S_{i-1} + h_{i-1}S_i}{6} \\ &= \left(\frac{S_i - S_{i-1}}{2}\right)h_{i-1}^2 + S_{i-1}h_{i-1} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{2h_{i-1}S_{i-1} + h_{i-1}S_i}{6}. \end{aligned}$$

We simplify this equation, and obtain

$$6\left(\frac{y_{i+1}-y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right) = h_{i-1}S_{i-1} + (2h_i + 2h_{i-1})S_i + h_iS_{i+1},$$

and because $(y_{i+1} - y_i)/h_i$ is the formula for the divided difference, which we notate as $f[x_i, x_{i+1}]$, we rewrite our equation as

$$6(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]) = h_{i-1}S_{i-1} + (2h_i + 2h_{i-1})S_i + h_iS_{i+1}.$$

This last equation applies at each internal point ($i = 1, 2, \ldots, n-1$) and gives us $n-1$ equations and $n+1$ unknown S_i values. To obtain $n+1$ equations we set S_0 and S_n equal to 0, which makes the end cubics flatten out near their respective endpoints. This technique leads to a natural spline.

We solve the system of equations whose matrix form is

$$\begin{bmatrix} 2(h_0+h_1) & h_1 & & & & \\ h_1 & 2(h_1+h_2) & h_2 & & & \\ & h_2 & 2(h_2+h_3) & h_3 & & \\ & & \ddots & & & \\ & & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix} \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_{n-1} \end{bmatrix}$$

$$= 6 \begin{bmatrix} f[x_1, x_2] - f[x_0, x_1] \\ f[x_2, x_3] - f[x_1, x_2] \\ f[x_3, x_4] - f[x_2, x_3] \\ \vdots \\ f[x_{n-1}, x_n] - f[x_{n-2}, x_{n-1}] \end{bmatrix}.$$

After solving this system and obtaining the S_i values, we find the coefficients for the cubic corresponding to each interval and thereby compute our curve.

$$\begin{aligned} a_i &= \frac{S_{i+1} - S_i}{6h_i}; \\ b_i &= \frac{S_i}{2}; \\ c_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{2h_i S_i + h_i S_{i+1}}{6}; \\ d_i &= y_i. \end{aligned}$$

2.1.1 A Cubic Spline Example

Let us consider the following example in which we are asked to fit a cubic spline to the data in Table 2.1. These data points come from the function $f(x) = \sin x - xe^{-x}$, which allows us to examine how well a cubic spline can approximate a function that is not a polynomial. The matrix representation of

x	y
0.350000	0.096257
3.200000	-0.188813
4.410000	-1.008232
5.560000	-0.683175
7.100000	0.723111

Table 2.1: Sample Data

the problem is

$$\begin{bmatrix} 2(2.85+1.21) & 1.21 & \\ 1.21 & 2(1.21+1.15) & 1.15 \\ & 1.15 & 2(1.15+1.54) \end{bmatrix} \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} = 6 \begin{bmatrix} -0.5772814 \\ 0.959864 \\ 0.630514 \end{bmatrix}.$$

We find that $S_0 = 0$, $S_1 = -0.70167$, $S_2 = 1.84669$, $S_3 = 0.30844$, and $S_4 = 0$. The coefficients for each of the cubics are in Table 2.2.

i	a	b	c	d
0	-0.0410	0	0.2333	0.096257
1	0.3510	-0.3508	-0.7666	-0.188813
2	-0.2229	0.9233	-0.4844	-1.008232
3	-0.0334	0.1542	0.7548	-0.683175

Table 2.2: Spline Coefficients

Figure 2.2 shows the actual function $f(x) = \sin(x) - xe^{-x}$ plotted on the same graph as the cubic spline. The true function is in green, and the spline is in blue. Notice that with only 5 points we get an accurate fit. We also note that our spline is more accurate near the data points. If we look at $x = -50$, we have no guarantee that our spline approximates $f(x)$ well although we postulate the behavior of $f(x)$ based on the trend of the data.

2.2 B-Splines

The definition of a cubic spline requires that the spline fits the data exactly, yet still retain smoothness. A B-spline does not necessarily pass through the data, but has the desirable property that it is contained entirely within the convex hull determined by the data points. The convex hull of a set of points S is the smallest convex set that contains all points in S. The data of a B-spline instead weights the curve rather than requiring an exact fit and a change to a data point

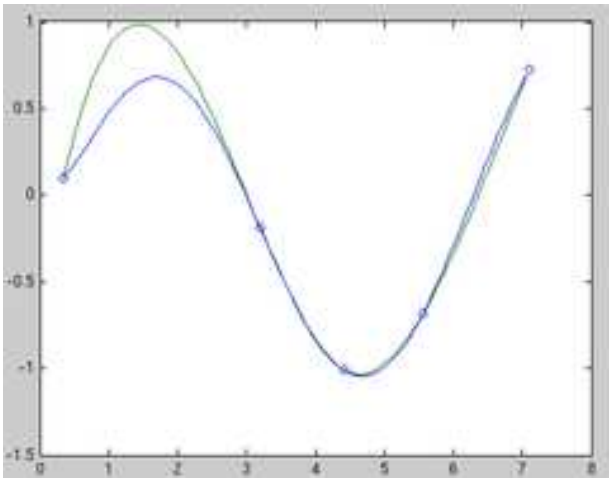


Figure 2.2: Cubic Spline plotted with actual function

will only affect the spline within a certain local interval rather that change the spline as a whole.

2.2.1 Bézier Curves

B-splines are a generalization of Bézier curves. In fact, they are a combination of Bézier curves and cubic splines in that B-splines are multiple Bézier curves pieced together into a spline that has many of the same properties of cubic splines (the curves meet in function values, as well as first and second derivatives). Bézier curves were developed in 1959 by a young engineer named Paul de Faget de Casteljau while working for the French car company Citroën. These curves were used to design smooth automobile bodies, and Citroën kept de Casteljau's work a company secret. About the same time as de Casteljau worked on his curves and surfaces at Citroën, Pierre Bézier worked on the same idea for Rénault, a competing French automaker. Although Bézier published his work in 1962, and thus received the credit for the curves and surfaces that bear his name, we know that both Bézier and de Casteljau developed nearly

identical results from different mathematical perspectives [7].

Bézier curves approximate $n+1$ points of data with a polynomial of degree n. These curves do not pass through all the data points, only the first and last points. The intermediate points determine the slope and curvature of the curve. In a cubic Bézier curve, the line connecting points p_0 and p_1 determines the slope of the curve at p_0 and the line connecting points p_2 and p_3 determines the slope of the curve at p_3. If we let the coordinates of each point represent the vector,

$$p_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

then the cubic Bézier curve is of the form

$$\begin{aligned} x(u) &= (1-u)^3 x_0 + 3(1-u)^2 u x_1 + 3(1-u)u^2 x_2 + u^3 x_3, \\ y(u) &= (1-u)^3 y_0 + 3(1-u)^2 u y_1 + 3(1-u)u^2 y_2 + u^3 y_3, \end{aligned}$$

where $0 \leq u \leq 1$. Notice that the point $(x(0), y(0))$ is equivalent to p_0 and $(x(1), y(1)) = p_3$. Also, note how the Bézier curve uses the points p_1 and p_2 to pull the curve toward them without forcing the curve to pass through them. This shows the regression aspect of the Bézier curve which B-splines inherit. Also, as previously mentioned, because $dx/du = 3(x_1 - x_0)$ and $dy/du = 3(y_1 - y_0)$ at $u = 0$ then the slope of the curve is $dy/dx = (y_1 - y_0)/(x_1 - x_0)$ which is the slope of the line connecting p_0 and p_1. Similarly the slope of the line at $u = 1$ is $dy/dx = (y_3 - y_2)/(x_3 - x_2)$ which is the slope of the line connecting p_2 and p_3.

Often we represent cubic Bézier curves as

$$P(u) = [u^3, u^2, u, 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

or

$$P(u) = u^T M_2 p.$$

2.2.2 B-Splines in the Plane

B-splines use Bézier curves as the basis for the spline. Just as cubic splines used portions of cubic polynomials between data points, B-splines use cubic Bézier curves. The general form for the cubic B-spline for the interval (p_i, p_{i+1}) (where p_i represent points in our data set) is

$$\begin{aligned} x_i(u) &= \frac{1}{6}(1-u)^3 x_{i-1} + \frac{1}{6}(3u^3 - 6u^2 + 4)x_i \\ &\quad + \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1)x_{i+1} + \frac{1}{6}u^3 x_{i+2}; \\ y_i(u) &= \frac{1}{6}(1-u)^3 y_{i-1} + \frac{1}{6}(3u^3 - 6u^2 + 4)y_i \\ &\quad + \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1)y_{i+1} + \frac{1}{6}u^3 y_{i+2}. \end{aligned}$$

Here again, $u \in [0, 1]$. We use the set of four points to generate the portion of the B-spline associated with the two inner points. Thus as we move along the data set we set $p_i = p_{i+1}$ for $i = 0, 1, 2$ and we set p_3 to the next point in the data set.

B-splines have the same characteristics as the cubic splines we discussed earlier: continuity of the curve and equality of the first and second derivatives. We examine each of these conditions:

(a) Continuity of the curve.

$$B_i(1) = B_{i+1}(0) = \frac{p_i + 4p_{i+1} + p_{i+2}}{6}.$$

(b) Equality of the first derivative.

$$B_i'(1) = B_{i+1}'(0) = \frac{-p_i + p_{i+2}}{2}.$$

(c) Equality of the second derivative.

$$B_i''(1) = B_{i+1}''(0) = p_i - 2p_{i+1} + p_{i+2}.$$

We can represent the cubic B-splines, much like Bézier curves, as

$$B_i(u) = \frac{1}{6}[u^3, u^2, u, 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{bmatrix}$$

or

$$B_i(u) = \frac{u^T M_b p}{6}.$$

B-splines are a slight variation on a Bézier curve, and in fact only differ in three ways [8].

1. For a B-spline, the curve does not begin and end at the extreme points.
2. The slopes of the B-splines do not have any simple relationship to lines drawn between the points.
3. The endpoints of a B-spline are in the vicinity of the two intermediate given points, but neither the x- nor the y- coordinates of these endpoints

normally equals the coordinates of the intermediate points.

Item 1 simply means that the curve is not fixed at either end. However, a spline between two points requires a total of four points (by adding the two neighboring points) so the splines at the end of the curve are computed with multiple copies of the endpoints. In this way, the ends of the curve are "weighted" toward the extreme point, but they are not required to evaluate to that value.

Item 2 refers to a characteristic of Bézier curves where the slope of the curve approaches the slope of the line connecting points p_0 and p_1 as the curve approaches p_0. The Bézier curve also approaches the slope of the line connecting p_2 and p_3 when the curve is near p_3.

From item 3 we can see that the spline is close to the anchor points, yet almost never equals those points. This implies that the spline smooths some of the bumps in the data and highlights one of the advantages a B-spline has over cubic splines. For example, a data point might force a cubic spline to curve outside of the trend of the data in order to pass through that point. Not only that, but because of the definition of the cubic spline, a change to that one point of data will affect the entire curve. Changes to data within a B-spline affect only local changes, and what's more, because the spline is not required to pass through the data points, we can reduce the effect of outliers in the data on the curve.

2.3 B-Spline Surfaces

So far we have discussed methods for fitting a smooth curve to data points in the plane, but the data for dose deposition is in three dimensional space. In this case we need to extend our method to a higher dimension and use tools to fit a smooth surface to data. Most splines for this purpose are bicubic (that is, cubic in x and cubic in y) simply because, as in dealing with fitting curves in the

plane, the cubic is the simplest polynomial that we have sufficient control over both slope and curvature. We choose the bicubic surface for the same reasons in creating a surface spline [6].

When we created B-spline curves we used four points to determine the spline between the two intermediate points. For surfaces, we extend the dimensions and use sixteen points to fit a patch between the four interior points. That is, for patch $S_{i,j}$ we use the matrix of data points (often called control points) given as

$$Q_{ij} = \begin{bmatrix} p_{i-1,j-1} & p_{i-1,j} & p_{i-1,j+1} & p_{i-1,j+2} \\ p_{i,j-1} & p_{i,j} & p_{i,j+1} & p_{i,j+2} \\ p_{i+1,j-1} & p_{i+1,j} & p_{i+1,j+1} & p_{i+1,j+2} \\ p_{i+2,j-1} & p_{i+2,j} & p_{i+2,j+1} & p_{i+2,j+2} \end{bmatrix}.$$

Patch $S_{i,j}$ fits a surface between points $p_{i,j}$, $p_{i,j+1}$, $p_{i+1,j}$, and $p_{i+1,j+1}$. The way we define the patches allows us to separate the domain of the spline into separate rectangular regions. Similar to the way in which we copied the endpoints for weighting for curves in the plane, we copy the edges of the data set so that each patch has sixteen control points to create the surface. Once again, copying the boundaries and the corners help us weight the surface toward the control points.

Figure 2.3 shows a B-spline patch with its control points. Because the only data needed to compute the surface patch is sixteen points surrounding the patch, any adjustments to the data or outliers within the data have only local effects. Once again, B-spline surfaces, like B-spline curves, avoid overfitting the data by examining the interpolation problem as small, localized regions rather than tackling the global problem all at once.

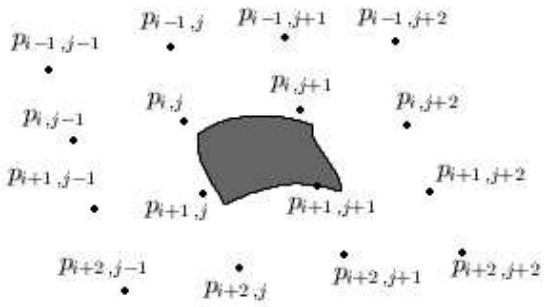


Figure 2.3: B-spline patch with its control points

2.3.1 Computing B-spline Surfaces

A B-spline surface is composed of patches over its domain. Given data points in an $m \times n$ grid we compute $(m-1)\times(n-1)$ patches $S_{rs}(u,v), r = 1,2,\ldots,m-1, s = 1,2,\ldots,n-1$. Each patch is a bicubic polynomial, meaning that it is expressed as

$$S_{rs}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} c_{ij} u^i v^j.$$

We compute these patches as

$$S_{rs}(u,v) = \frac{1}{36} U M Q_{rs} M^T V^T, 0 \leq u, v, \leq 1,$$

where $U = [u^3, u^2, u, 1], V = [v^3, v^2, v, 1]$,

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix},$$

and

$$Q_{rs} = \begin{bmatrix} p_{r-1,s-1} & p_{r-1,s} & p_{r-1,s+1} & p_{r-1,s+2} \\ p_{r,s-1} & p_{r,s} & p_{r,s+1} & p_{r,s+2} \\ p_{r+1,s-1} & p_{r+1,s} & p_{r+1,s+1} & p_{r+1,s+2} \\ p_{r+2,s-1} & p_{r+2,s} & p_{r+2,s+1} & p_{r+2,s+2} \end{bmatrix}.$$

When evaluating $S(x, y)$ we have that

$$S(x, y) = S_{rs}(x - r, y - s), (x, y) \in [1, m] \times [1, n],$$

where $r = \lfloor x \rfloor$ and $s = \lfloor y \rfloor$. In this manner, for a point (x, y) in the domain we know which patch corresponds to the region containing that point simply by evaluating the floor of x and y.

Computing the B-spline surface requires only matrix multiplication of 4×4 matrices. We store the matrices $1/36MQ_{rs}M^T$ as coefficients to the bicubic so that the description of each is

$$\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}.$$

Upon inspection of this form we notice that the first row has an x^3 term, the second row an x^2 term, etc, and we also notice that the first column has a y^3 term, the second column a y^2 term, etc. This insight allows us to compute the partial derivatives of S_{rs} quite easily. We obtain,

$$\frac{\partial}{\partial u}S_{rs} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 3a_{00} & 3a_{01} & 3a_{02} & 3a_{03} \\ 2a_{10} & 2a_{11} & 2a_{12} & 2a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix},$$

and

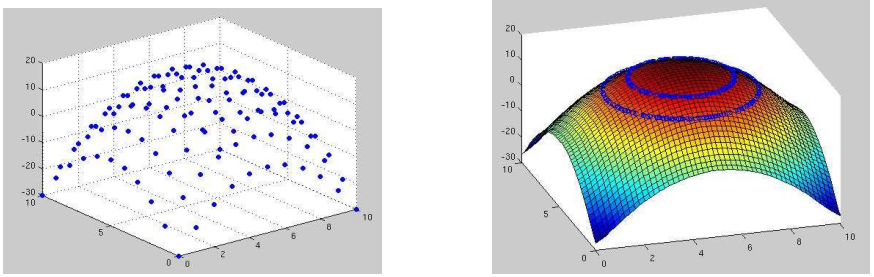
$$\frac{\partial}{\partial v}S_{rs} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 0 & 3a_{00} & 2a_{01} & a_{02} \\ 0 & 3a_{10} & 2a_{11} & a_{12} \\ 0 & 3a_{20} & 2a_{21} & a_{22} \\ 0 & 3a_{30} & 2a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}.$$

So differentiating only require shifting a row or column and multiplying by scalars, and thus Newton's method is an attractive option. Also, as was mentioned earlier, computing each patch only requires the data for the matrix Q_{rs}, which implies that we can parallelize this computation.

2.3.2 A B-spline Surface Example

To demonstrate how we fit a surface to data points, we generate data for the function $f(x, y) = 20 - (5 - x)^2 - (5 - y)^2$ such that $x, y \in [0, 10] \times [0, 10]$ and $x, y \in \mathbb{Z}$. Then, the data is a set of 121 points on a regular grid, and we fit 100 surface patches to the data. Figure 2.4 shows a scatter plots of the data points generated by the function and the B-spline surface obtained.

It is difficult to tell from the figures, but the B-spline surface tends to smooth the data; the peaks are not as high and the valleys are not as low. The surface pulls toward the mean. We can see that the corners of the domain seem to curve upward. This is due to the method duplicating the boundary points.



(a) Control points (b) B-Spline with Isocurves

Figure 2.4: Control points and resulting B-Spline surface

Chapter 3

The Hit and Run Algorithm

Now that we have a functional representation of the radiation distribution, we want to find a level curve of that surface. In order to accomplish this we explore a class of algorithms called the shake-and-bake algorithms, which will allow us to find a set of uniformly distributed points on the bounding curve for that isoregion. The particular algorithm we consider, called the hit-and-run algorithm, begins with an initial iteration, x_0, and randomly generates a direction to search. (Because we are searching within the plane we need only generate an angle θ on the uniform distribution from 0 to 2π and search along the vector $(\cos\theta, \sin\theta)$). We then search along this vector to find the point of intersection with our isodose curve. Once we have an intersection point, $\hat{x}_1$, we take the midpoint on the line segment connecting x_0 and $\hat{x}_1$ by setting $x_1 = x_0 + (\hat{x}_1 - x_0)/2$ and again generate a random direction in which to search.

To find the intersections along the vector with the isodose curve, we employ Newton's method because we have projected the problem into $\mathbb{R}^2$ by narrowing our search domain to the line defined by x_0 and θ. Newton's method converges in $O(n^2)$, so searching is reasonable. Additionally, because each search is done

independently of the others, we parallelize the searching. This is what is known as an "embarrassingly parallel" operation in that each operation is completely independent of the others, assuming of course that the random numbers generated by the processes are not the same.

The benefit of using the hit-and-run algorithm is that we have a certificate that the algorithm uniformly samples the isodose curve. Boender et al. [4] prove that all algorithms in the class SB (Shake-and-bake) converge to the uniform distribution on the boundary of a convex polyhedra.

Theorem. *For every algorithm in the class SB, the random sequence* $\{X^n\}_{n=0}^{\infty}$ *of iteration points converges to the uniform distribution on* ∂S*, independently of the starting point in the set* S*.*

Computationally, the first step is to choose some point that is on the interior of the isoregion. Our approach is to find the dose point with the maximum dose value (maximum height) and start from that point. More precisely, we start near that point by moving a small distance. This is to account for a numeric instability in Newton's method that occurs if $\nabla f(x, y) = 0$. The first directions we search are not entirely random. Our method first searches along eight vectors, for $\theta = n\pi/4, n = 0, 1, 2, \ldots, 7$, which attempts to diversify the search. To keep track of how many points we have on the curve, we use a vector and add new points as they are found. Also, we push new starting points onto a queue as we find points on the curve and compute the midpoint of that search vector.

3.1 Newton's Method

Newton's method is a widely used algorithm for root-finding. It uses the slope of the tangent to approximate linearly the root. This algorithm is iterated by continuing to move to the intersection of the tangent line and the x-axis and

recomputing the tangent line for that x-value. In computing the next iteration, we let

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Numerically, the algorithm becomes unstable near a horizontal asymptote or local extremum because $f'(x_n)$ approaches zero. Figure 3.1 shows an example of how Newton's method works. We see that each successive iteration moves closer to the root. The tangent lines are shown in green, and we see also how they are used to compute the next iteration point.

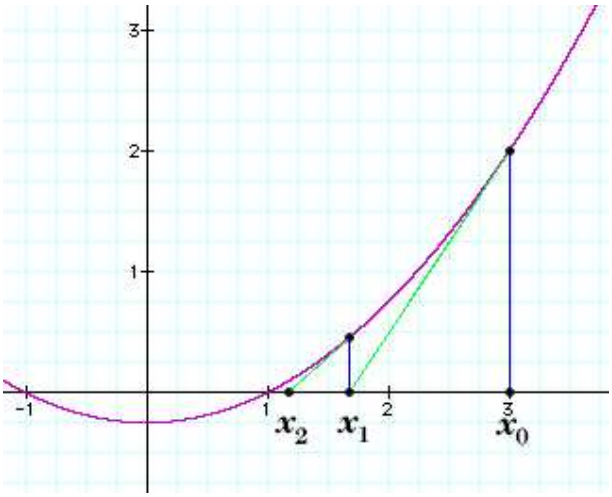


Figure 3.1: Newton's Method for three iterations [10]

The distance between the current point and the new approximation is called the Newton step. Sometimes in working with our B-spline surface, the Newton step attempts to jump beyond the domain. If that were allowed, because the function is undefined there, the program would crash. In order to compensate for attempts to escape the domain we place a simple check after each iteration of Newton's method. If our iteration fails to be in our domain, then we compute the intersect of the search vector with the boundaries of our spline's domain. Because the domain models the dimensions of a CAT scan, the domain is rect-

angular, so computing those intersections does not add any complexity to the algorithm.

Newton's method converges to a root quadratically ($O(n^2)$), which means that the number of accurate digits roughly doubles on each step. Numerical methods only seek to find an x-value that has a functional value within a specified tolerance of 0 because of the imprecision of floating point arithmetic. However, Newton's method does not always converge; it suffers from the possibility of cycling. It is plausible that Newton's method might step past the root and then on the next iteration return to the original point. In order to catch this we set a limit on the number of iterations Newton's method can perform before giving up. Theoretically, we should converge to a tolerance such as 0.000001 in only 3 iterations. (Recall that the algorithm converges quadratically). However, to be safe we wait 100 iterations before declaring it a cycle.

The hit-and-run algorithm efficiently (coupled with Newton's method) provides a uniform sampling of the boundary of the isoregion. Beyond the speed characteristics of the algorithm, it also gives us a certificate of uniform sampling, which allows us to see the form of the isodose curve.

Chapter 4

Computing the Isodose Curves

The sample radiation data used in these experiments comes from software developed at Trinity University as part of the RAD project (Radiotherapy OptimAl Design). The system computes dose deposition for treatments by inverse planning.

4.1 Newton's Method

As discussed earlier, Newton's method is our choice of a root-finding algorithm. In order to use this method we redefine the relationship between successive iteration points.

$$x_{n+1} = x_n + \alpha d$$

where

$$d = (\cos\theta, \sin\theta)$$

and

$$\alpha = \frac{k - f(x_n)}{\nabla f(x_n)^T d}$$

for some isolevel k. This adjustment to the method simply allows us to project the root-finding problem into two dimensions by using the line $x_0 + \lambda d$, $\lambda \in \mathbb{R}$. All movement must stay on that line, so we move a scalar multiple of d. Also, we use the directional derivative ($\nabla f(x)^T d$) to measure the Newton step.

The function for Newton's method looks like this

```
// ==== Begin Newton's Method ==== //
X0 = X;
if(X0[0] < 0 || X0[0] > (n-1) || X0[1] < 0 || X0[1] > (m-1))
    return;
int ctr = 0;
Vector2 OldX;
do
{
    double alpha = ((isoLevel-f(X,surface,m,n))/
                    (fpartialx(X,surface,m,n) * d[0] +
                     fpartialy(X,surface,m,n) * d[1]));
    OldX = X;
    if(ctr == 0)
    {
        // Make sure to step forward
        if(alpha < 0)
        {
            alpha *= -1;
        }
    }
    X = X + d * alpha;

    DomainCheck(X,theta,X0,m,n);
    ctr++;
    if(ctr >= 100)
    {
        cycle = true;
        break;
    }
}while(abs(f(X,surface,m,n)-isoLevel) > 0.00001);
// ==== End Newton's Method ==== //
```

This follows our mathematical understanding of Newton's method almost com-

pletely, with a few adjustments. First, the check to ensure that the first step is in the forward direction attempts to make sure that the root we find will be in the positive d direction. If our starting point is closer to one side of the boundary than the other then it may be possible for Newton's method to try to step backward (as we are thinking of it), which would reduce the number of random directions by half. This check does not guarantee that we will find a root in the positive d direction, but it helps to influence that trend.

The second adjustment is the DomainCheck function call. As previously discussed, we need to ensure that our iteration points stay within the defined domain for our B-spline. DomainCheck determines whether our new iteration point is outside of the domain and, if necessary, cuts the Newton step short in order to keep the problem well defined.

4.2 Domain Check

The DomainCheck function is essential to the algorithm's stability. Newton's method often attempts to step past the bounds, so our defense lies in this function. The check is quite simple, it looks at the x and y coordinates of the iteration point to determine if any of them exceed the boundaries. If this is the case, we compute values k such that $X_0 + kd$ hits a boundary value in one of the coordinates. That is to say, that $X_{0,x} + k \cdot \cos\theta = 0$ or $(n-1)$ or that $X_{0,y} + k \cdot \sin\theta = 0$ or $(m-1)$. We choose the minimum k value to ensure that we are within all bounds. Additionally, to account for errors in floating point arithmetic we decrease $|k|$ by a small constant (0.0001) to guarantee that our method strictly satisfies our boundary constraints. The following source code snippet of the DomainCheck function places additional checks on $\cos\theta$ and $\sin\theta$ to avoid any possible divisions by zero. Also, to double check the method we call the same check after adjusting the iteration point. If this check fails, the

program exits.

```
if(X[0] > (n-1) || X[0] < 0 || X[1] > (m-1) || X[1] < 0)
{
    Vector2 OldX = X;
    double k = 1000000;

    if(abs(cos(theta)) > 0.0001)
    {
        k = ((n-1)-X0[0])/cos(theta);
        if(abs(-X0[0]/cos(theta)) < abs(k))
            k = -X0[0]/cos(theta);
    }
    if(abs(sin(theta)) > 0.0001)
    {
        if(abs(((m-1)-X0[1])/sin(theta)) < abs(k))
            k = ((m-1)-X0[1])/sin(theta);
        if(abs(-X0[1]/sin(theta)) < abs(k))
            k = -X0[1]/sin(theta);
    }
    if(k<0 && k < -0.001)
        k+=0.001;
    else if(k > 0 && k > 0.001)
        k-=0.001;
    X = X0 + Vector2(cos(theta) * k, sin(theta)*k);
    if(!(X[0]<=(n-1)&&X[0]>=0&&X[1]<=(m-1)&&X[1]>=0))
    {
        cerr << "Error in Domain Check!\n";
        cerr << "OldX = " << OldX << " \t" << "X = " << X << endl;
        cerr << "Theta = " << theta << " \tk = " << k << endl;
        cerr << "X0 = " << X0 << endl;
        assert(X[0]<=(n-1)&&X[0]>=0&&X[1]<=(m-1)&&X[1]>=0);
    }
}
```

4.3 Example Cancer Treatment

The sample problem we examine is an acoustic neuroma. The CAT scan in Figure 4.1 shows the head, and we see that the clinic has highlighted the left eye, the spinal cord, and the tumor.

After computing a treatment plan with RAD, we run our method on the

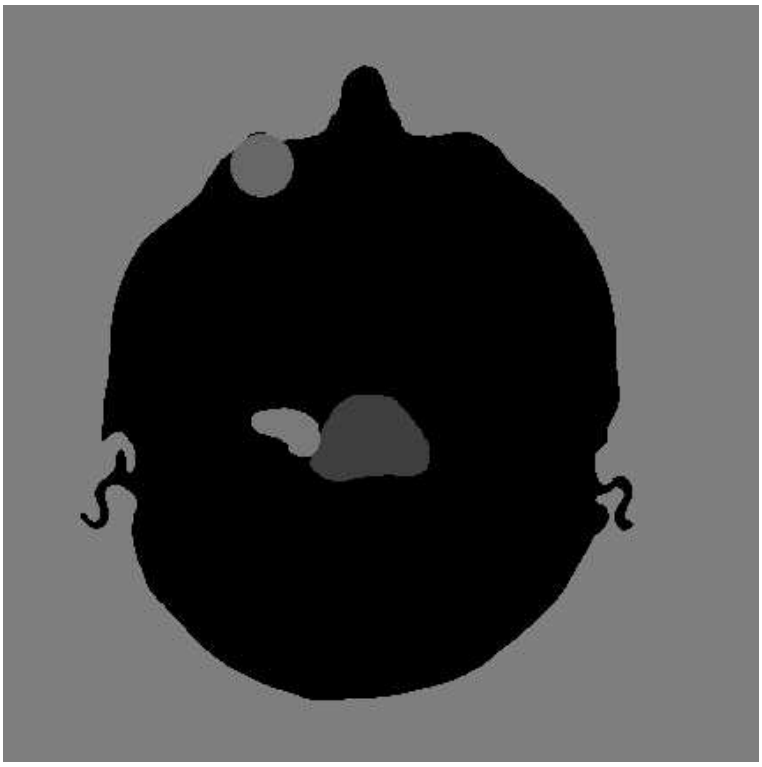


Figure 4.1: CAT Scan

surface spline over the deposited dose and compute the 50% isodose curve (50% of the prescribed target dose, or dose to the tumor), which is shown in Figure 4.2.

We find that Newton's method cycles when the surface approaches vertical asymptotes, which nearly occurs at the boundary of the target. When this happens we do not have points on that part of the boundary. This is a numerical flaw in our method; one that needs further investigation. However, we know that if we had a more consistent method of computing intersections with the boundary then we would have a uniform sampling [4].

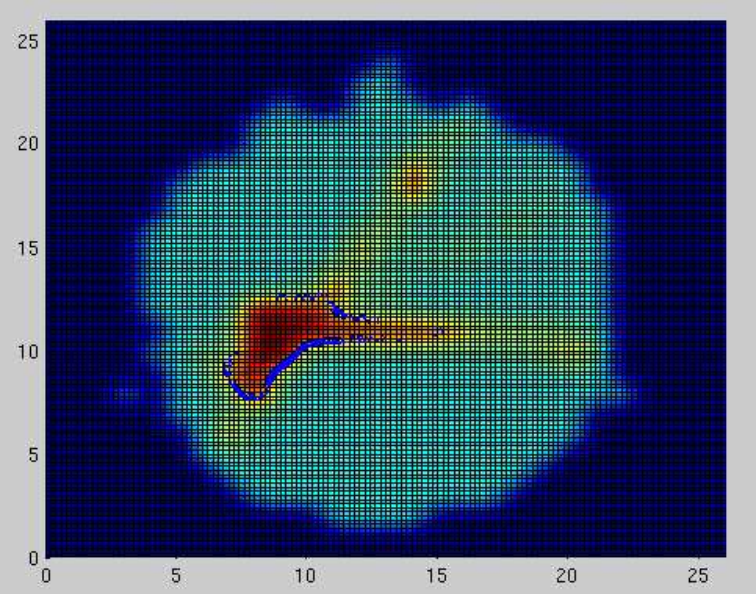


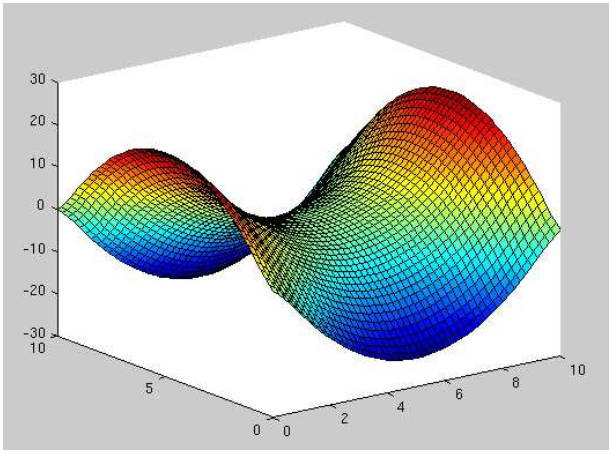
Figure 4.2: Treatment plan with 50% isodose curve

4.4 Drawbacks of the Method

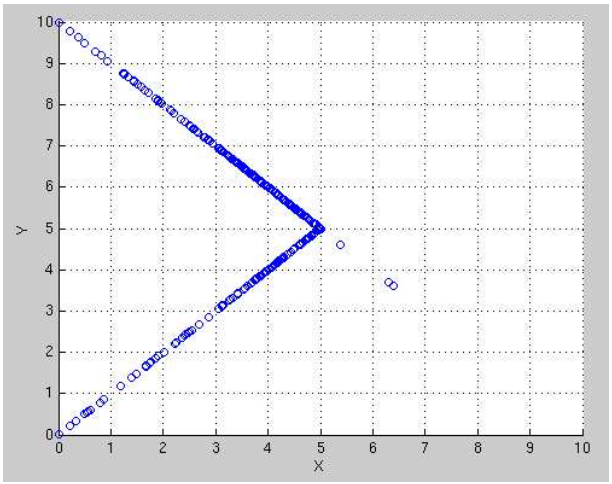
The method we use samples the isodose curve uniformly along the boundary, but this is not a true level curve of the splined function. While the scatter plot of these points is sufficient to identify the isodose curve, it might be considered to be an over approximation for the level curve. If we knew the sequence of the points around the boundary we could create a spline of those points using the methods discussed in Chapter 2. However the nature of the hit-and-run algorithm is such that we find boundary points is no particular order, and attempting to sort the points in any meaningful way is difficult.

The method also depends heavily on where we begin the hit-and-run algorithm. If we place the starting point within the isoregion then the algorithm behaves well. However, if there are two disjointed regions that represent a particular level of dose, we are limited to finding only the region in which we have an iteration point in the interior. For example, when we try to find an isolevel

curve for $f(x,y) = (5-x)^2 - (5-y)^2$ at $f(x,y) = 0$ we see that there are two distinct regions with values greater than zero. If we place our first iteration in the interior of the left-side region $(0 \leq x \leq 5, y \leq 10 - x, y \geq x)$. Then the boundary we sample is only of that region, with only a small chance of finding a few points on the boundary of right-side region.



(a) $f(x,y)$



(b) Example of the algorithm finding only one region

Figure 4.3: B-spline surface example for a saddle point

Figure 4.3 shows $f(x,y)$ along with a scatter plot of the boundary points

that the hit-and-run algorithm finds when the first iteration point is at $(0, 5)$. We can see that a few points are on the boundary for the right-side region, but for the the most part, the algorithm misses the right-side region. If we had no prior knowledge of the surface we might think those points were mistakes.

Because we notice that in the previous example the algorithm finds a few points on the second region, we explore the possibility that increasing the quota of points that the algorithm catalogs before stopping will improve our sampling over disjointed regions. To test this theory we examine a surface with many disjointed regions. The surface defined by $f(x, y) = \cos x \cdot \sin y$ appears in Figure 4.4.

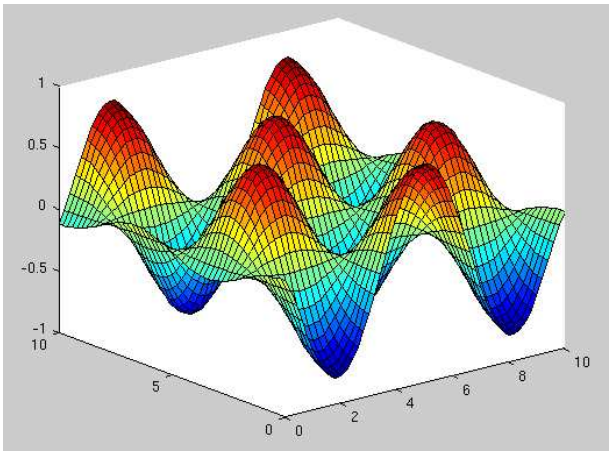


Figure 4.4: $f(x, y) = \cos x \cdot \sin y$

If we search for the level curve at $f(x, y) = 0.5$ there are isolevel curves surrounding each of the six peaks. The algorithm chooses to start at $(5, 0)$ and because of the close proximity of the disjointed regions the algorithm will hit the boundaries of several of the regions. We run the program with search quotas of 100, 200, 500, and 1000 boundary points and compare the results to determine if an increase in the sampling of the boundaries will better find disjointed regions. The resulting scatter plots are shown in Figures 4.5 – 4.8. We should find

that there are six circles outlined by the blue points, and we conjecture that increasing the number of boundary points will improve the resolution of those circles.

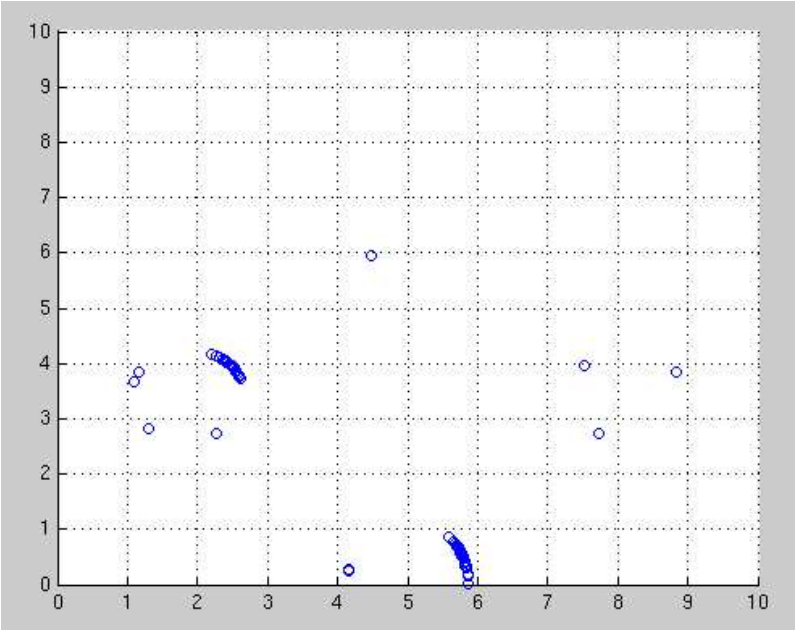


Figure 4.5: Sampling 100 Points

We can see that increasing the search quota for the algorithm does not necessarily improve the quality of the level curves. Rather than increasing the number of sample points, increasing the number of starting points to include those other regions will sample all boundaries. The problem lies in how we try to place starting points. If we include all points of the data set that achieve the max of the set, then we can improve our sampling, but we ignore those regions which do not achieve the same height as that max.

Fortunately for us, radiation distributions behave well. That is to say that we seldom have disjointed isoregions due to the nature of treatment planning. Most radiation accrues near the isocenter and inside the tumor and from there

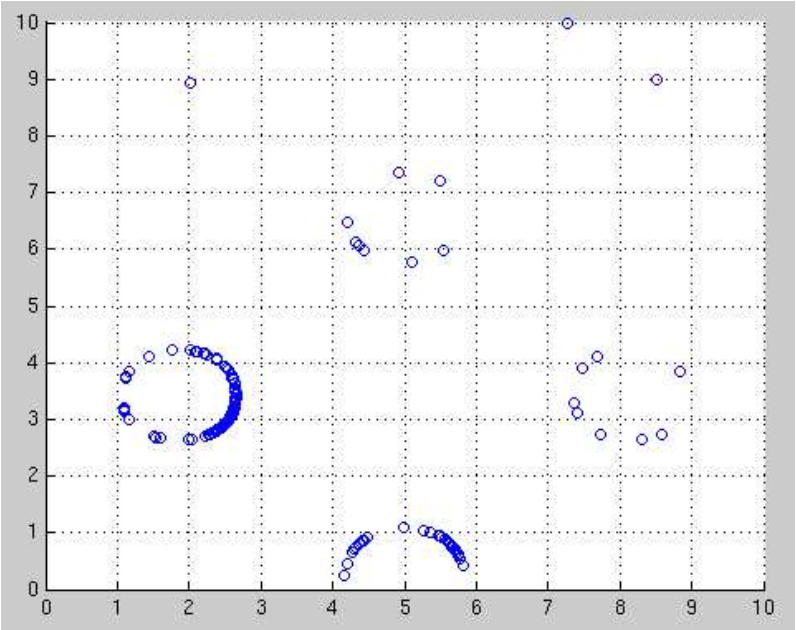


Figure 4.6: Sampling 200 Points

the level of radiation falls away.

4.5 Conclusions

While this method is not perfect, it achieves its goal. It provides us with a spatial understanding of how a treatment deposits radiation into the anatomy. Our novel approach provides us with fast executions and an interesting perspective on the isodose problem.

By benefit of the theorem in Boender et al. [4], we know that this is a valid approach to finding isodose curves. What problems exist with the method have to do with our implementation of the principles of finding the boundary intersection points. Improvements to Newton's method in terms of finding multiple roots (up to 3 for this problem) will help to improve the quality of our isodose curves as well as assist in finding the disjointed isoregions.

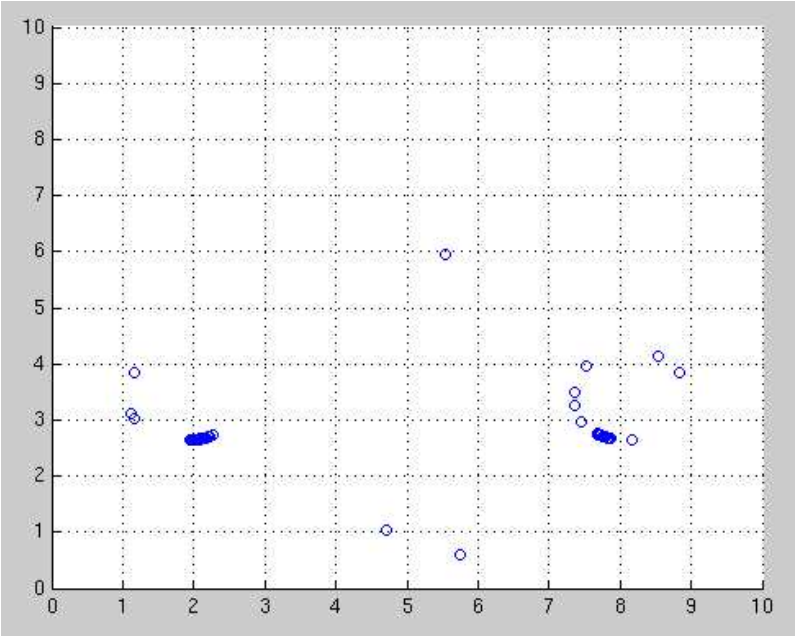


Figure 4.7: Sampling 500 Points

Anticipated future work will examine the possibility of achieving a sampling of points in a known order so that we can fit a spline to the sampled points. Another idea that we can consider is to use the CAT scan's pixel resolution to our advantage and limit our search to that discrete set. If we color the pixels that correspond to our sample points we may find that we indeed have what appears to the eye to be a continuous curve. Curiosity to try adjustments to the method implies that a solution may never truly be finished, yet takes the form of an ever-changing (and hopefully improving) entity while always tackling the same problem.

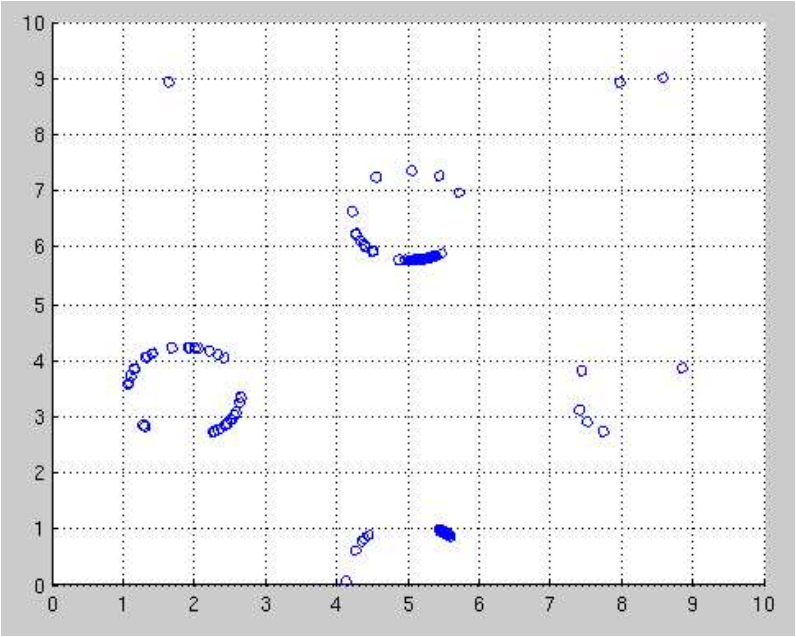


Figure 4.8: Sampling 1000 Points

Bibliography

[1] Sacred Heart Cancer Center.

[2] The TomoAdvantage: Simulated Hi-Art treatment plan for breast cancer.

[3] Cancer: Number One Killer. *BBC News online*, November 2000.

[4] C. G. E. Boender, R. J. Caron, J. F. McDonald, A. H. G. Rinnooy Kan, H. E. Romeijn, R. L. Smith, J. Telgen, and A. C. F. Vorst. Shake-and-Bake Algorithms for Generating Uniform Points on the Boundary of Bounded Polyhedra. *Operations Research*, 39(6), November-December 1991.

[5] D. Cheek, A. Holder, B. Salter, and M. Fuss. The Relationship Between the Number of Shots and the Quality of Gamma Knife Radiosurgeries. Technical Report 84, Trinity University, 2004.

[6] Fuhua Cheng and Ardeshir Goshtasby. A Parallel B-Spline Surface Fitting Algorithm. *ACM Transactions on Graphics*, 8(1):41–50, January 1989.

[7] Gerald Farin. A History of Curves and Surfaces in CAGD.

[8] Curtis F. Gerald and Patrick O. Wheatley. *Applied Numerical Analysis*. Addison Wesley Longman, Menlo Park, California, sixth edition, 1999.

[9] A. Holder and B. Salter. *A Tutorial on Radiation Oncology and Optimization*, chapter 4. 2004.

[10] David E. Joyce. Newton Basins. August 1994.

[11] Marilynn Marchione. Cancer Now No. 1 Killer Illness in United States. *The Arizona Republic Online*, January 2005.

[12] Inizio Pagina. La protezione del paziente in radioterapia. 15(4), April 2002.

Appendix A

BSplines.c

```
/**
  * BSplines.c
  *
  * @author: Ryan Acosta
  *
  * 02.02.2005
  * This program is intended to compute a B-Spline interpolating surface
  * for a given data set.
  *
  * 02.03.2005
  * At the present this program computes M Q Mt (Mt means M transposed)
  * but I still need to figure out how to incorporate the variables
  * into the patch equation.
  **/

#include <stdio.h>

void ReadDataPoints(char* filename, double*** DosePoints, int *m, int *n);
void PrintDataPoints(double** DosePoints, int m, int n);
void BSpline(double** DosePoints, int m, int n);

double** Transpose(double** A, int m, int n);
double** MatMult(double** A, double** B, int m, int n);
void ScalarMult(double** A, double k, int m, int n);

int main(int argc, char* argv[])
{
```

```
    double** Data;
    int m, n;
    if(argc != 2)
    {
        printf("Usage: %s DataPoints\n",argv[0]);
    }

    ReadDataPoints(argv[1], &Data, &m, &n);

//  PrintDataPoints(Data, m, n);

    BSpline(Data, m, n);

    return 0;
}

void ReadDataPoints(char* filename, double*** DosePoints, int* m, int *n)
{
    FILE *fptr;
    int i, j;

    fptr = fopen(filename, "r");
    if(fptr == NULL)
    {
        printf("Error opening file %s\n", filename);
        exit(1);
    }

    fscanf(fptr, "%d %d\n", m, n);

    (*DosePoints) = (double**)malloc((*m)*sizeof(double*));
    for(i = 0; i < *m; i++)
        (*DosePoints)[i] = (double*)malloc((*n)*sizeof(double));

    for(i = 0; i < (*m); i++)
    {
        for(j = 0; j < (*n); j++)
        {
            fscanf(fptr, "%lf", &(*DosePoints)[i][j]);
        }
    }

    fclose(fptr);
}

void PrintDataPoints(double** DosePoints, int m, int n)
```

```
{
    int i, j;

    for(i = 0; i < m; i++)
    {
        for(j = 0; j < n; j++)
            printf("%0.2lf ", DosePoints[i][j]);
        printf("\n");
    }
    printf("\n");
}

void BSpline(double** DosePoints, int m, int n)
{
    int i, j;
    int r, s;
    double** Q;
    double** M;
    double** MQM;
    FILE* fptr;

    Q = (double**)malloc(4 * sizeof(double*));
    M = (double**)malloc(4 * sizeof(double*));
    MQM = (double**)malloc(4 * sizeof(double*));
    for(i = 0; i < 4; i++)
    {
        Q[i] = (double*)malloc(4 * sizeof(double));
        M[i] = (double*)malloc(4 * sizeof(double));
        MQM[i] = (double*)malloc(4 * sizeof(double));
    }

    // Initialize M
    M[0][0] = -1; M[0][1] =  3; M[0][2] = -3; M[0][3] = 1;
    M[1][0] =  3; M[1][1] = -6; M[1][2] =  3; M[1][3] = 0;
    M[2][0] = -3; M[2][1] =  0; M[2][2] =  3; M[2][3] = 0;
    M[3][0] =  1; M[3][1] =  4; M[3][2] =  1; M[3][3] = 0;

    fptr = fopen("Patches.m", "w");

    fprintf(fptr, "function [Patches] = Patches()\n\n");
    fprintf(fptr, "Patches = [\n");

    for(r = 0; r < m-1; r++)
    {
        for(s = 0; s < n-1; s++)
        {
```

```
for(i = 0; i < 4; i++)
{
    for(j = 0; j < 4; j++)
    {
        if(r-1+i == -1)
        {
            if(s-1+j == -1)
                Q[i][j] = DosePoints[0][0];
            else if(s-1+j == n)
                Q[i][j] = DosePoints[0][n-1];
            else
                Q[i][j] = DosePoints[0][s-1+j];
        }
        else if(r-1+i == m)
        {
            if(s-1+j == -1)
                Q[i][j] = DosePoints[m-1][0];
            else if(s-1+j == n)
                Q[i][j] = DosePoints[m-1][n-1];
            else
                Q[i][j] = DosePoints[m-1][s-1+j];
        }
        else if(s-1+j == -1)
        {
            if(r-1+i == -1) // Should never happen
                Q[i][j] = DosePoints[0][0];
            else if(r-1+i == m)
                Q[i][j] = DosePoints[m-1][0];
            else
                Q[i][j] = DosePoints[r-1+i][0];
        }
        else if(s-1+j == n)
        {
            if(r-1+i == -1)
                Q[i][j] = DosePoints[0][n-1];
            else if(r-1+i == m)
                Q[i][j] = DosePoints[m-1][n-1];
            else
                Q[i][j] = DosePoints[r-1+i][n-1];
        }
        else
            Q[i][j] = DosePoints[r-1+i][s-1+j];
    } // for i
} // for j

// Q holds the necessary Data points
```

```
            MQM = MatMult(M, MatMult(Q, Transpose(M, 4, 4), 4, 4), 4, 4);

            ScalarMult(MQM, 1/36.0, 4, 4);

            for(i = 0; i < 4; i++)
            {
                for(j = 0; j < 4; j++)
                    fprintf(fptr, "%lf ", MQM[i][j]);
                fprintf(fptr, ";\n");
            }
        }
    }

    fprintf(fptr, "];\n");
    fclose(fptr);

    for(i = 0; i < 4; i++)
    {
        free(Q[i]);
        free(M[i]);
    }
    free(Q);
    free(M);
}

double** Transpose(double** A, int m, int n)
{
    double** At;
    int i, j;

    At = (double**)malloc(m * sizeof(double*));
    for(i = 0; i < m; i++)
        At[i] = (double*)malloc(n * sizeof(double));

    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
            At[i][j] = A[j][i];

    return At;
}

double** MatMult(double** A, double** B, int m, int n)
{
    double** C;
    int i, j, k;
```

```
    double sum;

    C = (double**)malloc(m * sizeof(double*));
    for(i = 0; i < m; i++)
        C[i] = (double*)malloc(n * sizeof(double));

    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
            sum = 0;
            for(k = 0; k < m; k++)
                sum += A[i][k] * B[k][j];
            C[i][j] = sum;
        }

    return C;
}

void ScalarMult(double** A, double k, int m, int n)
{
    int i, j;

    for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
            A[i][j] *= k;
}
```

Appendix B

Vector2.h

```
#ifndef VECTOR2_H
#define VECTOR2_H

/**
  * Vector2.h
  *
  * @author: Ryan Acosta
  *
  * 02.10.2005
  * This class defines a mathematical vector in R2 and some of the
  * normal, useful operations on such vectors.
  **/

#include <iostream>

using namespace std;

class Vector2
{
    public:
        Vector2();
        Vector2(double x, double y);
        Vector2(const Vector2& _v);
        ~Vector2();

        Vector2 operator*(double k) const;
        Vector2 operator+(const Vector2& _v) const;
```

```
        Vector2& operator=(const Vector2& _v);

        const double& operator[](int i) const;
        double& operator[](int i);

        friend ostream& operator<<(ostream &os, const Vector2& _v);
    private:
        double v[2];
};

#include "Vector2.cpp"

#endif
```

Appendix C

Vector2.cpp

```
/**
  * Vector2.cpp
  *
  * @author: Ryan Acosta
  *
  * 02.10.2005
  * This is the implementation of the member functions
  * of the Vector2 class.
  **/

Vector2::Vector2()
{
}

Vector2::Vector2(double x, double y)
{
    v[0] = x, v[1] = y;
}

Vector2::Vector2(const Vector2& _v)
{
    v[0] = _v[0], v[1] = _v[1];
}

Vector2::~Vector2()
{
}
```

```
Vector2 Vector2::operator*(double k) const
{
    return Vector2(v[0]*k,v[1]*k);
}

Vector2 Vector2::operator+(const Vector2& _v) const
{
    return Vector2(v[0]+_v[0],v[1]+_v[1]);
}

Vector2& Vector2::operator=(const Vector2& _v)
{
    v[0] = _v[0], v[1] = _v[1];
    return *this;
}

const double& Vector2::operator[](int i) const
{
    if(0<=i&&1>=i)
        return v[i];
    else
        return v[0];
}

double& Vector2::operator[](int i)
{
    if(0<=i&&1>=i)
        return v[i];
    else
        return v[0];
}

ostream& operator<<(ostream &os,const Vector2& _v)
{
    os << "(" << _v[0] << ", " << _v[1] << ")";
    return os;
}
```

Appendix D

hitandrun.cpp

```
/**
  * Ryan Acosta
  * hitandrun.cpp
  *
  * This program will use the hit-and-run algorithm to find an isodose
  * curve.
  **/
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <queue>
#include <string>

#include "Vector2.h"

using namespace std;

#define PI 3.14159

/**
 * Class: patch
 *
 * This class holds the coefficients for a bicubic patch.
 **/
class patch
{
```

```
    public:
        double partialCoeffs[2][16];
        double coeffs[16];
};

void ReadDosePoints(vector <vector <double> > &dosePoints, int&m, int&n);
void ReadPatchCoeffs(vector <patch> &surface, int m, int n);
Vector2 FindMaxDosePoint(vector<vector<double> > dosePoints, int m, int n);
void PreparePartialDerivs(vector<patch> &surface);

void printHeader(string line);
double f(Vector2 X, vector<patch> surface, int m, int n);
double fpartialx(Vector2 X, vector<patch>surface, int m, int n);
double fpartialy(Vector2 X, vector<patch>surface, int m, int n);

void HitAndRun(Vector2 Max, queue<Vector2> &startingPoints,
               vector<Vector2> &PtsOnCurve,
               vector<patch> &surface, int m, int n,
               double isoLevel, int numPoints);
void NewtonsMethod(Vector2 &X, Vector2 d, Vector2 &X0,
                   double theta, vector<patch>surface, int m,int n,
                   double isoLevel,bool &cycle);
void DomainCheck(Vector2 &X, double theta, Vector2 X0, int m, int n);

void PrintCoefficients(vector<patch> surface);

int main(int argc, char* argv[])
{
    int m, n;
    vector <patch> surface;
    vector <vector <double> > dosePoints;
    vector <Vector2> PtsOnCurve;
    queue<Vector2> startingPoints;
    double isoLevel = 0.0;
    int numPoints = 200;
    // Switches
    bool printCoeffs = false;

    if(argc > 1)
    {
        if(strcmp(argv[1],"-Coeffs") == 0)
        {
            printCoeffs = true;
            isoLevel = atof(argv[2]);
            numPoints = atoi(argv[3]);
        }
```

```
    else if(strcmp(argv[1],"-help") == 0)
    {
        cout << "Usage: " << argv[0] << " <isoLevel> <numPoints>\n";
        exit(1);
    }
    else
    {
        isoLevel = atof(argv[1]);
        numPoints = atoi(argv[2]);
    }
}

// Read dose points from file
ReadDosePoints(dosePoints, m, n);
Vector2 Max = FindMaxDosePoint(dosePoints,m,n);

ReadPatchCoeffs(surface, m, n);
PreparePartialDerivs(surface);

// Show I received the correct input
if(printCoeffs)
    PrintCoefficients(surface);

printHeader("Done with Input");

HitAndRun(Max,startingPoints,PtsOnCurve,surface,m,n,isoLevel,numPoints);

printHeader("IsoDose Curve Points");

ofstream out;
out.open("IsoPoints.m");
out << "IsoX = [\n";
for(int i = 0; i < PtsOnCurve.size(); i++)
{
    out << PtsOnCurve[i][0];
    if(i != PtsOnCurve.size()-1)
        out << ";\n";
    else
        out << "\n";
}
out << "];\n\n";
out << "IsoY = [\n";
for(int i = 0; i < PtsOnCurve.size(); i++)
{
    out << PtsOnCurve[i][1];
    if(i != PtsOnCurve.size()-1)
```

```
            out << ";\n";
        else
            out << "\n";
    }
    out << "];\n\n";
    out << "IsoZ = " << isoLevel << "*ones(length(IsoX),1);\n";
    out << "scatter3(IsoX,IsoY,IsoZ)\n";
    out.close();

    return 0;
}

void ReadDosePoints(vector <vector <double> > &dosePoints, int &m, int &n)
{
    // Read in the original dose points.
    printHeader("Read Dose Points");
    ifstream in;
    in.open("Level0.dat");
    in >> m >> n;
    vector<double> tempVec;
    double temp;
    for(int i = 0; i < m; i++)
    {
        tempVec.clear();
        for(int j = 0; j < n; j++)
        {
            in >> temp;
            tempVec.push_back(temp);
        }
        dosePoints.push_back(tempVec);
    }
    in.close();
}

Vector2 FindMaxDosePoint(vector<vector<double> > dosePoints, int m, int n)
{
    // Choose the maximum point as our starting place.
    printHeader("Find Max Dose Point");
    double maxValue = dosePoints[0][0];
    int maxX = 0, maxY = 0;
    for(int i = 0; i < dosePoints.size(); i++)
    {
        for(int j = 0; j < dosePoints[i].size(); j++)
        {
            if(dosePoints[i][j] > maxValue)
            {
```

```
                maxValue = dosePoints[i][j];
                maxX = j; maxY = (m-1)-i;
            }
        }
    }
    return Vector2(maxX, maxY);
}

void printHeader(string line)
{
    if(line.length()<=40)
    {
        cout << "[===  ";
        for(int i = 0; i < (40-line.length())/2; i++)
            cout << " ";
        cout << line;
        for(int i = 0; i < (40-line.length())/2; i++)
            cout << " ";
        if(line.length()%2)
            cout << " ";
        cout << "  ===]\n";
    }
    else
    {
        cerr << "Header string cannot be over 40 characters in length\n";
    }
}

void HitAndRun(Vector2 Max, queue<Vector2> &startingPoints,
               vector<Vector2> &PtsOnCurve,
               vector<patch> &surface, int m, int n,
               double isoLevel, int numPoints)
{
    double theta;
    bool cycle = false;
    Vector2 X = Max + Vector2(0.01, 0.01);
    startingPoints.push(X);
    startingPoints.push(X+Vector2(-0.01, 0.00));
    startingPoints.push(X+Vector2(-0.02, 0.00));
    startingPoints.push(X+Vector2(-0.02,-0.01));
    startingPoints.push(X+Vector2(-0.02,-0.02));
    startingPoints.push(X+Vector2(-0.01,-0.02));
    startingPoints.push(X+Vector2( 0.00,-0.02));
    startingPoints.push(X+Vector2( 0.00,-0.01));

    srand(static_cast<unsigned>(time(NULL)));
```

```
int iterations = 0;
while(startingPoints.size() != 0 && PtsOnCurve.size() < numPoints)
{
    X = startingPoints.front();
    startingPoints.pop();
    theta = ((double)rand()/(double)RAND_MAX)*2*PI;
    switch(iterations)
    {
        case 0:
        case 1:
            theta = PI*0.25;
            break;
        case 2:
        case 3:
            theta = PI*0.5;
            break;
        case 4:
        case 5:
            theta = PI*0.75;
            break;
        case 6:
        case 7:
            theta = PI;
            break;
        case 8:
        case 9:
            theta = PI*1.25;
            break;
        case 10:
        case 11:
            theta = PI*1.5;
            break;
        case 12:
        case 13:
            theta = PI*1.75;
            break;
        case 14:
        case 15:
            theta = 0.0;
            break;
        default:
            break;
    }
//  theta = PI/2;   // For Test purposes

    Vector2 X0;
```

```
        Vector2 start = X;

        cycle = false;
        for(int iter = 0; iter < 2; iter++)
        {
            X = start;
            if(iter == 1)
                theta += PI*0.5;

            Vector2 d = Vector2(cos(theta), sin(theta));
            NewtonsMethod(X,d,X0,theta,surface,m,n,isoLevel,cycle);

            if(!(X[0] < 0 || X0[0] > (n-1) || X0[1] < 0 || X0[1] > (m-1)))
            {
                if(!cycle && abs(f(X,surface,m,n)-isoLevel)<0.00001)
                {
                    PtsOnCurve.push_back(X);

                    // Put the midpoint into startingPoints
                    startingPoints.push(Vector2(X0[0]+0.5*(X[0]-X0[0]),
                                                X0[1]+0.5*(X[1]-X0[1])));
                    startingPoints.push(Vector2(Max[0]+0.5*(X[0]-Max[0]),
                                                Max[1]+0.5*(X[1]-Max[1])));
                }
                else
                {
                    cerr << "Cycle on " << start << endl;
                }
            }
            iterations++;
        }
    }
}

void NewtonsMethod(Vector2 &X, Vector2 d, Vector2& X0,
                   double theta, vector<patch>surface, int m,int n,
                   double isoLevel,bool &cycle)
{
    X0 = X;
    if(X0[0] < 0 || X0[0] > (n-1) || X0[1] < 0 || X0[1] > (m-1))
        return;
    int ctr = 0;
    Vector2 OldX;
    do
    {
        double alpha = ((isoLevel-f(X,surface,m,n))/
```

```
                        (fpartialx(X,surface,m,n) * d[0] +
                         fpartialy(X,surface,m,n) * d[1]));
        OldX = X;
        if(ctr == 0)
        {
            // Make sure to step forward
            if(alpha < 0)
            {
                alpha *= -1;
            }
        }
        X = X +
            d * alpha;

        DomainCheck(X,theta,X0,m,n);
        ctr++;
        if(ctr >= 100)
        {
            cycle = true;
            break;
        }

    }while(abs(f(X,surface,m,n)-isoLevel) > 0.00001);
    // ==== End Newton's Method ==== //
}

void DomainCheck(Vector2 &X, double theta, Vector2 X0, int m, int n)
{
    if(X[0] > (n-1) || X[0] < 0 || X[1] > (m-1) || X[1] < 0)
    {
        Vector2 OldX = X;
        double k = 1000000;

        if(abs(cos(theta)) > 0.0001)
        {
            k = ((n-1)-X0[0])/cos(theta);
            if(abs(-X0[0]/cos(theta)) < abs(k))
                k = -X0[0]/cos(theta);
        }
        if(abs(sin(theta)) > 0.0001)
        {
            if(abs(((m-1)-X0[1])/sin(theta)) < abs(k))
                k = ((m-1)-X0[1])/sin(theta);
            if(abs(-X0[1]/sin(theta)) < abs(k))
                k = -X0[1]/sin(theta);
        }
```

```
        if(k<0 && k < -0.001)
            k+=0.001;
        else if(k > 0 && k > 0.001)
            k-=0.001;
        X = X0 + Vector2(cos(theta) * k, sin(theta)*k);
        if(!(X[0]<=(n-1)&&X[0]>=0&&X[1]<=(m-1)&&X[1]>=0))
        {
            cerr << "Error in Domain Check!\n";
            cerr << "OldX = " << OldX << " \t" << "X = " << X << endl;
            cerr << "Theta = " << theta << " \tk = " << k << endl;
            cerr << "X0 = " << X0 << endl;
            assert(X[0]<=(n-1)&&X[0]>=0&&X[1]<=(m-1)&&X[1]>=0);
        }
    }
}

double f(Vector2 X, vector<patch> surface, int m, int n)
{
    // return the value of f based on which patch (x,y) is in
    if(X[1] == 0)
        X[1] = 0.02;
    int mypatch = (n-1)*(m-1-(int)(ceil(X[1]))) + (int)X[0];

    if((int)X[0] == (n-1))
        mypatch--;

    if(mypatch > surface.size()-1 || mypatch < 0)
    {
        cerr << "Bad mypatch value: " << mypatch << endl;
        cerr << "X = " << X << endl;
        exit(1);
    }

    double x = X[0] - (int)X[0];
    if((int)X[0] == (n-1))
        x = 1.0;
    double y = X[1] - (int)X[1];
    if((int)X[1] == (m-1))
        y = 1.0;

    return surface[mypatch].coeffs[0]*x*x*x*y*y*y +
           surface[mypatch].coeffs[1]*x*x*x*y*y +
           surface[mypatch].coeffs[2]*x*x*x*y +
           surface[mypatch].coeffs[3]*x*x*x +
           surface[mypatch].coeffs[4]*x*x*y*y*y +
           surface[mypatch].coeffs[5]*x*x*y*y +
```

```
            surface[mypatch].coeffs[6]*x*x*y +
            surface[mypatch].coeffs[7]*x*x +
            surface[mypatch].coeffs[8]*x*y*y*y +
            surface[mypatch].coeffs[9]*x*y*y +
            surface[mypatch].coeffs[10]*x*y +
            surface[mypatch].coeffs[11]*x +
            surface[mypatch].coeffs[12]*y*y*y +
            surface[mypatch].coeffs[13]*y*y +
            surface[mypatch].coeffs[14]*y +
            surface[mypatch].coeffs[15];
}

double fpartialx(Vector2 X, vector<patch> surface, int m, int n)
{
    // return the value of f based on which patch (x,y) is in
    if(X[1] == 0)
        X[1] = 0.02;
    int mypatch = (n-1)*(m-1-(int)(ceil(X[1]))) + (int)X[0];

    if((int)X[0] == (n-1))
        mypatch--;

    double x = X[0] - (int)X[0];
    if((int)X[0] == (n-1))
        x = 1.0;
    double y = X[1] - (int)X[1];
    if((int)X[1] == (m-1))
        y = 1.0;

    return surface[mypatch].partialCoeffs[0][0]*x*x*x*y*y*y +
           surface[mypatch].partialCoeffs[0][1]*x*x*x*y*y +
           surface[mypatch].partialCoeffs[0][2]*x*x*x*y +
           surface[mypatch].partialCoeffs[0][3]*x*x*x +
           surface[mypatch].partialCoeffs[0][4]*x*x*y*y*y +
           surface[mypatch].partialCoeffs[0][5]*x*x*y*y +
           surface[mypatch].partialCoeffs[0][6]*x*x*y +
           surface[mypatch].partialCoeffs[0][7]*x*x +
           surface[mypatch].partialCoeffs[0][8]*x*y*y*y +
           surface[mypatch].partialCoeffs[0][9]*x*y*y +
           surface[mypatch].partialCoeffs[0][10]*x*y +
           surface[mypatch].partialCoeffs[0][11]*x +
           surface[mypatch].partialCoeffs[0][12]*y*y*y +
           surface[mypatch].partialCoeffs[0][13]*y*y +
           surface[mypatch].partialCoeffs[0][14]*y +
           surface[mypatch].partialCoeffs[0][15];
}
```

```
double fpartialy(Vector2 X, vector<patch> surface, int m, int n)
{
    // return the value of f based on which patch (x,y) is in
    if(X[1] == 0)
        X[1] = 0.02;
    int mypatch = (n-1)*(m-1-(int)(ceil(X[1]))) + (int)X[0];

    if((int)X[0] == (n-1))
        mypatch--;

    double x = X[0] - (int)X[0];
    if((int)X[0] == (n-1))
        x = 1.0;
    double y = X[1] - (int)X[1];
    if((int)X[1] == (m-1))
        y = 1.0;

    return surface[mypatch].partialCoeffs[1][0]*x*x*x*y*y*y +
           surface[mypatch].partialCoeffs[1][1]*x*x*x*y*y +
           surface[mypatch].partialCoeffs[1][2]*x*x*x*y +
           surface[mypatch].partialCoeffs[1][3]*x*x*x +
           surface[mypatch].partialCoeffs[1][4]*x*x*y*y*y +
           surface[mypatch].partialCoeffs[1][5]*x*x*y*y +
           surface[mypatch].partialCoeffs[1][6]*x*x*y +
           surface[mypatch].partialCoeffs[1][7]*x*x +
           surface[mypatch].partialCoeffs[1][8]*x*y*y*y +
           surface[mypatch].partialCoeffs[1][9]*x*y*y +
           surface[mypatch].partialCoeffs[1][10]*x*y +
           surface[mypatch].partialCoeffs[1][11]*x +
           surface[mypatch].partialCoeffs[1][12]*y*y*y +
           surface[mypatch].partialCoeffs[1][13]*y*y +
           surface[mypatch].partialCoeffs[1][14]*y +
           surface[mypatch].partialCoeffs[1][15];
}

void ReadPatchCoeffs(vector <patch> &surface, int m, int n)
{
    printHeader("Read in Patch Coefficients");
    // Read in coefficients for all the patches
    char* dummy;
    char dummyChar;
    ifstream in;
    in.open("Patches.m");
    patch tempPatch;
    dummy = (char*)malloc(80 * sizeof(char));
```

```
in.getline(dummy, 80);
in.getline(dummy, 80);
in.getline(dummy, 80);
for(int i = 0; i < m-1; i++)
{
    for(int j = 0; j < n-1; j++)
    {
        // Read in the 4x4 matrix of coefficients
        // Store in row major order in a double array
        for(int k = 0; k < 16; k++)
        {
            in >> tempPatch.coeffs[k];
            if((k+1)%4 == 0)
                in >> dummyChar;
        }
        surface.push_back(tempPatch);
    }
}
free(dummy);
in.close();

// 3.21.2005
// The coefficient matrices don't behave as they should
// Perform the necessary changes to fix the matrix
// To get these numbers I replaced y with x and replaced
// x with (1-y)

for(int i = 0; i < m-1; i++)
{
    for(int j = 0; j < n-1; j++)
    {
        for(int k = 0; k < 16; k++)
        {
            tempPatch.coeffs[k] = surface[i*(n-1)+j].coeffs[k];
        }

        surface[i*(n-1)+j].coeffs[0] =   -tempPatch.coeffs[0];
        surface[i*(n-1)+j].coeffs[1] =  3*tempPatch.coeffs[0] +
                                          tempPatch.coeffs[4];
        surface[i*(n-1)+j].coeffs[2] = -3*tempPatch.coeffs[0] -
                                        2*tempPatch.coeffs[4] -
                                          tempPatch.coeffs[8];
        surface[i*(n-1)+j].coeffs[3] =    tempPatch.coeffs[0] +
                                          tempPatch.coeffs[4] +
                                          tempPatch.coeffs[8] +
                                          tempPatch.coeffs[12];
```

```
            surface[i*(n-1)+j].coeffs[4] =   -tempPatch.coeffs[1];
            surface[i*(n-1)+j].coeffs[5] =  3*tempPatch.coeffs[1] +
                                              tempPatch.coeffs[5];
            surface[i*(n-1)+j].coeffs[6] = -3*tempPatch.coeffs[1] -
                                            2*tempPatch.coeffs[5] -
                                              tempPatch.coeffs[9];
            surface[i*(n-1)+j].coeffs[7] =    tempPatch.coeffs[1] +
                                              tempPatch.coeffs[5] +
                                              tempPatch.coeffs[9] +
                                              tempPatch.coeffs[13];
            surface[i*(n-1)+j].coeffs[8] =    -tempPatch.coeffs[2];
            surface[i*(n-1)+j].coeffs[9] =   3*tempPatch.coeffs[2] +
                                               tempPatch.coeffs[6];
            surface[i*(n-1)+j].coeffs[10] = -3*tempPatch.coeffs[2] -
                                             2*tempPatch.coeffs[6] -
                                               tempPatch.coeffs[10];
            surface[i*(n-1)+j].coeffs[11] =    tempPatch.coeffs[2] +
                                               tempPatch.coeffs[6] +
                                               tempPatch.coeffs[10] +
                                               tempPatch.coeffs[14];
            surface[i*(n-1)+j].coeffs[12] =   -tempPatch.coeffs[3];
            surface[i*(n-1)+j].coeffs[13] =  3*tempPatch.coeffs[3] +
                                               tempPatch.coeffs[7];
            surface[i*(n-1)+j].coeffs[14] = -3*tempPatch.coeffs[3] -
                                            2*tempPatch.coeffs[7] -
                                               tempPatch.coeffs[11];
            surface[i*(n-1)+j].coeffs[15] =    tempPatch.coeffs[3] +
                                               tempPatch.coeffs[7] +
                                               tempPatch.coeffs[11] +
                                               tempPatch.coeffs[15];
        }
    }
}

void PreparePartialDerivs(vector<patch> &surface)
{
    // Prepare the partial derivates
    for(int p = 0; p < surface.size(); p++)
    {
        for(int dir = 0; dir < 2; dir++)
        {
            if(dir == 0)
            {
              for(int i = 0; i < 4; i++)
                for(int j = 0; j < 4; j++)
                  switch(i)
```

```
                {
                  case 0:
                    surface[p].partialCoeffs[dir][4*i+j] = 0.0;
                    break;
                  case 1:
                    surface[p].partialCoeffs[dir][4*i+j] =
                                        3*surface[p].coeffs[4*(i-1)+j];
                    break;
                  case 2:
                    surface[p].partialCoeffs[dir][4*i+j] =
                                        2*surface[p].coeffs[4*(i-1)+j];
                    break;
                  case 3:
                    surface[p].partialCoeffs[dir][4*i+j] =
                                        surface[p].coeffs[4*(i-1)+j];
                    break;
                }
            }
            else
            {
              for(int i = 0; i < 4; i++)
                for(int j = 0; j < 4; j++)
                  switch(j)
                  {
                    case 0:
                      surface[p].partialCoeffs[dir][4*i+j] = 0;
                      break;
                    case 1:
                      surface[p].partialCoeffs[dir][4*i+j] =
                                        3*surface[p].coeffs[4*i+j-1];
                      break;
                    case 2:
                      surface[p].partialCoeffs[dir][4*i+j] =
                                        2*surface[p].coeffs[4*i+j-1];
                      break;
                    case 3:
                      surface[p].partialCoeffs[dir][4*i+j] =
                                        surface[p].coeffs[4*i+j-1];
                      break;
                  }
            }
        }
    }
}

void PrintCoefficients(vector<patch> surface)
```

```
{
    for(int p = 0; p < surface.size(); p++)
    {
        cout << "[~~ Patch " << setw(2) << p << " ~~]\n";
        for(int i = 0; i < 4; i++)
        {
            for(int j = 0; j < 4; j++)
            {
                if(abs(surface[p].coeffs[4*i+j]) < 0.000001)
                    cout << setw(10) << "0" << " ";
                else
                    cout << setprecision(4) << setw(10)
                         << surface[p].coeffs[4*i+j] << " ";
            }
            cout << endl;
        }
        cout << endl << "x-partial" << endl;;
        for(int i = 0; i < 4; i++)
        {
            for(int j = 0; j < 4; j++)
            {
                if(abs(surface[p].partialCoeffs[0][4*i+j]) < 0.000001)
                    cout << setw(10) << "0" << " ";
                else
                    cout << setprecision(4) << setw(10)
                         << surface[p].partialCoeffs[0][4*i+j] << " ";
            }
            cout << endl;
        }
        cout << endl << "y-partial" << endl;
        for(int i = 0; i < 4; i++)
        {
            for(int j = 0; j < 4; j++)
            {
                if(abs(surface[p].partialCoeffs[1][4*i+j]) < 0.000001)
                    cout << setw(10) << "0" << " ";
                else
                    cout << setprecision(4) << setw(10)
                         << surface[p].partialCoeffs[1][4*i+j] << " ";
            }
            cout << endl;
        }
    }
}
```