AFRL-RH-BR-TR-2007-0075

2-Dimensional B-Spline Algorithms with Applications to Ray Tracing in Media of Spatially-Varying Refractive Index

Bonnie C. McAdoo
Taufiquar R. Khan

Clemson University

C. D. Clark III
Lance J. Irvin
Isaac D. Noojin

Northrop Grumman Information Technology

Dane A. Burrows
David A. Wooddell
Robert J. Thomas
Justin J. Zohner

Human Effectiveness Directorate
Directed Energy Bioeffects Division
Optical Radiation Branch

August 2007
Interim Report for June 2007 – August 2007

DESTRUCTION NOTICE – Destroy by any method that will prevent disclosure of contents or reconstruction of this document.

Distribution Approved for Public Release; Distribution Unlimited

Air Force Research Laboratory
Human Effectiveness Directorate
Directed Energy Bioeffects Division
Optical Radiation Branch

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U. S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Human Systems Wing (HSW/PA) Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RH-BR-TR-2007-0075 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

__//SIGNED//___________________________________
ALAN J. RICE, LT
Contract Monitor

__//SIGNED//___________________________________
GARRETT D. POLHAMUS, PhD, DR-IV
Chief, Directed Energy Bioeffects Division

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
August 2007	Technical Report	June 2007 – August 2007

4. TITLE AND SUBTITLE

2-DIMENSIONAL B-SPLINE ALGORITHMS WITH APPLICATIONS TO RAY TRACING IN MEDIA OF SPATIALLY-VARYING REFRACTIVE INDEX

5a. CONTRACT NUMBER F41624-02-D-7003

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER 62202F

5d. PROJECT NUMBER 7757

5e. TASK NUMBER B2

5f. WORK UNIT NUMBER 26

6. AUTHOR(S)

*McAdoo, Bonnie C.; *Khan, Taufiquar R.; ^Thomas, Robert J.; ^Zohner, Justin J.; ^Burrows, Dane A.; #Clark, Clifton D.; +Irvin, Lance J.; ^Wooddell, David A., Lt; #Noojin, Isaac

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

^Air Force Research Laboratory
Human Effectiveness Directorate
Directed Energy Bioeffects Division
Optical Radiation Branch
2624 Louis Bauer Dr.
Brooks City-Base, TX 78235-5128

#Northrop Grumman-IT
4241 Woodcock Dr.
Ste B-100
San Antonio, TX 78228

*Dept of Mathematical Sciences
Clemson University
Clemson, SC 29634-0975

+Ft Hayes University
Ft Hayes, KS

8. PERFORMING ORGANIZATION REPORT NUMBER

AFRL/RHDO

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Materiel Command
Air Force Research Laboratory
Human Effectiveness Directorate
Directed Energy Bioeffects Division
Optical Radiation Branch
2624 Louis Bauer Dr.
Brooks City-Base, TX 78235-5128

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-RH-BR-TR-2007-0075

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

Black and white

14. ABSTRACT

Presented are the ray-tracing methodologies for a modified Monte-Carlo approach to the solution of the radiative transport equation which has the unique feature of incorporating refractive index gradients within a multi-layer biological tissue model. In the approach, photon trajectories are computed using a solution of the Eikonal equation (ray-tracing methods) rather than linear trajectories. The method can be applied to the specific problem of incorporating thermal lensing and other non-linear effects in turbid media (biological tissues) by coupling the radiative transport solution into heat transfer and damage models.

15. SUBJECT TERMS:

B-Splines, Ray-Tracing, Eikonal Equation, Scattering, Monte-Carlo, Laser, Tissues, Damage

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Robert J. Thomas
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	SAR	67	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Distribution A. Approved for Public Release; Distribution Unlimited

This page intentionally left blank

Distribution A. Approved for Public Release; Distribution Unlimited

Contents

Table of Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Definitions 3
- 2.1 Splines . . . 3
- 2.2 B-splines . . . 4

3 Approximation in one dimension 5
- 3.1 The objective function in one dimension . . . 5
- 3.2 Necessary and sufficient conditions for minimization . . . 6

4 Approximation in two dimensions 9
- 4.1 The objective function in two dimensions . . . 9
- 4.2 Solving the system . . . 12

5 Tracing rays using Runge-Kutta methods 15
- 5.1 The ray trace equation . . . 15
- 5.2 Runge-Kutta methods . . . 15
  - 5.2.1 General Runge-Kutta Methods . . . 16
  - 5.2.2 Runge-Kutta-Fehlberg methods . . . 16
  - 5.2.3 Implementation . . . 17
  - 5.2.4 Stability . . . 17

6 Conclusion 21

A Code 23
- A.1 C code . . . 23
  - A.1.1 VBASIS . . . 23
  - A.1.2 VBNEUMANN . . . 26
  - A.1.3 LEAST SQUARES APPROXIMATION . . . 28
  - A.1.4 VSPLINE . . . 31
  - A.1.5 2D-SLICE COEFFICIENT . . . 32
  - A.1.6 INNER PRODUCT . . . 34
  - A.1.7 WEIGHTS . . . 37
  - A.1.8 PLOT BASIS . . . 38
  - A.1.9 SIGN . . . 39

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

A.1.10 SURF VALUE 40
A.2 An example in C 42
A.2.1 VBASIS 42
A.2.2 VBNEUMANN 43
A.2.3 VSPLINE 43
A.2.4 LEAST SQUARES APPROXIMATION 44
A.2.5 SLICE COEFFICIENTS 45
A.2.6 SURF VALUE 46
A.3 MATLAB Code 47
A.3.1 VBASIS 47
A.3.2 VBNEUMANN 49
A.3.3 LEAST SQUARES APPROXIMATION 50
A.3.4 VSPLINE 51
A.3.5 2D-SLICE COEFFICIENT 52
A.3.6 INNER PRODUCT 53
A.3.7 UNIFORM WEIGHTS 55
A.3.8 PLOT BASIS 55
A.3.9 SURF VALUE 55
A.3.10 DEFINING THE REFRACTIVE INDEX 56

References **59**

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

List of Figures

4.1 Two triangulations of a rectangular grid. The four corner points are hypothetical data, and the middle point on each is the interpolation in each case. 10

5.1 Although the solution approaches 0 as $t \to \infty$, the Euler method approximation using step size $h = 1/7$ does not. This is an example of instability. 18
5.2 The region of absolute stability for RK4 and RK5 given in (5.19) and (5.20). 20

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

List of Tables

3.1 Table of Variables 6

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 1

Introduction

Numerical models of physical phenomena often employ a representation of the computational space which is discrete. For example, a differential equation may be solved on a rectangular grid of discrete points through finite difference methods. While these representations of solutions are valuable, it is often desirable to approximate the solution in the regions between solution points or ”nodes”.

For slowly varying functions, a linear interpolation of values is often employed. This approach may provide sufficient accuracy in some cases. However, simulations which require many iterations are often adversely affected by the error introduced by linear interpolation. For example, small surface faceting errors for optical surfaces, introduced early in propagation are often manifested as significant aberrations.

Our current application for a generalized spline implementation is rooted in the need to solve coupled physical problems through differing methods. For example, the motivation for this technical report was the need to combine a heat transfer model with two different laser beam propagation codes. In both cases the temperature distribution in the material induces a change in the optical properties, such as refractive index. The coupled code for the laser beam propagation simulation require values of the optical properties on either a differing grid spacing (for fast Hankel transform methods), or at arbitrary points within the material (for ray tracing methods).

We have proposed a modified Monte-Carlo approach to the solution of the radiative transport equation which has the unique feature of incorporating refractive index gradients within a multi-layer biological tissue model. In the approach, photon trajectories are computed using a solution of the Eikonal equation (ray-tracing methods) rather than linear trajectories. The method can be applied to the specific problem of incorporating thermal lensing and other non-linear effects in turbid media (biological tissues) by coupling the radiative transport solution into heat-transfer and damage models. In turn, the method can be applied in the establishment of laser exposure limits for tissue-penetrating wavelengths, as well as a number of additional applications in imaging and spectroscopy as well as vision science.

Presented here is a short summary of the theory and methods for the implementation of a spline interpolation suitable for accurate one and two-dimensional functional distributions. Included is source code for both the MatLab and C++ programming languages. Example data are presented, along with a short stability and error analysis for problems of recent interest to our research.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 2

Definitions

2.1 Splines

Suppose $[a,b]$ is a closed, finite interval. We select N distinct points, or knots, $x_1,\dots,x_N$ strictly between a and b. That is, we have

$$a = x_0 < x_1 < x_2 < \dots < x_N < x_{N+1} = b. \tag{2.1}$$

The intervals between each consecutive pair, x_i and x_{i+1} for $i = 0\dots N$ form a partition of $[a,b]$, as long as we carefully define

$$I_i = [x_i, x_{i+1})$$

for all $i = 0\dots N-1$, and

$$I_N = [x_N, x_{N+1}]$$

so that the right endpoint b is in the partition, which we call Δ.
Let $\mathcal{P}_m$ be the space of polynomials of order m. Recall that

$$\mathcal{P}_m = \left\{ p(x) = \sum_{i=1}^{m} c_i x^{i-1} \quad c_i \in \mathbb{R} \right\}. \tag{2.2}$$

$\mathcal{P}_3$, for example, is the set of quadratic functions.
Finally, let $\bar{m}$ be an integer, $-1 \le \bar{m} \le m-2$. This will determine the smoothness we would like at our knots. If $\bar{m} = -1$, then we do not require the spline, or any of its derivatives, to be continuous at the knots.
$\mathcal{S}(\mathcal{P}_m;\bar{m};\Delta)$ denotes the space of polynomial splines of order m with knots $x_1,\dots,x_N$. We say that s is in this space if there exist polynomials $s_0,\dots,s_N \in \mathcal{P}_m$ such that

1. $s(x) = s_i(x)\ \forall x \in I_i, i = 0,1,\dots,N$, and
2. $D^j s_{i-1}(x_i) = D^j s_i(x_i)$ for $j = 0,1,\dots,\bar{m}$ and for $i = 1,\dots,N$.

Theorem 2.1.1. *[4] $\mathcal{S}(\mathcal{P}_m;\bar{m};\Delta)$ is a linear (vector) space of dimension $m + (m-\bar{m}-1)N$.*

Definition 2.1.2. *[4]Let the partition of $[a,b]$ $\{x_i\}_{i=1}^N$ and $\bar{m}$ be given. $\tilde{\Delta} = \{y_i\}_{i=1}^{2m+(m-\bar{m}-1)N}$ is the* extended partition associated with *$\mathcal{S}(\mathcal{P}_m;\bar{m};\Delta)$ provided the following conditions hold:*

1. $y_1 \le y_2 \le \cdots \le y_{2m+(m-\bar{m}-1)N}$
2. $y_1 \le \cdots \le y_m \le a$
3. $b \le y_{m+(m-\bar{m}-1)N+1} \le \cdots \le y_{2m+(m-\bar{m}-1)N}$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

4. $y_{m+1} \leq \cdots \leq y_{m+(m-\bar{m}-1)N} = \overbrace{x_1, \ldots, x_1}^{m-\bar{m}-1}, \ldots, \overbrace{x_N, \ldots, x_N}^{m-\bar{m}-1}$

2.2 B-splines

For numerical applications, a local, symmetric basis is more suitable than, for example, a one-sided basis [4]. We therefore choose to use a B-spline. Given integers i and $m > 0$, for all real x the *mth-order B-spline associated with knots y_i through y_{i+m}* is given by

$$Q_i^m(x) = \begin{cases} (-1)^m [y_i, \ldots, y_{i+m}](x-y)^{m-1} & \text{if } y_i < y_{i+m} \text{ and } x > y \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where [ ] denotes the divided difference. For a function f, the divided difference $[t_1, \ldots, t_{r+1}]f$ is given by

$$[t_1, \ldots, t_{r+1}]f = \frac{\det \begin{pmatrix} 1 & t_1 & \cdots & t_1^{r-1} & f(t_1) \\ 1 & t_2 & \cdots & t_2^{r-1} & f(t_2) \\ 1 & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_{r+1} & \cdots & t_{r+1}^{r-1} & f(t_{r+1}) \end{pmatrix}}{\det \begin{pmatrix} 1 & t_1 & \cdots & t_1^{r-1} & t_1^r \\ 1 & t_2 & \cdots & t_2^{r-1} & t_2^r \\ 1 & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_{r+1} & \cdots & t_{r+1}^{r-1} & t_{r+1}^r \end{pmatrix}} \tag{2.4}$$

Depending on spacing of the knots, however, B-splines can be very large or very small which can yield unfavorable results computationally. The *normalized B-spline* associated with knots $y_i, \ldots, y_{i+m}$ is given by

$$N_i^m(x) = (y_{i+m} - y_i) Q_i^m(x). \tag{2.5}$$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 3

Approximation in one dimension

3.1 The objective function in one dimension

Once we have determined the spline basis we will use, we need to determine the appropriate coefficients to approximate the function n at arbitrary points in the (r, z) plane.
For the time being, we will consider our basis for r. Let the ith basis function be given by ϕ_i, and let the number of basis elements (the dimension) be $K = m + (m - \bar{m} - 1)N$. Then our estimate $\tilde{n}$ for the function n at an arbitrary point r is given by

$$\tilde{n}(r) = \sum_{i=1}^{K} c_i \phi_i(r) \tag{3.1}$$

and $\mathbf{c} = (c_1, c_2, \ldots, c_K)$ are coefficients to be determined.
We would like to choose these coefficients so that they give us the best possible approximation to the true values of the function. Our idea of what kind of approximation is "best" depends on the particular application. However, certainly we would like the approximation to come close to the known values n at the gridpoints. Secondly, we would like the approximation to remain close to the data we have on the grid, without oscillating wildly between gridpoints.
We can state these goals mathematically as follows. We seek a vector $\mathbf{c_0}$ such that

$$J(\mathbf{c_0}) = \min_{\mathbf{c}} J \tag{3.2}$$

where

$$J(\mathbf{c}) = \sum_{j=1}^{N} \left|\tilde{n}(r_j) - n(r_j)\right|^2 + \alpha_1 \int_{r_0}^{r_f} |\tilde{n}(r) - l(r)|^2 \, dr + \alpha_2 \int_{r_0}^{r_f} \left|\tilde{n}'(r) - l'(r)\right|^2 \, dr. \tag{3.3}$$

Undefined variables are given in the following table.
We do not put a weight on the first part of the expression, $\sum_{j=1}^{N} \left|\tilde{n}(r_j) - n(r_j)\right|^2$ since we assume that the weight on this part of the expression is non-zero, and normalize it to one. Thus α_1 and α_2 are decided with respect to the importance of the first term.
Also, we choose $\alpha_1 \geq \alpha_2$, since the second and third terms are a linear combination of the squared L_2 norm and squared Sobolev norms. That is,

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Table 3.1: Table of Variables

N	the number of points at which we have data
K	the dimension of the spline; the number of basis functions
m	the order of the spline
$\bar{m}$	the smoothness of the spline at the knots
$n(r_j)$	the data at r_j
$l(r)$	linear interpolation between the known gridpoints
α_1	a weight that determines the importance of our approximation $\tilde{n}$ remaining close to $l(r)$
α_2	a weight that determines the importance of our approximation's derivative $\tilde{n}'$ remaining close to $l'(r)$

$$\begin{aligned} \alpha_1 \int_{r_0}^{r_f} |\tilde{n}(r) - l(r)|^2 \; dr + \alpha_2 \int_{r_0}^{r_f} \left|\tilde{n}'(r) - l'(r)\right|^2 \; dr &= a_1 \|\tilde{n}(r) - l(r)\|_{L^2}^2 + a_2 \|\tilde{n}(r) - l(r)\|_{H^1}^2 \\ &= a_1 \int_{r_0}^{r_f} |\tilde{n}(r) - l(r)|^2 \; dr + a_2 \Bigg(\int_{r_0}^{r_f} |\tilde{n}(r) \\ &\quad -l(r)|^2 \; dr + \int_{r_0}^{r_f} \left|\tilde{n}'(r) - l'(r)\right|^2 \Bigg) \\ &= (a_1 + a_2) \int_{r_0}^{r_f} |\tilde{n}(r) - l(r)|^2 \; dr \\ &\quad + a_2 \int_{r_0}^{r_f} \left|\tilde{n}'(r) - l'(r)\right|^2 \; dr. \end{aligned} \tag{3.4}$$

Thus we have

$$\begin{aligned} \alpha_1 &= a_1 + a_2 \\ &= a_1 + \alpha_2 \end{aligned} \tag{3.5}$$

$$\Rightarrow \alpha_1 - \alpha_2 = a_1 \geq 0. \tag{3.6}$$

3.2 Necessary and sufficient conditions for minimization

The necessary condition for J being minimized with respect to $\mathbf{c}$ is that

$$\frac{\partial J}{\partial c_k} = 0 \text{ for } k = 1 \ldots K. \tag{3.7}$$

That is, for $k = 1 \ldots K$,

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

$$
\begin{aligned}
\sum_{j=1}^{N}\left(\tilde{n}(r_j)-n(r_j)\right)\phi_k(r_j)+\alpha_1\int_{r_0}^{r_f}\left(\tilde{n}(r)-l(r)\right)\phi_k(r)\,dr & \\
+\alpha_2\int_{r_0}^{r_f}\left(\tilde{n}'(r)-l'(r)\right)\phi_k'(r)\,dr &= 0\\
\sum_{j=1}^{N}\left(\sum_{i=1}^{K}c_i\phi_i(r_j)-n(r_j)\right)\phi_k(r_j)+\alpha_1\int_{r_0}^{r_f}\left(\sum_{i=1}^{K}c_i\phi_i(r)-l(r)\right)\phi_k(r)\,dr & \\
+\alpha_2\int_{r_0}^{r_f}\left(\sum_{i=1}^{K}c_i\phi_i'(r)-l'(r)\right)\phi_k'(r)\,dr &= 0\\
\sum_{i=1}^{K}c_i\left(\sum_{j=1}^{N}\phi_i(r_j)\phi_k(r_j)\right)-\sum_{j=1}^{N}n(r_j)\phi_k(r_j)+\alpha_1\sum_{i=1}^{K}c_i\int_{r_0}^{r_f}\phi_i(r)\phi_k(r)\,dr & \\
-\alpha_1\int_{r_0}^{r_f}l(r)\phi_k(r)\,dr+\alpha_2\sum_{i=1}^{K}c_i\int_{r_0}^{r_f}\phi_i'(r)\phi_k'(r)\,dr-\alpha_2\int_{r_0}^{r_f}l'(r)\phi_k'(r)\,dr &= 0\\
\sum_{i=1}^{K}c_i\left(\sum_{j=1}^{N}\phi_i(r_j)\phi_k(r_j)+\alpha_1\int_{r_0}^{r_f}\phi_i(r)\phi_k(r)\,dr+\alpha_2\int_{r_0}^{r_f}\phi_i'(r)\phi_k'(r)\,dr\right) & \\
-\sum_{j=1}^{N}n(r_j)\phi_k(r_j)-\alpha_1\int_{r_0}^{r_f}l(r)\phi_k(r)\,dr-\alpha_2\int_{r_0}^{r_f}l'(r)\phi_k'(r)\,dr &= 0.
\end{aligned}
\tag{3.8}
$$

Now, expanding this for all k into matrix form,

$$
(\mathbf{A_1}+\alpha_1\mathbf{A_2}+\alpha_2\mathbf{A_3})\mathbf{c}=(\mathbf{b_1}+\alpha_1\mathbf{b_2}+\alpha_2\mathbf{b_3}) \tag{3.9}
$$

where

$$
\mathbf{A_1}=\begin{pmatrix}
\sum_{j=1}^{N}\phi_1(r_j)\phi_1(r_j) & \sum_{j=1}^{N}\phi_2(r_j)\phi_1(r_j) & \cdots & \sum_{j=1}^{N}\phi_K(r_j)\phi_1(r_j)\\
\sum_{j=1}^{N}\phi_1(r_j)\phi_2(r_j) & \sum_{j=1}^{N}\phi_2(r_j)\phi_2(r_j) & \cdots & \sum_{j=1}^{N}\phi_K(r_j)\phi_2(r_j)\\
\vdots & \ddots & \cdots & \vdots\\
\sum_{j=1}^{N}\phi_1(r_j)\phi_K(r_j) & \cdots & \cdots & \sum_{j=1}^{N}\phi_K(r_j)\phi_K(r_j)
\end{pmatrix} \tag{3.10}
$$

$$
\mathbf{A_2}=\begin{pmatrix}
\int_{r_0}^{r_f}\phi_1(r)\phi_1(r)\,dr & \int_{r_0}^{r_f}\phi_2(r)\phi_1(r)\,dr & \cdots & \int_{r_0}^{r_f}\phi_K(r)\phi_1(r)\,dr\\
\int_{r_0}^{r_f}\phi_1(r)\phi_2(r)\,dr & \int_{r_0}^{r_f}\phi_2(r)\phi_2(r)\,dr & \cdots & \int_{r_0}^{r_f}\phi_K(r)\phi_2(r)\,dr\\
\vdots & \ddots & \cdots & \vdots\\
\int_{r_0}^{r_f}\phi_1(r)\phi_K(r)\,dr & \cdots & \cdots & \int_{r_0}^{r_f}\phi_K(r)\phi_K(r)\,dr
\end{pmatrix} \tag{3.11}
$$

$$
\mathbf{A_3}=\begin{pmatrix}
\int_{r_0}^{r_f}\phi_1'(r)\phi_1'(r)\,dr & \int_{r_0}^{r_f}\phi_2'(r)\phi_1'(r)\,dr & \cdots & \int_{r_0}^{r_f}\phi_K'(r)\phi_1'(r)\,dr\\
\int_{r_0}^{r_f}\phi_1'(r)\phi_2'(r)\,dr & \int_{r_0}^{r_f}\phi_2'(r)\phi_2'(r)\,dr & \cdots & \int_{r_0}^{r_f}\phi_K'(r)\phi_2'(r)\,dr\\
\vdots & \ddots & \cdots & \vdots\\
\int_{r_0}^{r_f}\phi_1'(r)\phi_K'(r)\,dr & \cdots & \cdots & \int_{r_0}^{r_f}\phi_K'(r)\phi_K'(r)\,dr
\end{pmatrix} \tag{3.12}
$$

$$
\mathbf{b_1}=\begin{pmatrix}
\sum_{j=1}^{N}n(r_j)\phi_1(r_j)\\
\sum_{j=1}^{N}n(r_j)\phi_2(r_j)\\
\vdots\\
\sum_{j=1}^{N}n(r_j)\phi_K(r_j)
\end{pmatrix}
\qquad
\mathbf{b_2}=\begin{pmatrix}
\int_{r_0}^{r_f}l(r)\phi_1(r)dr\\
\int_{r_0}^{r_f}l(r)\phi_2(r)dr\\
\vdots\\
\int_{r_0}^{r_f}l(r)\phi_k(r)dr
\end{pmatrix} \tag{3.13}
$$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

$$\mathbf{b_3} = \begin{pmatrix} \int_{r_0}^{r_f} l'(r)\phi_1'(r)dr \\ \int_{r_0}^{r_f} l'(r)\phi_2'(r)dr \\ \vdots \\ \int_{r_0}^{r_f} l'(r)\phi_k'(r)dr \end{pmatrix}.$$

Note that all of these matrices are known or can be calculated. Also note that the top three are symmetric, and are generally banded since B-splines are local in nature.

The sufficient condition is that J must be convex (concave up) with respect to $\mathbf{c}$. We know, then, that the Hessian must be positive semi-definite. The Hessian is symmetric, and given by

$$\mathbf{H} = \mathbf{A_1} + \mathbf{A_2} + \mathbf{A_3}. \tag{3.14}$$

Hence the sufficient condition is satisfied if the eigenvalues of $\mathbf{H}$ are non-negative [6]. The matrices $\mathbf{A_1}$, $\mathbf{A_2}$, and $\mathbf{A_3}$ are banded, as mentioned above. If we can make the stronger assumption that $\mathbf{H}$ is diagonally dominant, which is probably reasonable for B-splines, the Gershgorin Circle Theorem [7] states that all real parts of the eigenvalues will be positive. Further, since these matrices are symmetric (Hermitian) $\mathbf{H}$ is also Hermitian; the eigenvalues are therefore real.

Hence, as long as the sum of these matrices is diagonally dominant, the eigenvalues will be positive, and the sufficient condition is satisfied.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 4

Approximation in two dimensions

4.1 The objective function in two dimensions

Recall that we would like to approximate $n(r,z)$, a function on two variables when given data on a two-dimensional grid. Letting ϕ represent spline basis functions in the r direction, and ψ represent spline basis functions in the z direction, we seek coefficients c_{ij} such that

$$\tilde{n}(r,z) = \sum_{i=1}^{K}\sum_{j=1}^{P} c_{ij}\phi_i(r)\psi_j(z) \tag{4.1}$$

where $\tilde{n}$ denotes our approximation of n, K is the size of the spline basis in r and P the size in z. Let $\mathbf{C}$ be defined by

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1K} \\ c_{21} & c_{22} & \cdots & c_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ c_{P1} & \cdots & \cdots & c_{PK} \end{bmatrix}. \tag{4.2}$$

Thus our spline approximation of the function can be rewritten in matrix form:

$$\tilde{n}(r,z) = \mathbf{\Psi}^T\mathbf{C}\mathbf{\Phi} \tag{4.3}$$

where $\Psi = (\psi_1(z),\ldots,\psi_P(z))^T$ and $\Phi = (\phi_1(r),\ldots,\phi_K(r))^T$.

We run into obstacles when we move to two dimensions from one, however, regarding what to use for an objective function. Of course we would like to minimize the distance between our approximation and our data at points where we have data; this is not hard to generalize. However, it is difficult to generalize the method of keeping the interpolating curve smooth between grid points.

In one dimension, the linear interpolation between grid points is uniquely defined by simply connecting the data points with line segments. The analog to this in two dimensions would be to define a plane between sets of three points. However, we are working with a rectangular grid, and there are many ways to choose sets of three points that will alter our approximation.

In Figure 4.1, we see a possible problem with triangulating the grid in an arbitrary way. If we divide the rectangles formed by the grid into half along the diagonal we see that one interpolation may give us a very different approximation for points not on the grid than the other, though this example is extreme.

In Figure 4.1 we see two different triangulations of the same grid. Call the first t_1 and the second t_2. Using a linear combination of t_1 and t_2 is one possible strategy to keep our approximation from oscillating too much without restricting it in an artificial manner. We define the objective function J as follows.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

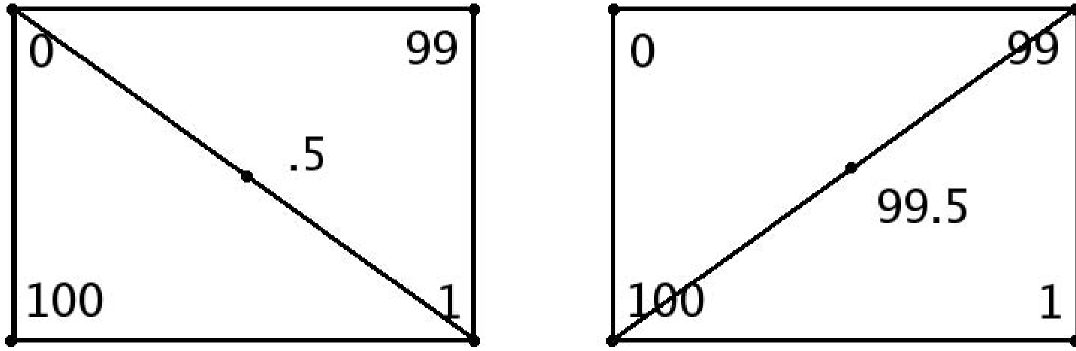

Figure 4.1: Two triangulations of a rectangular grid. The four corner points are hypothetical data, and the middle point on each is the interpolation in each case.

$$
\begin{aligned}
J(c) \quad = \quad & \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \left|\tilde{n}(r_i, z_j) - n(r_i, z_j)\right|^2 + \alpha_1 \left(\int_{r_0}^{r_f} \int_{z_0}^{z_f} |\tilde{n}(r,z) - t_1(r,z)|^2 \; dz\, dr \right. \\
& \left. + \int_{r_0}^{r_f} \int_{z_0}^{z_f} |\tilde{n}(r,z) - t_2(r,z)|^2 \; dz\, dr \right) + \alpha_2 \left(\int_{r_0}^{r_f} \int_{z_0}^{z_f} \left|\tilde{n}_z - t_{1,z}\right|^2 \; dz\, dr \right. \\
& + \int_{r_0}^{r_f} \int_{z_0}^{z_f} \left|\tilde{n}_r - t_{1,r}\right|^2 \; dz\, dr + \int_{r_0}^{r_f} \int_{z_0}^{z_f} \left|\tilde{n}_z - t_{2,z}\right|^2 \; dz\, dr \\
& + \int_{r_0}^{r_f} \int_{z_0}^{z_f} \left|\tilde{n}_r - t_{2,r}\right|^2 \; dz\, dr
\end{aligned} \tag{4.4}
$$

where t_1 represents the planar interpolation where the grid has been triangulated in one direction (see Figure 4.1 for illustration) and t_2 represents the other. Note that the definition of the Sobolev norm tells us that $\alpha_1 \geq \alpha_2$, as with the 1-D case.

Let δ, γ be integers such that $1 \leq \delta \leq N_1$, $1 \leq \gamma \leq N_2$ where N_1 is the number of grid points in the r direction and N_2 the number of grid points in the z direction. Then

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

$$
\begin{aligned}
\frac{\partial J}{\partial c_{\delta\gamma}} &= 2\sum_{i=1}^{N_1}\sum_{j=1}^{N_2}\left(\tilde{n}(r_i,z_j)-n(r_i,z_j)\right)\frac{\partial \tilde{n}}{\partial c_{\delta\gamma}} + 2\alpha_1\left(\int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}(r,z)-t_1(r,z))\frac{\partial \tilde{n}}{\partial c_{\delta\gamma}}\,dz\,dr\right. \\
&\quad \left. + \int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}(r,z)-t_2(r,z))\frac{\partial \tilde{n}}{\partial c_{\delta\gamma}}\,dz\,dr\right) + 2\alpha_2\left(\int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}_z - t_{1,z})\frac{\partial \tilde{n}_z}{\partial c_{\delta\gamma}}\,dz\,dr\right. \\
&\quad + \int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}_r - t_{1,r})\frac{\partial \tilde{n}_r}{\partial c_{\delta\gamma}}\,dz\,dr + \int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}_z - t_{2,z})\frac{\partial \tilde{n}_z}{\partial c_{\delta\gamma}}\,dz\,dr \\
&\quad \left. + \int_{r_0}^{r_f}\int_{z_0}^{z_f}(\tilde{n}_r - t_{2,r})\frac{\partial \tilde{n}_r}{\partial c_{\delta\gamma}}\,dz\,dr\right) \\
&= 2\left[\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\sum_{i=1}^{N_1}\phi_\delta(r_i)\phi_k(r_i)\sum_{j=1}^{N_2}\left(\psi_p(z_j)\psi_\gamma(z_j)\right) - \sum_{i=1}^{N_1}\phi_\delta(r_i)\sum_{j=1}^{N_2}n(r_i,z_j)\psi_\gamma(z_j)\right. \\
&\quad +\alpha_1\left(2\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi_k(r)\phi_\delta(r)\int_{z_0}^{z_f}\left(\psi_p(z)\psi_\gamma(z)\right)\,dz\,dr - \int_{r_0}^{r_f}\phi_\delta(r)\int_{z_0}^{z_f}(t_1(r,z)\right. \\
&\quad \left. +t_2(r,z))\,\psi_\gamma(z)\,dz\,dr\right) + \alpha_2\left(2\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi_k(r)\phi_\delta(r)\int_{z_0}^{z_f}\left(\psi'_p(z)\psi'_\gamma(z)\right)\,dz\,dr\right. \\
&\quad - \int_{r_0}^{r_f}\phi_\delta(r)\int_{z_0}^{z_f}(t_{1,z}+t_{2,z})\,\psi'_\gamma(z)\,dz\,dr + 2\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi'_k(r)\phi'_\delta(r) \\
&\quad \left.\left.\int_{z_0}^{z_f}\left(\psi_p(z)\psi_\gamma(z)\right)\,dz\,dr - \int_{r_0}^{r_f}\phi'_\delta(r)\int_{z_0}^{z_f}(t_{1,r}+t_{2,r})\,\psi_\gamma(z)\,dz\,dr\right)\right]
\end{aligned}
\tag{4.5}
$$

Our necessary condition is that all partials are zero. That is, after some regrouping, for every γ, δ,

$$
\begin{aligned}
&\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\sum_{i=1}^{N_1}\phi_\delta(r_i)\phi_k(r_i)\sum_{j=1}^{N_2}\psi_p(z_j)\psi_\gamma(z_j) + 2\alpha_1\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi_k(r)\phi_\delta(r) \\
&\int_{z_0}^{z_f}\psi_p(z)\psi_\gamma(z)\,dz\,dr + 2\alpha_2\left(\sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi_k(r)\phi_\delta(r)\int_{z_0}^{z_f}\psi'_p(z)\psi'_\gamma(z)\,dz\,dr\right. \\
&\left. + \sum_{k=1}^{K}\sum_{p=1}^{P}c_{kp}\int_{r_0}^{r_f}\phi'_k(r)\phi'_\delta(r)\int_{z_0}^{z_f}\psi_p(z)\psi_\gamma(z)\,dz\,dr\right) - \sum_{i=1}^{N_1}\phi_\delta(r_i)\sum_{j=1}^{N_2}n(r_i,z_j)\psi_\gamma(z_j) \\
&-\alpha_1\int_{r_0}^{r_f}\phi_\delta(r)\int_{z_0}^{z_f}T(r,z)\psi_\gamma(z)\,dz\,dr - \alpha_2\left(\int_{r_0}^{r_f}\phi'_\delta(r)\int_{z_0}^{z_f}T_r(r,z)\psi_\gamma(z)\,dz\,dr\right. \\
&\qquad\left. + \int_{r_0}^{r_f}\phi_\delta(r)\int_{z_0}^{z_f}T_z(r,z)\psi'_\gamma(z)\,dz\,dr\right) = 0
\end{aligned}
\tag{4.6}
$$

where $T(r,z) = t_1(r,z) + t_2(r,z)$.
In matrix notation, we have a system that looks like

$$
\mathbf{A_1CD_1} + 2\alpha_1\mathbf{A_2CD_2} + 2\alpha_2\left(\mathbf{A_3CD_2} + \mathbf{A_2CD_3}\right) = \mathbf{b_1} + \alpha_1\mathbf{b_2} + \alpha_2(\mathbf{b_3} + \mathbf{b_4}) \tag{4.7}
$$

where

$$
\mathbf{A_1} = \begin{pmatrix} \sum_{j=1}^{N_2}\psi_1(z_j)\psi_1(z_j) & \cdots & \sum_{j=1}^{N_2}\psi_P(z_j)\psi_1(z_j) \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{N_2}\psi_1(z_j)\psi_P(z_j) & \cdots & \sum_{j=1}^{N_2}\psi_P(z_j)\psi_P(z_j) \end{pmatrix} \tag{4.8}
$$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

$$\mathbf{D_1} = \begin{pmatrix} \sum_{i=1}^{N_1} \phi_1(r_i)\phi_1(r_i) & \cdots & \sum_{i=1}^{N_1} \phi_1(r_i)\phi_K(r_i) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{N_1} \phi_K(r_i)\phi_1(r_i) & \cdots & \sum_{i=1}^{N_1} \phi_K(r_i)\phi_K(r_i) \end{pmatrix} \tag{4.9}$$

$$\mathbf{A_2} = \begin{pmatrix} \int_{z_0}^{z_f} \psi_1(z)\psi_1(z)\,dz & \cdots & \int_{z_0}^{z_f} \psi_1(z)\psi_P(z)\,dz \\ \vdots & \ddots & \vdots \\ \int_{z_0}^{z_f} \psi_P(z)\psi_1(z)\,dz & \cdots & \int_{z_0}^{z_f} \psi_P(z)\psi_P(z)\,dz \end{pmatrix} \tag{4.10}$$

$$\mathbf{D_2} = \begin{pmatrix} \int_{r_0}^{r_f} \phi_1(r)\phi_1(r)\,dr & \cdots & \int_{r_0}^{r_f} \phi_1(r)\phi_K(r)\,dr \\ \vdots & \ddots & \vdots \\ \int_{r_0}^{r_f} \phi_K(r)\phi_1(r)\,dr & \cdots & \int_{r_0}^{r_f} \phi_K(r)\phi_K(r)\,dr \end{pmatrix} \tag{4.11}$$

$$\mathbf{A_3} = \begin{pmatrix} \int_{z_0}^{z_f} \psi'_1(z)\psi'_1(z)\,dz & \cdots & \int_{z_0}^{z_f} \psi'_1(z)\psi'_P(z)\,dz \\ \vdots & \ddots & \vdots \\ \int_{z_0}^{z_f} \psi'_P(z)\psi'_1(z)\,dz & \cdots & \int_{z_0}^{z_f} \psi'_P(z)\psi'_P(z)\,dz \end{pmatrix} \tag{4.12}$$

$$\mathbf{D_3} = \begin{pmatrix} \int_{r_0}^{r_f} \phi'_1(r)\phi'_1(r)\,dr & \cdots & \int_{r_0}^{r_f} \phi'_1(r)\phi'_K(r)\,dr \\ \vdots & \ddots & \vdots \\ \int_{r_0}^{r_f} \phi'_K(r)\phi'_1(r)\,dr & \cdots & \int_{r_0}^{r_f} \phi'_K(r)\phi'_K(r)\,dr \end{pmatrix} \tag{4.13}$$

$$\mathbf{b_1} = \begin{pmatrix} \psi_1(z_1) & \cdots & \psi_1(z_{N_2}) \\ \vdots & \ddots & \vdots \\ \psi_P(z_1) & \cdots & \psi_P(z_{N_2}) \end{pmatrix} \begin{pmatrix} n(r_1, z_1) & \cdots & n(r_{N_1}, z_1) \\ \vdots & \ddots & \vdots \\ n(r_1, z_{N_2}) & \vdots & n(r_{N_1}, z_{N_2}) \end{pmatrix} \begin{pmatrix} \phi_1(r_1) & \cdots & \phi_K(r_1) \\ \vdots & \ddots & \vdots \\ \phi_1(r_{N_1}) & \cdots & \phi_K(r_{N_1}) \end{pmatrix} \tag{4.14}$$

$$\mathbf{b_2} = \int_{r_0}^{r_f} \begin{pmatrix} \phi_1(r) \\ \vdots \\ \phi_K(r) \end{pmatrix} \int_{z_0}^{z_f} T(r,z) \begin{pmatrix} \psi_1(z), & \cdots, & \psi_P(z) \end{pmatrix} dz\,dr \tag{4.15}$$

$$\mathbf{b_3} = \int_{r_0}^{r_f} \begin{pmatrix} \phi_1(r) \\ \vdots \\ \phi_K(r) \end{pmatrix} \int_{z_0}^{z_f} T_z \begin{pmatrix} \psi'_1(z), & \cdots, & \psi'_P(z) \end{pmatrix} dz\,dr. \tag{4.16}$$

$$\mathbf{b_4} = \int_{r_0}^{r_f} \begin{pmatrix} \phi'_1(r) \\ \vdots \\ \phi'_K(r) \end{pmatrix} \int_{z_0}^{z_f} T_r \begin{pmatrix} \psi_1(z), & \cdots, & \psi_P(z) \end{pmatrix} dz\,dr. \tag{4.17}$$

The matrices $\mathbf{A_i}$, $\mathbf{D_i}$ are banded (since our bases are B-splines) and symmetric for $i = 1, 2, 3$; $\mathbf{A_i}$, $\mathbf{D_i}$ as well as $\mathbf{b}$ can be calculated. However, this system is difficult to solve in its current form. We tackle this in the next section.

4.2 Solving the system

We replace multiplication of three matrices with multiplication by a matrix and a vector to make the system easier to solve. That is, for each term $\mathbf{A_i C D_j}$,

$$\mathbf{A_i C D_j} \rightarrow \mathbf{A'_m c}$$

and for each i

$$\mathbf{b_i} \rightarrow \mathbf{b'_i}$$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

where $\mathbf{c}$ and $\mathbf{b_i'}$ are vectors of length $k \times p$ and $\mathbf{A_m'}$ is a square matrix with $k \times p$ rows and columns. Then we have the linear system

$$\left(\mathbf{A_1'} + 2\alpha_1 \mathbf{A_2'} + 2\alpha_2 \left(\mathbf{A_3'} + \mathbf{A_4'}\right)\right) \mathbf{c} = \mathbf{b_1'} + \alpha_1 \mathbf{b_2'} + \alpha_2 \left(\mathbf{b_3'} + \mathbf{b_4'}\right). \tag{4.18}$$

The modified system could be formed in different ways, but for consistency we describe one approach. We obtain $\mathbf{c}$ by stacking up the columns of $\mathbf{C}$, one on top of the other from left to right, and do the same for $\mathbf{b_i'}$. We construct the matrix $\mathbf{A_m'}$ from the matrices $\mathbf{A_i}$ and $\mathbf{D_j}$ in the following manner. For any term $\mathbf{A_i C D_j}$ if

$$\mathbf{D_j} = \begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1K} \\ d_{21} & d_{22} & \cdots & d_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ d_{K1} & \cdots & \cdots & d_{KK} \end{pmatrix} \tag{4.19}$$

then we let

$$\mathbf{A_m'} = \begin{pmatrix} d_{11}\mathbf{A_i} & d_{21}\mathbf{A_i} & \cdots & d_{K1}\mathbf{A_i} \\ d_{12}\mathbf{A_i} & d_{22}\mathbf{A_i} & \cdots & d_{K2}\mathbf{A_i} \\ \vdots & \vdots & \ddots & \vdots \\ d_{1K}\mathbf{A_i} & d_{2K}\mathbf{A_i} & \cdots & d_{KK}\mathbf{A_i} \end{pmatrix}. \tag{4.20}$$

Once we determine the values for $\mathbf{c}$ from Equation 4.18, we can reassemble $\mathbf{c}$ back into its original matrix form $\mathbf{C}$ by unstacking the columns, and use Equation 4.3 to find an approximation of the function n at any point (r, z).

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 5

Tracing rays using Runge-Kutta methods

We can apply the method outlined in the previous chapters to determine the refractive index at any point in two dimensions. Next we assume that, through these methods or analytically, we can determine the refractive index at any point in a medium. This information in combination with the ray trace equation gives us an approximation for the path a ray may travel through the medium. We discuss a method for doing this in this chapter.

5.1 The ray trace equation

Recall that the ray trace equation can be expressed [5]

$$\frac{d^2\mathbf{r}}{dt^2} = n\nabla n \tag{5.1}$$

where $\mathbf{r} = x\hat{\imath} + y\hat{\jmath} + z\hat{k}$, and $n(\mathbf{r})$ is the refractive index distribution at $\mathbf{r}$. t is defined as

$$t = \int \frac{ds}{n} \tag{5.2}$$

where ds is an arc length measure.
Introducing the optical ray vector $\mathbf{T}$ as

$$\mathbf{T} \equiv \frac{d\mathbf{r}}{dt} \tag{5.3}$$

allows us to rewrite the second-order ray trace equation as a system of first-order equations:

$$\frac{d\mathbf{r}}{dt} = \mathbf{T} \tag{5.4}$$

$$\frac{d\mathbf{T}}{dt} = n\nabla n \tag{5.5}$$

To determine the path of a ray through a medium, we now need only numerically solve this system of ordinary differential equations.

5.2 Runge-Kutta methods

The Runge-Kutta methods are a set of methods that numerically approximate the solution of an ordinary differential equation or a system of ordinary differential equations. We will describe the idea of Runge-Kutta methods, the classical Runge-Kutta method, and finally the variation we chose to use for this problem, a Runge-Kutta-Fehlberg method known as RKF45.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

5.2.1 General Runge-Kutta Methods

Consider the initial value problem

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0. \tag{5.6}$$

Since we often cannot solve such a system analytically, as is the case with the ray equation, we use the information given in the initial value problem to find a numerical solution. We know the initial value of the vector $\mathbf{y}$, and we know the derivative of $\mathbf{y}$ with respect to t. Using the derivative, we can take small steps along t starting with t_0 and estimate $\mathbf{y}$ at each step. That is, for a reasonably small value of the step size h,

$$\mathbf{y}(t + h) \approx \mathbf{y}(t) + h\,\mathbf{f}(t, \mathbf{y}(t)). \tag{5.7}$$

We can iteratively use the above expression starting with the initial value to estimate $\mathbf{y}$ at different values of t. This method is called Euler's method or the polygon method [6].
Euler's method, however is a method of order 1, which means that the magnitude of the difference between the true difference quotient and the difference quotient for the approximation is $O(h^1) = O(h)$.
Runge-Kutta methods are one-step methods that generalize Euler's method by using more points to estimate the function value at each step. Using a Runge-Kutta method instead of Euler's method allows us to achieve greater accuracy with larger step sizes. For example, the classical Runge-Kutta method [6] is of order 4. The classical method is given by

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{5.8}$$

with

$$\mathbf{k}_1 \equiv \mathbf{f}\left(t_{n-1}, \mathbf{y}_{n-1}\right) \tag{5.9}$$

$$\mathbf{k}_2 \equiv \mathbf{f}\left(t_{n-1} + \frac{h}{2}, \mathbf{y}_{n-1} + \frac{h\mathbf{k}_1}{2}\right) \tag{5.10}$$

$$\mathbf{k}_3 \equiv \mathbf{f}\left(t_{n-1} + \frac{h}{2}, \mathbf{y}_{n-1} + \frac{h\mathbf{k}_2}{2}\right) \tag{5.11}$$

$$\mathbf{k}_4 \equiv \mathbf{f}\left(t_{n-1} + h, \mathbf{y} + h\mathbf{k}_3\right). \tag{5.12}$$

Note the similarity to Simpson's rule. While this method achieves better accuracy than Euler's method, we are left with the problem of how to choose h. It would be inefficient to use trial and error with various step sizes, trying to determine if our approximation is good enough. This is the motivation for Runge-Kutta-Fehlberg methods.

5.2.2 Runge-Kutta-Fehlberg methods

Runge-Kutta-Fehlberg methods use Runge-Kutta methods to determine whether the correct step size h is being used at each step, and to choose the next step size [6]. Specifically, at step n two approximations are made: one, say y_n, using a Runge-Kutta method of order p, and the other, say z_n a Runge-Kutta method of order $p + 1$. If the difference $|y_n - z_n|$ is below a certain tolerance, one of these approximations is accepted, a step size for the next step is calculated, and the procedure is repeated for the next step.
Since at any step n we calculate two approximations, y_n of order p and z_n of order $p + 1$, we much choose which one to use. While it seems logical to take the higher-order approximation, and this is often done, the error analysis done automatically as we perform the Runge-Kutta-Fehlberg method applies to the order p approximation. It is therefore advisable to take the order p approximation y_n particularly in the case of stiff problems. [2]

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

5.2.3 Implementation

We employ a Runge-Kutta pair consisting of methods of order 4 and 5 as given in [3]. To move from y_{n-1} to y_n we must compute the following six vectors.

$$\mathbf{k}_1 = \mathbf{f}(t_{n-1}, \mathbf{y}_{n-1}) \tag{5.13}$$

$$\mathbf{k}_2 = \mathbf{f}\left(t_{n-1} + \frac{1}{4}h, \mathbf{y}_{n-1} + \frac{1}{4}\mathbf{k}_1 h\right) \tag{5.14}$$

$$\mathbf{k}_3 = \mathbf{f}\left(t_{n-1} + \frac{3}{8}h, \mathbf{y}_{n-1} + \left(\frac{3}{32}\mathbf{k}_1 + \frac{9}{32}\mathbf{k}_2\right)h\right) \tag{5.15}$$

$$\mathbf{k}_4 = \mathbf{f}\left(t_{n-1} + \frac{12}{13}h, \mathbf{y}_{n-1} + \left(\frac{1932}{2197}\mathbf{k}_1 - \frac{7200}{2197}\mathbf{k}_2 + \frac{7296}{2197}\mathbf{k}_3\right)h\right) \tag{5.16}$$

$$\mathbf{k}_5 = \mathbf{f}\left(t_{n-1} + h, \mathbf{y}_{n-1} + \left(\frac{439}{216}\mathbf{k}_1 - 8\mathbf{k}_2 + \frac{3680}{513}\mathbf{k}_3 - \frac{845}{4104}\mathbf{k}_4\right)h\right) \tag{5.17}$$

$$\mathbf{k}_6 = \mathbf{f}\left(t_{n-1} + \frac{1}{2}h, \mathbf{y}_{n-1} + \left(-\frac{8}{27}\mathbf{k}_1 + 2\mathbf{k}_2 - \frac{3544}{2565}\mathbf{k}_3 + \frac{1859}{4104}\mathbf{k}_4 - \frac{11}{40}\mathbf{k}_5\right)h\right) \tag{5.18}$$

Using these six vectors, two approximations are made. The first is of order 4:

$$\mathbf{y}_n = \mathbf{y}_{n-1} + \left(\frac{25}{216}\mathbf{k}_1 + \frac{1408}{2565}\mathbf{k}_3 + \frac{2197}{4104}\mathbf{k}_4 - \frac{1}{5}\mathbf{k}_5\right)h \tag{5.19}$$

and the second, of order 5:

$$\mathbf{z}_n = \mathbf{y}_{n-1} + \left(\frac{16}{135}\mathbf{k}_1 + \frac{6656}{12825}\mathbf{k}_3 + \frac{28561}{56430}\mathbf{k}_4 - \frac{9}{50}\mathbf{k}_5 + \frac{2}{55}\mathbf{k}_6\right)h. \tag{5.20}$$

To determine whether we should accept one of these approximations for the nth step, we test whether the difference in the two approximations is less than a predetermined error control tolerance, ϵ. That is, we accept the 5th-order approximation if

$$|\mathbf{y}_n - \mathbf{z}_n| < \epsilon. \tag{5.21}$$

The value of h for the next step is then chosen by finding a scalar q using the following expression.

$$q = \left(\frac{\epsilon h}{2\,|z_n - y_n|}\right)^{1/4} \tag{5.22}$$

We determine our next step size, say h_{new}, by multiplying q with h. That is

$$h_{new} = qh \tag{5.23}$$

Naturally we must consider the possibility of the denominator of q being 0 when $y_n = z_n$. For numerical purposes, we choose a maximum step size value, say h_{max}, so that if at any time q is very large, or if the denominator of q is 0, we take h_{max} as our step size for the next iteration. We also choose a minimum step size, h_{min}, to prevent the program from becoming too expensive to run.

5.2.4 Stability

While knowing the order of the Runge-Kutta method we use gives us an estimate in terms of h of the order of the error per step in our method, we still must keep in mind the possibility of instability in our method, leading to an approximation of the solution to the differential equation that grows further away from the solution as t increases.
To consider the possibility of stability problems, we find our region of absolute stability in the complex plane. Consider the ordinary differential equation

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

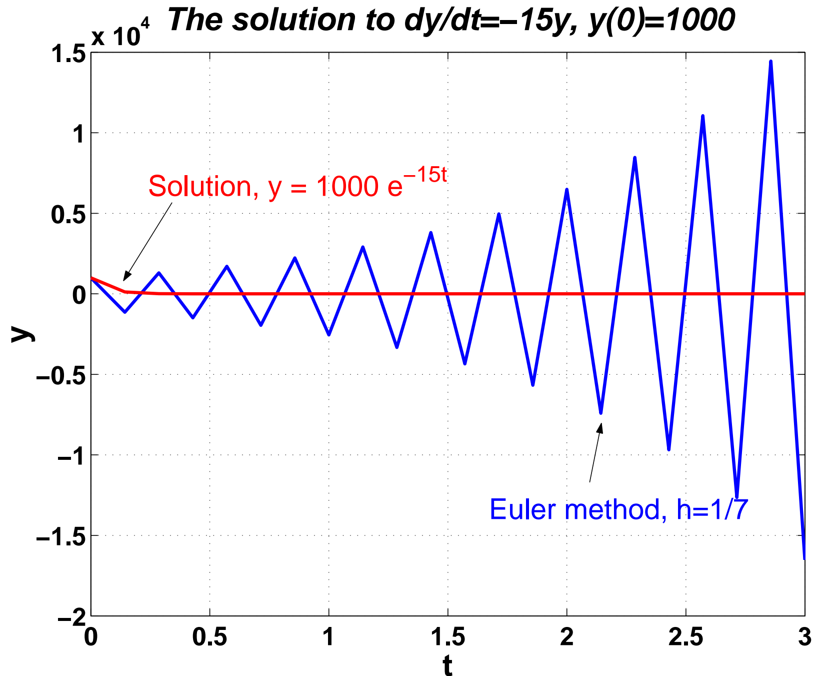

Figure 5.1: Although the solution approaches 0 as $t \to \infty$, the Euler method approximation using step size $h = 1/7$ does not. This is an example of instability.

$$y'(t) = \lambda y(t). \tag{5.24}$$

We can express a Runge-Kutta method in vector form as follows [1].

$$\begin{aligned}
Y_1 &= y_{n-1} && (5.25)\\
Y_2 &= y_{n-1} + ha_{21} f(Y_1) && (5.26)\\
&\vdots \\
Y_s &= y_{n-1} + h\left[a_{s1} f(Y_1) + a_{s2} f(Y_2) + \cdots + a_{s,s-1} f(Y_{s-1})\right], && (5.27)\\
y_n &= y_{n-1} + h\left[b_1 f(Y_1) + b_2 f(Y_2) + \cdots b_2 f(Y_s)\right] && (5.28)
\end{aligned}$$

In the fifth-order method described in (5.20), for example, $k_i = f(Y_i)$ for $i = 1, 2, \ldots, s = 6$. This can equivalently be written:

$$\mathbf{Y} = y_{n-1}\mathbf{e} + h\mathbf{A}\mathbf{f}(\mathbf{Y}). \tag{5.29}$$

where $\mathbf{Y} = [Y_1, Y_2, \ldots, Y_s]^T$, $\mathbf{e} = [1, 1, \ldots, 1]^T$ and

$$\mathbf{A} = \begin{pmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ a_{21} & 0 & \cdots & \cdots & 0 \\ a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ a_{s1} & \cdots & \cdots & a_{s,s-1} & 0 \end{pmatrix}. \tag{5.30}$$

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Also, let $\mathbf{b} = [b_1, b_2, \ldots, b_s]^T$ and let $z = \lambda h$. Using the properties of the simple ODE (1) along with (5.29) gives us

$$\mathbf{Y} = y_{n-1}\mathbf{e} + z\mathbf{A}\mathbf{Y} \tag{5.31}$$

$$y_n = y_{n-1} + z\mathbf{b}^T\mathbf{Y} \tag{5.32}$$

We would like to find the region of stability. From [1] the function $r(z)$ determining this is

$$\begin{aligned}
r(z) &= \frac{y_n}{y_{n-1}} &(5.33)\\
&= 1 + \frac{z\mathbf{b}^T\mathbf{Y}}{y_{n-1}} \text{ (using Equation 5.32)}\\
&= 1 + z\mathbf{b}^T\frac{\mathbf{Y}}{y_{n-1}}\\
&= 1 + z\mathbf{b}^T\left(\mathbf{e} + \frac{z}{y_{n-1}}\mathbf{A}\mathbf{Y}\right) \text{ (using Equation 5.31)}\\
&= 1 + z\mathbf{b}^T\left[\mathbf{e} + \frac{z}{y_{n-1}}\begin{pmatrix} 0 & \cdots & \cdots & \cdots & 0\\ a_{21} & 0 & \cdots & \cdots & 0\\ a_{31} & a_{32} & 0 & \cdots & 0\\ \vdots & \ddots & \ddots & \ddots & 0\\ a_{s1} & \cdots & \cdots & a_{s,s-1} & 0\end{pmatrix}\begin{pmatrix} y_{n-1}\\ y_{n-1} + ha_{21}f(Y_1)\\ \vdots\\ y_{n-1} + h\left[a_{s1}f(Y_1)) + \cdots + a_{s,s-1}f(Y_{s-1})\right]\end{pmatrix}\right]\\
&= 1 + z\mathbf{b}^T\left(\mathbf{I} + z\mathbf{A} + z^2\mathbf{A}^2 + \cdots + z^{s-1}\mathbf{A}^{s-1}\right)\mathbf{e} &(5.34)
\end{aligned}$$

For a Runge-Kutta method of order p, if $k \leq p$

$$\mathbf{b}^T\mathbf{A}^{k-1}\mathbf{e} = \frac{1}{k!} \tag{5.35}$$

from [1]. Thus

$$r(z) = 1 + z + \frac{z^2}{2!} + \cdots + \frac{z^p}{p!} + c_{p+1}z^{p+1} + \cdots + c_s z^s \tag{5.36}$$

where for $i = p+1, p+2, \ldots s$, the coefficient $c_i = \mathbf{b}^T\mathbf{A}^{i-1}\mathbf{e}$. Now, for our situation we have

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0\\ \frac{1}{4} & 0 & 0 & 0 & 0 & 0\\ \frac{3}{32} & \frac{9}{32} & 0 & 0 & 0 & 0\\ \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & 0 & 0 & 0\\ \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & 0 & 0\\ -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & 0\end{pmatrix} \tag{5.37}$$

and

$$\mathbf{b}^T = \left[\frac{16}{135}, 0, \frac{6656}{12825}, \frac{28561}{56430}, \frac{-9}{50}, \frac{2}{55}\right]. \tag{5.38}$$

We therefore have the polynomial

$$r(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} + \frac{z^5}{120} + .00048076923077z^6. \tag{5.39}$$

To determine the stability region, we must find out when $r(z) < 1$. Using a routine from [1] we plotted the stability region in the complex plane, the interior of which is the set of values of z for which the approximation of the test function $y'(t) = \lambda y(t)$ is stable.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

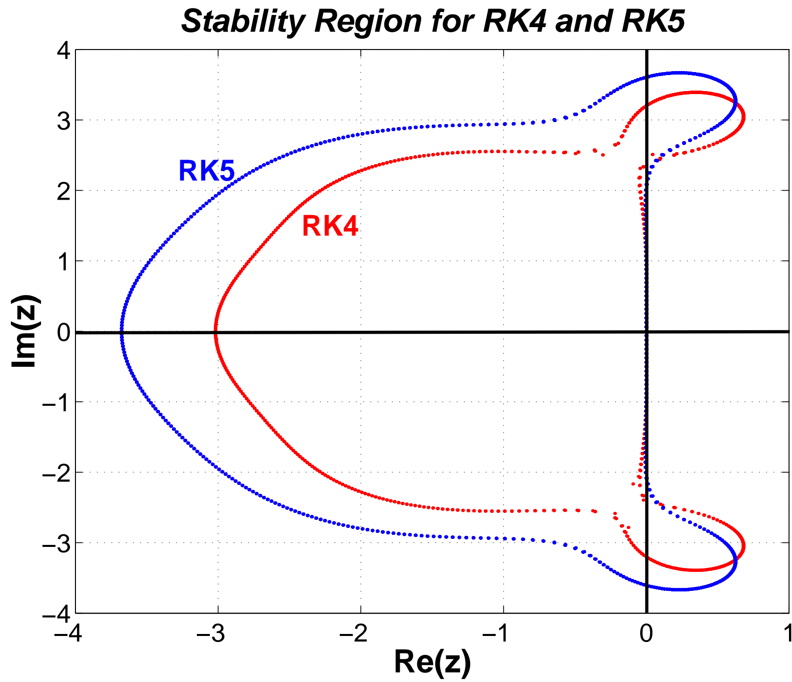

Figure 5.2: The region of absolute stability for RK4 and RK5 given in (5.19) and (5.20).

While we use a Runge-Kutta-Fehlberg method rather than a Runge-Kutta method, our approximations are in fact determined by the Runge-Kutta method of order 4 or 5 whose stability region is shown. Therefore, assuming h is always chosen such that $z = \lambda h$ is in the intersection of the two stability regions, our method is also stable.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Chapter 6

Conclusion

Given data values for a function on a two-dimensional rectangular grid, using B-splines allows us to estimate with relative computational ease reasonable values for the function at any point in the space covered by the grid. Two advantages that B-splines provide are their local nature, and the ability to combine them using tensor products. We demonstrated in this report how to make the system of tensors that results from using a B-spline basis in two dimensions into a more standard matrix system.

The next step is to extend the use of a B-spline basis into three dimensions, and then to n dimensions for any integer $n \geq 1$. For a cylindrically symmetric three-dimensional space, the two-dimensional procedure that we have outlined should be sufficient if the problem is set up to exploit the symmetry. For a more general space, extension to n uses the same idea as the problem in two dimensions, and in theory is not much more difficult. The challenge, however, lies in restructuring a system of high-rank tensors into a system which can be implemented more easily on a computer.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

Appendix A

Code

Given data on a two-dimensional grid, we implement the theory mentioned in the previous sections to approximate the function of interest. We develop the following routines to achieve this. An implementation in C is given in A.1, followed by an example. MATLAB code is given in A.3.

A.1 C code

A.1.1 VBASIS

```
#include "spline.h"

output vBasis(double x[], double grid[], int order, int derivative, int xLength,
 int gridLength)
{

/********************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *  This routine evaluates the values of the B-spline basis
 *  (or the derivatives) functions at given points. The grid
 *  points and the order of the splines are specified by the
 *  user, however, additional grid points outside of the
 *  interval [xmin, xmax] are chosen by the program to provide
 *  a complete basis.
 *
 * @Usage:
 *  output <name>=vBasis(x, grid, order, derivative, xLength, gridLength);
 *   Input:
 *    x : array of values for x on which the basis functions
 *      are to be evaluated.
 *    grid : the grid points in ascending order, all grid points must
 *      be distinct. The interval on which the spline basis functions
 *      are defined are given by:
 *         [grid[0], grid[N]].
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
*      where N is the length of the array grid.
* order : order of the spline functions.
* derivative : order of derivative needed.
* xLength : an integer value showing the length of the array x.
* gridLength : an integer value showing the length of the array grid.
*
*   Output:
*    v : an array of dimension order +1 by M, where M is the length of
*      the array x.
*    ndim : total number of basis elements, ndim=N+order-1.
*    index  : indices of the basis elements with non-zero values at a
*      point x.  index is a 2 by M array,
*         index[0][k]  -- lowest index of non-zero basis element at x[k].
*         index[1][k]  -- highest index of non-zero basis element at x[k].
* @Note:
*  Output is a structure defined in functions.h.
*******************************************************************************/

 output out;
 int i, j, k, factor, lcount=0, rcount=0, acount=0, counter=0, M=xLength,
N=gridLength, lgridLength=order, rgridLength=order;
 double localg[2*order+1], trunc[2*order+2], dgrid, n, lgrid[lgridLength],
rgrid[rgridLength], agrid[lgridLength+gridLength+rgridLength];

 out.ndim = N+order-1;
 out.index=Array2D<int>(2, M);
 out.v=Array2D<double>(order+1, M);

 //Construct the augmented grid
 //
 dgrid=grid[1]-grid[0];
 for(i=0;i<lgridLength;i++)
 {
   lgrid[i]=dgrid*i + grid[0]-order*dgrid;
   lcount++;
 }
 dgrid=grid[N-1]-grid[N-2];
 for(i=0;i<rgridLength;i++)
 {
     rgrid[i]=grid[N-1] +dgrid*(i+1);
     rcount++;
 }
 for(i=0;i<lcount;i++)
 {
   agrid[acount]=lgrid[i];
   acount++;
 }
 for(i=0;i<gridLength;i++)
 {
   agrid[acount]=grid[i];
   acount++;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
}
for(i=0;i<rcount;i++)
{
  agrid[acount]=rgrid[i];
  acount++;
}

//Main loop over points x
//
for(k=0; k<M; k++)
{
  for(j=0; j<N-1; j++)
  {
    if((sign(x[k]-grid[j])*sign(grid[j+1]-x[k]))>=0)
      break;
  }
  if(x[k]<grid[0])
  {
    j=0;
  }
  if(x[k]>grid[N-1])
  {
    j=N-2;
  }

  //Evaluate the values of the basis functions (or derivatives) at x(k)
  //
  // 1. Evaluate the values of the truncated polynomials
  //
  factor=1;
  if(derivative >0)
  {
    for(i=0; i<=derivative-1;i++)
    {
      factor=factor*(order-i);
    }
  }
  for(i=0;i<2*order+2;i++)
  {
    trunc[i]=0;
  }
  counter=0;
  for(i=j;i<=j+2*order+2;i++)
  {
    localg[counter]=agrid[i];
    counter++;
  }
  for(i=0;i<order*2+2;i++)
  {
    if(0<=(x[k]-agrid[j+i]))
      trunc[i]=x[k]-agrid[j+i];
```

```
      if(order > derivative)
        trunc[i]=factor*(pow(trunc[i], order-derivative));
      else if(order == derivative)
    trunc[i]=factor*sign(trunc[i]);
      else
      {
        cout << "The spline function is not differentiable";
        exit(-1);
      }
    }

    //2. Compute the divided differences
    //
    int l, ll;
    for(l=0;l<order+1;l++)
    {
      for(ll=0; ll<2*order+2-l; ll++)
      {
double tmp=(trunc[ll+1]-trunc[ll])/(localg[ll+l+1]-localg[ll]);
trunc[ll]=tmp;
      }
    }

    //3. Store the value in the vector v
    //
    double itrunc[2][1];
    itrunc[0][0]=j;
    out.index[0][k]=j;
    itrunc[1][0]=j+order;
    out.index[1][k]=j+order;
    for(i=0;i<order+1;i++)
    {
      out.v[i][k]=trunc[i];
    }
  }

  return out;
}
```

A.1.2 VBNEUMANN

```
#include "spline.h"

output vbneumann(double x[], double grid[], int order, int derivative,
int xLength, int gridLength)
{

/********************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
```

```
*
* @Description:
*  Evaluuates the value of the basis elements of spline functions of the
*  specified order on the given grid which satisfies the Neumann boundary
*  conditions.
*
* @Usage:
*      output <name>=vbneumann(x, grid, order, derivative, xLength,
*        gridLength);
*   Input:
*      x               : array of values for x on which the basis functions
*                        are to be evaluated.
*      grid            : the grid points in ascending order, all grid points
*                         must be distinct. The interval on which the spline
*                         basis functions  are defined are given by:
*                           [grid[0], grid[N]].
*                        where N is the length of the array grid.
*      order           : order of the spline functions.
*      derivative      : order of derivative needed.
*      xLength         : an integer value showing the length of the array x.
*      gridLength      : an integer value showing the length of the array grid.
*
*   Output:
*      v               : an array of dimension order +1 by M, where M is the
*                        length of
*                        the array x.
*      ndim            : total number of basis elements, ndim=N+order-1.
*      index           : indices of the basis elements with non-zero values at a
*                        point x.  index is a 2 by M array,
*                           index[0][k]  -- lowest index of non-zero basis
*                                           element at x[k].
*                           index[1][k]  -- highest index of non-zero basis
*                                           element at x[k].
* @Note:
*      Output is a structure defined in functions.h.
******************************************************************************/


 Array2D<double> tau, v, u;
 Array2D<int> index;
 output tmp, out;
 int M=xLength, N=gridLength, k, i, ndim;
 double interval[2];

 interval[0]=grid[0];
 interval[1]=grid[N-1];
 tmp = vBasis(interval, grid, order, 1, 2, gridLength);
 tau=tmp.v;
 tmp = vBasis(x, grid, order, derivative, xLength, gridLength);
 ndim=tmp.ndim;
 u=tmp.v;
```

```
  v=u.copy();
  index=tmp.index;

  for(k=0; k<M; k++)
  {
    if(index[0][k]!=1)
      index[0][k]=index[0][k];
    else
    {
      for(i=0;i<order;i++)
        v[i][k]=u[i+1][k]-tau[i+1][0]*u[0][k]/tau[0][0];
    }
    if(index[1][k]!=ndim)
      index[1][k]=index[1][k]-1;
    else
    {
      for(i=0;i<order;i++)
        v[i][k]=v[i][k]-tau[i][1]*u[order+1][k]/tau[order+1][1];
      index[1][k]=ndim-2;
    }
    ndim=ndim-2;
    out.ndim=ndim;
    out.v=v;
    out.index=index;

    return out;
  }
}
```

A.1.3 LEAST SQUARES APPROXIMATION

```
#include "spline.h"

Array1D<double> lsqapp(Array1D<double> xdata, Array1D<double>  ydata,
 Array1D<double> wdata, Array1D<double> xgrid, int order, double alpha0,
 double alpha1)
{

/********************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *      Compute the least square approximation of the data set using a given
 *      set polynomial spline functions. The optimization functional is given
 *      by
 *   J(coef) = \sum^N_{J=1} wdata_j|L(t_j)-S(t_j)|^2
 *    \alpha_0\int^{t_max}_{t_min} |L[t]-S[t]|^2dt
 *    \alpha_1\int^{t_max}_{t_min} |L'[t]-S'[t]|^2dt,
 * where:
```

```
*   N : number of data points.
*   L : liner spline interpolation of the data.
*   S : polynomial spline function.
*        \alpha_0 : where on the L_2 norm.
*        \alpha_1 : weight on the H_1 norm.
*
* @Usage:
*      Array1D<double> <name>=vBasis(xdata, ydata, wdata, xgrid, order,
*                             alpha0, alpha1);
*   Input:
*      xdata           : data values for the independent variable.
*      ydata : data values for the dependent variable.
*      wdata : weights on the data points.
*      xgrid : grid for the spline function.
*      order  : order of the polynomial spline.
*      alpha0 : weight on the L_2 norm.
*      alpha1 : weights on the H_1 norm.
*
*   Output:
*      coef : coefficients for the optimal spline function.
*
******************************************************************************/


 output start=vBasis(xdata, xdata, 1, 0, xdata.dim(), xdata.dim());
 Array2D<double> P1(start.ndim,start.ndim), P2, P3, W(wdata.dim(),
wdata.dim()), A1, A2, A3, A, Q, intp_tmp;
 Array1D<double> r1, r2, r3, r, coef, intp;
 double xmin, xmax;
 int i, j, k;

 //Evaluate the pointwise term.
 //
 for(i=0;i<start.ndim;i++)
 {
   for(j=0;j<start.ndim;j++)
   {
     P1[j][i]=0;
   }
 }

 for(k=0; k<start.ndim; k++)
 {
   for(j=start.index[0][k]; j<=start.index[1][k]; j++)
   {
     P1[k][j] = start.v[j-start.index[0][k]][k];
   }
 }
 intp=inverse(P1)*ydata;
 output filter=vBasis(xdata, xgrid, order, 0, xdata.dim(), xgrid.dim());
 Q=Array2D<double> (start.ndim, filter.ndim);
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
for(int k=0;k<start.ndim; k++)
{
  for(int j=filter.index[0][k]; j<=filter.index[1][k];j++)
  {
    Q[k][j]=filter.v[j-filter.index[0][k]][k];
  }
}
for(i=0;i<wdata.dim();i++)
{
  for(j=0;j<wdata.dim();j++)
  {
    W[j][i]=0;
  }
}
for(i=0;i<wdata.dim();i++)
{
  W[i][i]=wdata[i];
}
for(i=0;i<intp_tmp.dim1();i++)
{
  for(int k=0; k<intp_tmp.dim2(); k++)
  {
    intp[i]+=intp_tmp[i][k]*ydata[k];
  }
}
A1=Array2D<double> (filter.ndim, filter.ndim);
r1=transpose(Q)*W*ydata;
A1=transpose(Q)*W*Q;

//Evaluate the L_2 term
//
xmin=xdata[0];
for(i=1;i<xdata.dim();i++)
{
  if(xdata[i]<xmin)
    xmin=xdata[i];
}
for(i=0;i<xgrid.dim();i++)
{
  if(xdata[i]<xmin)
    xmin=xgrid[i];
}
xmax=xdata[0];
for(i=1;i<xdata.dim();i++)
{
  if(xdata[i]>xmax)
    xmax=xdata[i];
}
for(i=0;i<xgrid.dim();i++)
{
  if(xdata[i]>xmax)
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
      xmax=xgrid[i];
  }
  A=Array2D<double> (filter.ndim, filter.ndim);
  P2=Array2D<double> (xgrid.dim(), xdata.dim());
  A2=Array2D<double> (filter.ndim, filter.ndim);
  A3=Array2D<double> (filter.ndim, filter.ndim);
  P3=Array2D<double> (xgrid.dim(), xdata.dim());
  P2=innprd(xgrid, order, 0, xdata, 1, 0, xmin, xmax, xgrid.dim(),
    xdata.dim());
  r2=P2*intp;
  A2=innprd(xgrid, order, 0, xgrid, order, 0, xmin, xmax, xgrid.dim(),
     xgrid.dim());

  //Evaluate the H_1 term
  //
  P3=innprd(xgrid, order, 1, xdata, 1, 1, xmin, xmax, xgrid.dim(), xdata.dim());
  r3=P3*intp;
  A3=innprd(xgrid, order, 1, xgrid, order, 1, xmin, xmax, xgrid.dim(),
    xgrid.dim());

  //Solve for the optimal coefficients
  //
  r=r1+alpha0*r2+alpha1*r3;
  A=A1+alpha0*A2+alpha1*A3;
  coef = inverse(A)*r;

  return coef;
}
```

A.1.4 VSPLINE

```
#include "spline.h"

Array1D<double> vspline(Array1D<double> x, Array1D<double> grid,
 int order, int derivative, Array1D<double> coef)
{

/*******************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *  Evaluate a given polynomial spline function.
 *
 * @Usage:
 *  Array2D<double> <name>=vspline(x, grid, order, derivative, coef);
 *
 *   Input:
 * x : values of the independant variable.
 * xgrid : grid on which the splines are defined.
```

```
 * order : order of spline.
 * dev : order of derivative.
 * coef : coefficients with respect to the standard basis.
 *
 *   Output:
 * v : value of the spline.
 *
 ******************************************************************************/


  Array1D<double> v(x.dim());
  output tmp;
  int i, j;

  tmp=vBasis(x, grid, order, derivative, x.dim(), grid.dim());

  if(tmp.ndim!=coef.dim())
  {
    cout << "The dimension of the coefficient vector is wrong. "
         << tmp.ndim << " " << coef.dim() << endl;
    exit(-1);
  }

  //Calculate the spline values.
  //
  for(i=0; i<x.dim(); i++)
  {
    v[i]=0;
    for(j=tmp.index[0][i]; j<=tmp.index[1][i]; j++)
    {
      v[i]=v[i]+coef[j]*tmp.v[j-tmp.index[0][i]][i];
    }
  }

  return v;
}
```

A.1.5 2D-SLICE COEFFICIENT

```
#include "spline.h"

Array2D<double> slice_coef(Array1D<double> z, Array1D<double> nr,
 Array2D<double> r, Array2D<double> nval, Array1D<double> zgrid,
 Array1D<double> nrgrid, Array2D<double> rgrid, int order, double alpha0,
 double alpha1)
{

/******************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
*
* @Description:
*  Evaluate the coeficient for each slice at p_{i} to approximate the data set
*  r(i,1:nr[i]) and nval(i,1:nr[i]). This approximation is done using one
*  dimensional approximation.
*
* @Usage:
*      Array2D<double> <name>=slice_coef(z, nr, r, nval, zgrid, nrgrid, rgrid,
*      order, alpha0, alpha1);
*
*   Input:
*    zgrid : grid points in z.
*    nrgrid : number of grid points in r at each slice.
*    rgrid : grid points in r.
*    order : order of the spline requested.
*    z : z values for data.
*    nr : number of r data points at each slice.
*    r : r measurements.
*    nval : intensity measurements.
*    alpha0 : weight for data approximation.
*    alpha1 : weight for derivative approximation.
*
*   Output:
* coefz : lenz by (nt[i]+order-1) 2 dimensional array containing
*   the approximation coefficients at each slice.
*
******************************************************************************/

 int lenz=z.dim(), i, j;
 Array2D<double> coefz(lenz, (int)nr[0]+order-1);

 for(i=0; i<lenz; i++)
 {
   Array1D<double> tmp, tmpr, tmpnval, ones, tmprgrid;
   tmpr=Array1D<double> ((int)nr[i]);
   tmpnval=Array1D<double> ((int)nr[i]);
   ones=Array1D<double> ((int)nr[i]);
   tmprgrid=Array1D<double> ((int)nrgrid[i]);

   for(j=0; j<nr[i]; j++)
   {
     tmpr[j]=r[i][j];
     tmpnval[j]=nval[i][j];
     ones[j]=1;
   }
   for(j=0; j<nrgrid[i]; j++)
   {
     tmprgrid[j]=rgrid[i][j];
   }
   tmp=lsqapp(tmpr, tmpnval, ones, tmprgrid, order, alpha0, alpha1);
   for(j=0; j<tmp.dim(); j++)
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
    {
      coefz[i][j]=tmp[j];
    }
  }

  return coefz;
}
```

A.1.6 INNER PRODUCT

```
#include "spline.h"

Array2D<double> innprd(double grd1[], int ord1, int dev1, double grd2[],
 int ord2, int dev2, double xmin, double xmax, int grd1Length, int grd2Length)
{

/*****************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *  Computes the matrix of the inner product of two families of polynomial
 *  spline basis functions. If {B_K}, k=1, ...., N and {C_j}, j=1, ..., M,
 *  then the matrix is given by:
 *  A_{k, j}=<B_k, C_j>.
 *
 * @Usage:
 *      Array2D <name>=innprd(grd1, ord1, dev1, grd2, ord2, dev2, xmin, xmax);
 *   Input:
 *      grd1            : grid of points of the first group of polynomial spline
 *        functions.
 *      ord1 : the order of the first group of spline functions.
 *      dev1 : order of the derivatives of the first group of
 *          spline functions.
 *      grd2 : grid points of the second group of polynomial spline
 *        functions.
 *      ord2 : the order of the second group of spline functions.
 *      dev2 : order of the derivatives of the second group of
 *        spline functions.
 *      xmin : lower bound of the interval of integration.
 *      xmax : upper bound of the interval of integration.
 *
 *   Output:
 *      A               : the matrix of the inner product.
 * @Note:
 *  Uses the wt function to determine the weights (needs to be changed
 *  if values other than 1 are desired).
 *
 *****************************************************************************/
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
 int z=0, nint=5, i, j, k=0, n=grd1Length, lgridLength=ord1, acount=0,
rgridLength=ord1, N1, N2, icount=0;
 double weight[5], x[5], dgrd=grd1[1]-grd1[0], lgrd[lgridLength],
     rgrd[rgridLength],
agrd1[lgridLength+grd1Length+rgridLength], grid1[1], grid2[1], u;
 double alpha[5], y[5];
 Array2D<double> A;
 output fltr1, fltr2;

 weight[0]=0.2369268851;
 weight[1]=0.4786286705;
 weight[2]=0.5688888889;
 weight[3]=weight[1];
 weight[4]=weight[0];
 x[0]=-0.9061798459;
 x[1]=-0.5384693101;
 x[2]=0;
 x[3]=-x[1];
 x[4]=-x[0];

 //construct the combined grid
 //

 for(i=0;i<lgridLength;i++)
 {
   lgrd[i]=dgrd*i + grd1[0]-ord1*dgrd;
 }
 dgrd=grd1[n-1] - grd1[n-2];
 for(i=0;i<rgridLength;i++)
 {
     rgrd[i]=grd1[n-1] + dgrd*(i+1);
 }
 for(i=0;i<ord1;i++)
 {
   agrd1[acount]=lgrd[i];
   acount++;
 }
 for(i=0;i<grd1Length;i++)
 {
   agrd1[acount]=grd1[i];
   acount++;
 }
 for(i=0;i<ord1;i++)
 {
   agrd1[acount]=rgrd[i];
   acount++;
 }
 lgridLength=ord2;
 rgridLength=ord2;
 n=grd2Length;
 acount=0;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
dgrd=grd2[1]-grd2[0];
for(i=0;i<lgridLength;i++)
{
  lgrd[i]=dgrd*i + grd2[0]-ord2*dgrd;
}
double agrd2[lgridLength+grd2Length+rgridLength];
dgrd=grd2[n-1]-grd2[n-2];
for(i=0;i<rgridLength;i++)
{
    rgrd[i]=grd2[n-1] +dgrd*(i+1);
}
for(i=0;i<ord2;i++)
{
  agrd2[acount]=lgrd[i];
  acount++;
}
for(i=0;i<grd2Length;i++)
{
  agrd2[acount]=grd2[i];
  acount++;
}
for(i=0;i<ord2;i++)
{
  agrd2[acount]=rgrd[i];
  acount++;
}
double cgrid[ord1*2+ord2*2+grd1Length+grd2Length];
int ccount=0;
for(i=0;i<ord1*2+grd1Length;i++)
  cgrid[ccount++]=agrd1[i];
for(i=0;i<ord2*2+grd2Length;i++)
  cgrid[ccount++]=agrd2[i];
int elements= sizeof(cgrid)/sizeof(double);
sort(cgrid, elements+cgrid);

double igrd[2+ccount];
igrd[icount++]=xmin;
double cx=cgrid[0];
i=0;
while(cx!=cgrid[ccount-1])
{
  if(cx >= xmax)
    break;
  if(igrd[k]<cx)
  {
    igrd[icount++]=cx;
    k++;
  }
  cx=cgrid[i++];
}
igrd[icount++]=xmax;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
  grid1[0]=grd1[0];
  grid2[0]=grd2[0];
  fltr1=vBasis(grid1, grd1, ord1, dev1, 1, grd1Length);
  fltr2=vBasis(grid2, grd2, ord2, dev2, 1, grd2Length);

  //Calculate the inner product matrix
  //
  N1=fltr1.ndim;
  N2=fltr2.ndim;
  A=Array2D<double>(N1,N2);
  for(i=0;i<fltr2.ndim;i++)
  {
    for(j=0;j<fltr1.ndim;j++)
    {
      A[j][i]=0;
    }
  }
  for(z=0;z<icount-1;z++)
  {
    double a=igrd[z];
    double b=igrd[z+1];
    for(i=0;i<5;i++)
      y[i]=(b+a)/2+(b-a)*x[i]/2;
    fltr1=vBasis(y, grd1, ord1, dev1, 5, grd1Length);
    fltr2=vBasis(y, grd2, ord2, dev2, 5, grd2Length);
    wt(y, 5, alpha);
    for(i=0;i<nint;i++)
    {
      for(k=fltr1.index[0][i]; k<=fltr1.index[1][i]; k++)
      {
        for(j=fltr2.index[0][i];j<=fltr2.index[1][i];j++)
        {
          u=fltr1.v[k-fltr1.index[0][i]][i]*fltr2.v[j-fltr2.index[0][i]][i];
          A[k][j]=A[k][j]+(b-a)*u*weight[i]*alpha[i]/2;
        }
      }
    }
  }

  return A;
}
```

A.1.7 WEIGHTS

```
#include "spline.h"

void wt(double y[], int length, double alpha[])
{

/*******************************************************************************
 * @Author: Dane Burrows
```

```
 *
 * @Date 9-July-07
 *
 * @Description:
 *  Returns a weight for calculating the inner product.
 *
 * @Usage:
 *      wt(y, length, alpha);
 *
 *   Input:
 *      y : array of dimension: length which can be used for
 *         evaluating the weight.
 *      length : length of the array: y.
 *      alpha : empty array to be filled with the result.
 *
 ******************************************************************************/


  for(int i=0;i<5;i++)
    alpha[i]=1;
}
```

A.1.8 PLOT BASIS

```
#include "spline.h"

Array2D<double> plotBasis(Array2D<double> v, Array2D<int> index, double x[],
int order, int vLength, int ndim)
{

/******************************************************************************
 * @Name: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *      This routine evaluates the values of the B-spline basis.
 *
 * @Usage:
 *      output <name>=vBasis(x, grid, order, derivative, xLength, gridLength);
 *   Input:
 *      v                : an array of dimension order +1 by M, where M is the
 *                         length
 *                        of the array x
 *      index            : indices of the basis elements with non-zero values
 *                         at a point x.  index is a 2 by M array,
 *                            index[0][k]  -- lowest index of non-zero basis
 *                                            element at x[k].
 *                            index[1][k]  -- highest index of non-zero basis
 *                                            element at x[k].
```

```
 *      x               : array of values for x on which the basis functions
 *                        are to be evaluated.
 *      order           : order of the spline functions.
 *      vLength : length of the array v.
 *      ndim            : total number of basis elements, ndim=N+order-1.
 *
 *   Output:
 *      u               : an array that can be graphed to show the basis
 *                        functions.
 *
 ****************************************************************************/

  int i, j, k;
  Array2D<double> u(ndim, vLength);
  for(i=0;i<vLength;i++)
  {
    for(j=0;j<ndim;j++)
    {
      if(index[0][i]<=j&&j<=index[1][i])
      {
        u[j][i]=v[j-index[0][i]][i];
      }
      else
       u[j][i]=0;
    }
  }

  return u;
}
```

A.1.9 SIGN

```
#include "spline.h"

double sign(double a)
{

  /****************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *  Checks the sign of a number and returns the sign as either 1, -1, or 0.
 *
 * @Usage:
 *      output <name>=vBasis(x, grid, order, derivative, xLength, gridLength);
 *   Input:
 *      a               : A double value to have its sign evaluated.
 *   Output:
 *      x               : A double value of either 1.0 (positive) -1.0
```

```
 *                         (negative) or 0.0.
 *
 ****************************************************************************/


  double x=0;
  if(a==0)
    x=0;
  if(a<0)
    x=-1;
  if(a>0)
    x=1;

  return x;

}
```

A.1.10 SURF VALUE

```
#include "spline.h"

Array2D<double> surf_value(Array1D<double> x, Array1D<double> y,
Array1D<double> zgrid, Array1D<double> nrgrid, Array2D<double> rgrid,
 int rorder, int zdev, int rdev, Array2D<double> coefz)
{

/*****************************************************************************
 * @Author: Dane Burrows
 *
 * @Date 9-July-07
 *
 * @Description:
 *  Evaluate the approximation for a finite number of slices at given points
 *  zgrid[i] using interpolation in z.
 *
 * @Usage:
 *      Array2D <name>=surf_val(x, y, zgrid, nrgrid, rgrid, rorder, zdev, rdev,
 *                      coefz);
 *   Input:
 *    x : z values where surface is requested.
 *    y : r values where surface is requested.
 *    zgrid : grid points in z.
 *    nrgrid : number of r points at each z grid.
 *    rgrid : r grid points.
 *    order : order of spline.
 *    dev : order of derivative.
 *    coefz : coefficents with respect to the standard basis at each
 *      z values in zgrid[i].
 *
 *   Output:
 *    v : value of the z interpolating slice function.
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
 *
 ******************************************************************************/

  int i, j, k, lenx, n;
  double weight1, weight2;
  Array2D<double> v;

  n=zgrid.dim();
  lenx=x.dim()-1;

  for(i=0; i<lenx; i++)
  {
    Array1D<double> tmp;
    for(j=0;j<n-1; j++)
    {
      //Interpolate between the first two slices
      //
      Array1D<double> tmprgrid1((int)nrgrid[j]),
      tmprgrid2((int)nrgrid[j+1]),
tmp_grid1((int)nrgrid[j]), tmp_grid2((int)nrgrid[j+1]),
tmp_grid_z((int)nrgrid[j]), tmp_grid_r((int)nrgrid[j]),
tmp_coef1(coefz.dim2()), tmp_coef2(coefz.dim2()),
tmp_coef_z(coefz.dim2()), tmp_coef_r(coefz.dim2());
      for(k=0;k<nrgrid[j];k++)
      {
        tmprgrid1[k]=rgrid[j][k];
        tmprgrid2[k]=rgrid[j+1][k];
      }
      if((x[i] >= zgrid[j])&&(x[i]<zgrid[j+1]))
      {
        weight1=(x[i]-zgrid[j])/(zgrid[j+1]-zgrid[j]);
        tmp_grid1=tmprgrid1;
        tmp_grid2=tmprgrid2;
        if(zdev==1)
        {
          weight2=1/(x[i]*(log(zgrid[j+1])-log(zgrid[j])));
          for(k=0;k<tmp_grid1.dim();k++)
            tmp_grid_z[k]=(tmp_grid2[k]-tmp_grid1[k])*weight2;
        }
        for(k=0;k<tmp_grid1.dim();k++)
        {
          tmp_grid_r[k]=tmp_grid1[k]+(tmp_grid2[k]-tmp_grid1[k])*weight1;
        }
        for(k=0;k<coefz.dim2();k++)
        {
          tmp_coef1[k]=coefz[j][k];
          tmp_coef2[k]=coefz[j+1][k];
        }
        if(zdev==1)
        {
          weight2=1/(x[i]*(log(zgrid[j+1])-log(zgrid[j])));
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
          for(k=0;k<tmp_grid1.dim();k++)
            tmp_coef_z[k]=(tmp_coef2[k]-tmp_coef1[k])*weight2;
        }

        for(k=0;k<tmp_coef1.dim();k++)
        {
          tmp_coef_r[k] = tmp_coef1[k]+(tmp_coef2[k]-tmp_coef1[k])*weight1;
        }
      tmp=vspline(y, tmp_grid_r, rorder, zdev, tmp_coef_r);
      }
    }
    // Calculate the surface value at points requested
    //
    if(i==0)
      v=Array2D<double> (tmp.dim(), lenx);
    for(j=0;j<tmp.dim(); j++)
    {
      v[j][i]=tmp[j];
    }
  }

  return v;
}
```

A.2 An example in C

A.2.1 VBASIS

```
#include "spline.h"

int main()
{
  int order, derivative;
  int xlength, i, gridLength, j;
  gridLength=11;
  xlength=101;
  Array1D<double> grid(gridLength), x(xlength);
  Array2D<double> plots;
  ofstream output1("vBasis.out");

  for(i=0;i<gridLength;i++)
    grid[i]=i;
  for(i=0; i<xlength; i++)
    x[i]=i/10.0;

  order=3;
  derivative=0;

  output vbout=vBasis(x, grid, order, derivative, xlength, gridLength);
  plots=plotBasis(vbout.v, vbout.index, x, order, vbout.v.dim2(), vbout.ndim);
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
  for(j=0;j<vbout.ndim;j++)
  {
    for(i=0;i<vbout.v.dim2();i++)
    {
      output1 <<x[i] << " " << plots[j][i]<<endl;
    }
    output1 <<endl;
  }

  return 0;
}
```

A.2.2 VBNEUMANN

```
#include "spline.h"

int main()
{
  int order, derivative;
  int xlength, i, gridLength, j;
  gridLength=11;
  xlength=101;
  Array1D<double> grid(gridLength), x(xlength);
  for(i=0;i<gridLength;i++)
    grid[i]=i;
  for(i=0; i<xlength; i++)
    x[i]=i/10.0;
  ofstream output1("vbneumann.out");

  order=3;
  derivative=0;

  output vbout=vbneumann(x, grid, order, derivative, xlength, gridLength);
  Array2D<double> plots = plotBasis(vbout.v, vbout.index, x, order,
  vbout.v.dim2(), vbout.ndim);

  for(i=0;i<vbout.ndim;i++)
  {
    for(j=0;j<vbout.v.dim2();j++)
    {
      output1 << x[j] << " " << plots[i][j] << endl;
    }
    output1 << endl;
  }

  return 0;
}
```

A.2.3 VSPLINE

```
#include "spline.h"
```

```
int main()
{
  int order, derivative;
  int xlength, i, xdataLength, gridLength, zLength=11, yLength=101,
aLength=21, j;
  xdataLength=11;
  gridLength=11;
  xlength=101;
  ofstream output1("ydata.out");
  ofstream output2("vspline.out");
  Array1D<double> xdata(xdataLength), grid(gridLength), ydata(xdataLength),
x(xlength), vspln;
  for(i=0;i<xdataLength;i++)
    xdata[i]=i;
  for(i=0;i<gridLength;i++)
    grid[i]=i;
  for(i=0; i<xdataLength; i++)
  {
    ydata[i]=exp(-xdata[i]);
    output1 << xdata[i] << " " << ydata[i] << endl;
  }
  for(i=0; i<xlength; i++)
    x[i]=i/10.0;
  order=3;
  derivative=0;
  Array1D<double> approx;
  approx=lsqapp(xdata, ydata, ydata, grid, order, 0.1, 0.01);
  vspln=vspline(x, grid, order, derivative, approx);

  for(int i=0; i < vspln.dim(); i ++)
    output2 << i/((double)xlength-1)*10.0 << " " << vspln[i]<< endl;

  return 0;
}
```

A.2.4 LEAST SQUARES APPROXIMATION

```
#include "spline.h"

int main()
{
  int order;
  int xlength, i, j, xdataLength, gridLength;
  xdataLength=11;
  gridLength=11;
  Array1D<double> xdata(xdataLength), grid(gridLength), ydata(xdataLength);
  for(i=0;i<xdataLength;i++)
    xdata[i]=i;
  for(i=0;i<gridLength;i++)
    grid[i]=i;
  for(i=0; i<xdataLength; i++)
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
    ydata[i]=exp(-xdata[i]);
  ofstream output1("lsqapp.out");
  order=3;
  Array1D<double> approx;
  approx=lsqapp(xdata, ydata, ydata, grid, order, 0.1, 0.01);

  for(i=0;i<approx.dim();i++)
  {
    output1 << approx[i] << endl;
  }

  return 0;
}
```

A.2.5 SLICE COEFFICIENTS

```
#include "spline.h"

int main()
{
  int order, derivative;
  int i, zLength=11, aLength=21, j;

  order=3;
  derivative=0;
  Array1D<double> z(zLength), nr(zLength);
  for(i=0; i<zLength; i++)
    nr[i]=aLength;
  Array2D<double> r(zLength, aLength);
  Array2D<double> nval(zLength, aLength);
  ofstream output1("slice.out");

  for(i=0; i<zLength; i++)
    z[i]=i;
  for(i=0; i<zLength; i++)
  {
    for(int j=0; j<nr[0]; j++)
    {
      r[i][j]=-10+j;
      nval[i][j]=-0.001*exp(-r[i][j]*r[i][j])*exp(-z[i])+1.35;
    }
  }

  Array2D<double> slice;
  slice=slice_coef(z, nr, r, nval, z, nr, r, order, .1, .01);
  for(i=0;i<slice.dim2();i++)
  {
    for(j=0;j<slice.dim1();j++)
    {
      output1 << j << " " << i << " " << slice[j][i] << endl;
    }
```

```
    output1 << endl;
  }

  return 0;
}
```

A.2.6 SURF VALUE

```
#include "spline.h"

int main()
{
  int order, derivative;
  int xlength, i, gridLength, zLength=11, yLength=101, aLength=21, j;
  xlength=101;
  Array1D<double>  x(xlength), y(yLength);
  for(i=0; i<xlength; i++)
    x[i]=i/10.0;
  for(i=0; i<yLength; i++)
    y[i]=-10+i/5.0;
  ofstream output1("surf.out");

  order=3;
  derivative=0;
  double temp[1];
  Array1D<double> z(zLength), nr(zLength);
  for(i=0; i<zLength; i++)
    nr[i]=aLength;
  Array2D<double> r(zLength, aLength);
  Array2D<double> nval(zLength, aLength);

  for(i=0; i<zLength; i++)
    z[i]=i;
  for(i=0; i<zLength; i++)
  {
    for(int j=0; j<nr[0]; j++)
    {
      r[i][j]=-10+j;
      nval[i][j]=-0.001*exp(-r[i][j]*r[i][j])*exp(-z[i])+1.35;
    }
  }

  Array2D<double> slice, surf;
  slice=slice_coef(z, nr, r, nval, z, nr, r, order, .1, .01);
  surf=surf_value(x, y, z, nr, r, order, 0, 0, slice);
  for(i=0;i<surf.dim2();i++)
  {
    for(j=0;j<surf.dim1();j++)
    {
      output1 << j << " " << i << " " << surf[j][i] << endl;
    }
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
    output1 << endl;
  }

  return 0;
}
```

A.3 MATLAB Code

A.3.1 VBASIS

```
function [v,ndim,index]=vbasis(x,grid,order,derivative)
%
% Description: This routine evaluates the values of the B-spline basis
%    (or the derivatives) functions at given points. The grid points and
%    the order of the splines are specified by user, however, additional
%    grid points outside of the interval [xmin, xmax] are choosen by
%    the program to provide a complete basis.
% Usage:
%       [v,ndim,index]=vbasis(x,grid,order,derivative);
%    Input:
%       x          : row vector of values for x on which the basis functions
%                    are to be evaluated.
%       grid       : the grid points in ascending order, all grid points
%                    must be distinct. The interval on which the spline
%                    basis functions are defined are given by:
%                       [grid(1),grid(N)]
%                    where N is the length of the vector grid.
%       order      : order of the spline functions.
%       derivative : order of derivative needed.
%                    derivative=0, value of the function is requested.
%    Output:
%       v          : an array of dimension order+1 by M, where M is the
%                    length of vector x.
%       ndim       : total number of basis elements, ndim = N+order-1.
%       index      : the indices of basis elements with non-zero values
%                    at a point x. index is a 2 by M matrix,
%                        index(1,k)  -- lowest index of non-zero basis element
%                                       at x(k).
%                        index(2,k)  -- highest index of non-zero basis element
%                                       at x(k).
%
       N=length(grid);
       M=length(x);
       ndim=N+order-1;
%
% Construct the augmented grid.
%
       dgrid=grid(2)-grid(1);
       lgrid=grid(1)-order*dgrid:dgrid:grid(1)-dgrid;
       dgrid=grid(N)-grid(N-1);
       rgrid=grid(N)+dgrid:dgrid:grid(N)+order*dgrid;
```

```
       agrid=[lgrid, grid, rgrid];
%
% Main loop over points x.
%
       v=[];
       index=[];
       for k=1:M
%
% Determine the interval [grid(j), grid(j+1)] in which x(k) belongs.
%
           for j=1:N-1
              if sign(x(k)-grid(j))*sign(grid(j+1)-x(k)) >= 0
                 break;
              end;
           end;
           if x(k) < grid(1)
              j=1;
           end;
           if x(k) > grid(N)
              j=N-1;
           end;
%
% Evaluate the values of the basis functions (or derivatives) at x(k).
%   1. Evaluate the values of the truncated polynomials.
%
           factor=1;
           if derivative > 0
              for i=0:derivative-1
                  factor=factor*(order-i);
              end;
           end;
           trunc = zeros(1,2*order+2);
           localg = agrid(j:j+2*order+1);
           for i=1:2*order+2
               trunc(i)=max([(x(k)-agrid(j+i-1)),0]);
               if order > derivative
                  trunc(i)=factor*(trunc(i)^(order-derivative));
               elseif order == derivative
                  trunc(i)=factor*sign(trunc(i));
               else
                  disp('The spline function is not differentiable');
                  return;
               end;
           end;
%
%   2. Compute the divided differences.
%
           for l=1:order+1
               for ll=1:2*order+2-l
                   trunc(ll)=(trunc(ll+1)-trunc(ll))/(localg(ll+l)-localg(ll));
               end;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
          end;
%
%   3. Store the value in the vector v.
%
          itrunc=[j;j+order];
          index = [index itrunc];
          trunc = trunc(1:order+1);
          v = [v trunc'];
       end;
```

A.3.2 VBNEUMANN

```
function [v,ndim,index]=vbneumann(x,grid,order,derivative)
%
% Description: evaluates the value of basis elements of spline functions
%    of the specified order on the given grid which satisfies the
%    Neumann boundary conditions.
% Usage:
%       [v,ndim,index]=vbneumann(x,grid,order,derivative);
%    Input:
%       x          : row vector of values for x on which the basis functions
%                    are to be evaluated.
%       grid       : the grid points in ascending order, all grid points
%                    must be distinct. The interval on which the spline
%                    basis functions are defined are given by:
%                       [grid(1),grid(N)]
%                    where N is the length of the vector grid.
%       order      : order of the spline functions.
%       derivative : order of derivative needed.
%                    derivative=0, value of the function is requested.
%    Output:
%       v          : an array of dimension order+1 by M, where M is the
%                    length of vector x.
%       ndim       : total number of basis elements, ndim = N+order-3.
%       index      : the indices of basis elements with non-zero values
%                    at a point x. index is a 2 by M matrix,
%                        index(1,k)  -- lowest index of non-zero basis element
%                                       at x(k).
%                        index(2,k)  -- highest index of non-zero basis element
%                                       at x(k).
%
        N = length(grid);
        interval = [grid(1),grid(N)];
        [tau,ndim,index] = vbasis(interval,grid,order,1);
        [u,ndim,index] = vbasis(x,grid,order,derivative);
        M = length(x);
        v=u;
%
% Impose the boundary conditions.
%
        for k=1:M
```

```
          if index(1,k) ~= 1
             index(1,k) = index(1,k)-1;
          else
             for i=1:order
                 v(i,k) = u(i+1,k)-tau(i+1,1)*u(1,k)/tau(1,1);
             end;
          end;
          if index(2,k) ~= ndim
             index(2,k) = index(2,k)-1;
          else
             for i=1:order
                 v(i,k) = v(i,k)-tau(i,2)*u(order+1,k)/tau(order+1,2);
             end;
             index(2,k) = ndim-2;
          end;
      end;
      ndim = ndim-2;
      return;
```

A.3.3 LEAST SQUARES APPROXIMATION

```
function [coef]=lsqapp(xdata,ydata,wdata,...
                       xgrid,order,filter,alpha0,alpha1)
%
% Description: Compute the least square approximation of the data set
%   using a given set polynomial spline functions. The optimization
%   functional is given by
%     J(coef) = \sum^N_{j=1} wdata_j|L(t_j)-S(t_j)|^2
%            \alpha_0\int^{t_max}_{t_min} |L(t)-S(t)|^2dt
%            \alpha_1\int^{t_max}_{t_min} |L'(t)-S'(t)|^2dt,
%   where:
%      N         : number of data points.
%      L         : linear spline interpolation of the data.
%      S         : polynomial spline function.
%      \alpha_0  : weight on the L_2 norm.
%      \alpha_1  : weight on the H_1 norm.
% Usage:
%    [coef]=lsqapp(xdata,ydata,wdata,xgrid,order,filter,alpha0,alpha1)
%  where:
%     xdata      : data values for the independent variable.
%     ydata      : data values for the dependent variable.
%     wdata      : weights on the data points.
%     xgrid      : grid for the spline function.
%     order      : order of the polynomial spline.
%     filter     : routine for evaluation of the spline functions.
%     alpha0     : weight on the L_2 norm.
%     alpha1     : weight on the H_1 norm.
%     coef       : coefficients for the optimal spline function.
%
      [u,ndata,index]=vbasis(xdata,xdata,1,0);
%
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
% Evaluate the pointwise term.
%
      P1 = zeros(ndata,ndata);
      for k=1:ndata
          for j=index(1,k):index(2,k)
              P1(k,j) = u(j-index(1,k)+1,k);
          end;
      end;
      intp = inv(P1)*ydata';
      [u,ndim,index]=feval(filter,xdata,xgrid,order,0);
      Q = zeros(ndata,ndim);
      for k=1:ndata
          for j=index(1,k):index(2,k)
              Q(k,j) = u(j-index(1,k)+1,k);
          end;
      end;
      W = diag(wdata,0);
      r1 = Q'*W*ydata';
      A1 = Q'*W*Q;
%
% Evaluate the L_2 term.
%
      xmin = min([xdata xgrid]);
      xmax = max([xdata xgrid]);
      [P2] = innprd(xgrid,order,0,filter,xdata,1,0,'vbasis',...
                   'unif',xmin,xmax);
      r2 = P2*intp;
      [A2] = innprd(xgrid,order,0,filter,xgrid,order,...
                    0,filter,'unif',xmin,xmax);
%
% Evaluate the H_1 term.
%
      [P3] = innprd(xgrid,order,1,filter,xdata,1,1,'vbasis',...
                   'unif',xmin,xmax);
      r3 = P3*intp;
      [A3] = innprd(xgrid,order,1,filter,xgrid,order,...
                    1,filter,'unif',xmin,xmax);
%
% Solve for the optimal coefficients.
%
      r = r1+alpha0*r2+alpha1*r3;
      A = A1+alpha0*A2+alpha1*A3;
      coef = (inv(A)*r)';
      return;
```

A.3.4 VSPLINE

```
function [v]=vspline(x,xgrid,order,dev,filter,coef)
%
% Description: Evaluate a given polynomial spline function.
% Usage:
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
%        [v]=vspline(x,xgrid,order,dev,filter,coef);
% where:
%       x      : values of the independent variable.
%       xgrid  : grid on which the splines are defined.
%       order  : order of spline.
%       dev    : order of derivative.
%       filter : filter used to impose boundary conditions.
%       coef   : coefficients with respect to the standard basis.
%       v      : value of the spline.
%
       [u,ndim,index]=feval(filter,x,xgrid,order,dev);
       if ndim ~= length(coef)
          disp('The dimension of the coefficient vector is wrong.');
          return;
       end;
       v = zeros(size(x));
%
% Calculate the spline values.
%
       for k=1:length(x)
           for j=index(1,k):index(2,k)
               v(k) = v(k)+coef(j)*u(j-index(1,k)+1,k);
           end;
       end;
       return;
```

A.3.5 2D-SLICE COEFFICIENT

```
function [coefp]=slice_coef(p,nt,t,ival,pgrid,ntgrid,tgrid,order,filter,...
                           alpha0,alpha1)
%
% Description: Evaluate coefficient for each slice at p_{i} to approximate
%   the data set t(i,1:nt(i)) and ival(i,1:nt(i)).  This approximation
%   is done using one dimensional approximation.
% Usage:
%        [coef] = slice_coef(pgrid,ntgrid,tgrid,order,filter,p,nt,t,ival,...
%                            alpha0,alpha1);
%
% Inputs:
%       pgrid        : grid points in z
%       ntgrid       : number of grid points in r
%       tgrid        : grid points in r
%       order        : order of the spline requested.
%       filter       : filter requested.
%       p            : z values  for data.
%       nt           : number r data points at each slice.
%       t            : r measurements.
%       ival         : refractive index measurements
%       alpha0       : weight for data approximation.
%       alpha1       : weight for derivative approximation.
%
```

```
% Outputs:
%      coefp        : lenp by (nt(i)+order-1) matrix containing the
%                     approximation coefficients at each slice.
%
       lenp = max(size(p));
       coefp = [];
       for i=1:lenp
           [tmp]=lsqapp(t(i,1:nt(i)),ival(i,1:nt(i)),ones(1,nt(i)),...
                        tgrid(i,1:ntgrid(i)),order,filter,alpha0,alpha1);
           coefp = [coefp;tmp];
       end;
       return;
```

A.3.6 INNER PRODUCT

```
function [A]=innprd(grd1,ord1,dev1,fltr1,grd2,ord2,dev2,fltr2,wt,xmin,xmax)
%
% Description: Compute the matrix of the inner product of two families of
%    polynomial spline basis functions. If {B_k}, k=1,...,N and {C_j},
%    j=1,...,M, then the matrix is given by
%            A_{k,j} = <B_k, C_j>.
% Usage:
%    [A]=innprd(grd1,ord1,dev1,'fltr1',grd2,ord2,dev2,'fltr2',wt,xmin,xmax);
%    Input:
%         grd1      : grid points of the first group of polynomial spline
%                     functions.
%         ord1      : the order of the first group of spline functions.
%         dev1      : order of the derivatives of the first group of
%                     spline functions.
%         fltr1     : filter that modifies the basis elements in order
%                     to satisfy the boundary condition.
%         grd2      : grid points of the second group of polynomial spline
%                     functions.
%         ord2      : the order of the second group of spline functions.
%         dev2      : order of the derivatives of the second group of
%                     spline functions.
%         fltr2     : filter that modifies the basis elements in order
%                     to satisfy the boundary condition.
%         wt        : weight function in the inner product.
%         xmin      : lower bound of the interval of integration.
%         xmax      : upper bound of the interval of integration.
%    Output:
%         A          : the matrix of inner product.
%
        weight = [0.2369268851, 0.4786286705, 0.5688888889];
        weight = [weight weight(2) weight(1)];
        x = [-0.9061798459, -0.5384693101, 0];
        x = [x -x(2) -x(1)];
%
% Construct the combined grid.
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
%
        nint = 5;
        n = length(grd1);
        dgrd=grd1(2)-grd1(1);
        lgrd=grd1(1)-ord1*dgrd:dgrd:grd1(1)-dgrd;
        dgrd=grd1(n)-grd1(n-1);
        rgrd=grd1(n)+dgrd:dgrd:grd1(n)+ord1*dgrd;
        agrd1=[lgrd, grd1, rgrd];
        n = length(grd2);
        dgrd=grd2(2)-grd2(1);
        lgrd=grd2(1)-ord2*dgrd:dgrd:grd2(1)-dgrd;
        dgrd=grd2(n)-grd2(n-1);
        rgrd=grd2(n)+dgrd:dgrd:grd2(n)+ord2*dgrd;
        agrd2=[lgrd, grd2, rgrd];
        cgrd = [agrd1 agrd2];
        cgrd = sort(cgrd);
%
% Choose the maximum and the minimum grid point.
%
        igrd = xmin;
        k = 1;
        for cx = cgrd
            if cx >= xmax
               break;
            end;
            if igrd(k) < cx
               igrd = [igrd cx];
               k = k+1;
            end;
        end;
        igrd = [igrd xmax];
        [v1,N1,index1] = feval(fltr1,grd1(1),grd1,ord1,dev1);
        [v2,N2,index2] = feval(fltr2,grd2(1),grd2,ord2,dev2);
%
% Calculate the inner product matrix.
%
        A = zeros(N1,N2);
        for k= 1:length(igrd)-1
          a = igrd(k);
          b = igrd(k+1);
          y = (b+a)/2+(b-a)*x/2;
          [v1,ndim,index1] = feval(fltr1,y,grd1,ord1,dev1);
          [v2,ndim,index2] = feval(fltr2,y,grd2,ord2,dev2);
          alpha = feval(wt,y);
          for i = 1:nint
          for k = index1(1,i):index1(2,i)
          for j = index2(1,i):index2(2,i)
             u = v1(k-index1(1,i)+1,i)*v2(j-index2(1,i)+1,i);
             A(k,j) = A(k,j)+(b-a)*u*weight(i)*alpha(i)/2;
          end;
          end;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

```
        end;
      end;
      return;
```

A.3.7 UNIFORM WEIGHTS

```
function [v]=unif(x)
%
% Description:
%   Uniform weight function.
%
    v = ones(size(x));
    return;
```

A.3.8 PLOT BASIS

```
function plot_basis(x,knots,order,deriv,filter)
% This function plots the basis functions for a particular polynomial splines
% with the given order and derivative for the knots specified.
[v,ndim,index] = feval(filter,x, knots, order, deriv);
nx = length(x);
v2 = zeros(ndim,nx);
for j=1:nx
    for i=index(1,j):index(2,j)
        v2(i,j) = v(i-index(1,j)+1,j);
    end;
end;
plot(x,v2(1,:));
hold on;
pause;
for i=2:ndim
    plot(x,v2(i,:));
    pause;
end;
return;
```

A.3.9 SURF VALUE

```
function [v]=surf_value(x,y,pgrid,ntgrid,tgrid,torder,pdev,tdev,filter,coefp)
%
% Description: Evaluate the approximation for a finite number of slices at
%  given z points pgrid(i) using logarithmic interpolation in p.
% Usage:
%       [v] = surf_val(x,y,pgrid,ntgrid,tgrid,order,dev,filter,coefp);
% where:
%      x      : z values where surface is requested.
%      y      : r values where surface is requested.
%      pgrid  : grid points in z.
%      ntgrid : number of r grid points at each z grid.
%      tgrid  : r grid points.
%      order  : order of spline.
%      dev    : order of derivative.
```

```
%       filter : filter used to impose boundary conditions.
%       coefp  : coefficients with respect to the standard basis at each
%                pressure values in pgrid(i).
%       v      : value of the p-logarithmic interpolating slice function.
%
        lenx = length(x);
        v = [];
        n = length(pgrid);
        for i=1:lenx
            for j=1:n-1
%
% Interpolate between first two slices.
%
            if ((x(i) >=pgrid(j)) & (x(i) < pgrid(j+1)))
               %weight1=(log(x(i))-log(pgrid(j)))/(log(pgrid(j+1))-
log(pgrid(j)));
               weight1=(x(i)-pgrid(j))/(pgrid(j+1)-pgrid(j));
               tmp_grid1 = tgrid(j,1:ntgrid(j));
               tmp_grid2 = tgrid(j+1,1:ntgrid(j+1));
               if (pdev == 1)
                  weight2 = 1/(x(i)*(log(pgrid(j+1))-log(pgrid(j))));
                  tmp_grid_p = (tmp_grid2-tmp_grid1)*weight2;
               end;
               tmp_grid_t = tmp_grid1 + (tmp_grid2-tmp_grid1)*weight1;
               tmp_coef1 = coefp(j,:);
               tmp_coef2 = coefp(j+1,:);
               if (pdev == 1)
                  weight2 = 1/(x(i)*(log(pgrid(j+1))-log(pgrid(j))));
                  tmp_coef_p = (tmp_coef2-tmp_coef1)*weight2;
               end;
               tmp_coef_t = tmp_coef1 + (tmp_coef2-tmp_coef1)*weight1;
            end;
            end;
%
% Calculate the surface value at points requested.
%
            tmp = vspline(y,tmp_grid_t,torder,tdev,filter,tmp_coef_t);
            v = [v,tmp'];
        end;
        return;
```

A.3.10 DEFINING THE REFRACTIVE INDEX

```
function [n] = rindex_profile(r,z,mua,theta,A,nbar)
%This function computes the refractive index profile from this class of
functions:
%            n(r,z) = A*exp(-r^2/theta^2)*exp(-mua*z)
%
% Input:
%    x                   :   x grid of size nx x 1
%    y                   :   y grid of size ny x 1
```

```
%    z                  :   z grid of size nz x 1
%    mua                :   absorption coefficient (scalar)
%    theta              :   spread parameter (scalar)
%    A                  :   percent deviation from nbar (scalar)
%    nbar               :   background refractive index (scalar)
%
% Output:
%    n                  :   refractive index profile at (r,z) of size nr x nz
%
nr = length(r);
nz = length(z);
n = zeros(nr,nz);
for i=1:nr
    for j=1:nz
        n(i,j) = -A*exp(-(r(i)^2)/(theta^2))*exp(-mua*z(j))+nbar;
    end;
end;
return;
```


DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited

References

[1] J.C. Butcher. *The Numerical Analysis of Ordinary Differential Equations*. J. Wiley, 1987.

[2] D. W. Harder. Numerical Methods for Electrical and Computer Engineers, 2006. Accessed July 20, 2007. http://www.ece.uwaterloo.ca/~ece204/TheBook/14IVPs/rkf45/theory.html.

[3] John H. Mathews. Module for Runge-Kutta-Fehlberg method for O.D.E's, 2004. Accessed June 12, 2004. http://math.fullerton.edu/mathews/n2003/RungeKuttaFehlbergMod.html.

[4] Larry Schumaker. *Spline Functions Basic Theory*. Wiley, 1980.

[5] Anurag Sharma, D. Vizia Kumar, and A.K. Ghatak. Tracing rays through graded-index media: a new method. *Appl. opt.*, 21(6):984–987, 1982.

[6] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer, 1980.

[7] Eric Weisstein. Gershgorin circle theorem. From *Mathworld*–A Wolfram Web Resource. Accessed July 3, 2007. http://mathworld.wolfram.com/GershgorinCircleTheorem.html.

DISTRIBUTION A: Approved for Public Release; Distribution Unlimited